Modelling and Programming Embedded Controllers with Timed Automata and Synchronous Languages

T. Bourke

Ph.D.

2009

UNSW
THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • AUSTRALIA

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation, and linguistic expression is acknowledged.

Signed .....

Date .....

Abstract

Embedded controllers coordinate the behaviours of specialised hardware components to satisfy broader application requirements. They are difficult to model and to program. One of the greatest challenges is to express intricate timing behaviours—which arise from the physical characteristics of components—while not precluding efficient implementations on resource-constrained platforms. Aspects of this challenge are addressed by this thesis through four distinct applications of timed automata and the synchronous languages Argos and Esterel.

A novel framework for simulating controllers written in an imperative synchronous language is described. It includes a transformation of synchronous models into timed automata that accounts for timing properties which are important in constrained implementations but ignored by the usual assumption of synchrony. The transformation provides an interface between the discrete time of synchronous programs and a continuous model of time. This interface is extended to provide a way for simulating Argos programs within the widely-used Simulink software.

Timed automata are well-suited for semantic descriptions, like the aforementioned transformation, and for modelling abstract algorithms and protocols. This thesis also includes a different type of case study. The timing diagram of a small-scale embedded component is modelled in more detail than usual with the aim of studying timing properties in this type of system. Multiple models are constructed, including one of an assembly language controller. Their interrelations are verified in Uppaal using a construction for timed trace inclusion testing.

Existing constructions for testing timed trace inclusion do not directly address recent features of the Uppaal modelling language. Novel solutions for the problems presented by selection bindings, quantifiers, and channel arrays in Uppaal are presented in this thesis. The first known implementation of a tool for automatically generating a timed trace inclusion construction is described.

The timed automata case study demonstrates one way of implementing application timing behaviours while respecting implementation constraints. A more challenging, but less detailed, example is proposed to evaluate the adequacy of Esterel for such tasks. Since none of the standard techniques are completely adequate, a novel alternative for expressing delays in physical time is proposed. Programs in standard Esterel are recovered through syntactic transformations that account for platform constraints.

Acknowledgements

My supervisor A. Sowmya has been unwavering in her support, encouragement, and enthusiasm throughout my time at UNSW. I owe much to her persistence and faith. My co-supervisor Rob van Glabbeek has always been generous with his time and patient in explaining his insights. His lectures on process algebra and semantics were a high point of my study at UNSW.

I have enjoyed the company of many good people at both CSE and NICTA. Especially Leonid Ryzhyk and Peter Gammie, who consistently took an interest in my work, listened to my ideas, shared their ideas, gave feedback, and reviewed drafts.

Gerwin Klein's understanding while I finished this manuscript was invaluable.

NICTA support me financially, but more importantly, it drew many talented staff and visitors to Australia which enriched my time as a graduate student immeasurably.[1]

I am grateful to the reviewers, Reinhard von Hanxleden, Florence Maraninchi, and S. Ramesh, for their detailed and thoughtful comments.

Mirjam endured another long preoccupation, and helped to maintain my health, sanity, and spirits.

Background. The material on transition systems in §2.1 follows R.J. van Glabbeek's notes on the topic. The descriptions of equivalence relations and preorders in Appendix C and of process algebra in Appendix A are, besides the references given, informed by his COMP9152 course at UNSW in 2005.

An infrared sensor and Validating timed trace inclusion in Uppaal. Several iterations of the timed automaton model of §4.3 were presented at regular 'timed automata teas' where Ansgar Fehnker, Peter Gammie, and Rob van Glabbeek asked insightful questions and made helpful suggestions.

During a pleasant afternoon at Radboud University in 2006, Frits Vaandrager suggested the testing construction in Figure 4.21, and the timed trace inclusion testing technique, which is applied in Chapter 4 and extended in Chapter 5. Ansgar Fehnker and Rob van Glabbeek provided support and insight during the development of the extensions. Kim Larsen suggested the railway example.

Delays in Esterel. S. Ramesh of GM Research Labs in Bangalore, India gave useful feedback on an early draft. The ideas were discussed with, and reviewed by Peter Gammie and Leonid Ryzhyk, both of who had valuable insights. The anonymous reviewers for EMSOFT 2007 offered accurate, encouraging, and insightful comments.

[1] NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

Related publications

[BS05] T. Bourke and A. Sowmya. Formal models in industry standard tools: An Argos block within Simulink. In F. E. Tay, ed., *International Journal of Software Engineering and Knowledge Engineering: Selected Papers from the 2005 International Conference on on Embedded and Hybrid Systems*, vol. 15, pp. 389–395. World Scientific, Singapore, Apr. 2005.

This paper describes an early version of the concepts and software described in Chapter 3. It forms the basis of the text and diagrams in §3.3.2 and the example in §3.4.1.

[BS06] —. A timing model for synchronous language implementations in Simulink. In S. L. Min and Y. Wang, eds., *Proceedings of 6th ACM International Conference on Embedded Software (EMSOFT'06)*, pp. 93–101. ACM Press, Seoul, South Korea, Oct. 2006. ISBN 1-59593-542-8.

The formal model of Chapter 3 (§§3.2 and 3.3.1) is taken directly from this paper with some minor corrections, as is the comparison with related work of §3.5.

[BS08] —. Automatically transforming and relating Uppaal models of embedded systems. In *Proceedings of 8th ACM International Conference on Embedded Software (EMSOFT'08)*, pp. 59–68. ACM Press, Atlanta, Georgia USA, Oct. 2008.

Chapter 5 improves upon this paper in several ways:

- The illustrative example is presented in more detail (§§5.1.1 and 5.5).
- An alternative 'partitioning' technique is described (§5.3.2.1).
- The example in Figure 5.12 has been improved.
- The implementation is discussed in more detail (§5.4).

[BS10] —. Delays in Esterel. In A. Benveniste, S. A. Edwards, E. Lee, K. Schneider, and R. von Hanxleden, eds., *SYNCHRON 2009*, no. 09481 in Dagstuhl Seminar Proceedings, pp. 55–84. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, Dagstuhl, Germany, 2010.

The ideas in Chapter 6 were presented at the 2009 Synchron workshop at Dagstuhl. An abridged version of the chapter appears in the proceedings.

Related software

1. An Argos compiler and Simulink block (refer Chapter 3).
2. Software for manipulating Uppaal models (refer Chapter 5).
3. A backhoe loader simulation (refer §2.4.3.1).

Contents

1 Introduction **1**
- 1.1 Aim . . . 2
- 1.2 Scope and general approach . . . 2
  - 1.2.1 Programs in physical time . . . 2
  - 1.2.2 Real embedded controllers . . . 3
  - 1.2.3 Verification of timed models . . . 4
  - 1.2.4 Implementations . . . 4
- 1.3 Contributions and significance . . . 4
- 1.4 Outline . . . 5

2 Background **7**
- 2.1 Transition Systems . . . 7
- 2.2 Timed Models . . . 10
  - 2.2.1 Why model timing detail? . . . 10
  - 2.2.2 Timed Transition Systems . . . 11
  - 2.2.3 Timed Automata . . . 17
- 2.3 Uppaal . . . 25
  - 2.3.1 Variables . . . 25
  - 2.3.2 Communication and priority . . . 26
  - 2.3.3 Semantics . . . 27
  - 2.3.4 Model Checking . . . 28
- 2.4 Synchronous Languages . . . 29
  - 2.4.1 Core concepts . . . 29
  - 2.4.2 Argos . . . 35
  - 2.4.3 Esterel . . . 43
- 2.5 Concluding remarks . . . 54

3 Simulating synchronous execution **55**
- 3.1 Simulink and Stateflow . . . 56
  - 3.1.1 Simulink . . . 56
  - 3.1.2 Stateflow . . . 61
  - 3.1.3 Executing Stateflow within Simulink . . . 64
- 3.2 Modelling synchronous execution . . . 65
  - 3.2.1 Execution parameters . . . 66
  - 3.2.2 Timed Automaton Model . . . 68
- 3.3 Implementation of an Argos block . . . 72
  - 3.3.1 Embedding within Simulink . . . 72
  - 3.3.2 Practicalities . . . 74
- 3.4 Experience . . . 76
  - 3.4.1 Sensor failure detection . . . 76
  - 3.4.2 Bang-bang temperature controller . . . 81
- 3.5 Comparisons with related work . . . 86
  - 3.5.1 Synchronous execution machines . . . 86
  - 3.5.2 Simulink to Lustre . . . 87
  - 3.5.3 TAXYS framework . . . 87
  - 3.5.4 AASAP Semantics . . . 87
- 3.6 Reflections and conclusions . . . 88

4 An infrared sensor **91**
4.1 Introduction . . . 91
4.2 The Sharp GP2D02 range sensor . . . 92
4.2.1 Overview . . . 92
4.2.2 Timing Diagram . . . 93
4.3 Timing diagram model . . . 95
4.3.1 Rationale and guiding philosophy . . . 95
4.3.2 Choosing an alphabet . . . 96
4.3.3 An explanation of the model . . . 97
4.3.4 Liveness and progress . . . 102
4.3.5 Limitations . . . 103
4.4 Driver/sensor split model . . . 103
4.4.1 Action direction and synchronization . . . 104
4.4.2 The split models . . . 104
4.4.3 Verifying implementation . . . 108
4.4.4 Verifying transmission correctness . . . 114
4.5 Assembly language implementation . . . 116
4.5.1 MCS51 Program . . . 118
4.5.2 Program model . . . 119
4.5.3 Verifying the program model . . . 121
4.5.4 Discussion . . . 123
4.6 Discussion and future work . . . 124
4.6.1 Summary and contributions . . . 124
4.6.2 Possible extensions and improvements . . . 125

5 Validating timed trace inclusion in Uppaal **127**
5.1 Introduction . . . 127
5.1.1 A flexible railway controller . . . 128
5.2 Uppaal . . . 133
5.2.1 Variables, expressions, and valuations . . . 134
5.2.2 Channels and actions . . . 134
5.2.3 Processes . . . 135
5.2.4 Automata . . . 136
5.2.5 Validation automata . . . 136
5.3 Transforming Uppaal models . . . 137
5.3.1 Elementary channels . . . 138
5.3.2 Channel arrays . . . 143
5.3.3 Inverting invariants . . . 149
5.3.4 Urgent Channels and shared variables . . . 149
5.4 Implementation . . . 150
5.4.1 Experience and implementation language . . . 152
5.4.2 Ensuring determinism . . . 154
5.5 Validation automaton for the railway controller . . . 154
5.6 Discussion . . . 156

6 Delays in Esterel **159**
6.1 Introduction . . . 159
6.2 Motivating example: a microprinter controller . . . 160
6.3 Expressing delays in Esterel . . . 164
6.3.1 Pause statements . . . 164
6.3.2 Timing inputs . . . 165
6.3.3 External timers . . . 167
6.3.4 External intervals . . . 169
6.3.5 Non-deterministic pause . . . 170
6.3.6 Quantitative watchdogs . . . 171
6.3.7 Other techniques . . . 171
6.4 An alternative . . . 172
6.4.1 Desired characteristics . . . 173
6.4.2 Esterel+delay . . . 173

6.4.3 Comparison to related work . . . 181
6.5 Loose ends and further possibilities . . . 183
6.5.1 Relating specifications and implementations . . . 183
6.5.2 Strictly event-driven programs . . . 184
6.5.3 Suspension . . . 186
6.5.4 Indeterminate delays . . . 187
6.6 Summary . . . 188

7 Conclusion **189**
7.1 Summary, contributions, and significance . . . 189
7.1.1 Programs in physical time . . . 189
7.1.2 Real embedded controllers . . . 191
7.1.3 Verification of timed models . . . 192
7.1.4 Implementations . . . 194
7.2 Limitations and future work . . . 194
7.3 Final remarks . . . 196

A Process Algebra **199**
A.1 Mathematical syntax . . . 199
A.2 Unobservable actions . . . 200
A.3 Interleaved concurrency . . . 202
A.4 Synchronized/handshake communication . . . 203
A.5 Determinism . . . 205
A.6 Semantic models . . . 206

B Input/Output Automata and related approaches **209**
B.1 Input-enabling and composition . . . 209
B.2 Modelling and specification . . . 211
B.3 Fair executions and analysis . . . 213
B.4 Related techniques . . . 215
B.5 Timed I/O Automata . . . 217

C Preorders **219**
C.1 Motivation . . . 219
C.1.1 Semantic theory . . . 219
C.1.2 Design and development . . . 219
C.1.3 Verification . . . 220
C.2 Technical Background . . . 220
C.3 Traces . . . 223
C.4 Simulations . . . 225
C.5 Remarks on Process Algebra and IOA . . . 232
C.6 Omissions . . . 233

D Textual Argos **235**
D.1 Explanation . . . 235
D.2 Lexical details . . . 235
D.3 Grammar . . . 235

E Argos Case Studies **237**
E.1 Sensor failure detection . . . 237
E.2 Bang-bang temperature controller . . . 239

F Formal Timing Diagram Model **241**

List of Figures

2.1 Categorisations in different timed sequence models 13
2.2 Dyadic clock regions for equalities 19
2.3 Dyadic clock regions for constant comparison 19
2.4 Dyadic clock regions for clock differences 19
2.5 Mixed dyadic clock regions 19
2.6 Example timed automaton to demonstrate clock regions 20
2.7 An example of the time successor relation 21
2.8 Urgent location macro 22
2.9 Three types of urgency 23
2.10 Interpretation of self-loops in Argos: no loops 37
2.11 Interpretation of self-loops in Argos: with loops 37
2.12 Encapsulation in Argos giving a non-deterministic result 40
2.13 Expressing strong abortion with inhibition and refinement 41
2.14 Inter-level outgoing transitions 41
2.15 Expansion of the temporized state macro 42
2.16 Backhoe loader example: input and output signals 45
2.17 Main backhoe loader controller module 45
2.18 Variations of the abort statement 48
2.19 Conceptual sketch of parallel synchronization in Esterel 49

3.1 Example Simulink model 57
3.2 Simulink Block 57
3.3 Basic steps of a Simulink simulation 60
3.4 Algebraic loops in Simulink 61
3.5 Standalone Stateflow Flow Diagram 62
3.6 Conceptual sketch of internal and external synchrony 66
3.7 External timing parameters 67
3.8 Translation of ABRO to a sample-driven system 69
3.9 Urgent inputs 71
3.10 Simulation components 73
3.11 Tool chain for Argos block 75
3.12 Mapping Simulink signals to logical signals 75
3.13 Fault-tolerant Fuel Controller 77
3.14 Fuel control subsystem in Argos 79
3.15 Bang-bang temperature controller 82
3.16 Bang-bang temperature controller in Argos 84
3.17 Argos simulation results 85

4.1 Physical appearance of sensor 93
4.2 Using the sensor 93
4.3 Sensor timing diagram 94
4.4 Sensor test signal 95
4.5 Non-instantaneous signal changes 96
4.7 Models of two events: composite and interleaved 97
4.6 Timing diagram model 98
4.8 Model of two events: causal 99
4.9 Model of two events: instantaneous causal 99

4.10 Model of two events: bounded causal 99
4.11 Start sequence from sensor timing diagram 99
4.12 Countdown of bits in sensor timing diagram 99
4.13 Sampling detail 100
4.14 Modelling data transmission: change event 101
4.15 Modelling data transmission: non-deterministic choice 101
4.16 Split models 105
4.17 Alternative driver model with τ-transitions 107
4.18 Deviation from the usual verification of timed trace inclusion 110
4.19 Trace inclusion testing in the paired synchronization model 111
4.20 Trace inclusion testing in the broadcast model 112
4.21 Validating transmission 115
4.22 Assembly language driver implementation 117
4.23 $\mathsf{DRIVER}_{\mathsf{test}}$: timed trace inclusion tester for $\mathsf{DRIVER}_{\mathsf{evt}}$ 122

5.1 Basic railway crossing 128
5.2 Simple railway control system 129
5.3 Original and modified controllers 130
5.4 Flexible railway controller 131
5.5 Status array for the flexible railway controller 132
5.6 No selection bindings or quantifiers 140
5.7 Construction for $\forall s_{11},\ldots,s_{1n_1},\ldots,s_{m1},\ldots,s_{mn_m}.\,\mathsf{neg}(g_1 \vee \cdots \vee g_m)$. 141
5.8 Selection bindings but no quantifiers 141
5.9 Selection bindings/negated guard clash 141
5.10 Construction for $\forall s_{11},\ldots,s_{mn_m}.\,\exists a_{11},\ldots,a_{mn'_m}.\,\mathsf{neg}(g_1 \vee \ldots \vee g_m)$ . . . 143
5.11 Channel selections without bindings (partitioned) 145
5.12 Channel selections without bindings (generalised) 148
5.13 Channel selections with selection bindings 150
5.14 Channel selections with differing ranges 151
5.15 Validating an array of urgent channels 152
5.16 High-level structure of Urpal 152
5.17 Validation automaton for Railway controller 155

6.1 Physical structure of the microprinter example 161
6.2 Typical microprinter motor control signals 162
6.3 Stepper motor controller in Esterel 163
6.4 Granularity of timing inputs 166
6.5 POLIS seatbelt alarm controller 168
6.6 Effect of suspend on delays 169
6.7 Delaying with feedback from a clock variable 172
6.8 Concrete syntax of platform statements 175
6.9 Phase relationships in an Esterel+delay program 180
6.10 Esterel+delay: artifacts and relations 183

A.1 Three process expressions and a corresponding process graph 200
A.2 Example for discussing the effect of τ transitions 201
A.3 CCS expressions and corresponding LTSs 202
A.4 CCS expressions with process graphs for synchronization and restriction 204
A.5 Sketch of relations between process algebras, models, and actual processes 206
A.6 Example failure sets 207

B.1 'Pasting' together Input/Output Automata (IOA) traces 211
B.2 Example: Pathfinder model 212

C.1 Relative granularity of equivalence classes 222
C.2 Simplification of the linear time–branching time spectrum 223
C.3 Simulations between two processes 226
C.4 Failure to find a ready simulation 228

C.5 Ready simulations between two processes 228
C.6 Failure to find a bisimulation between two processes 229
C.7 Trace equivalence via forward and backward simulation. 231

F.1 Different ways of formalizing a timing diagram 241
F.2 RTSTD version of timing diagram 243

Acronyms

AASAP	Almost As Soon As Possible
ACP	Algebra of Communicating Processes
API	Application Programming Interface
BDD	Binary Decision Diagram
BMM	Boolean Mealy Machine
CCS	Calculus of Communicating Systems
COTS	Commercial Off The Shelf
CRP	Communicating Reactive Processes
CSP	Communicating Sequential Processes
DBM	Difference Bound Matrices
DFA	Deterministic Finite Automata
DNF	Disjunctive Normal Form
EA	Enable All (processor bit flag)
EGO	Exhaust Gas Oxygen
FIFO	First-in-First-Out
FIN	Finite Internal Nondeterminism
FPGA	Field Programmable Gate Array
FTS	Fair Transition System
IE	Interrupt Enable (processor bit flag)
IO	Input/Output
IOA	Input/Output Automata
LED	Light Emitting Diode
LOTOS	Language Of Temporal Ordering Specification
LTS	Labelled Transition System
LSB	least significant bit
MAP	Manifold Absolute Pressure
MSB	most significant bit
NASA	(U.S.) National Aeronautical and Space Administration
NFA	Non-deterministic Finite Automata

ODE Ordinary Differential Equation

OS Operating System

POSIX Portable Operating System Interface

PSPACE Polynomial Space

RTOS Real Time Operating System

RTCCS Real Time Calculus of Communicating Systems

RTSTD Real Time Symbolic Timing Diagrams

SCCS Synchronous Calculus of Communicating Systems

SML Standard Meta Language

SOS Structural Operational Semantics

SSM Safe State Machines

STD Symbolic Timing Diagram

TD Timing Diagram

TIOA Timed Input/Output Automata

TLA$^+$ Temporal Logic of Actions

TLRCS Temporal Logic of Reactive and Concurrent Systems

TPTL Timed Propositional Temporal Logic

TTA Time Triggered Architecture

TTS Timed Transition System

VHDL VHSIC Hardware Description Language

VHDL/S VHSIC Hardware Description Language/Statecharts

WCET Worst Case Execution Time

XML Extensible Markup Language

Chapter 1

Introduction

Embedded systems are ubiquitous and important. There are computers ticking away inside cars, industrial plants, medical equipment, power stations, aeroplanes, rockets, and satellites. All of this equipment is programmed in one way or another, whether the programs are executed on a microprocessor or encoded in electronic circuits. Most such programs are *reactive*: they continually respond to events received from sensors by sending signals that are translated into physical actions. Design faults in a program, that may only be revealed after seemingly unlikely sequences of events, may have catastrophic consequences including damage to property and loss of life. Techniques that detect design faults or reduce their likelihood are thus of real importance.

Designing and programming embedded systems is difficult. One of the greatest challenges is the inescapable influence of physical time. In many other computer systems, time can be ignored or reduced to abstract properties like termination, progress, fairness, or liveness. Its quantitative aspects may be of little importance; total or partial event orderings may be adequate models. But time cannot be ignored in embedded systems that interact closely with the physical world, where two facts are undeniable:

1. *Timing behaviour is integral to specification and interaction.* The meaning or effect of an event often depends on its time of occurrence.

2. *Behaviours in time are physically constrained.* They are realised by devices that are subject to physical laws.

Systems with strict timing requirements and restrictions involve masses of interrelated and intricate details that encumber specification, design, analysis, implementation, and trouble-shooting. Engineers come to understand and tackle these complications through abstraction and modelling.

Rigorously defined models can facilitate and improve designs. Models may exist in the mind alone, or as sketches on paper, but there are many benefits to expressing them in a rigorous notation; that is, a notation with a formal meaning and mathematical properties, one that can be analyzed, simulated, compiled, or otherwise manipulated precisely and algorithmically. A good notation provides *precision*—it has is a single, unambiguous meaning; *clarity*—it focuses on the essentials and strives for simplicity; and *rigour*—it is amenable to verifiable arguments and analysis by cases.

Choosing a suitable level of abstraction, however, is not easy. Models that are too abstract simply shift problems and risks from designs to implementations. Conclusions drawn from analyzing or simulating such models may not hold of the systems they purport to represent. Implementations derived from such models may be impracticable. Conversely, models that are too concrete may not clarify anything. Such models may not make the original problem any more understandable or any easier to solve. It may not be possible to analyse or to simulate them effectively. While it may be easier to derive efficient implementations from more concrete models, the models themselves are usually specific to a certain platform; abstract requirements and implementation specifics are likely to be interwoven and it may be difficult or even infeasible to adapt a design to changes in the underlying platform or to account for changes in requirements.

Systems with complex timing requirements exacerbate these problems. Not only must models represent numerical details at suitable levels of abstraction, but they must also manage the tension between application timing requirements and implementation timing constraints. Simulation and analysis become more challenging, and it becomes more difficult to derive faithful and efficient implementations.

1.1 Aim

The aim of the work described in this thesis is to study and improve the application of synchronous language and timed automata models, in combination and alone, to the design and implementation of embedded controllers with intricate timing requirements on platforms with strict timing constraints.

1.2 Scope and general approach

Of the many approaches to designing and implementing embedded controllers with complex timing constraints, two are chosen as the focus of this thesis: timed automata and synchronous languages.

Timed automata are a formalism for modelling sequential systems in continuous time. They are often, when certain restrictions are respected, analyzable automatically by tools like the Uppaal model checker. Multiple timed automata models may be necessary to describe a single system at various levels of abstraction, or from various perspectives. The relations between them can sometimes be verified automatically.

Imperative synchronous languages, like Argos and Esterel, are for programming sequential systems. They are supported by rigorous mathematical models and efficient compilation techniques. In contrast to timed automata, time in the synchronous languages is discrete and qualitative. Synchronous languages are distinguished by an assumption on the behaviour of implementation platforms: that they are much faster than their environments.

There are perhaps two extreme approaches in embedded systems design. At one extreme is systems engineering which is concerned with specific techniques for efficient and robust implementation, for example the details of scheduling algorithms or cache utilisation. At the other extreme is theoretical computer science, where applications are often considered at the high level of abstraction necessary to tease out general principles and fundamental rules. Both viewpoints are important but they are hard to reconcile. The models and techniques developed in this thesis are a compromise. They attempt to incorporate some of the low level details of importance to engineers but of less interest to theoreticians without becoming lost in the minutiae of specific systems and their optimisation.

Four broad themes cut across and unite the specific applications of timed automata and synchronous languages to embedded systems design in this thesis. They are presented in turn over the following subsections:

§1.2.1 Programs in physical time
§1.2.2 Real embedded controllers
§1.2.3 Verification of timed models
§1.2.4 Implementations

1.2.1 Programs in physical time

Embedded controller programs act in physical time, subject to the two types of constraints mentioned in the opening: application requirements and implementation characteristics. This theme is pursued through three developments: models and tools for simulating synchronous language programs in Simulink, an assembly language driver for a small sensor component, and an extension to Esterel for expressing reactive behaviours in physical time. Implementation timing details are resolved, in each of the developments, by transformations between models.

Backhoe loader	(Original example)	§2.4.3
Sensor failure detection	(Mathworks demonstration)	§3.4.1
Bang-bang temperature controller	(Mathworks demonstration)	§3.4.2
Infrared sensor	(Real application)	Chapter 4
Extended railway controller	(Extension of standard example)	§§5.1.1 and 5.5
Microprinter controller	(Real application)	§6.2

Table 1.1: Motivating and demonstrative examples

Simulink is a widely-used tool for modelling and simulating embedded systems, an add-on, called Stateflow, allows the inclusion of discrete control logic. Stateflow is rich in features but also more complicated and less rigorously defined than the synchronous language Argos, which is a similar but more austere alternative. A technique for simulating Argos programs in Simulink is proposed. The embedding of the discrete timing model of synchronous programs into the continuous one of Simulink is defined by a transformation of Mealy machines into timed automata. The transformation provides a precise specification of a Simulink block for Argos and addresses implementation timing issues that are usually ignored in the synchronous approach. It effectively combines application requirements, expressed as a synchronous program, with implementation restrictions, characterised by two idealised parameters.

Timed automata are employed more directly in a case study of a small sensor component. A timing diagram describes the behaviour of the sensor in physical time. This timing diagram is modelled as a timed automaton, which, after various other developments, becomes the specification for an assembly language driver. In contrast to the abstract approach of the synchronous languages, where program execution time is assumed away, the assembler driver meets its timing requirements by exploiting platform execution characteristics. A timed automaton model of the driver is generated by transforming the assembler program into timed automata. The transformation incorporates the timing details of individual instructions. This model is verified against the specification in the Uppaal model checker.

A more challenging case study is proposed to evaluate the suitability of Esterel for expressing and implementing controller behaviours in terms of physical time. Solutions using various standard techniques are attempted, but each requires information about an eventual implementation platform. An alternative that allows such decisions to be delayed and made more portably is proposed. It is is based on syntactic transformations of delays expressed in physical time. The transformations again account for implementation parameters.

1.2.2 Real embedded controllers

Various examples of embedded controllers are presented throughout this thesis. They are summarised in Table 1.1. Admittedly, the backhoe controller, sensor failure detection, temperature controller, and railway controller examples are more abstract than the systems usually faced by engineers. The sensor and microprinter are, however, completely realistic. They are studied as they are, not as they could or should be.

The sensor makes and communicates range readings. Its interface is described by a deceptively simple-looking timing diagram. Working solely from this diagram, as would most embedded systems engineers, the specification is modelled as a single timed automaton. Utmost care is taken to render it faithfully and in detail. Two variations of a second model that clarifies the distinct roles of driver and sensor are developed. The relation of each variation to the original model is validated in Uppaal using a construction for testing timed trace inclusion. One of the variations is taken as a specification for the assembly language driver described in the previous subsection.

The study of delays in Esterel is motivated from the challenges of controlling a microprinter. These components are commonly found in cash registers and printing calculators. The timing constraints of the microprinter are more challenging than those of the sensor, though they are not presented in as much detail. Esterel is almost ideal

for programming the required controller, except that it turns out to be surprisingly difficult to express the real-time requirements in a portable way.

1.2.3 Verification of timed models

Verification is important for embedded controllers because faults are difficult to avoid and, if they do occur, may cause damage or injury. Automatic verification techniques like model checking are especially relevant because embedded control software is typically finite—dynamic data structures and recursive functions are usually avoided—and some faults may only occur after long and intricate sequences of events. Verification is even more necessary when continuous timing features are present, because numerical details aggravate the challenge of expressing and understanding the behaviour of systems. Uppaal is a model checker for timed automata. It is applied repeatedly in the sensor case study, and becomes the focus of extensions to a technique for testing timed trace inclusion in a subsequent chapter.

The models in the sensor case study are created in Uppaal. Properties are verified of the models themselves, namely transmission correctness of the sensor and driver, and absence of deadlock and liveness in the assembly driver model. But the focus is otherwise on the relationships between the various models. A construction for testing timed trace inclusion is used to validate that some models faithfully implement others.

It turns out that the standard construction for testing timed trace inclusion cannot address certain recent features of the Uppaal language directly. The construction is thus extended to handle selection bindings, quantifiers, and channel arrays. Several new techniques are necessary, and it is shown that the construction is not always possible.

1.2.4 Implementations

The work described in this thesis is supported by two software implementations. A tool chain for compiling Argos programs and simulating them in Simulink, and an implementation of the construction for testing timed trace inclusion. Besides providing useful tools, the development of these programs clarified concepts and revealed subtleties which may have otherwise been overlooked.

The Argos tool chain comprises three main programs: a compiler of (textual) Argos programs to dataflow equations, a compiler of dataflow equations to C, and a Simulink s-function. An s-function is essentially a custom Simulink block. It must export a series of functions that are called by the simulation algorithm. Special conventions are used to exchange information and implement memory. Debugging s-functions is challenging; when they crash so does Simulink. The implementation is demonstrated and evaluated by reimplementing two standard Stateflow examples.

The tool for constructing testing automata for validating timed trace inclusion automates a task that is tedious and error-prone to perform manually. An automatic tool is especially advantageous because timed trace inclusion testing often involves several iterations: testing finds flaws, models are adjusted, new testing automata are created, and the process is repeated. The tool is written in Standard Meta Language (SML), and includes a parser and algebraic data types for Uppaal models. The legibility of output models is improved by formatting them with Graphviz. Besides the construction for testing timed trace inclusion, the tool provides other basic manipulations on timed automata, and rudimentary support for modelling MCS51 assembly language programs.

1.3 Contributions and significance

Several specific contributions are made by this thesis:

1. An improved approach for expressing delays in Esterel based on syntactic transformations that account for abstract platform characteristics is motivated and described. The limitations of this approach are discussed and directions for future research are proposed.

	Chapter 3 (Simulink)	Chapter 4 (Sensor)	Chapter 5 (Timed traces)	Chapter 6 (Esterel+delay)
Timed models §2.2	∗	∗	∗	∗
Uppaal §2.3		∗	∗	
Synchronous languages §2.4	∗			∗

Table 1.2: Explicit relationships between background sections and technical chapters

2. A standard approach for testing timed trace inclusion is extended to account for recent features of Uppaal, namely selection bindings, quantifiers, and channel arrays. The various constructions and techniques developed to treat these features may be applicable in other similar transformations.

3. The existing and extended timed trace inclusion testing techniques have been implemented in a functional programming language. No other similar software is known to exist. The software and source code are available for download. Routines and algebraic data types were developed to parse and represent Uppaal models. They are suitable for reuse in other applications that manipulate Uppaal models.

4. A detailed case study of a small-scale but realistic embedded component, whose timing diagram is modelled in full detail with timed automata. The case study differs from others in its focus on an embedded hardware component, rather than on algorithms or protocols, and in its level of detail. Two different techniques for modelling protocols in Uppaal are compared; one of which highlights a limitation in current restrictions on broadcast channels.

5. A transformation of synchronous language models into timed automata that accounts for the execution mode and two idealised implementation parameters. The transformation formalises and clarifies details that are usually ignored or taken for granted.

6. The design and implementation of a Simulink block for simulating Argos programs according to the details of the aforementioned transformation. Simulink is a de facto standard in industry for modelling and simulation.

7. A new microprinter controller case study is proposed to illustrate the challenges of expressing the timing behaviours of a certain class of embedded application.

8. The various approaches for expressing delays in Esterel, including a new proposal, are surveyed.

9. A novel backhoe loader example for introducing the basics of Esterel. A graphical simulator for the example is available for download.

10. A detailed summary of the Argos synchronous language including careful treatments of implicit self-loop transitions and the macro for temporized states.

More generally, this thesis contributes to the larger problem of applying formal modelling and programming techniques to the detailed timing requirements and constraints that engineers encounter in practice; especially applications of the Uppaal modelling language and the synchronous languages Argos and Esterel.

1.4 Outline

The body of this thesis comprises a background chapter and four distinct technical chapters.

Chapter 2: Background The background chapter begins with a unifying overview of various transition system models. It otherwise contains three main sections. The first

two address modelling timed systems. The fundamentals are presented in §2.2; particular attention is given to the definition and fundamentals of timed (safety) automata. The specifics of modelling with timed automata in Uppaal are described in §2.3. The last section addresses synchronous languages. It contains an overview of the core concepts in §2.4.1, a comprehensive description of Argos in §2.4.2, and a tutorial introduction to Esterel in §2.4.3. The dependencies between the four main sections and the four technical chapters are summarised in Table 1.2.

Chapter 3: Simulating synchronous execution In this first technical chapter timed automata and synchronous languages are applied together to the simulation of embedded controllers in Simulink. The chapter begins with an overview of Simulink and the Stateflow extension in §3.1. An idealised model that accounts for imperfections in implementations of synchronous programs is presented in §3.2. It forms a specification for a Simulink block for simulating Argos programs, which is described in §3.3 and evaluated in §3.4 against two examples from the Stateflow software package. The chapter ends with comparisons to related work in §3.5 and an evaluation in §3.6.

Chapter 4: An infrared sensor The second technical chapter focuses on modelling an infrared sensor component with timed automata. It begins with an informal description of the sensor and its timing diagram in §4.2. Three types of models are described: a single timed automaton in §4.3, networks of timed automata in §4.4, and a model of an assembly language controller in §4.5. Refinement relations between the models are verified in Uppaal using a construction for testing timed trace inclusion. A summary of the work and possible improvements are presented in §4.6.

Chapter 5: Validating timed trace inclusion in Uppaal The construction for testing timed trace inclusion is extended, in the third technical chapter, to include more recent features of the Uppaal modelling language. The chapter begins with an extension of the well-known railway controller example that demonstrates the newer features. Then follows in §5.2, a formalisation of Uppaal that is sufficient to develop the transformation described in §5.3. The transformation is introduced in stages, beginning with basic elementary channels, then introducing selection bindings, then quantifiers, and finally addressing arrays of channels. The implementation of the transformation in software is discussed in §5.4. The result of applying the software to the railway controller example is presented in §5.5.

Chapter 6: Delays in Esterel The last technical chapter returns to the central theme of specifying embedded controllers with complex timing requirements. A motivating example is presented in §6.2, and used throughout §6.3 to demonstrate the different ways of expressing timing delays in Esterel. In §6.4, an alternative that addresses some of the inadequacies of existing techniques is proposed. The limitations of this approach and directions for future work are summarised in §6.5.

Chapter 7: Conclusion Finally, the thesis contents are recounted and evaluated in terms of the four main themes, and several specific ideas for future research are proposed.

Chapter 2

Background

The forthcoming technical chapters build on several well-developed topics. The details and context necessary to understand and appreciate them are provided in this chapter, whose sections can be divided into two categories: §§2.1–2.3 focus on modelling using transition systems with an emphasis on continuous time, and §2.4 is about programming using synchronous languages. Both categories are relevant to the design and analysis of embedded controllers.

Transition systems and their traces are fundamental models of discrete behaviour. Some basic definitions, terminology, and important variations are outlined in §2.1.

A special type of transition system, called a Timed Transition System (TTS), is used to model discrete behaviour in continuous time. The passage of time is represented by delay transitions. Timed automata are another formalism with the same purpose, but abstract clock variables, rather than explicit delay transitions, are used to express timing constraints. TTSs and timed automata are presented in §2.2.

Uppaal is a system for modelling and verifying real-time systems using timed automata extended with features for concurrency, communication, data variables, and priority. Uppaal is described in §2.3.

The focus changes in §2.4 from modelling to programming when the synchronous languages Argos and Esterel are introduced. The distinction is only minor, though, since both languages are characterised by their direct connection to transition system models. The synchronous languages were expressly developed for programming embedded controllers.

2.1 Transition Systems

Transition systems model the essential characteristics of discrete systems.

Definition 2.1.1
Given sets A, of *actions*, and P of *predicates*, a Labelled Transition System (LTS) over A and P is a tuple $(S, S_0, \longrightarrow, \models)$, where

- S is a set (of *states*),
- $S_0 \subseteq S$ is a non-empty set (of *initial states*),
- $\longrightarrow$ is a set of relations $\xrightarrow{a} \subseteq S \times S$, for each $a \in A$, (the *transitions*) and,
- $\models \subseteq S \times P$, assigns to each state a set of predicates. ■

When presenting an LTS, the sets A and P, and the predicate component $\models$ will only be written explicitly when required. The notation $s \xrightarrow{a}$ abbreviates $\exists s' . s \xrightarrow{a} s'$, and $s \not\xrightarrow{a}$ abbreviates $\nexists s' . s \xrightarrow{a} s'$. When an a-transition relates state s to state s', $s \xrightarrow{a} s'$, the latter is termed an *a-derivative*, or just *derivative*, of the former.

Definition 2.1.2
A *Kripke structure* over P is an LTS over singleton A, and P where the transition relation, written $\longrightarrow$, is total: $\forall s \in S . \exists s' \in S . s \longrightarrow s'$. ■

Kripke structures are an appropriate model when the properties of states and potential future states are of primary interest. In graphical depictions, the states are normally drawn as open circles with applicable predicates written inside or alongside each. The transition relation expresses the structure of future possibilities from any state. Properties of Kripke structures are usually expressed in temporal logic, and, when S is finite, can, in principle, be verified by model checking [CJGP00, §2.1].

Definition 2.1.3
A *process graph* over A is an LTS over A and $P = \{\surd\}$ where $S_0 = \{s_0\}$. ■

Process graphs are an appropriate model when the branching structure of action possibilities is of primary importance. The sole predicate of Definition 2.1.3 indicates the possibility of successful termination. Other predicates are sometimes also needed. In depictions of process graphs, the states are often drawn as dots, sometimes with an extra ring when the termination predicate holds. Transitions represent the instants when an action occurs. Process graphs are the usual semantic model for process algebras.

A distinction is sometimes made between LTSs and process graphs, whereby an LTS expresses the domain of potential processes, of which process graphs are elements [Mil89]. The distinction is not important in this thesis.

The set of LTSs may be partitioned in two based on whether the transition relation can be considered a function or not.

Definition 2.1.4
Given an LTS $(S, S_0, \longrightarrow, \vDash)$ over A, a state $s \in S$ is *deterministic* iff, for all $a \in A$, $s \xrightarrow{a} s'$ and $s \xrightarrow{a} s''$ implies that $s' = s''$. An LTS is deterministic iff every $s \in S$ is deterministic and S_0 contains one state. ■

For a non-deterministic state and an action there may be a choice of successor states. Non-determinism is an abstraction; it is either unknown, uncertain, unimportant, or intentionally unspecified how the choice is actually resolved in the system being modelled. Determinism is often desired of implementations.

Definition 2.1.5
An *automaton* over a set A is an LTS over A and $P = \{\odot\}$ where $S_0 = \{s_0\}$. Both S and A must be finite. ■

Note that an automaton is technically the same as a finite process graph, however it is interpreted as a compact representation of a *language*, which is a set of *words*. A word is a sequence of symbols (actions). The predicate $\odot$ marks *acceptance states*. Different interpretations of acceptance states give different types of automata.

Non-deterministic Finite Automata (NFA) define languages containing words of finite length. To determine if a word belongs to the language defined by an NFA, the current state of the automaton is tracked, beginning at s_0. Symbols of the word are considered one-by-one in sequence against the current state. A word is accepted if, for each current state and symbol, there exists a transition labelled with the symbol, the destination becomes the new current state, and, after considering all symbols, the current state is accepting. The theory of NFAs is well developed [HMU01].

An ω-automaton defines a language containing words of infinite length. The various acceptance conditions are described in §2.2.3.4 in the context of timed automata.

The language theoretic perspective, particular complexity theory, is important in the compilation and analysis of reactive systems. NFAs delimit the possibilities of finite state systems and underlie regular expressions. Augmenting NFAs with pushdown stacks defines the class of context-free languages which find application in the parsing of computer programs. Augmenting NFAs with infinite storage, to give Turing machines, defines the limit of what is computable, that is what can and cannot be done by a computer program.

The *Moore machines* [Moo56] and *Mealy machines* [Mea55] that underlie the theory of sequential circuit design are essentially automata where $P = \varnothing$ and the set of actions is partitioned into inputs I and outputs O. In a Moore machine, outputs are a function of state alone, that is for each $o \in O$ there is a function $f_o(s)$. Moore outputs are, in effect, sustained. In a Mealy machine, outputs are a function of both state and inputs,

that is for each $o \in O$ there is a function $f_o(s,i)$. Mealy outputs are, in effect, instantaneous. Mealy and Moore machines are the foundation for implementations of many types of reactive systems, including systems described with synchronous languages.

Moore and Mealy machines, and LTSs in general, are static structures that model dynamic processes. Executions and traces are one way of recovering the intended dynamic behaviours. Other, less linear, possibilities are described in Appendix C.

The potential of a system is transmuted in concert with its environment into a linear record of state changes and events, which is formalised as an *execution*.[1]

Definition 2.1.6
An *execution* α of an LTS over A $(S, S_0, \longrightarrow)$ is an alternating sequence, either finite $s_0, a_0, s_1, a_1, \ldots, a_{n-1}, s_n$, or infinite $s_0, a_0, s_1, a_1, \ldots$, of elements of S and A, satisfying:

- $s_0 \in S_0$ *(initiality)*, and,
- $s_i \xrightarrow{a_i} s_{i+1}$ *(consecution)* for all $i \geq 0$, and where for finite α, $i < n$. ■

Definition 2.1.7
An *infinite execution* is an execution of the form $s_0, a_0, s_1, a_1, \ldots$. ■

The set of all possible *executions* of an LTS $\mathcal{T}$ will be written $execs(\mathcal{T})$. Note that an execution must begin, and if finite, end, with a state. Sets of executions are prefix-closed.

While it can be argued that real systems do not have infinite executions, they are a useful abstraction nonetheless, permitting descriptions of ideally unending executions, or at least those of indeterminate length.

The subsequence of states in an execution exposes the internal details of a transition system, which is not always feasible or desirable. In models with structured states it is possible to partially obscure the internal structure. In labelled transition systems, and typically for process graphs, all internal information is concealed by considering *traces* rather than executions.

Definition 2.1.8
The *strong (finite) trace* of a finite execution $\alpha = s_0, a_0, s_1, a_1, \ldots, a_n, s_n$ of an LTS over A is the restriction of that sequence to elements in A: $a_0, a_1, \ldots, a_n$ ■

Definition 2.1.9
Given an LTS over A and a distinguished subset of (internal or local) actions $L \subseteq A$, a *(weak finite) trace* is a strong trace of the LTS restricted to actions in $A \setminus L$. ■

The trace corresponding to a finite execution α will be denoted $trace(\alpha)$. The set of traces of an LTS $\mathcal{T}$ is written $traces(\mathcal{T}) = \{trace(\alpha) \mid \mathsf{finite}(\alpha) \wedge \alpha \in execs(\mathcal{T})\}$. Trace sets are prefix-closed. The empty trace is denoted ϵ.

Definition 2.1.10
If σ is a weak trace of an execution between two states s, s' of an LTS, then s' is termed a *descendant* of s; the relation is written $s \overset{\sigma}{\Longrightarrow} s'$. ■

Traces represent externally observable behaviour, representing in some way the what rather than how of a system. The system is treated as a 'black box'. Two transition systems may give rise to identical trace sets despite differences in internal structure. Related equivalence and refinement relations are discussed in Appendix C.

Proposition 1
For each trace of a deterministic LTS there is exactly one corresponding execution. ■

The practical importance of this property should not be overlooked. Deterministic systems are easier to test and debug than non-deterministic ones; repeating the same sequence of actions gives the same results [Ber00a, §2.3].

[1] The concept is fundamental but not the terminology. This presentation is a mix [LT87, §2.1][MP92].

2.2 Timed Models

The quantitative aspects of time are usually ignored in transition system models. In languages whose semantics are defined as transition systems, either no assumptions are made of the relative execution speeds of concurrent processes, or at best, only the minimum expressible through either progress, that is some non-zero execution speed, or fairness, that is some non-infinite relative execution speed. Observations of transition system behaviours, like executions and traces, cannot distinguish fast processes from slow ones. Quantitative timing features can, in fact, be modelled directly in LTSs, but there are several subtleties. Furthermore, specialised data structures and algorithms are required to encode and analyze them. The motivations for modelling quantitative timing details are discussed in §2.2.1.

Much has been written in the last two decades on modelling the timing details of systems. The focus of this section is on fundamental semantic models rather than on the various proposals for expressing timing behaviours, or on issues of computational complexity and the related subclasses and extensions of basic models. The review focuses on transition system models, namely TTSs, in §2.2.2, and timed automata, in §2.2.3, and their timed executions and traces.

2.2.1 Why model timing detail?

Ignoring quantitative timing details yields many advantages. Untimed models are simpler to represent, manipulate, and reason about. Systems designed, and possibly verified, without assumptions on timing are resilient to variations in implementation details like scheduling, component temperatures, and component interconnections [Dij68, §§1 and 2]. Implementors have greater flexibility in realising untimed designs. Where timing requirements are necessary, it may still be possible to treat them independently from other design aspects, including logical correctness [Hoa85, §1.1]. For instance, one way to do this is to introduce compilation directives, so called 'pragmas', which are ignored for all purposes but implementation [Hoa85, §7.2.6]. Ultimately, untimed system models are to be preferred when adequate.

While it is appropriate to ignore timing details in many types of systems and algorithms, or to treat them separately as performance requirements, there are situations where precise timing is either significant or integral to design and description. For instance:

- The meaning of signals and the effect of actions may depend on precisely when they occur, as in embedded controllers, and other systems that also interact with physical processes.
- The passage of time can yield additional information, for instance on signal absence and process failures, as in the design and realisation of distributed algorithms and systems [Lyn96, Chapters 23–25][Tel00, Chapters 12 and 15].
- Timing assumptions can be essential to correctness. For example, in Fischer's mutual-exclusion algorithm [AL94] certain erroneous execution paths are ruled out by a careful choice of two timing parameters. Equivalent untimed algorithms may not exist, or they may lack desired properties.
- Time multiplexing can increase resource efficiency, as in time-sharing operating systems, or minimise hardware, as in shared buses and other interconnection paths.
- Time-triggered systems [KB03, Kop97, Pon01] trade dynamic efficiency for simplicity and predictability. Time can also be used to reduce or eliminate control dependencies between processes [KB03, BCLG+02].

Modelling languages that can express timing details are necessary to accurately describe and analyze such time-dependent features.

Models with time can be used to calculate quantitative properties of systems, for instance in some approaches to Worst Case Execution Time (WCET) estimation [WEE+08, §2.3.4], and to estimate or optimise system parameters.

Both timed and untimed models may be used to describe aspects of a single design. Any refinement relation between timed and untimed models is necessarily asymmetric. Timing information is either added or removed.

Timing information is added when an untimed specification is refined into a timed one. Development may, for instance, begin with an assumption that a system responds quickly enough, while later refinements toward an implementation provide detailed justifications [SGSAL94]. Alternatively, a timed model may implement an untimed specification. For instance, the correctness of Fischer's algorithm rests on its timed behaviour, but its purpose is to maintain an untimed mutual exclusion property [SGSAL94].

Timing information is also sometimes removed from a model to form an untimed abstraction. For instance, even though an untimed model may be sufficient or desired for particular analyses or design steps, a timed model may be needed for others. Or, the simplest way of formalising a system or specification may be to transcribe it directly into a timed model. The abstraction of timing details can then be justified separately from the issue of model fidelity. Such abstractions may anyway depend upon the specific properties to be analyzed.

Other properties lie between the extremes of untimed and continuous-time models. There are models with distinguished timeout transitions, which typically differ in priority from other transitions. There are also discrete-time models, which are suitable for many purposes, including the provision of sound abstractions for certain properties and models in continuous time [HMP92]. Discrete-time models are, if only by virtue of their relative closeness to LTSs, easier to reason about and to analyze. They often give suitable expressions of computer systems and programs which are, in their very nature, discrete.

Continuous time is a useful abstraction even if physical time is not truly continuous, or if modelling a discrete system. There is no need to specify a smallest granularity; even if individual systems are bound to clocks of minimum period, their relative offset may not be. Assumptions of minimum granularity could introduce hidden synchronizations between components.

2.2.2 Timed Transition Systems

Timed behaviour can be modelled by an LTS. In the usual approach, transitions are partitioned into two categories: discrete transitions, that occur instantaneously, and delay transitions, that represent the passage of time. The structure of delay transitions between states is restricted so as to model expectations of physical time. The subclass of LTSs with structured, real-number delay transitions is called Timed Transition Systems (TTSs).

Definition 2.2.1
A TTS is an LTS $(S, S_0, \longrightarrow)$ over A where $\mathbb{R}^{>0} \subseteq A$ and the subset of the transition relation $\{\xrightarrow{d} \mid d \in \mathbb{R}^{>0}\}$ satisfies:

- $s \xrightarrow{d+d'} s' \implies \exists s''.\, (s \xrightarrow{d} s'' \text{ and } s'' \xrightarrow{d'} s')$ *(time interpolation)*, and,
- $\exists s''.\, (s \xrightarrow{d} s'' \text{ and } s'' \xrightarrow{d'} s') \implies s \xrightarrow{d+d'} s'$ *(time additivity)*.

Time interpolation and additivity are minimum expectations on the progression of time: the duration of a delay is significant, not the number of transitions that express it. They exclude many types of unrealistic behaviour; but not so called *Zeno traces*, which will be addressed separately. The two properties combined are termed 'time continuity' [Wan90, Lemma 2.3]. When TTSs serve as the model for a higher-level language, time interpolation and additivity are usually shown as properties of the language and its semantic mapping [Wan90, §3.2][BM02, §§2.1.2 and 4.1.2].

In some approaches [Wan90, NSY93, HR95, BM02] delay transitions are deterministic, that is,

$$\forall d \in \mathbb{R}^{>0}. \, p \xrightarrow{d} p' \text{ and } p \xrightarrow{d} p'' \implies p' = p''.$$

It has been argued, however, that time non-determinism generalises naturally to hybrid systems and the expression of features like clock drift [LV96, §A.1]. Time determinism may be a property of a given timed language, but it need not be imposed on the definition of TTSs.

The time domain of Definition 2.2.1 is fixed as the set of strictly positive real numbers $\mathbb{R}^{>0}$. In other approaches [LV96, §A.2][NSY93, §2.1][Wan90] it is parameterised over a (dense) time domain with a zero constant and an addition operator.

Traces were proposed, in §2.1, as a linear expression of the dynamics of an LTS. While the same definitions apply to TTSs, they are less adequate. The issues are described in §2.2.2.1. Two alternatives, *timed sequence pairs* and *continuous traces*, are discussed in §§2.2.2.2 and 2.2.2.3, respectively. The three approaches offer different perspectives on the behaviours expressible in timed models. Their comparison yields insights into some of the subtle aspects of modelling with continuous time.

2.2.2.1 Timed traces

Since TTSs are a subclass of LTSs, the standard definitions of traces, Definitions 2.1.8 and 2.1.9, apply directly. These traces are an interleaving of finite or infinite sequences of discrete actions and delay actions. The appeal of (ordinary) traces is their generality and familiarity. A familiarity that is, perhaps, not completely justified: comparing traces and hiding internal actions requires care, and certain intuitions are strained.

Distinct traces of discrete LTSs represent distinct behaviours. More care is required, however, when comparing traces of a TTS. For instance, for a single execution where an a-action is performed after waiting for one unit of time there are, due to time interpolation and additivity, an infinite number of traces, for example:

$$\begin{array}{l} 1, a, 1, 1, \ldots, 1, \ldots \\ 0.5, 0.3, 0.2, a, 1, 1, \ldots \\ 1, a, 2, 2, \ldots, 2, \ldots \\ \vdots \end{array}$$

Such irrelevant differences between traces can be overcome by defining equivalence classes represented by a canonical form that has at most one delay action initially and between any two discrete actions, and that ends in a single delay action if closed, or an infinite sequence of 1-actions if admissible [LV96, §2.3.1]. Zeno traces with a finite discrete subsequence must either be excluded or represented by extra notation. These compensations are, in any case, additional complications.

Care is also required when defining weak traces [LV96]. Consider a system that waits for one unit of time, performs a τ-action, and then waits indefinitely [LV96]. Initially it can only take actions d, for $0 < d \leq 1$; any larger delay action would exceed the time when the internal step must occur. Such discontinuities violate the supposed non-detectability of internal steps. Hiding them is again an additional complication.

Traces and other timed sequences can be partitioned into three categories [LV96, KLSV06] based on the sum of their delay transitions and whether they are finitely or infinitely long. The sum of delay transitions is called the *limit time*.

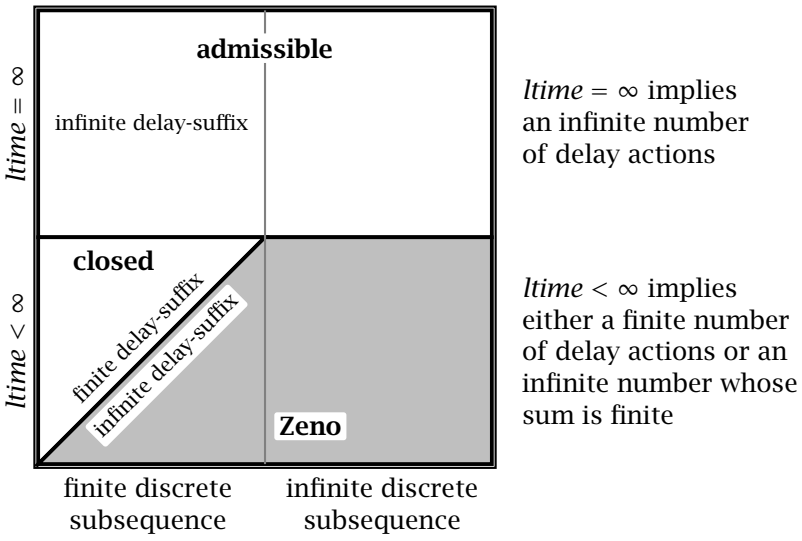
Definition 2.2.2
The *limit time* of a trace α, $ltime(\alpha)$, is defined:

$$ltime(\alpha) = \begin{cases} \infty & \text{if } \alpha \lceil \mathbb{R}^{>0} \text{ diverges} \\ \sum \alpha \lceil \mathbb{R}^{>0} & \text{otherwise,} \end{cases}$$

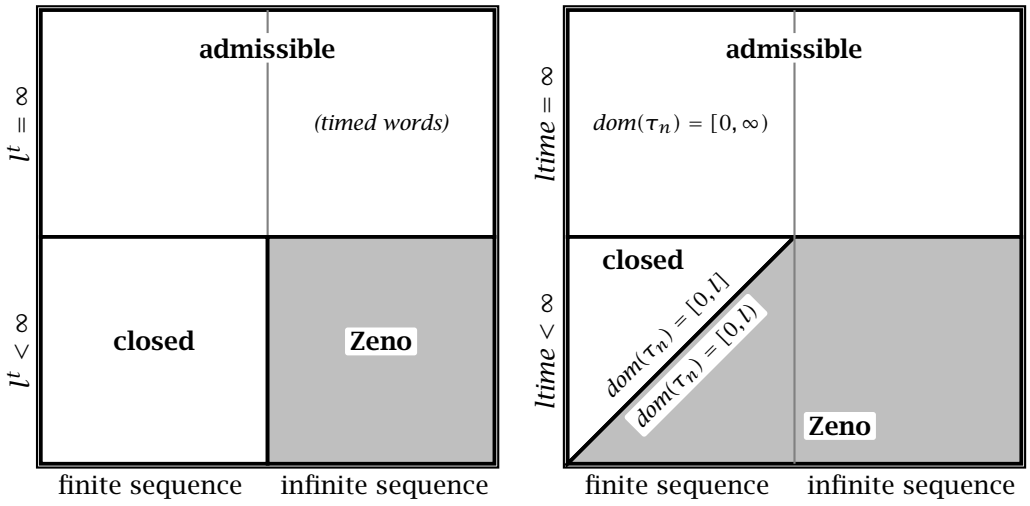
where $\alpha \lceil \mathbb{R}^{>0}$ is the subsequence of delay actions in the trace. ■

A trace α is *admissible* if $ltime(\alpha) = \infty$, *closed*[2] if the subsequence of delay actions is finite, and otherwise *Zeno*. The three categories are shown in Figure 2.1a, which is

[2] Sometimes termed 'finite' [LV96].



(a) Ordinary traces



(b) Timed sequence pairs (α^t, l^t)

(c) (A, V)-sequences

Figure 2.1: Categorisations in different timed sequence models

split into four quadrants, one for each combination of infinite or finite *ltime* with finite or infinite subsequences of discrete actions. The quadrant for finite *ltime* with finite discrete actions, at bottom left, is further divided based on whether there are finitely or infinitely many delay actions. The bottom right quadrant, finite *ltime* with infinitely many discrete actions, could be similarly subdivided but the categorisation would be unchanged. It is clear from the figure that the Zeno traces are those where either an infinite number of discrete actions occur in a finite amount of time, like:

$$\begin{array}{ll} & 1, a, a, \ldots, a, \ldots \\ \text{or :} & 1, a, \frac{1}{2}, a, \frac{1}{4}, a, \frac{1}{8}, \ldots \end{array}$$

Or those with an infinitely long suffix of delay actions whose sum converges, like:

$$a, a, a, \tfrac{1}{2}, \tfrac{1}{4}, \tfrac{1}{8}, \ldots$$

Note that the magnitudes of delay actions in an admissible trace can decrease without bound, as for example in $a, 1, a, \frac{1}{2}, a, \frac{1}{3}, a, \frac{1}{4}, a, \frac{1}{5}, a, \ldots$, provided their sum does not converge to a finite limit.

2.2.2.2 Timed sequence pairs

There are at least two objections to the model of delay transitions. First, it is questionable whether periods of delay are observable in the same way that discrete actions are. Second, delay transitions violate the idea that time passes in states and that transitions between states are instantaneous. It might be more natural to envisage an observer equipped with a stopwatch, of infinitely fine precision, which is started simultaneously with the system and stopped, if at all, at some point. Each discrete action could then be recorded along with the absolute time of its occurrence. This leads to the idea of a *timed sequence* [LV96, §2.3.1].

Definition 2.2.3
A *timed sequence* over A_D, where $A_D \cap \mathbb{R}^{\geq 0} = \emptyset$, is a finite or infinite sequence of pairs $(a_0, t_0), (a_1, t_1), \ldots, (a_i, t_i), \ldots$ where, for all i, $a_i \in A_D$, $t_i \in \mathbb{R}^{\geq 0}$, and for consecutive pairs (a_i, t_i) and (a_{i+1}, t_{i+1}), $t_i \leq t_{i+1}$. ■

A finite timed sequence may represent either a finite observation or an infinitely long observation where no more actions occur after some point. This limitation is overcome by pairing the sequence with the length of observation [LV96, §2.3.1], giving a *timed sequence pair*.

Definition 2.2.4
A *timed sequence pair* is a pair (α^t, l^t) where $\alpha^t = (a_0, t_0), \ldots, (a_i, t_i), \ldots$ is a timed sequence and $l^t \in (\mathbb{R}^{\geq 0} \dot{\cup} \infty)$, such that if α^t is infinite $l^t = \mathit{limit}(t_0, \ldots, t_i, \ldots)$, and otherwise $l^t \geq t_i$ for all i. ■

Timed sequence pairs can also be categorised by length and limit time. A timed sequence pair is admissible if $l^t = \infty$, closed if $l^t < \infty$ and α^t is finite, and Zeno otherwise. The three categories are shown in Figure 2.1b. As there are no subsequences of delay transitions to consider, it is not possible to distinguish an observation where the final time would be reached in finitely many delay transitions and one where it would only be approached by an infinite sequence of delay transitions. The difference is unimportant if such Zeno executions are excluded from the outset [LV96, §2.3.2].

Timed sequence pairs overcome the problems of timed traces. Initial delays, delays between discrete actions, and final delays are represented uniquely, even if, for the last, some representational power is lost. Actions can be hidden by simply omitting pairs. Furthermore, the passage of time is not represented by instantaneous transitions but by more intuitive clock readings. Translations between timed sequence pairs and timed traces are readily defined [LV96, §2.3.1].

The *timed words* of the language-theoretic approach [AD94] are, in the present terminology, admissible, infinite timed sequence pairs where the time components increase strictly; though strictness is not essential [AD94].

2.2.2.3 Trajectories and continuous traces

In a timed trace, delays between actions are represented as sequences of discrete delay transitions. In a timed sequence pair, the length of delays are recorded explicitly. A third alternative is to represent delays as continuous intervals called *trajectories*. Trajectories form the basis of a unified representation of continuous executions and traces which generalises naturally to hybrid systems. The unified representation also avoids a technical issue that arises in the absence of time determinism. The modern approach to Timed Input/Output Automata (TIOA) [KLSV06] is based on trajectories. This subsection repeats its core definitions.

TTSs describe the structure of transitions between states; they abstract from the representation of the states themselves. Trajectories are best described, however, in terms of the continuous evolution of variable values. For the following definitions, a state should be understood as a valuation of an implicit, finite set of variables. In any case, the variables are ultimately discarded in the definition of continuous traces.

Definition 2.2.5
A *closed V-trajectory* τ of *limit time* $0 \leq t < \infty$ is a function from the interval $[0, t]$ to a valuation of variables in V.[3] ■

Definition 2.2.6
An *open V-trajectory* τ of *limit time* $0 < t \leq \infty$ is a function from the interval $[0, t)$ to a valuation of variables in V. ■

Definition 2.2.7
A *V-trajectory* τ, also termed a *trajectory* if the value of V is clear or unimportant, is either a closed V-trajectory or an open V-trajectory. ■

The notation $ltime(\tau)$ stands for the limit time of τ. The notation $\tau^t(x)$ stands for the value of variable $x \in V$ at time t in trajectory τ. For closed trajectories only, the shorthand τ^{ltime} is used instead of $\tau^{ltime(\tau)}$. The *domain* of a trajectory $dom(\tau)$ is the interval $[0, ltime(\tau)]$ if τ is closed or the interval $[0, ltime(\tau))$ if τ is open. Trajectories with domain $[0, 0]$ are called *point trajectories*.

Trajectories are functions from anchored intervals of positive reals to values of variables. They model variable changes over continuous intervals of time. Different types of variables are characterised by restrictions on how their values may change during an interval. Discrete variables must, for instance, be constant within a trajectory. Clock variables must change linearly, and often at a single common rate. Various subclasses of hybrid variables are defined by restrictions on their continuity and derivatives.

Definitions of a prefix ordering and a concatenation operator on trajectories are necessary for later descriptions.

Definition 2.2.8
Given V-trajectories τ and υ, τ is a prefix of υ, written $\tau \leq \upsilon$, if $ltime(\tau) \leq ltime(\upsilon)$ and $\tau^t(v) = \upsilon^t(v)$, for all $v \in V$ and all $t \in dom(\tau)$. ■

Definition 2.2.9
The *concatenation* of V-trajectories τ and τ', with τ closed, written $\tau \frown \tau'$, is defined on $[0, ltime(\tau) + ltime(\tau')]$ if τ' is closed, and on $[0, ltime(\tau) + ltime(\tau'))$ otherwise:

$$(\tau \frown \tau')(t) = \begin{cases} \tau(t) & \text{if } t \leq ltime(\tau) \\ \tau'(t - ltime(\tau)) & \text{if } t > ltime(\tau) \end{cases}$$ ■

When two trajectories τ and τ' are concatenated, a single valuation must be chosen at their juncture. In Definition 2.2.9 [KLSV06, §3.3.3] the last valuation of τ is chosen, but sometimes the first valuation of τ' is preferred [Pnu94]. The concatenation of an infinite sequence of trajectories is well-defined [KLSV06, §3.3.3].

Continuous executions and traces are both defined in terms of an alternating sequence of trajectories and discrete actions.

[3] Although trajectories and the silent action are both written with the symbol 'τ', there is no ambiguity because the two are used in distinct formalisms.

Definition 2.2.10
Given a set A of discrete actions and a finite set V of variables, a *finite* (A,V)*-sequence* is an alternating sequence of V-trajectories and elements of A, where the first and last elements are trajectories: $\alpha = \tau_0, a_0, \tau_1, a_1, \tau_2, \ldots, \tau_n$. All trajectories but the last must be closed. ■

Definition 2.2.11
Given a set A of discrete actions and a finite set V of variables, an *infinite* (A,V)*-sequence* is an alternating sequence of closed V-trajectories and elements of A, where the first element is a trajectory: $\alpha = \tau_0, a_0, \tau_1, a_1, \tau_2, \ldots$ ■

Definition 2.2.12
An (A,V)-sequence is either a finite (A,V)-sequence or an infinite (A,V)-sequence. ■

Definition 2.2.13
The *limit time of an* (A,V)*-sequence* $\alpha = \tau_0, a_0, \tau_1, a_1, \tau_2, \ldots$, written $ltime(\alpha)$, is equal to the limit of $\sum ltime(\tau_i)$. ■

The strict alternation between trajectories and actions in an (A,V)-sequence is a convenient simplification. It is not limiting since point trajectories can be inserted between actions that occur with no intervening delay. In contrast to concatenation, when two trajectories abut at an action there is no need to choose between the last valuation of one and the first valuation of the other. A choice may yet be required, however, if actions are removed from a sequence.

(A,V)-sequences can be classified similarly to traces and timed sequence pairs. They are admissible when $ltime = \infty$, closed when finite in length and ending in a closed trajectory, and Zeno otherwise. See Figure 2.1c.

The *continuous executions* of a model can be represented by (A,V)-sequences. They differ fundamentally from their discrete counterparts [MP93]. While continuous executions have no gaps, discrete models involve countably many samplings. Two factors require consideration when reasoning about continuous executions through sets of samplings. First, sampling instants must be chosen somehow to capture all important events [MP93], which is less of an issue for timed models with explicit events than for hybrid models where actions are represented solely by changes in state variables. Second, it is possible to define a TTS where, despite the fact that every sampling for an interval is defined, no continuous mapping from the interval exists [LV96, §A]. The time additivity clause of Definition 2.2.1 can be strengthened to forbid such anomalies by insisting that delay transitions have corresponding trajectories [LV96, §2.1].

The *continuous traces* of a model can be represented by $(A,\emptyset)$-sequences, that is (A,V)-sequences where the set of variables is empty. In a continuous trace, the trajectories map from an interval to an empty valuation: the only information present is the time between successive actions. A restriction operation on (A,V)-sequences [KLSV06, §3.4.4] serves to map continuous executions into continuous traces, and, more generally, to localize variables and actions.

Definition 2.2.14
The *restriction* of an (A,V)-sequence α to actions in A' and variables in V', $\alpha \lceil (A',V')$, is the sequence defined inductively by

$$(\tau, a, \tau', \alpha) \lceil (A',V') = \begin{cases} \tau, a, ((\tau'\alpha) \lceil (A',V')) & \text{if } a \in A' \\ (\tau \frown \tau'\alpha) \lceil (A',V') & \text{if } a \notin A' \end{cases}$$

with the domains of trajectories in the result restricted to $V \cap V'$. ■

When discrete actions are removed by restriction, the adjacent trajectories are concatenated, ensuring that the result is an (A,V)-sequence and solving the problems that occur when hiding internal actions in timed traces.

TIOA are defined directly in terms of trajectories and (A,V)-sequences. Appendix B contains a brief description.

2.2.3 Timed Automata

TTSs are a fundamental model for discrete transition systems in continuous time. But directly expressing timing behaviour in terms of delay transitions can be difficult. Any model that expresses non-trivial timing behaviours will have infinitely many states, with each having either zero outgoing delay transitions or infinitely many. Moreover, having delay transitions blurs the intuition that time passes in states and that transitions represent instantaneous actions.

The most common alternative is to represent time using state variables [Haa81, AL94, AH97] called *clocks*. Using clocks makes it easier to distinguish the discrete control structure of a model from its timed characteristics. In particular, a set of states can be clustered into *locations* that each give the same valuation to the subset of discrete variables. It can then be said that time passes in locations.

Timing behaviour and requirements are expressed as constraints and operations on clocks. Clocks are not, however, like ordinary program variables: they are inextricably bound to the flow of time. Their approximation in physical systems is not a simple issue. It is certainly more involved, for example, than the usual, if not always justified, approximation of natural numbers by machine words. Care must be taken when modelling systems for implementation to ensure that operations and constraints on clocks in the model can be realised in practice. Furthermore, models with clocks still have infinitely many states and decision procedures only exist for certain subclasses where operations and constraints on the clocks are restricted in one way or another.

Timed automata are the archetypal clock-based formalism. They are defined by adding clocks to finite automata (Definition 2.1.5) [AD94, HNSY94]. Although there are similar formalisms that permit an unbounded number of control states [AL94, KLSV06], the restriction to a finite number of states facilitates automated analysis, which is particularly important for timed models where checking properties involves considering both concurrent interleavings and intricate numerical detail.

Timed automata are defined in §2.2.3.1. The definition includes restrictions on expressions involving clock variables that make possible the structured representation of sets of clock values described in §2.2.3.2. Techniques for delaying actions or forcing them to occur within a definite period are presented in §2.2.3.3. More abstract considerations, and some of the problems that can occur when modelling in continuous time, are presented in §2.2.3.4.

2.2.3.1 Definition and semantics

Some restrictions on expressions involving clocks are required before the definition of timed automata. Two classes of expressions will be defined: *clock expressions* and *invariant clock expressions*. The latter are characteristic of a specific class of timed automata.

Definition 2.2.15
The *clock expressions over K*, written E_K, where K is a set of clocks, is the smallest set that satisfies:

$$\frac{k \in K \quad c \in \mathbb{Q}^{\geq 0}}{k \, R \, c \in E_K} \qquad \frac{k_1, k_2 \in K \quad c \in \mathbb{Q}^{\geq 0}}{k_1 - k_2 \, R \, c \in E_K} \qquad \frac{p, q \in E_K}{p \wedge q \in E_K}$$

where $R \in \{<, \leq, =, \geq, >\}$ and $\mathbb{Q}^{\geq 0}$ is the set of positive rational numbers and zero. ■

The form of clock expressions varies [AD94, Definition 3.6][HNSY94, Definition 3.1]. In this definition [BW04, §2.1], negation and disjunction are not permitted directly, but, for clock expressions used as transition guards in timed automata, they can be treated as a shorthand for conversion to Disjunctive Normal Form (DNF): relations in terms are changed to effect negation, and a separate transition is introduced for each disjunctive clause. Invariant clock expressions are less flexible.

Definition 2.2.16
The *invariant clock expressions over K*, written I_K, are the subset of the clock expressions over K where $R \in \{<, \leq\}$. ■

Invariant clock expressions are used to state *location invariants* in *timed safety automata* [HNSY94], which are otherwise essentially *timed transition tables* [AD94, Definition 3.7] with a single initial state. Timed safety automata have proved especially suitable for modelling and model-checking real-time systems. Timed transition tables can also be extended in other ways to yield different formalisms. In particular, abstract variants, namely timed Büchi automata and timed Muller automata [AD94, §§3.5–3.7], are fundamental to timed-language theory and to the semantics of timed logics. The term 'timed automata' will henceforth refer specifically to *timed safety automata* as typically defined for the Uppaal model checker [BW04].

Definition 2.2.17
A *timed automaton* $\mathcal{A} = (L, l_0, K, \mathsf{inv}_K, T)$ on A, a set of discrete actions, comprises finite sets of locations L and clocks K, an initial location l_0, a function inv_K from L to I_K, and a set of transitions $T \subseteq L \times E_K \times A \times 2^K \times L$. A transition $(l, g, a, R, l') \in T$ is written $l \xrightarrow{g}{a}{}_T^R\, l'$. ■

A transition between locations has three labels: a *guard* clock expression g which determines when the transition may occur, a discrete action a, and a set of clocks R that are reset to zero if the transition occurs. The meaning of these labels, the set of clocks K and the location invariants inv_K, are made precise by a semantic mapping to TTSs [BW04, Definition 2].

Definition 2.2.18
The *semantic model* of a timed automaton $\mathcal{A} = (L, l_0, K, \mathsf{inv}_K, T)$ over A and with n distinct clocks $K = \{k_1, \ldots, k_n\}$, is the TTS $[\![\mathcal{A}]\!] = (S, S_0, \longrightarrow)$:

- $S = L \times (\mathbb{R}^{\geq 0})^n$,
- $S_0 = \{(l_0, (0, \ldots, 0))\}$
- $\longrightarrow$ is the smallest relation such that

$$\frac{l \xrightarrow{g}{a}{}_T^R\, l' \quad [\![g]\!]_{\vec{v}}}{(l, \vec{v}) \xrightarrow{a} (l', \mathit{reset}_R(\vec{v}))}\ \text{action} \qquad \frac{d \in \mathbb{R}^{>0} \quad [\![\mathsf{inv}_K(l)]\!]_{\vec{v}+d}}{(l, \vec{v}) \xrightarrow{d} (l, \vec{v} + d)}\ \text{delay}$$

where $\vec{v} = (v_{k_1}, \ldots, v_{k_n})$ is a tuple of values, one for each element of K, $[\![\psi]\!]_{\vec{v}}$ is the value of the predicate ψ when atoms in K are interpreted as corresponding values in $\vec{v}$, $\mathit{reset}_R(\vec{v})$ sets element i to 0 if $k_i \in R$ and otherwise leaves it unchanged, and $(v_{k_1}, \ldots, v_{k_n}) + d = (v_{k_1} + d, \ldots, v_{k_n} + d)$. The mapping is undefined for timed automata where $[\![\mathsf{inv}_K(l_0)]\!]_{(0,\ldots,0)}$ does not hold, or, where, for any reachable state $(l, \vec{v})$, $[\![\mathsf{inv}_K(l)]\!]_{\vec{v}}$ does not hold. ■

Each state of the semantic model comprises a discrete control location and a value for each clock from the non-negative real numbers. All clocks are set to zero initially. There are two rules for transitions. An action transition at a location can only occur for those clock valuations where the guard expression is satisfied. When an action transition occurs the discrete location may change and a subset of the clocks are reset to zero. The values of the other clocks are not changed and thus discrete actions are instantaneous.[4] Delay transitions can occur provided that they do not violate the relevant location invariant. When a delay transition occurs, all clocks are incremented by the same amount and the discrete location is unchanged. The limited form of invariant clock expressions ensures that time interpolation and additivity hold. More liberal invariants necessitate the stronger antecedent $\forall \epsilon \leq d.\, [\![\mathsf{inv}_K(l)]\!]_{\vec{v}+\epsilon}$, that is, the invariant must be satisfied for the whole interval [BST97, Definition 2].

The semantic mapping of Definition 2.2.18 is only defined for timed automata where the location invariant at a state is never violated by an incoming action transition. This is the approach taken in the Uppaal model checker and in some semantic accounts of it [BV08]; a model where discrete actions violate an invariant is

[4] This decision is fundamental to a few programming languages, see §2.4 on the synchronous languages, and almost all timed modelling languages [Wan90, NS94, HNSY94, AL94, BM02, Lam02, BW04, KLSV06].

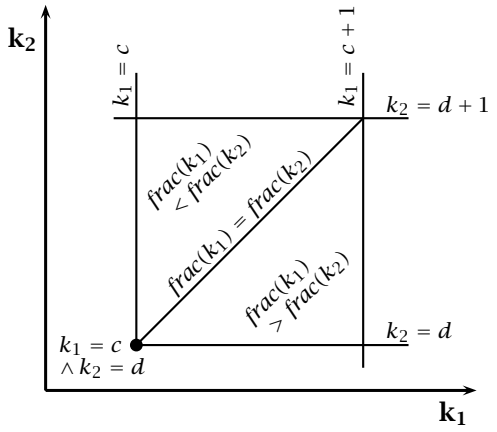
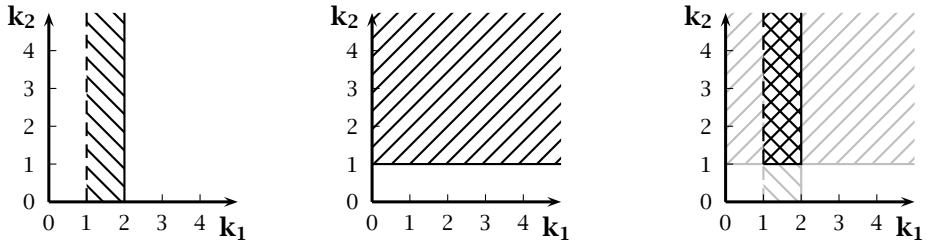
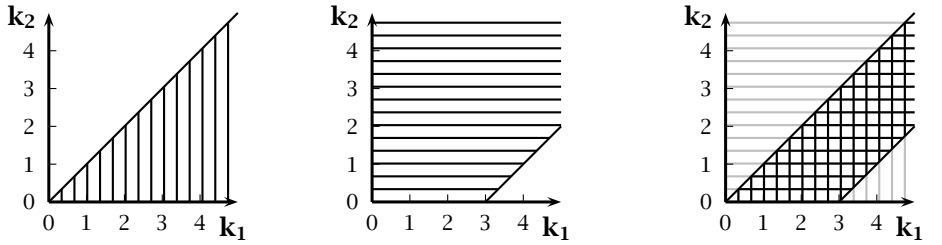


Figure 2.2: Dyadic clock regions for equalities



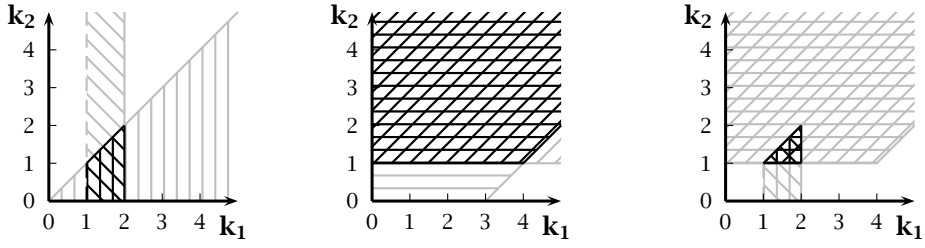
(a) $k_1 > 1 \wedge k_1 \leq 2$ **(b)** $k_2 \geq 1$ **(c)** $k_1 > 1 \wedge k_1 \leq 2 \wedge k_2 \geq 1$

Figure 2.3: Dyadic clock regions for constant comparison



(a) $k_1 - k_2 \geq 0$ **(b)** $k_1 - k_2 \leq 3$ **(c)** $k_1 - k_2 \geq 0 \wedge k_1 - k_2 \leq 3$

Figure 2.4: Dyadic clock regions for clock differences



(a) $k_1 > 1 \wedge k_1 \leq 2 \wedge k_1 - k_2 \geq 0$ **(b)** $k_2 \geq 1 \wedge k_1 - k_2 \leq 3$ **(c)** $k_1 > 1 \wedge k_1 \leq 2 \wedge k_2 \geq 1 \wedge k_1 - k_2 \geq 0 \wedge k_1 - k_2 \leq 3$

Figure 2.5: Mixed dyadic clock regions

considered invalid. A different approach is taken by other definitions [HNSY94, Definitions 3.3 and 3.7] and semantic accounts of Uppaal [BW04, Definition 2] where a discrete transition cannot occur if it would violate the location invariant of the destination state. Definition 2.2.18 can be altered to express this idea by adding another premise $[\![\mathsf{inv}_K(l')]\!]_{reset_R(\vec{v})}$ to the action rule, and by omitting the second part of the last clause that describes when the mapping is undefined.

The semantics of other timed formalisms can also be given in terms of TTSs, but timed automata themselves are often used instead [NSY93, JMO93].

2.2.3.2 Clock regions

The restricted form of clock expressions and invariant clock expressions allows a structured representation of sets of clock values as *clock regions* [AD94, §4.2]. For systems of two clocks, that is $K = \{k_1, k_2\}$, clock regions are readily visualized as two-dimensional graphs. Some examples are presented in Figures 2.2–2.5 where the values of k_1 and k_2 are plotted on the horizontal and vertical axes, respectively. No generality is lost in assuming that the constants, c and d in the figures, have integer values: for the set of rational constants in a given timed automaton, a smallest granularity can always be chosen [AD94, §4.1].

Constraints with equality are drawn as points and straight lines. In Figure 2.2, the dot at bottom-left represents the clock region where $k_1 = c$ and $k_2 = d$. The horizontal and vertical lines represent clock regions where only one of the clocks is constrained. The diagonal line represents the clock region where both clocks have the same fractional part. And the triangular regions above and below the diagonal line are the clock regions where the fractional part of k_1 is, respectively, strictly less than or strictly greater than the fractional part of k_2.

Inequalities are drawn as convex polygons. Some examples of constraints built by comparing single clocks with constants are shown in Figure 2.3. The clock region that corresponds to the conjunction of two constraints is the intersection of the individual clock regions. Clock regions are always contiguous because disjunction is forbidden. Some examples of constraints built by comparing clock differences with constants are shown in Figure 2.4. The slope of the diagonal lines in clock regions is always equal to one because multiplication is not allowed in clock expressions. Some examples of conjunctions of single clock and clock difference inequalities are shown in Figure 2.5.

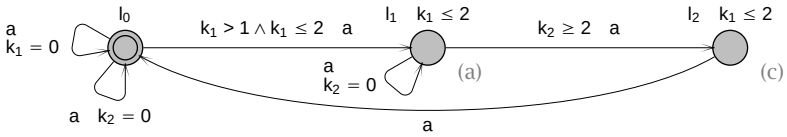
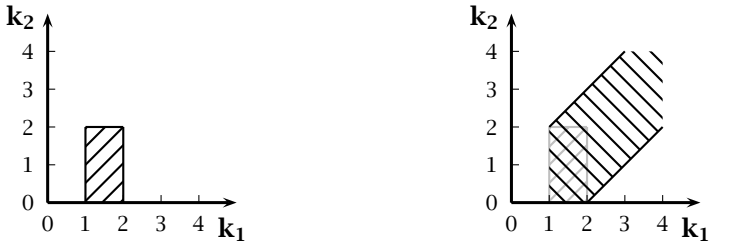


Figure 2.6: Example timed automaton to demonstrate clock regions

By way of example, consider the timed automaton with two clocks and three locations in Figure 2.6. The initial location, l_0, is marked by an inner circle. All of the transitions are labelled with the discrete action a. There are two self-loops at location l_0, one resets clock k_1, the other resets clock k_2. The absolute and relative values of the clocks are thus unconstrained. The transition to l_1 can only be taken when $1 < k_1 \leq 2$ and k_1 cannot exceed 2 at l_1 due to the location invariant. The set of possible clock values at l_1 is described by the clock region of Figure 2.3a. Clock k_2 can be reset at l_1, but clock k_1 cannot. The transition to l_2 may only occur when $k_2 \geq 2$, and the set of possible clock values at l_2 is described by the clock region of Figure 2.3c. This is a simple example. Locations need not always be paired with the same set of possible clock values.

For any given timed automaton there are a finite number of clock regions [AD94, Lemma 4.5], that is, for a given number of clocks and the (scaled) value of the largest constant occurring in guards or invariants, only so many lines, points, and polygons can be formed. It is thus possible to construct finite abstractions, called *region automata*, that are sound and complete for deciding language emptiness [AD94, §4.3].



(a) $k_1 \geq 1 \wedge k_1 \leq 2 \wedge k_2 \leq 2$

(b) Time successors of $k_1 \geq 1 \wedge k_1 \leq 2 \wedge k_2 \leq 2$

Figure 2.7: An example of the time successor relation

This theoretical result underpins the possibility of finite model-checking of timed automata, which otherwise have an infinite state space. In practice, sets of clock values are usually encoded as Difference Bound Matrices (DBM) rather than represented explicitly as clock regions [BW04].

Computations that involve sets of clock values, like the construction of a region automaton, where discrete locations are paired with convex clock regions, are not defined in terms of delay transitions, but rather through a *time successor* relation [AD94]. The time successors of a dyadic clock region are readily visualized as the clock region encompassed by diagonal lines of unit slope extending up and to the right from each corner of the original region. An example is given in Figure 2.7. The diagonals are of unit slope because all clocks increase at the same constant rate.

2.2.3.3 Guards, invariants, and urgency

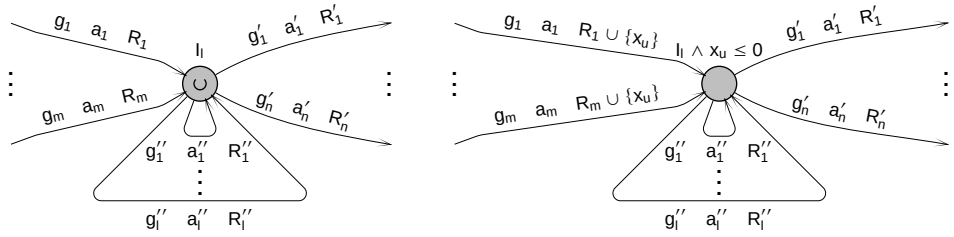
In addition to expressing sequences of actions, timed automata enable occurrences of actions to be placed precisely in time. That is, they can convey when actions may occur and when they must occur. In timed automata, the 'may' is expressed in transition guard expressions and the 'must' in location invariants. But there are alternative approaches and for the purpose of comparison the expression of actions in time is usefully divided into three types of mechanism: those for delay, those for timeout, and those for urgency. Each is now considered in turn, with particular attention devoted to urgency and its treatment in two extensions of timed automata.

Delays before an action may occur are modelled in timed automata with $>/\geq/=$ conjuncts on transition guards. They are relative to the instants when the clocks in the guard were last reset, or, for clocks that have never been reset, to the system start time. Constraints can thus be defined over paths through a model. Other formalisms, notably timed process algebras [Wan91, NSY93, LL97, BM02], introduce special notations for expressing delays.[5] For instance [Wan90], $\epsilon(3).a.0$ may indicate a process that waits for 3 units of time before it is willing to engage in an a action. In this case and others, and in contrast to timed automata, the delay is relative to the instant of entry into a state. Constraints over paths can be recovered using parallel composition and synchronisation provided such operators are available.

Timeouts are limits on when actions may occur. Different formalisms interpret the expiry of a time limit on an action in different ways, but two basic approaches are discernable: either the possibility of the action elapses, or the action (or another) must occur without further delay. Timed automata take the former approach: the possibility of performing a transition whose guard contains $</\leq/=$ conjuncts expires when the guard is no longer satisfied. The elimination of possible actions by the progression of time is sometimes termed 'timeout' [NSY93] or 'weak choice' [BM02, p. 33]. A delay transition that crosses an upper bound on an action effectively moves from a state where there is a transition labelled with that action to a state where there is not.

A different approach to timeouts is taken in RTCCS [Wan90, Wan91], a timed extension of the process algebra CCS. In RTCCS, action possibilities may only be removed

[5] That some of these are based on discrete-time does not invalidate the comparison.



(a) Macro notation **(b)** Expanded form (x_u is a new clock)

Figure 2.8: Urgent location macro

by the occurrence of a τ-transition (refer §A.2). This is a consequence of a *maximal progress assumption* which mandates that τ-transitions occur with priority over delay transitions. Upper bounds on actions are modelled as the choice between the action and a delay-prefixed τ-action.[6] Besides allowing the expression of timeouts, an assumption of maximal progress reduces non-determinism in processes and forces synchronisations between processes and other hidden actions to occur as soon as possible. Such an assumption may not always be necessarily wanted or warranted.

Urgency is a more general way to assert the priority of (certain) discrete actions over delay actions, and, under the assumption that time cannot be stalled indefinitely, of specifying that an action must occur immediately. In timed automata, urgency can be expressed through location invariants, including the particular case of *urgent locations*, or one of two extensions for marking individual transitions.

A location invariant states when the enabled actions at a location must occur without any delay. Time may only pass—equivalently, delay transitions may only occur—at a location while the corresponding location invariant is satisfied. For closed location invariants, like $k \leq 3$, time effectively stops at the upper bound. For open location invariants, like $k < 3$, time may not pass beyond the given limit. The expectation that physical time cannot really be stopped is expressed in models by rejecting executions and traces where it does, that is, only admissible traces or sequences are usually considered. There are therefore two possibilities in a model when a location invariant impedes the flow of time. If discrete transitions are enabled from that location then they become *urgent*: one of them must occur immediately. Otherwise the model must be rejected as infeasible.

Location invariants allow the specification of so called *urgent locations* where time may not pass at all. These locations are useful enough that a macro notation has been introduced to represent them [BW04]: the letter 'u' is drawn inside a location to mark it as urgent, see Figure 2.8a. A timed automaton with urgent locations can be transformed into a standard timed automaton by both introducing a new clock x_u, which is reset on entry to an urgent location,[7] and adding the clause $x_u \leq 0$ to the invariants of all locations marked urgent, see Figure 2.8b.

While location invariants are more discriminating than a universal assumption of maximal progress, they cannot readily express the same type of urgency: that a transition should be urgent only whenever it is enabled. This deficiency can be remedied by extending timed automata with either *deadlines* [BST97] or *urgent actions* [BW04].

Transitions in timed automata with deadlines are labelled, in addition to the standard guard g, action a, and reset elements R, with a deadline expression d. At a location, the negated disjunction of the deadline expressions of all outgoing transitions, $\neg(d_1 \vee \ldots \vee d_n)$, is effectively treated as a location invariant. A timed automaton with deadlines is only considered valid if this implicit location invariant is right closed, and also if for each transition, $d \implies g$. Under these conventions, time can only progress at a location while none of the deadline expressions on outgoing transitions are satisfied. In other words, a transition is urgent whenever its deadline expression is satisfied. The restriction on the implicit invariant expression prevents the possibility of a

[6] For example, an upper bound of 3 on an action a could be expressed: $a.\text{NIL} + \epsilon(3).\tau.\text{NIL}$.

[7] It is immaterial whether or not the clock is reset by self-loop transitions.

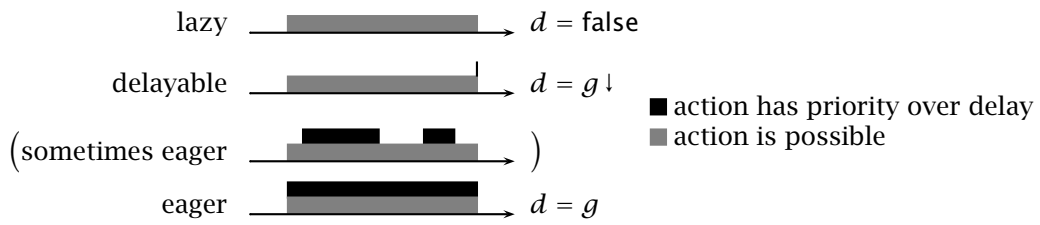


Figure 2.9: Categories of urgency for timed automata with deadlines [BST97, Figure 1]

deadline being approached but never reached. The restriction on individual deadline expressions ensures that if a transition is urgent then it is also enabled. The effects of deadline transitions could also be achieved, albeit in a less structured way, by relaxing the restrictions on location invariants.

Three different categories of urgency have been proposed for timed automata with deadlines. They are shown on time lines in Figure 2.9. The gray intervals represent periods when an action is possible because its guard g is satisfied. The black intervals represent periods when an action is urgent because its deadline d is satisfied. *Lazy* transitions are those where $d = \mathsf{false}$. They are never urgent. *Delayable* transitions are those where $d = g\downarrow$, where $g\downarrow$ is the falling-edge of a right-closed guard; for instance if $g \leq 7$ then $g\downarrow = 7$. *Eager* transitions are those where $d = g$. They are urgent whenever enabled and hence behave as if under an assumption of maximal progress. It is also possible to express transitions that are only urgent at particular times; that is transitions that are *sometimes eager*.

Only the lazy and delayable modes can be expressed in standard timed automata. A transition is lazy if its guard g is not curtailed by the source location invariant i, that is if $g \wedge i = g$, and delayable otherwise.

Urgent actions are an alternative way of expressing eagerness. The idea is simple. Given a timed automaton over A, an extra set $U \subseteq A$ is specified. The actions in U are considered eager. They must occur with urgency whenever they are enabled. At this point, that means whenever their guard and the location invariant at the destination location are satisfied, but, in the networks of timed automata discussed in the next section it will also depend on the behaviour of other components.

2.2.3.4 Progress, liveness, and feasibility

While timed automata alone are adequate for expressing possible behaviours, additional mechanisms are required to express certain necessary behaviours. Progress is one such mechanism and, in tandem with location invariants, it is adequate for situations where concrete timing bounds are known or can be assumed. When more abstract liveness constraints are required, they are sometimes stated together with a timed automata model as additional acceptance sets or logical formulas; though care must be taken to avoid any undue interference with the step-by-step operation of the model. Besides such explicit requirements, models are usually implicitly required to be feasible, in the sense that they do not specify so called Zeno executions where time cannot progress beyond a fixed bound.

Under an assumption of progress, a transition system cannot rest indefinitely in a state where actions are always possible; one of the actions must eventually occur. Progress is assumed for many untimed formalisms. It is less potent for timed automata, however, because they may stay forever in any location where the invariant is equivalent to true, even if one or more discrete transitions from that location are always enabled, since progress is still made, in a sense, through delay transitions. Still, the combination of progress and non-trivial location invariants is adequate for expressing bounded liveness requirements, where concrete deadlines are set on the occurrence of discrete transitions. Bounded liveness is easy to reason about operationally and readily incorporated into model checking algorithms.

Liveness constraints that are more abstract are necessary when it is impossible or

unnatural to place specific, concrete bounds on behaviours. They are typically affixed to transition system specifications to filter out certain infinite executions that would otherwise be permissible. Executions that do not satisfy the additional constraints can be ignored when proving properties of a model, but their impossibility must be ensured when implementing a model. Two approaches typically applied to untimed formalisms, namely acceptance sets and pairings with formulas of temporal logic, are just as applicable for stating abstract liveness constraints on timed automata models.

In language-based timed models [AD94], liveness constraints are specified by designating subsets of *accepting locations* and an *acceptance condition*. As for (untimed) ω-automata, there are two standard approaches. In one, *timed Büchi automata*, a single subset of accepting locations is specified, and an execution satisfies the *Büchi acceptance condition* if it returns to at least one of those locations infinitely often. In the other, *timed Muller automata*, a set of subsets of accepting locations is specified, and an execution satisfies the *Muller acceptance condition* if the set of locations it returns to infinitely often is included in one of those subsets.

Liveness constraints can also be specified by pairing models with formulas of temporal logic [AL94, SGSAL94]. Due to the potential subtleties of such formulas, they are often limited to forms that express fairness alone; but most other liveness properties can be deduced from these [Lam02, §8]. Fairness assumptions are, in a way, a discriminating and compositional alternative to universal assumptions of progress.

When liveness constraints are paired with automata (timed or not), they should only be allowed to influence choices repeated across the infinite totality of an execution but not to influence the step-by-step choice of transitions. A paired automaton and liveness constraint where every finite execution of the former can be extended into an infinite execution that satisfies the latter is termed *machine closed* [AL94, AL91].[8]

The assumption that time increases without bound is so fundamental that it is usually an implicit liveness constraint, and a model is termed *feasible* if it respects this expectation. The concept of machine closure makes possible a succinct definition of feasibility. A timed automaton that contain a single clock called *now*, which is never reset, is feasible if it is machine closed with respect to the liveness property: $\forall t \in \mathbb{R}.\, \Diamond(now > t)$ [AL94]. Feasibility can also be defined in terms of the categorisations discussed in §§2.2.2.1–2.2.2.3 and shown in Figure 2.1: in a feasible model there is at least one admissible execution from every reachable state [KLSV06, §4.3.2].

Although feasibility involves, in general, considerations of liveness, some causes of infeasibility can be identified by static approximations. For instance, a system is *time reactive* [BST97] if at least one discrete action is enabled from a state where time is inhibited.[9] Time reactivity is necessary for feasibility, but not sufficient. A time reactive system could, for instance, still perform an infinite sequence of discrete actions without letting time pass.

The concept of *Zenoness* is closely related to that of feasibility. A trace or execution where time does not increase without bound is termed Zeno.[10] Any timed model where delay is possible has Zeno traces; for instance, traces that simply degenerate into an infinite number of successively smaller fractional steps. These traces are simply rejected because they cannot occur in reality. In some modelling frameworks, it is possible to distinguish whether the cause of Zenoness is a system, its environment, or the two together [SGSAL94][KLSV06, §6.3]. A model is not feasible if it contains reachable states from which only Zeno traces are possible. Such states usually indicate an error in modelling or specification since they represent systems that cannot really exist let alone be implemented. Note, though, that feasibility is necessary for implementability but not sufficient. For instance, timed automata where arbitrarily many actions can occur in a finite period of time, or where actions may be arbitrarily close to one another [AD94, §3.6] are feasible but not implementable.

[8] Additional care is required to treat liveness constraints when a distinction is made between a system and its environment [SGSAL94].

[9] The original, informal definition is: 'time can progress at any state unless an untimed transition is enabled'. It could also be interpreted as a form of maximal progress assumption.

[10] This term refers to the ancient author of apparent paradoxes that declare the impossibility of crossing a distance because first half of it must be crossed, and then half of that, and so on in unending recursion.

2.3 Uppaal

Timed automata are fundamentally expressive enough to model a large class of systems, but the modelling task becomes more manageable if additional features are introduced. Moreover, since it is especially tricky to model timed systems and specifications, simulation and analysis tools are all but essential.

Uppaal [LPW97, BDL04, BW04] is a software tool for creating models in an extended version of timed automata and for verifying certain of their properties by exhaustive state space exploration. It comprises three main components: a graphical user interface for specifying networks of timed automata using diagrams and a C-like description language, an interface for running interactive simulations and viewing traces, and a model checking engine for deciding whether a model satisfies given formulas.

In Uppaal, timed automata are extended in two main ways: finite variables are allowed in addition to control locations, and sets of timed automata can run in parallel and intercommunicate. Variables and related features are described in §2.3.1. The different types of communication and the various priority rules are described in §2.3.2. Although the semantics of Uppaal models is involved, the fundamentals can be understood through the approximation of synchronizing timed automata presented in §2.3.3. The three subsections present the most important Uppaal modelling constructs with a minimum of formal definition. A more precise account is presented in Chapter 5.

The model checking interface is described briefly in §2.3.4. The focus is on the language for specifying properties rather than on algorithmic details [BW04] or practical techniques for effective verification [BDL04, §6].

2.3.1 Variables

Strictly speaking, Uppaal models are not composed of timed automata with locations and transitions, but rather of *processes* with *nodes* and *edges*. Processes may have *local data variables* in addition to clock variables. Furthermore, it is not processes that are manipulated in the Uppaal graphical editor, but rather *parameterised processes*, which are instantiated with concrete parameter values to give processes. This facilitates the creation of multiple components that share the same control structure.

There are three basic types of data variables: bounded integers, booleans, and scalar variables; and two type constructors: records and arrays. Scalar variables are a special feature of Uppaal. They take values from a finite set and the only operations allowed on them are equality testing, assignment, and array indexing. These restrictions allow the size of the reachable state space of models containing scalar variables to be compressed using a technique called *symmetry reduction* [BDL+06]. Type aliasing (`typedef`) is especially important for scalar types because each declaration introduces a unique type.

Process edges incorporate and exploit data variables through three enhancements to standard transitions. First, their guard expressions can contain predicates over variables. Second, in addition to clock resets, an edge can be labelled with a sequence of assignments and function calls that update the value of data variables when the edge is taken. The assignment expressions and the functions are defined using operators and control structures based on the C language [BDL+06], and, additionally, with bounded existential and universal quantifiers. Third, edges can be labelled with a list of *selection bindings*. Each selection binding pairs a variable name with a data type. The names are bound over the guards, action expressions, and update sequences of corresponding edges. In one way, selection bindings represent a non-deterministic choice since their values are effectively chosen arbitrarily from their associated types when an edge is taken. But, because they may be used in guard and action expressions, the choice of values in fact determines whether an edge is enabled or not, and it is more accurate to visualize an edge with selection bindings as a set of simpler edges where selection bindings have been instantiated with explicit values. This form of implicit iteration make selection bindings especially useful in combination with arrays.

The semantics of data variables and the extra edge labels can be defined by mapping

every node and variable valuation in a process to a distinct location in a corresponding timed automaton. An edge from one node to another is mapped to multiple transitions that account for its selection bindings, the variable valuations where its guard remains satisfiable, and the effect of assignments and function calls on variables. A precise definition of this mapping is presented in §5.2.3.

2.3.2 Communication and priority

An Uppaal model comprises channels, processes (instantiated templates), shared variables, shared clocks, and priority declarations. The edges of a process can be categorised by whether they are local to a process, or whether they synchronise with other processes on a channel. They are subject to a variety of priority rules.

There are two types of action in Uppaal: *local actions* and *channel actions*.

Local actions always involve a single process. They are represented graphically as edges without action labels. When a local action occurs on an edge that does not change any shared variables and that cannot be influenced by priority rules, it is effectively unobservable and may be considered a τ-action (refer §A.2). Otherwise, it may be possible to observe local actions by their effect on shared variables, and also to influence them through the priority mechanisms described below.

Channel actions usually involve more than one process. Channels are names that allow processes to communicate without identifying one another directly. To each channel c are associated output and input actions. These are indicated, respectively, by suffixing the channel name with an exclamation mark $c!$ or a question mark $c?$. When processes synchronise on a channel, the assignments and functions associated with the output action are processed before those of any input actions. This allows, in particular, value-passing synchronisations to be simulated using shared variables.

There are two types of channels in Uppaal. *Handshake channels*, or often just 'channels', synchronize a single input action with a single output action. *Broadcast channels* synchronize a single output action with all enabled input actions on the same channel. A broadcast output can occur on a channel even when no corresponding input actions are enabled. The updates of a broadcast output occur before those of any synchronized inputs, which are evaluated in the order of their relative instantiation.[11]

In Uppaal models, the enabledness of edges depends not just on their guards, but also on the semantics of communication through channels which are subject to several forms of priority, namely: urgent channels, channel and process priorities, and committed nodes.

Enabled actions on an *urgent channel* take priority over delays. They are, like the eager transitions of timed automata with deadlines, a way of locally specifying maximal progress. Unlike eager transitions though, they are declared per channel rather than per transition and their enabledness may depend on the willingness of other processes to communicate. Both handshake and broadcast channels may be marked urgent.

While channel urgency gives priority to discrete actions over delays, *channel priority* gives enabled edges on some channels priority over those on other channels. A channel priority declaration defines a total but not necessarily strict ordering between channels. A special 'default' channel identifier defines the status of channels that are not listed explicitly and also of local actions, which are subject to the same rules. An edge cannot be triggered when there is, in the same system, an enabled edge of strictly higher priority. The possible triggerings of edges with the same channel priority are further determined by a separate *process priority* ordering (also total and non-strict), where the priorities of all processes involved in potential synchronisations are considered and compared.[12] Delay transitions are unaffected by channel and process priorities.

Regardless of channel and process priorities, the outgoing edges of *committed nodes* have priority over delays and the outgoing edges of uncommitted nodes. Committed nodes allow sequences of edges to be made atomic, at least with respect to delays and to edges from uncommitted nodes; the edges of simultaneously active com-

[11] See the language reference distributed with Uppaal, under the section 'Templates/Edges'.

[12] The rules for determining edge priorities are given in the language reference distributed with Uppaal under the section 'Priorities'

mitted nodes are still interleaved with one another. Committed nodes were used in earlier versions of Uppaal to model broadcast communication and for array manipulations requiring iteration. They can reduce the number of possible interleavings and hence the state space of a model. While the committed attribute is conferred on nodes, treating it as an attribute of edges facilitates the definition of a compositional semantics [BV08]. The status of nodes cannot be completely neglected, however, since an active committed node without any (enabled) outgoing edges will still block delays and edges from uncommitted locations.

2.3.3 Semantics

Defining the semantics of Uppaal models requires more attention than it is often given. Recent accounts [BV08] give precise definitions that include most features and whose operators have the properties expected of them. Presently it will suffice, however, to describe the fundamentals—that is, excluding data variables, priorities, channel urgency, and broadcast channels—in terms of timed automata that communicate through paired synchronisations [Sto02, §7.5.2]. An Uppaal model is, in this case, n concurrent timed automata composed using parallel and restriction operators:[13]

$$(\mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_n) \backslash \mathsf{Acts},$$

where Acts is the set of all actions excluding the local action τ. The two operators are defined in the manner of CCS.[14] Given two timed automata, each may either act alone, or alternatively, transitions labelled with complementary actions may synchronise resulting in a silent action that precludes further synchronisation.

Definition 2.3.1
The *parallel composition* of two timed automata $\mathcal{A} = \mathcal{A}_1 \parallel \mathcal{A}_2$ gives a timed automaton $\mathcal{A} = (L, l_0, K, \mathsf{inv}_K, T)$ where

$$\begin{aligned} L &= L_1 \times L_2 \\ l_0 &= (l_{0_1}, l_{0_2}) \\ K &= K_1 \dot{\cup} K_2 \\ \mathsf{inv}_K(l_1, l_2) &= \mathsf{inv}_{K_1}(l_1) \wedge \mathsf{inv}_{K_2}(l_2) \end{aligned}$$

and T is the smallest relation such that

$$\frac{l_1 \xrightarrow[a_1]{g_1\ R_1}_T l_1'}{(l_1, l_2) \xrightarrow[a_1]{g_1\ R_1}_T (l_1', l_2)}\ \text{left} \qquad \frac{l_2 \xrightarrow[a_2]{g_2\ R_2}_T l_2'}{(l_1, l_2) \xrightarrow[a_2]{g_2\ R_2}_T (l_1, l_2')}\ \text{right} \qquad \frac{l_1 \xrightarrow[a]{g_1\ R_1}_T l_1' \quad l_2 \xrightarrow[\bar{a}]{g_2\ R_2}_T l_2'}{(l_1, l_2) \xrightarrow[\tau]{g_1 \wedge g_2\ R_1 \cup R_2}_T (l_1', l_2')}\ \text{both}$$

where $a_1, a_2 \in \mathsf{Acts}_\tau$ and $a \in \mathsf{Acts}$. ∎

Location invariants only restrict which delay transitions may occur. Their violation by a discrete transition indicates a modelling flaw per Definition 2.2.18.

Note that, unlike the process algebras, see Appendix A, and other programming languages with concurrency, like the synchronous languages, see §2.4, concurrency in Uppaal is neither nimble nor hierarchical; in any model, a fixed number of processes run at a single level.

The *restriction* operator prunes all transitions labelled with actions from a given set.

Definition 2.3.2
The *restriction* of a timed automaton $\mathcal{A} = (L, l_0, K, \mathsf{inv}_K, T)$ to actions over channels not in a set P is written $\mathcal{A} \setminus P = (L, l_0, K, \mathsf{inv}_K, T')$, where T' is the smallest relation

[13] Relating an Uppaal 'network of timed automata' to a parallel composition and restriction of individual timed automata actually requires more care than the present treatment infers [BV08, §4].
[14] refer §A.3

such that

$$\frac{l \xrightarrow[a]{g\ R}_T l' \quad a \notin (P? \cup P!)}{l \xrightarrow[a]{g\ R}_{T'} l'}\ \text{restrict}$$

where $P?$ and $P!$ are, respectively, sets of input and output actions on channels in P. ■

A symbolic execution of an Uppaal model is an alternating sequence of sets of states, where variables and locations are fixed but clock values may vary, and completed actions.

The actions of processes in an Uppaal model are asynchronously interleaved, while all clocks increase at the same constant rate. Sets of executions that differ only in the values of clocks, so called *symbolic executions*,[15] are represented by an alternating sequence of completed actions and states. A completed action is either a local action, a complementary pair of synchronized handshake actions, or a set of broadcast actions containing one output action and all input actions enabled at the instant of occurrence. A state is a triple of the active process nodes, the current variable valuation, and a symbolic representation of clock values and their time successors. Only executions containing completed communications are considered. Thus, in contrast to the *open system* interpretation made elsewhere in this chapter, Uppaal models are interpreted as *closed systems* [Sto02, §7.5.2]—processes are always simulated and analyzed in combination with an explicit model of their environment.

2.3.4 Model Checking

Uppaal is used to create and simulate timed models, but its real strength is in automatically verifying their properties using an efficient reachability analysis algorithm.

Given a closed TTS $\mathcal{T}$ and a property φ, Uppaal decides whether the former satisfies the latter, that is, whether $\mathcal{T} \vDash \varphi$. Properties are evaluated over paths of alternating delay and action transitions from the initial state s_0. They are restricted to the forms:

$\mathsf{E}\Diamond p$, asserting that it is possible to reach a state satisfying p on a path from s_0.

$\mathsf{E}\Box p$, asserting that there is a path from s_0 along which all states satisfy p. Either the path is infinite or ends in a state s with either no outgoing transitions or where $s \xrightarrow{d} s'$ is possible for all $d \in \mathbb{R}^{\geq 0}$.

$\mathsf{A}\Diamond p$, which is equivalent to $\neg(\mathsf{E}\Box \neg p)$.

$\mathsf{A}\Box p$, which is equivalent to $\neg(\mathsf{E}\Diamond \neg p)$.

$p \rightsquigarrow q$, which is equivalent to $\mathsf{A}\Box\,(p \text{ implies } \mathsf{A}\Diamond q)$.

In practice, state properties, written here as p and q, are formulated over the nodes and variable valuations of the component processes from which the TTS is derived. Additionally, a **deadlock** property asserts that no action transitions leave a state or its time successors (states reachable via a delay transition).

Uppaal provides witness traces for properties asserting existence, that is true properties of the form $\mathsf{E}\Diamond p$ or $\mathsf{E}\Box p$, or counter-examples for failed properties of the form $\mathsf{A}\Diamond p$ or $\mathsf{A}\Box p$. Not only do this feature give insight into the failure of a property to hold, but it is also useful for generating specific traces to see, for instance, whether a model is capable of performing them, or to better understand the behaviour of a system.

Model checking can be a very effective technique, but it is not usually 'push button verification'. There are two main reasons. First, although model checking algorithms are highly optimised, nothing can alter the exponential complexity of parallel composition. The state space of many models is simply too large to check effectively. It is not always clear in advance when this limit may be reached. It may, for example, be

[15] They are called, misleadingly, 'symbolic traces' in the Uppaal documentation.

possible to analyze a system of three processes in minutes, but, if the number is increased to four, the model checker may run indefinitely or crash for lack of memory. Analyzing complex models usually requires the application of modelling tricks and abstraction techniques. Second, most model checkers analyze finite state models, but many systems and protocols are designed for an indefinite number of processes, participants, or resources. Showing, for instance, that it is possible to infer a property of an arbitrary number of processes from an analysis of a finite number requires specific expertise [Hol03, Chapter 5].

2.4 Synchronous Languages

There are many synchronous languages. They can be categorised according to two main criteria: *classical* or *derivative*, *imperative* or *dataflow*. Both criteria are somewhat arbitrary. They aid in the discussion of what is a wide body of work, but, perhaps like all such categorisations, they should not be construed as fundamental.

The first criterion is historical. The classical languages are, in this thesis, those appearing in both the first book on synchronous programming [Hal93] and the 1991 special issue of the *Proceedings of the IEEE*: Esterel [BS91], Lustre [HCRP91], and Signal [LGGLBLM91]. The book also describes Argos, here considered one of the first derivative languages.[16] Many derivative languages have been developed to explore different design choices and possible extensions. None yet enjoy great success outside academia, but most are a rich source of ideas and techniques.

The second criterion refers to concepts, notation, and culture. Programming in an *imperative language* requires conceptualizing control foci that move around within the text of a program. At any instant there are active statements that influence the response of a program and inactive ones that do not. Programs are depicted as sequences of imperative statements, as Esterel usually is, as the circles and arrows of state machines, as Argos usually is, or as a mix of both [And96, Tec05]. The imperative style is traditionally associated with electronic and software engineering where systems are often designed as Mealy machines and implemented in algorithmic languages.

Programming in a *dataflow language* requires conceptualizing streams of values—sometimes as infinite sequences, sometimes as the successive results of unending iteration—as they are transformed by networks of interconnected functions. All functions are simultaneously active at an instant. Programs are either depicted as a sequence of function declarations, as the interconnected rectangles of a flow diagram, or, more usually, as a mix of the two. Lustre and Signal have each a textual syntax and a graphical syntax. Dataflow programming developed in computer science [WA85] and is natural in lazy functional languages [CP95], but similar concepts also occur in the mathematics and design of discrete feedback controllers and digital signal processing applications.

Many systems contain both imperative and dataflow elements. Modern synchronous languages often try to integrate both styles.

This thesis focuses on the imperative languages Argos and Esterel. Few of the modern derivative languages, besides Argos, are mentioned or discussed in any detail; not because they are unimportant, but rather because issues of timing and triggering can be satisfactorily studied without them. Version 5 of Esterel in pure form is adopted instead of the feature-rich, but more complicated version 7.

This section has three parts. The first, §2.4.1, discusses concepts common to all synchronous languages. The second and third describe specific languages, Argos in §2.4.2 and Esterel in §2.4.3.

2.4.1 Core concepts

The synchronous languages are typified by a focus on specific domains of application, §2.4.1.1, a common approach to execution, §2.4.1.2, underlying mathematical models,

[16] The term ‘derivative’ is used here with the sense of ‘derived from’ or ‘influenced by’. No negative connotations are intended.

§2.4.1.3, and the adoption of two fundamental principles that relate the behaviour of models to physical time, §2.4.1.4. These two principles require a new approach to issues of causality, §2.4.1.5, and influence the approaches to communication, §2.4.1.6, and concurrency, §2.4.1.7.

2.4.1.1 Domains of application

The three classical synchronous languages, Esterel, Lustre, and Signal, were initially and independently developed for rigorous programming in specific embedded programming domains, respectively: real-time sequencing and coordination [BMR83], discrete feedback controllers [HCRP91, Cas01], and digital signal processing [LGGLBLM91]. Different specialisations have developed over time. Esterel is used, predominantly, for circuit [Ber92] and system [Tec05] design. Lustre is used in the design of embedded controllers, as a compilation target for other modelling languages [CCM+03, MH96], and for research into dataflow programming [CP95]. Signal is the focus of ongoing research into constraint-oriented specification [GTL03]. New synchronous languages have since been developed, including Reactive C [Bou91], which is essentially a restricted version of Esterel that is easier to implement in software, and Argos [MR01], which is a simple graphical formalism that borrows from both Communicating Sequential Processes (CSP) and Statecharts.

While the applications, design methods, and details of the domains addressed by the various languages are quite different, the actual implementations of controllers share the same characteristics, basic structure, and underlying formal model.

The defining characteristic of such controllers is that they loop continually and maintain an ongoing interaction that is driven by their environment. These properties define the class of *reactive systems* [HP85, MP92], by contrast with *transformational systems*, where termination is desired and interaction is essentially limited to taking inputs initially and producing outputs at conclusion, and *interactive systems*, where the system determines the speed and nature of interaction with the environment [Ber00a, §2.1]. In truth, real systems rarely fall neatly into one or other of these categories [Ber00a, §2.1].

2.4.1.2 Execution schemes

Typical embedded controller programs have, at least in principle, one of two basic structures [BCE+03]:

sample-driven:

```
for each clock tick do
  sample inputs
  compute outputs
  update memory
```

event-driven:

```
for each input event do
  compute outputs
  update memory
```

The two differ in the means of triggering. A sample-driven program is triggered at regular intervals. The triggering may be implemented, for example, by timer interrupts, sleep commands, a hardware clock, or a periodic scheduler. An event-driven program executes in response to events. The triggering may be implemented, for example, by interrupts, polling,[17] function calls, or a queue-driven scheduler. The sample-driven mode is natural for clocked sequential circuits and also for control domain applications [Cas01]. The event-driven scheme is more general, since periodic clock ticks are but one triggering input. It is also harder to realise accurately in practice.

2.4.1.3 Mathematical models

Embedded controllers that follow either of the execution schemes can be formalized as Deterministic Finite Automata (DFA). Both adjectives are important. Both have practical and theoretical ramifications.

[17] Effectively a less structured form of sample-driven behaviour.

Determinism entails that system behaviours are, at least in principle, readily reproducible, which simplifies testing and fault-analysis [BS91, Ber00a]. Deterministic systems can be given an adequate semantics in terms of finite traces [Hoa85, §1], avoiding the need for more complicated theories [Ber00a, §2.3].

A finite program can run in a fixed amount of memory, which is an important consideration because embedded controllers usually have limited memory and no swap files. It is, furthermore, an important correctness concern because systems that use unbounded amounts of memory can fail due to stack overflows or failed heap allocations, and their response times are difficult or impossible to predict. Finiteness is ensured in synchronous languages by carefully designed programming structures, as in Esterel, and by clock calculi, as in Lustre [Cas92]. In principle, finite systems can be verified automatically with basic model-checking techniques [Hal93, Part III][Bou97]. In practice, because variables are used, abstraction and more advanced analysis techniques are often necessary. Synchronous languages attend to this issue too. In Esterel, for instance, control flow and data handling are explicitly separated in the grammar of statements and expressions making it easier to form an abstraction for verification.

Systems can be designed directly using DFA, usually in the form of Mealy or Moore machines. The event-driven execution scheme corresponds closely with Mealy machines where *input event* becomes a transition guard, *output event* a transition output, and *update memory* a state change. Both Esterel and Argos can be seen, at a minimum, as a more structured way to describe such machines. Although dataflow programs are not expressed directly in terms of states and transitions, DFA are used in their compilation to efficient executables [HCRP91][Hal93, §6.2.3].

The direct link with mathematical models, like DFA, is central to the synchronous approach [BB91]. In contrast to much related work, the models do not result from *post hoc* formalisation of a language, but rather integral development with a language. They aim to be mathematical as well as formal [Ber00a, §6.1.1].

Even though reactive programming languages are often formalised as Mealy machines, the interplay of instantaneous actions and those with durations that occurs in practice is usually subtle and a satisfactory resolution is perhaps yet to be found. There are various approaches though, amongst them Statecharts [Har87, MMP91], with its activities, variants like Stateflow [Mat04], with `during` actions, the Communicating Reactive Processes (CRP) extension to Esterel [BRS93], and formalisms like hybrid automata [MP93]. This issue is considered again in Chapters 3 and 6.

2.4.1.4 Fundamental principles

Two principles underlie all purely synchronous languages. The first comes of the observation that the two execution schemes both give rise to a totally-ordered sequence of loop iterations, termed *reactions*:

> **The temporal behaviour of a system can be reduced to totally-ordered sequences of discrete reactions.**

The clarity of this observation is important, but more remarkable are the ramifications of adhering to it resolutely. First, it implies a certain orderliness and non-interruptibility that is absent in many other types of systems. For instance, with interrupt-driven routines or preemptive schedulers there is the possibility that any single thread of behaviour could be temporarily displaced by a higher-priority one. Second, the intervals between reactions are ignored. This abstraction from continuous time is the essence of a discrete system, but most other types of software systems have finer granularities. In the synchronous approach, delays, time-stamps, and time-outs are not expressed in real-time, or its approximation by a low-level clock, but rather in terms of relative order and number of reactions. The second principle is related and distinctive:

> **A reaction is computed instantaneously.**

To argue that this is impossible is to miss the point. It is an idealisation or abstraction made with the intent of simplifying system design, in the same manner as classical physics suffices for many tasks despite its neglect of relativistic effects [Ber00a,

§6][Ber00b, §3]. The principle underlies synchronous circuits and is shared by most related formal models: in automata, including timed and hybrid variants, transitions are instantaneous and time only passes in states, in process algebra, actions are instantaneous.[18] It is also adopted to some degree by Statecharts [Har87, §8] and similar approaches. Of course, computation does take time, so in practice the so called *synchrony hypothesis* is only justified if a *system reacts rapidly enough to perceive all external events in suitable order* [Hal93, p. 6], or, similarly, if reactions can be computed quickly enough relative to the frequency of input events.

2.4.1.5 Pure synchrony

Assuming away execution times gives a simple model of external behaviour, but what does it really mean to react instantaneously? The synchronous approach is defined by the asking and answering of this question. It means that outputs occur synchronously with inputs, and further that all intermediate internal actions and communications also occur synchronously and in zero time [BB91]. While the inputs and outputs of Mealy machines also occur together on instantaneous transitions, Mealy machines have no internal structure. Synchronous programs, however, are constructed from interacting components.

The central philosophical problem is whether internal events can be at once simultaneous and causally ordered.

In Argos, they are simply simultaneous. Whilst communication between components can usually be interpreted in causal terms, for example, input a causes event b which triggers output c, the potential transitions from active states actually yield a set of constraints to be solved and the events a, b, and c occur simultaneously solely on the basis of constraint satisfaction. Programs whose constraints cannot be satisfied are rejected since they do not specify a response to an input—reactive programs cannot restrict the environment and thus must be input-enabled.[19] Programs whose constraints have multiple solutions are also rejected since they are not deterministic.

The *logical semantics* of Esterel is similar but the situation is more intricate due to variables and instantaneous control structures. Consistent signal valuations sometimes run contrary to intuitions on control flow and computing them is unnatural and potentially expensive. Thus, a concept termed *constructive causality* has been developed [Ber99] to respect the causes and effects inherent in imperative control structures and to allow a more natural, monotonic calculation of signal values [BS00]. It also turns out that constructive causality gives a model that corresponds neatly with the important class of delay-insensitive digital circuits [Ber99].

Constructive causality is, arguably, a compromise between the original conception of synchrony with only one causal order of events: the total order of reactions; and the micro-step approach of other languages like Statecharts, where there are two orders: between reactions and within a reaction. Most formalisms, in fact, take the latter approach. For instance, in timed and hybrid automata the order of actions at a single instant of time is significant and can be formalised as two-layered 'super dense' time [MP93]. There is, in the *constructive semantics* of Esterel, an acceptance of the ordering of computation steps within a reaction, an order intended by the programmer and expressed in the program structure, but nevertheless also a desire to treat internal communications as simultaneous, which accords well with the original notion of broadcast communication.

2.4.1.6 Communication

In both Argos and Esterel, components communicate through *signals*, which are named broadcast communication channels. Every signal has a single, consistent status at a reaction, either present or absent. The status of (system) input signals is set by the environment before a reaction is computed. The status of internal and output signals—the

[18] Esterel is sometimes credited as an influence on timed process algebras [HR95]. Furthermore, there are similarities between the assumptions of synchrony and of maximal progress (§2.2.3.3), since in both all internal actions must occur instantaneously and immediately.

[19] Input-enabledness is, in the synchronous language literature, sometimes termed 'reactivity'.

former are lexically scoped and the latter are visible to the environment—is computed during a reaction. In Argos and the logical semantics of Esterel, an assignment of statuses to signals is found as the solution of a set of transition constraints. The constructive semantics of Esterel is closer to operational intuition. At the beginning of a reaction, non-input signals have an indeterminate status, written $\perp$. They are marked present only when a corresponding output action is executed, termed an *emission*. They are only marked absent if and when it can be determined that no such action can occur in the reaction.

Two consequences of the signalling mechanism are especially unique. For one, after a signal has been set additional emissions in the same reaction do not affect its status. In many languages, for instance Statecharts, several identically-named events can occur at an instant. They are distinguished by name and relative order. This is easier to implement in software [HLN+88] and its formalisation is relatively straightforward [HPSS87], but there are drawbacks. Notably, the relation between states used in the expression of a program and states in the underlying model becomes more complicated and it is more difficult to bound the number of computation steps in a reaction.

The second consequence is that triggers and conditions within a program can refer to the absence of an event relative to a reaction. Far from being an obscurity, referring to absence is a fundamental way to specify priority. For instance in Structural Operational Semantics (SOS), the possibility of a lower priority transition may depend on the impossibility of a higher priority one. Furthermore, signal absence is integral to the meaning of certain control structures, notably suspension and preemption, where components can only execute when certain events do not occur. Incorporating absence in full generality gives more opportunity to reduce a language to a *small set of primitive statements* [BMR83]. But it also makes distributed implementation and desynchronization more challenging, and there is some conceptual discord with the event-driven execution scheme.

The complications of computing signal absence in Esterel, particularly in software, have led to other proposals. Most notable is the approach taken by Reactive C [Bou91] and related languages [BS96, Puc98]. There, absence is only detectable at the end of a reaction; further computation and state change may occur, but further signal emissions may not. For instance, in a conditional statement that tests a single signal, further outputs may occur in the branch taken when the signal is present, but not in the other branch. This approach can be seen as another response to the problems that motivated the development of the constructive semantics of Esterel. The result is simpler to implement in software but less general and also less natural for circuit implementations.

Besides idempotence of emission and perceivable absence, synchronous broadcasts have several other interesting characteristics, namely: non-blocking sends, persistence and global visibility, contrasts with queued communications, and advantages for program construction and verification.

Senders can always emit signals. They are thus isolated by default from the behaviour of other components.[20] The potential drawbacks of not knowing whether a communication has been received are mitigated by global synchronization and the possibility of so called *instantaneous dialogues* [MR01, §3.2.2.7] within a reaction.

Synchronous broadcast communications are persistent within a reaction and visible at all components with consistent status and value; as if all components were in the same room speaking to, listening to, and overhearing one another [Ber89b]. Persistence and consistency together ensure that communications are not lost due to the internal ordering of a reaction.

As signal statuses are reset between reactions, components can miss communications if they are not in a receptive state a signal is emitted. This would not happen were communications queued, but queues have other drawbacks. Unbounded queues introduce the possibility of indeterminate delays between message sending and receipt, and of unbounded memory use. Bounded queues, when full, require either the loss of

[20] A similar design principle is embodied in the *temporal firewalls* of the Time Triggered Architecture (TTA) [KB03].

communications or blocking of the sender. Nevertheless, queuing is sometimes appropriate and queues can be expressed in synchronous languages [Jef93, KRSW98, SA01]. Furthermore, doing so explicitly has three advantages: the bounding is explicit, the timing behaviour is clear, and exceptional cases can be handled as most appropriate for a specific application.

One practical advantage of global visibility, together with sender isolation and synchronous concurrency, is that systems are easily extended without inadvertently altering their existing behaviour. New components can be added in parallel to 'listen' and act on system activity without interfering (providing they do not emit existing signals).[21] This capability can be exploited for verification by encoding a safety property to be checked as a *synchronous observer* [HLR93, HR99] that emits an error signal if the property is violated. Observers can either be expressed directly in the same programming language or translated into it from a specification language like temporal logic [JPO95]. A model checker can guarantee that the error signal is never emitted, and hence that the property holds of all reachable states, or it can provide a counter-example trace. In Chapter 4, similar techniques are applied to a non-synchronous modelling language, which has both broadcast and paired communication mechanisms.

2.4.1.7 Concurrency

The concurrent components of a synchronous program operate in lockstep. This approach has similarities with the synchronous model of distributed systems [Lyn96, §2][Tel00, §12] and with synchronous process algebras. Synchronous concurrency differs from the interleaving model common to many software systems [Dij68], although it can be implemented by interleaving component actions. Ultimately, it is closer to the type of concurrency found in clocked digital circuits.

A parallel can be drawn between the sequences of discrete reactions in synchronous programs and the sequences of discrete *rounds* in synchronous distributed systems. A round has two distinct phases: communication and computation. All components have first the opportunity to send and receive messages, and then each may update its local state before the next round begins. The two models have, by virtue of this similarity, several properties in common. All components are synchronized to a global and totally-ordered notion of time. Each component has regular opportunities to make progress, and thus fairness is less of an issue. Behaviour is not affected by the order of message receipt and transmission. And absence is significant, for example, for fault detection in synchronous distributed systems [Tel00, §15].

But synchronous languages differ from synchronous distributed systems in the treatment of communication between concurrent components. Communications in the former are broadcast whereas in the latter they are usually directed. More significantly, while reactions and communications within reactions are instantaneous, rounds must take time: in a distributed system *message transmission delay is not negligible compared to the time between events in a single process* [Lam77]. Moreover, communication and response within a reaction are not so rigidly structured as they are in a round, rather the two are intermingled, particularly during instantaneous dialogues.[22]

There are also similarities between synchronous languages and synchronous process algebras. The Synchronous Calculus of Communicating Systems (SCCS) [Mil83] is a process algebra distinguished by having an operator for synchronous concurrency, rather than one defined in terms of interleaving [Hoa78, Mil89]. All concurrent components participate in a system transition, their individual actions are combined by a product operator and unwanted actions are removed by a restriction operator. The instantaneous dialogues characteristic of synchronous programs can be mimicked using such operators; Argos, in fact, takes this very approach. Although there is no explicit operator for asynchronous concurrency in SCCS, asynchronous processes can still be expressed [Mil83, §7]. Similarly, asynchronous behaviours can also be expressed in synchronous languages [BS01, HB02].

[21] Similarly to the 'publish-subscribe' or 'plugin' mechanisms of some component frameworks.

[22] Similar cooperations could be achieved in synchronous distributed systems, but components would have to share many more details of their individual states and control structures than would be natural.

In contrast, operators for asynchronous concurrency do not combine the actions of components in the same way. Instead they interleave individual actions, usually in arbitrary, that is non-deterministic, order. Varying degrees of synchronization through communication are possible but usually only through pairs of complementary actions.[23] The intermingling of component actions within a synchronous reaction can be implemented by interleaving, using varying degrees of static [WBC+00, Edw00] and dynamic [EL03] scheduling. While the interleaving may be non-deterministic, the results, in terms of signal valuations and state change, are still determinate [BS01].

Although synchronous languages were originally developed for programming real-time software [BMR83], the same notion of concurrency with instantaneous communication is also a good model for synchronous digital circuits, which are defined against the regular pulses of a global clock. The voltage levels on the interconnecting wires of circuits correspond to the logical values of signals and, in well-behaved circuits, they too converge to a consistent state. For these reasons, synchronous programs that have no explicit data manipulations are readily compiled into circuits [Ber92] and there is a close connection between well-behaved Esterel programs and well-behaved cyclic circuits [Ber99, §4.4]. The circuit form can be realised directly or simulated in software. Reaction computations in hardware are truly concurrent rather than interleaved.

For all its advantages, the synchronous model of concurrency has some limitations due to the possibility of signal absence and the rigidity that comes with predictability.

The tight binding of components within a reaction and the need to treat signal absence make distribution challenging, although techniques do exist [CGP94]. There is a fundamental conflict between the ideal of delay-free communication in synchronous programs and the communication delays that define distributed systems. Yet the simplicity of reasoning about synchronous specifications and the inherent appeal of adapting them to situations where causality is partially ordered have motivated recent research [BCLG99, BCLG+02].

Synchronous programs are predictable but rigid. The two attributes are concomitant: for example, a group meeting for lunch each day at the same place and time is predictable and need only make arrangements at the outset, whereas choosing a different place and time everyday involves more communication, and possibly unbounded time, but gives more adaptability, flexibility, and variety. Compiling away concurrency and communication means that tasks cannot simply be added at run-time, as is possible in general-purpose Operating Systems (OSs). This is appropriate for hard real-time systems, which must account for worst-case scenarios, but too expensive for other types of system which are expected to be more dynamic. Synchronous languages are optimised for expressing control structures and calculations, but not for specifying the kind of data structure manipulations that are also required in systems software and other types of embedded or low-level programming. There have been attempts to make the synchronous style more applicable to systems with such features [BCG+97, MM97], but they have not proved especially effective.

2.4.2 Argos

Argos is a set of operators for combining Boolean Mealy Machines (BMMs) [MR01]. It is similar to Statecharts in motivation and syntax but much simpler in definition; most notably, inter-level transitions and history junctions are forbidden. The design of Argos also blends concepts from the classical synchronous languages, particularly Esterel, and process algebra, particularly Calculus of Communicating Systems (CCS).

There are four Argos operators for combining basic BMMs: *parallel composition* for the synchronous product of two machines, *encapsulation* for enforcing synchronization and hiding signals, *inhibition* for restricting participation in reactions, and *refinement* for introducing hierarchical structure. The operators, with standard [MR01] graphical and mathematical notations, are presented in §§2.4.2.1–2.4.2.5. Programs are sometimes also encoded in the textual syntax described in Appendix D.

[23] Although the encapsulation function of Algebra of Communicating Processes (ACP) is more flexible, see §A.4.

The semantics of Argos are defined recursively over its syntax by a function which maps each syntactic operator to a corresponding semantic operator over BMMs [MR01, §4.3]. The semantic operators are defined in §§2.4.2.1–2.4.2.5, following the standard presentation [MR01], rather than the SOS version [Mar92], with only minor notational differences. The semantic operators are closed over BMMs, but they do not, in general, preserve determinism or input-enabledness. Programs that map to non-deterministic BMMs or to those that are not input-enabled are regarded as incorrect. The semantic function is slightly more complicated as a result since it is undefined when any subprogram is undefined.

It is reasonable to consider some programs as incorrect [MR01, §1.5]. In statically-typed languages, for instance, programs are rejected if they are not type correct. Checking for determinism and input-enabledness in Argos, however, is more expensive. In principle it means analyzing the reachable state space and considering all input assignments, which is only practical for the smallest programs. In practice, some form of approximate detection must be adopted and certain valid programs will be rejected; again analogously with statically-typed languages.

The notion of equivalence in Argos is bisimulation of correct programs [MR01, Definition 8]. The semantics are termed compositional [Mar92], because this equivalence is a congruence for all the operators [MR01]. In practice, this fact justifies reasoning about the behaviour of a program in terms of the individual behaviours of its parts.

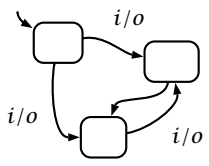
Although valued extensions have been sketched [MR01, §4.5], Argos programs are, essentially, free of variables and limited to pure Boolean signals. These limitations enable a concise notation and a relatively simple semantic definition, but make it difficult to express sophisticated behaviours and to program real systems. Argos is nevertheless suitable for expressing a certain class of reactive controllers, as is demonstrated in Chapter 3.

There are good reasons for studying Argos, despite the fact that languages like Esterel, particularly in the graphical form of Safe State Machines (SSM) [And95], are a better choice for programming most systems since they have variables, data structures, host language function calls, and industrial-strength tool support. For one, Argos treats many of the essential ideas of Statecharts in an uncomplicated way, and, in doing so, provides a means of expressing Mealy machine models without recourse to a weightier language like Esterel. It demonstrates, in a simple setting, many of the fundamental concepts and issues in the design of imperative synchronous languages. The syntax of Argos, though, is too simple to demonstrate the constructive causality and modern compilation techniques of Esterel, where subtle interpretations are read into the more sophisticated, domain-specific constructs.

Another advantage of Argos is that a relatively simple and efficient means of compilation exists [MH96], whereby programs are transformed into dataflow streams; essentially those of Lustre. A separate stream is introduced for every state, local signal, and output. The streams can also be computed, analysed, and converted to a sequential program directly using Binary Decision Diagrams (BDDs).

2.4.2.1 Boolean Mealy Machines

notation:
(S, s_0, I, O, T)



BMMs are the basis of the Argos language. They are depicted graphically as state transition diagrams and in the mathematical notation as a tuple, whose formal definition [MR01, Definition 1] follows. Semantically, the tuples stand for themselves. There is a direct relationship between diagrams and tuples: states are drawn as rectangles; the initial state is marked by an otherwise unconnected arrow in its upper-left corner; and transitions are drawn between pairs of states and labelled by input and output expressions separated by a slash. States may be labelled with names. But neither names, nor the relative positions of graphical elements have any semantic significance. The four operators of Argos act over BMMs.

Definition 2.4.1
A Boolean Mealy Machine (BMM) over disjoint finite sets I (input signals) and O (output signals) is a deterministic LTS over $A = 2^{I \cup O}$ and $P = \emptyset$ where S is finite and $S_0 = \{s_0\}$. A BMM will be written as a tuple (S, s_0, I, O, T), where $T \subseteq S \times 2^I \times 2^O \times S$. Each $(s, I', O', s') \in T$ denotes a transition $s \xrightarrow{a} s'$, where $a = I' \cup O'$. ■

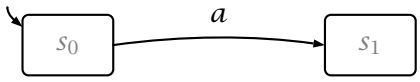
It is usual, in Argos, to specify subsets of input signals by treating the elements of I as variables and combining them into Boolean expressions with negation, written by overlining them ($\overline{a}$), and conjunction, written as a dot ($a \cdot b$). The set of Boolean expressions with variables in I will be written $\mathrm{A}B(I)$ [MR01, Definition 1]. A transition guard φ/O', with $\varphi \in \mathrm{A}B(I)$, then denotes a set of transitions on the actions in:

$$\{I' \cup O' \mid I' \subseteq I \wedge I' \vDash \varphi\},$$

where $I' \vDash \varphi$ means that φ is satisfied by assigning true to all variables in I' and false to those in $I \setminus I'$. Transitions are sometimes also labelled with lists of expressions in $\mathrm{A}B(I)$, delimited by plus symbols ($\varphi_1 + \cdots + \varphi_n$), which is a short-hand for multiple transitions, one for each φ_i, that share the same source state, destination state, and output set. An absent input label is treated as shorthand for $\varphi =$ true. An absent output label is treated as shorthand for $O' = \emptyset$ and the dividing slash is omitted.

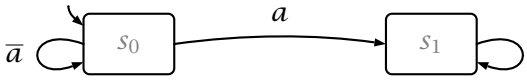
State names are ignored in the semantics of Argos because they limit compositionality [MR01]. Programs with the same external behaviour should be interchangeable regardless of their internal structure. The in() predicate of Statecharts is, for this reason, not made available but its effect can be recovered by synchronizing on local signals.

BMMs in Argos must be deterministic and input-enabled. That is, in every state,[24] for any valuation of inputs, exactly one transition must be possible. This causes a minor problem of interpretation. For example, consider the BMM diagram:



(Figure 2.10)

The corresponding BMM has the form $(\{s_0, s_1\}, s_0, I, O, T)$. The values of I and O are not made explicit in the diagram,[25] beyond that $a \in I$, but they are not important at present. There are two relevant possibilities for T. The *literal interpretation* is $T_l = \{(s_0, a, \emptyset, s_1)\}$, but this BMM would not be input-enabled since it does not specify what should happen in s_0 in reactions where a is absent. Instead, it is usual to *omit the transitions from a state to itself, if they do not emit signals* [MR01, §2.2], giving the *self-loop interpretation*:



(Figure 2.11)

where $T_s = T_l \cup \{(s_0, \neg a, \emptyset, s_0), (s_1, \mathsf{true}, \emptyset, s_1)\}$. These implicit self-loop transitions can be inferred for each state from the set of explicit outgoing transitions. In particular, the exact value of I is irrelevant. There is a problem, however, when it comes to refinement, because, as will become clear in §2.4.2.5, an explicit transition from a state terminates any machine that refines that state, whereas an implicit transition should not. Furthermore, the explicit transition may be a self-loop, which would restart a refining machine.[26] The precise policy could be stated: assume the literal interpretation for refined BMMs and the self-loop interpretation otherwise. And thus, *the machine that serves as the* (refined) *controller is not necessarily reactive* [MR01, §3.2.4]. This policy is implicitly adopted in the compilation to dataflows [MH96].

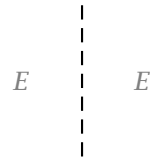
In the following subsections the class of BMMs and a special value $\perp$ representing an incorrect program is written $M_\perp$. The subset of deterministic BMMs is written M^d.

[24] It is usual [MR01, Definition 3] to consider all states, not just those reachable from the initial state.
[25] They are sometimes given explicitly [MR01, Figure 1].
[26] In some Statecharts-like languages, Stateflow for example, there is a distinction between 'internal' and 'external' transitions.

2.4.2.2 Operator: parallel composition

notation:
$E \parallel E$

operator:
$\times : M_\perp \times M_\perp \Rightarrow M_\perp$



The parallel operator joins the behaviours of two Argos programs which then act independently, but, true to the synchronous paradigm, simultaneously. It is expressed in the graphical notation by drawing two programs next to one another with a dashed line between them, and in the mathematical notation by placing a $\parallel$ symbol between two expressions.

The meaning of parallel composition is given in terms of synchronous composition, written $\times$, which maps two BMMs to a third.

Definition 2.4.2
The *synchronous composition* of two BMMs $M_1 \times M_2$ is defined

$$\begin{pmatrix} S_1, \\ s_0^1, \\ I_1, \\ O_1, \\ T_1 \end{pmatrix} \times \begin{pmatrix} S_2, \\ s_0^2, \\ I_2, \\ O_2, \\ T_2 \end{pmatrix} = \begin{pmatrix} S_1 \times S_2, \\ (s_0^1, s_0^2) \\ I_1 \cup I_2, \\ O_1 \cup O_2, \\ T' \end{pmatrix}$$

where T' is the smallest set such that

$$\begin{pmatrix} s_1, \\ m_1, \\ o_1, \\ s_1' \end{pmatrix} \in T_1 \ \wedge \ \begin{pmatrix} s_2, \\ m_2, \\ o_2, \\ s_2' \end{pmatrix} \in T_2 \Longrightarrow \begin{pmatrix} (s_1, s_2), \\ m_1 \wedge m_2, \\ o_1 \cup o_2, \\ (s_1', s_2') \end{pmatrix} \in T'$$

■

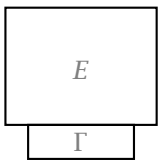
Both parallel components are synchronized in the resulting BMM, acting simultaneously in each of its reactions. The resulting set of states is the Cartesian product of the original sets. Such definitions are typical of both synchronous and asynchronous process models. The characteristic feature of synchronous composition is that transitions are combined with conjunction. Between any two composite states all possible pairings of transitions from one component with those of the other are allowed. The new guards are conjunctions of the component guards, and the new outputs are the union of the component outputs.

Synchronous composition alone does not enforce consistency in local signal valuations and thus does not completely resolve communications between components. For instance the conjunction of transitions $(s_0, \neg a, \varnothing, s_0')$ and $(s_1, \mathsf{true}, \{a\}, s_1')$ gives the transition $((s_0, s_1), \neg a, \{a\}, (s_0', s_1'))$, which is self-contradictory: the transition is only triggered when a is absent but it itself emits a. Signals that occur in both input guards and as outputs must, at some point, be declared local using the encapsulation operator, described next in §2.4.2.3, which removes such inconsistencies. Argos is, in this regard, similar to CCS where the unsynchronized transitions of a parallel composition can later be removed with a signal hiding operator.

2.4.2.3 Operator: encapsulation

notation:
$\overline{E^\Gamma}$

operator:
$\setminus : M_\perp \times \mathcal{P}(A) \Rightarrow M_\perp$



The encapsulation operator makes a set of signals local to a subprogram. Local signals are used for broadcast communication within the scope of the operator. They are not visible outside the scope. Encapsulation is represented graphically by a rectangle drawn around a subprogram with a smaller rectangle underneath in which the signals to hide are written. It is written mathematically by drawing a line above the expression in scope, with the signals to be hidden next to it in superscript.

The corresponding semantic operator is also called encapsulation.

Definition 2.4.3
The *encapsulation* of a BMM $M = (S, s_0, I, O, T)$ by a set of signals Γ, written $M \setminus \Gamma$, is the BMM $(S, s_0, I \setminus \Gamma, O \setminus \Gamma, T')$ where T' is the smallest set such that[27]

$$(s, m, o, s') \in T \wedge \forall x \in \Gamma.\, (((x \wedge m) = m \implies x \in o) \ \wedge\ ((\neg x \wedge m) = m \implies x \notin o)) \implies (s, \exists \Gamma.\, m, o \setminus \Gamma, s') \in T' \quad \blacksquare$$

The matrix of the second premise in Definition 2.4.3 can be replaced by the biconditional expression $((x \wedge m) = m) \iff x \in o$ only if guards are restricted to complete monomials, otherwise there are actually three possibilities for each $x \in \Gamma$: x is positive in m, x is negative in m, or x does not occur in m.

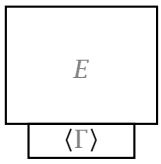
Encapsulation does not preserve input-enabledness or determinism. An example is shown in Figure 2.12a, where there are two deterministic BMMs M_1 and M_2 in parallel, and the local signals a and b are encapsulated: $(M_1 \parallel M_2) \setminus \{a, b\}$. The synchronous product before the removal of encapsulated signals is shown in Figure 2.12b. It is deterministic; one transition emits signals a and b, the other does not. The result after encapsulation is shown in Figure 2.12c. It is not deterministic; in state (A_1, B_1) there are two possible transitions when the external signal i is present. Each transition is consistent in itself, depending either on the absence of a and b and not emitting either, or requiring both and also emitting them. Examples for a loss of input-enabledness after encapsulation are also readily constructed [MR01, 3.2.2.4].

The example of Figure 2.12 also demonstrates the view of synchrony taken in Argos, that events within a reaction are strictly simultaneous. In particular, it is not possible to argue that the guards of a transition are evaluated before its outputs are emitted, and thus that only the transition with guard $i \wedge \overline{a} \wedge \overline{b}$ could occur.

2.4.2.4 Operator: inhibition

notation:
$\overline{E^{\langle \gamma \rangle}}$

operator:
whennot : $M_\perp \times A \Rightarrow M_\perp$



The inhibition operator prevents a subprogram from acting in reactions where a given signal is present.[28] Its graphical and mathematical representations are the same as those of the encapsulation operator except that only a single signal may be given and it must be written between angle braces.

The corresponding semantic operator is also called inhibition.

Definition 2.4.4
The *inhibition* of a BMM $M = (S, s_0, I, O, T)$ by a signal $\gamma \notin I$, written M whennot γ, is the BMM $(S, s_0, I \cup \{\gamma\}, O, T')$ where T' is the smallest set such that

$$(s, m, o, s') \in T \implies \begin{array}{ll} & (s, m \wedge \neg\gamma, o, s') \in T' \\ \wedge & (s, m \wedge \gamma, \varnothing, s) \in T' \end{array} \quad \blacksquare$$

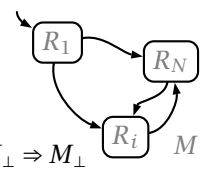
The only possible transition when the inhibiting signal is present is a self-loop without any outputs. When the inhibiting signal is absent the original responses are possible.

The use of inhibition in features related to refinement, namely enter-by-history, strong abortion, and inter-level outgoing transitions, is discussed in §§2.4.2.5–2.4.2.7.

2.4.2.5 Operator: refinement

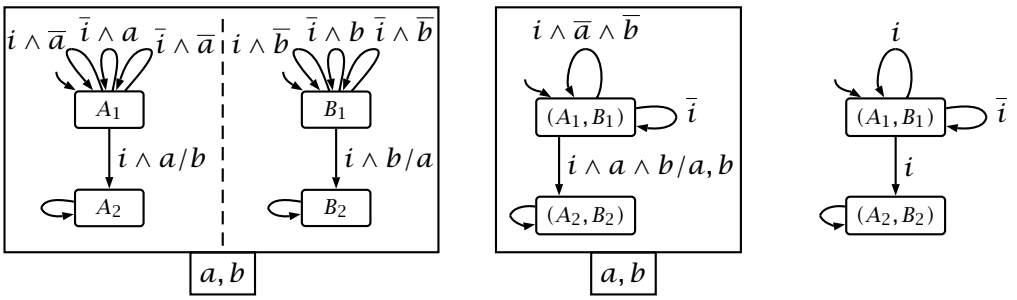
notation:
$\mathbf{R}_M(R_1, \ldots, R_n)$
$R = E \mid \mathsf{NIL}$

operator:
$\rhd : M^d \times M_\perp \times \cdots \times M_\perp \Rightarrow M_\perp$



The refinement operator, like the parallel operator, combines BMMs, which may then act simultaneously and intercommunicate. Unlike parallel composition, however, refinement introduces a hierarchical relationship between BMMs. A controlling BMM determines the lifetimes of subordinate BMMs. Compared to Statecharts, refinement in Argos is simpler but less flexible.

[27] The equivalence between expressions here is equivalence under all assignments to free variables.
[28] Also called the 'inhibiting operator' [MR01].



(a) Basic Program **(b)** After composition **(c)** After encapsulation

Figure 2.12: An encapsulation that gives a non-deterministic result [MR01, §3.2.2.4]

There is not really a single corresponding semantic operator, but rather a family of semantic operators, one for each non-zero natural number.

Definition 2.4.5
The *refinement* of a BMM $M = (S, s_0, I, O, T)$, where $S = \{s_0, \ldots, s_n\}$, by a set of refining BMMs $\mathcal{M} = \left\{(S^i, s_0^i, I^i, O^i, T^i) \mid 0 \le i \le n\right\}$ is $M \rhd \mathcal{M} = (S', s_0^0, I \cup \bigcup I^i, O \cup \bigcup O^i, T')$ where

$$S' = \dot{\bigcup}_{i=0}^{n} \left\{s_j^i \mid 0 \le j \le n_i\right\}$$

and T' is the smallest set satisfying the two rules

$$\left(\begin{array}{cc} & (s_k, m, o, s_{k'}) \in T \\ \wedge & \left(s_l^k, m', o', s_{l'}^k\right) \in T^k \end{array} \right) \Longrightarrow \left(s_l^k, m \wedge m', o \cup o', s_0^{k'}\right) \in T' \qquad \text{(outer)}$$

$$(s_l^k, m', o', s_{l'}^k) \in T^k \Longrightarrow \left(s_l^k, m' \wedge \bigwedge_{(s_k, m, \cdot, \cdot) \in T} \neg m, o', s_{l'}^k\right) \in T' \qquad \text{(inner)}$$

■

The refinement comprises the states of the refining BMMs, but its transitions account for the priority of the controlling BMM and the possibility of abortion.

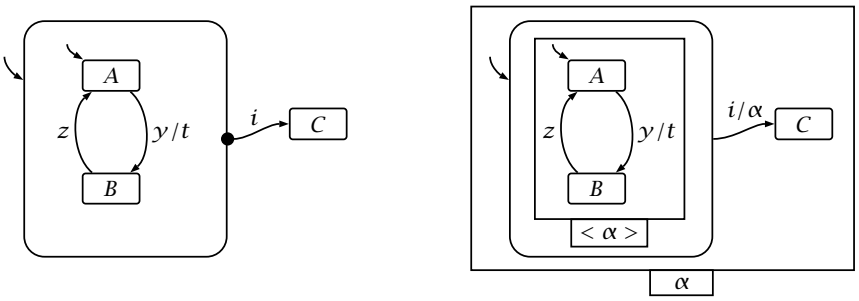
The *outer* rule preserves the transitions of the controlling BMM and defines their effect on the refining BMMs. The BMM that refines the source state is allowed to act in the transition that terminates it—so called *weak abortion* [Ber00a, §3.6]—in fact, because both transitions occur synchronously, a refining BMM can terminate itself by emitting signals that trigger a transition in its parent. The BMM that refines the destination state is started in its initial state and it may not act until the next reaction. This asymmetry between actions of the terminated machine and actions of the initialized machine sidesteps the complexities of *process schizophrenia* [MR01, §3.2.4][Ber99, §12][TS05], where the possibility of simultaneous termination and initialization of programs mandates nested signal replication. It also precludes practicable encodings of incoming inter-level transitions and conditional initial states.

The *inner* rule allows the active refining BMM to act when none of the controlling transitions are possible. A conjunction of negated controlling guards expresses the required priority.

The mapping of states to BMMs in Definition 2.4.5 is encoded by consecutively numbering states in the controlling BMM, starting from zero for the initial state. While this is awkward, it does not present any fundamental problems. In the graphical notation, the mapping is expressed by drawing each refining BMM within a state of the controlling BMM. State labels are used in textual encodings; see, for example, Appendix D.

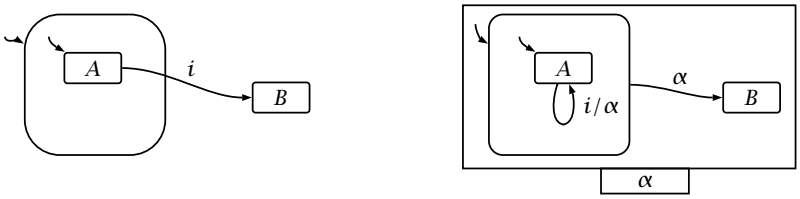
The refinement operator requires a BMM for each controlling state. A place-holder BMM can be used for states that are not refined.

Definition 2.4.6
The *Nil BMM* is defined $\mathsf{NIL} = (\{\mathsf{NIL}\}, \mathsf{NIL}, \varnothing, \varnothing, \{(\mathsf{NIL}, \mathsf{true}, \varnothing, \mathsf{NIL})\})$. ■



(a) Macro notation **(b)** Expansion

Figure 2.13: Expressing strong abortion with inhibition and refinement



(a) Statecharts **(b)** Argos equivalent

Figure 2.14: Inter-level outgoing transitions

Different combinations of the inhibition and refinement operators are able to express enter-by-history, strong abortion, and inter-level outgoing transitions. The last two are described in subsequent sections where special notations are introduced.

Enter-by-history [Har87, §2] is a characteristic feature of Statecharts. A history junction (H) can be added to a refining state machine to indicate that its active state should not be forgotten when it is aborted, but should instead become the initial state when the machine is next activated. Argos does not support enter-by-history, but the same effect can be achieved, with some effort, using local signals, parallel composition, and inhibition. A Statecharts diagram that use enter-by-history is shown in Figure 3.13, and an encoding of it in Argos is given in Figure 3.14.

2.4.2.6 Macro: strong abortion

The parallel, encapsulation, inhibition, and refinement operators are the *core operators* of Argos. They are defined as direct manipulations of BMMs. *Macro notations*, in contrast, are defined as syntactic transformations.

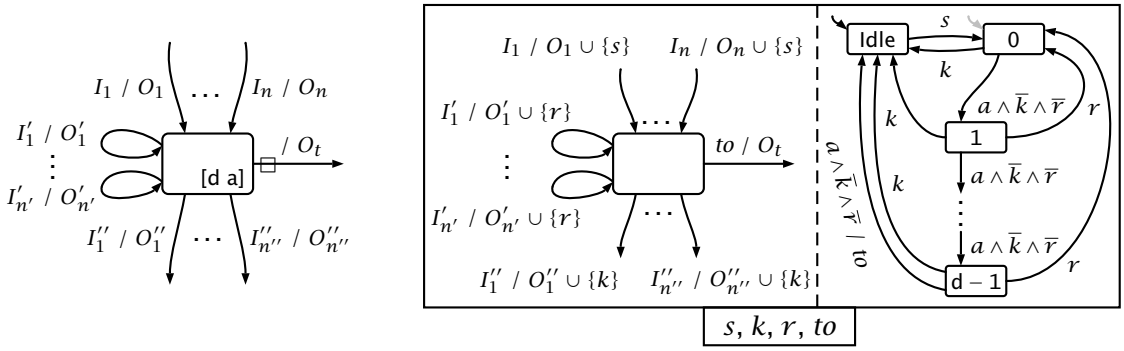
While abortion in Argos is weak by default, it is possible to express *strong abortion*, where the aborted subprogram cannot contribute outputs to the reaction in which it terminates. The macro notation suggested [MR01, §3.3.2] for indicating strong-abortion is a black dot at the base of a transition, as in Figure 2.13a.[29] The meaning of this notation is defined in terms of the primitive operators. The contents of each strongly-aborted state are enclosed in an inhibition operator that is labelled by a fresh local signal, which is added to the outputs of all strongly-aborting transitions sourced at that state, as in Figure 2.13b.

2.4.2.7 Macro: outgoing inter-level transitions

Inter-level transitions are typical of Statecharts and similar languages. They connect states at different levels of the hierarchy. They are either *incoming* from a shallower state to a deeper one, or *outgoing* from a deeper state to a shallower one.

Incoming inter-level transitions are effectively a way to choose the starting states of refining state machines down to and including the destination state. They cannot be

[29] The term 'inhibiting transition' is also used [MR01].



(a) Macro notation **(b)** Expansion in parallel

Figure 2.15: Expansion of the temporized state macro

modelled naturally in Argos because the initial states of refining BMMs are fixed and the refining BMMs themselves do not participate in the initializing reaction.

Outgoing inter-level transitions are effectively a way of terminating controlling state machines up to the level of the destination state. They can be modelled in Argos by replacing the multi-level transition with two new transitions: a self-loop at the source state and a new transition from the ancestor of the source state at the level of the destination state to the destination state itself. The new self-loop transition inherits the guard of the original and emits a fresh local signal that triggers the other new transition. Any other outputs can be emitted by either of the new transitions. For example, the inter-level transition in Figure 2.14a can be replaced by those in Figure 2.14b.

2.4.2.8 Macro: temporized states

The *temporized state* macro is for specifying time-limited states. These states have a label of the form `[d a]`, for example `[5 SEC]`, and a single timeout transition designated by a square box, see Figure 2.15a. Timeout transitions may not have guard expressions, but they may emit outputs. If the temporized state is still active after d occurrences of signal a the timeout transition is triggered.

The expansion of a temporized state to basic operators is shown in Figure 2.15b. Four new local signals are introduced: s for 'start', k for 'kill', r for reset, and *to* for 'timeout'. Fresh names must be chosen, in particular when there are multiple temporized states in a single BMM. A counter automaton is introduced for each temporized state. The s signal is added to the outputs of all transitions from other states into the temporized state; its emission starts the counter automaton. The k signal is added to the outputs of all transitions from the temporized state to other states, except the timeout transition. Its emission resets the counter automaton. The r signal is added to the outputs of all explicit self-loop transitions at the temporized state, except the timeout transition. Its emission restarts the counter automaton. The timeout transition becomes a normal transition that is triggered by the *to* signal which the counter automaton emits if a timeout occurs. The temporized state becomes a normal state.

The construction of Figure 2.15b must be altered if the temporized state is an initial state or if the timeout transition is a self-loop. In the former case, state `0`, rather than `Idle`, is made the initial state. In the latter case, the transition from state `d − 1` that emits *to* goes to state `0` rather than `Idle`. These adjustments are readily combined.

The construction presented here differs slightly from the original [MR01, Figure 9] by accounting for self-loops and initial temporized states. The former must be defined explicitly because self-loops can be addressed in two ways: either resetting the timer, as here, or not. The natural interpretation of self-loops in the original, adding both s and k signals to their outputs, is incorrect because it would leave the counter automaton in the `Idle` state.

An alternative expansion refines the temporized state with the counter automaton. The counter automaton is placed in parallel with any other refining subprograms. The guard of the transition that emits *to* must include the negated disjunction of all

	syntax	**notation**
primitive		
no-op statement	**nothing**	0
signal broadcast	**emit** s	$!s$
unit delay	**pause**	1
signal test	**present** s **then** p **else** q **end**	$s?p,q$
suspension	**suspend** p **when** s	$s \supset p$
sequencing	p ; q	$p;q$
looping	**loop** p **end**	$p*$
concurrency	$p \parallel q$	$p \mid q$
exceptions	**trap** T **in** p **end**	$\{p\}, \uparrow p$
	exit T	k with $k \geq 2$
local signals	**signal** s **in** p **end**	$p \setminus s$
derived		
immediate suspension	**suspend** p **when immediate** s **end**	$s \supset\!\!\!\cdot\, p$
abortion	**abort** p **when** s **end**	$p \gg s$

Table 2.1: Syntax and notation of the main Esterel statements [Ber99, §5]

guards on transitions from the temporized state. As refinement manages the lifetime of subprograms, the `Idle` state and the signals s, k, and r are not required. State `0` is made initial. The transition that emits *to* goes to state `0` rather than to `Idle`. All other transitions to `Idle`, and the state itself, are omitted.

The expansion gives the semantics of temporized states, but in practice, counters are used rather than explicit states.

Temporized states embody the multiform notion of time that is typical of synchronous languages. Alternatively, they can be interpreted in real time [JMO93]. Timeouts are then expressed in the form `[d]`, that is with an implicit reference to time rather than to a signal, and programs are defined semantically in terms of timed graphs rather than BMMs. This possibility is discussed again in Chapter 6 (§6.3.6).

2.4.3 Esterel

Esterel is a larger and more complex language than Argos and, unlike Uppaal, it is not an extension of a simpler underlying formalism. The treatment of time and events in Esterel distinguishes it from algorithmic languages like Ada or C.

Esterel can be viewed from two complementary perspectives. In one sense it is a means of describing DFA from which efficient hardware or software can be generated. The advantage of expressing designs in Esterel, rather than directly, is to avoid duplication [Ber00a, §3.2][Ber00b], and to ensure that small changes to the requirements do not entail large changes to programs [Ber00b, §4.2]. In another sense, Esterel is a domain-specific notation for describing reactive behaviours. There are statements for describing event handling, concurrency, preemption, and suspension. Furthermore, the sequential behaviour of statements is defined tractably and rigorously.

The main Esterel statements are described in §2.4.3.1 through the development of a small example. An overview of the various semantic definitions and compilation techniques is presented in §2.4.3.2. Comprehensive descriptions of the Esterel language [Ber00a][PBEB07, Part I] and its semantics [Ber99][PBEB07, Parts II and III] are available in the literature. The literature also contains several extensions and variations of Esterel, as well as case-studies and applications to diverse domains.

2.4.3.1 Overview

At a glance, most Esterel statements, see the central column of Table 2.1, look like those of any other imperative programming language; and especially to Ada and Pascal whose syntax also derives from Algol. The behaviour of Esterel is, however, quite

```
halt                          =  loop pause end

sustain s                     =  loop emit s; pause end

present s then p end          =  present s then p else nothing end

await s                       =  trap T in
                                   loop
                                     pause;
                                     present s then exit T end
                                   end loop
                                 end

await immediate s             =  trap T in
                                   loop
                                     present s then exit T end;
                                     pause
                                   end loop
                                 end

suspend                       =  suspend
  p                                  present s then pause end; p
when immediate s                 when s

abort                         =  trap T in
  p                                  suspend p when s; exit T
when (immediate) s               ||
                                   await (immediate) s; exit T
                                 end

weak abort                    =  trap T in
  p                                  p; exit T
when (immediate) s               ||
                                   await (immediate) s; exit T
                                 end

loop                          =  loop
  p                                  abort p; halt when s
each s                           end loop

every (immediate) s do        =  await (immediate) s;
  p                              loop p each s
end every

repeat e times                =  trap T in
  p                                  var C : int = e in
end repeat                             loop
                                         if C = 0 then exit T end;
                                         C := C - 1; p
                                       end loop
                                     end var
                                   end trap

await e s                     =  repeat e times await s end repeat
```

Table 2.2: Derived statements of Esterel [Ber00a, BG92]

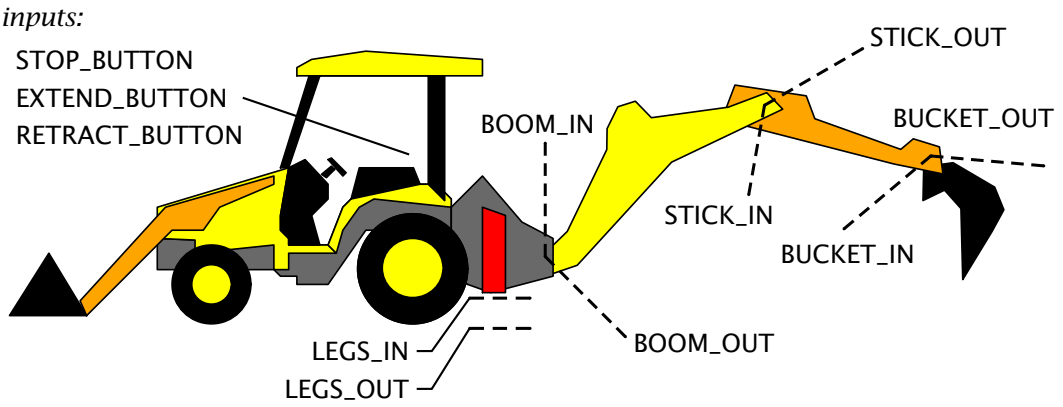


Figure 2.16: Backhoe loader example: input and output signals

```
1  module BACKHOE:
2
3  input BOOM_OUT, BOOM_IN;
4  relation BOOM_OUT # BOOM_IN;
5
6  input STICK_OUT, STICK_IN;
7  relation STICK_OUT # STICK_IN;
8
9  input BUCKET_OUT, BUCKET_IN;
10 relation BUCKET_OUT # BUCKET_IN;
11
12 input LEGS_OUT, LEGS_IN;
13 relation LEGS_OUT # LEGS_IN;
14
15 input STOP_BUTTON, EXTEND_BUTTON, RETRACT_BUTTON;
16
17 output BOOM_PUSH, BOOM_PULL, BOOM_DRIVE;
18 output STICK_PUSH, STICK_PULL, STICK_DRIVE;
19 output BUCKET_PUSH, BUCKET_PULL, BUCKET_DRIVE;
20 output LEGS_RAISE, LEGS_LOWER, LEGS_STOP;
21
22 % Controller implementation:
23 halt
24
25 end module
```

Figure 2.17: Main backhoe loader controller module

different to most other languages. It is perhaps most easily understood through a concrete example. For that reason, the concepts and constructs of Esterel are introduced in this subsection as they are needed to develop a controller for the backhoe loader system shown in Figure 2.16. The subsection begins with an outline of the example.

Backhoe loaders, or just backhoes, are specialised tractors. A backhoe is fitted with a loader unit at front for pushing and carrying, and an extensible arm at rear for digging, lifting, and pushing. The arm has three segments: the *boom*, *stick*, and *bucket*. A driver is able to swivel his seat and coordinate the segments using joysticks. Now, suppose a control unit is added to automate some sequences of movements in response to buttons pressed by the driver. The controller receives input from the eleven sensors shown at the top of Figure 2.16. There are two sensors for each segment: one triggered when the segment is driven against its minimum position, ∗_IN, the other when it is driven against its maximum position, ∗_OUT. Similar sensors exist for a pair of stabilising legs which must be extended before the arm can be moved. In addition, there is a sensor for each of three buttons in the cabin: *stop*, *extend*, and *retract*. The controller can send twelve different commands. There are three commands each for the hydraulic pistons attached to each arm segment. Two of them set the position of an internal valve that determines whether the segment moves toward the maximum position, ∗_PULL, or the minimum position, ∗_PUSH, when driven. The arm only moves when a third control, ∗_DRIVE, is sent continuously, otherwise it is held in position. The legs operate differently. They are pushed downward when engaged to the drive train by a LEGS_EXTEND command and similarly pulled upward by a LEGS_RETRACT command. Either movement can be stopped by a LEGS_STOP command.

An Esterel system has three layers [BG92]:

1. an *interface* that converts between external events and signals, and implements one of the event-driven or sample-driven execution schemes,
2. a *reactive kernel* that executes compiled Esterel programs, and,
3. *data-handling* functions written in an algorithmic language.[30]

For the backhoe controller, an event-driven interface layer is assumed and no data-handling functions are necessary. Only the reactive kernel is relevant.

Esterel programs are structured using *modules* which comprise a name, a list of interface declarations, and a reactive statement. A top-level module for the backhoe controller is presented in Figure 2.17. An **input** signal is declared for each sensor and an **output** signal is declared for each command. They are all pure signals, meaning that they have a *status* at each reaction, either present or absent, but not a *value*. Esterel programs must usually be input-enabled, that is they must usually specify what happens for any combination of input signals in any state. A **relation** declaration relaxes this requirement by either stating that two signals are never present simultaneously, using # as shown in the figure, or that the presence of one signal implies another, using => (but not shown in the figure). Compilers and interpreters use relations between signals for optimisation and to ignore behaviours that would otherwise be illegal. Modules may also contain declarations of types, constants, functions, and procedures [Ber00a, §7.3]. It is also possible to declare **inputoutput** signals that are both emitted within the module and received from other parts of the program. Such signals cannot be inputs from the external environment.

The module template in Figure 2.17 contains a **halt** statement where control flow will stop indefinitely. The analogue in Argos would be a state with self-loop transitions for each input, none of which emit any outputs. The **halt** statement in the template is a placeholder for the real controller program. Using the **nothing** statement instead would have given a program that terminated immediately. Besides their behaviour, there are two other distinctions between **halt** and **nothing** that warrant discussion before returning to the example, namely the differences between *primitive* and *derived* statements, and between *instantaneous* and *non-instantaneous* statements.

Each statement of Esterel can be categorised as either primitive or derived. The semantics of primitive statements are defined explicitly whereas derived statements are

[30] The latest versions of Esterel also provide features for expressing classical computation.

defined syntactically as translations into other statements [BMR83][Ber99, §2.2], as are the macro notations of Argos. This division reduces the effort needed to define and understand the semantics, to develop program transformations and analyses, and to prove properties of the language or programs written in it. The primitive statements, which include **nothing**, are listed in the upper part of Table 2.1. The lower part of that table contains two derived statements that are so convenient for reasoning semantically that they are given their own mathematical notation. These two derived statements and several others are presented with their definitions in Table 2.2. In particular, the **halt** statement is defined as a **pause** statement within a **loop** statement.

Each statement of Esterel can also be categorised as either instantaneous or non-instantaneous.[31] Instantaneous statements begin and end in a single reaction and thus, in principle, their execution takes no time. Execution of a non-instantaneous statement begins in one reaction but may end in a subsequent reaction. Of the primitive statements only **pause** is non-instantaneous. When a control flow reaches a **pause** it stops and, if not otherwise influenced, continues again from that statement in the next reaction. A reaction ends when all control flows have either stopped at **pause** statements or terminated. The **nothing** statement thus begins and ends in a single reaction whereas the **loop** and **pause** of **halt** effect repeated delays from each reaction to the next.

Returning now to the example, suppose pressing the extend button should make the controller to lower its legs fully, and then to lift the three arm segments simultaneously. The first half of this operation is expressible as a sequence of three statements:

```
await EXTEND_BUTTON;
emit LEGS_EXTEND;
await LEGS_OUT.
```

The **await** and **emit** statements are composed by the sequence ';' operator. Sequencing is instantaneous: when the first statement terminates, the second begins immediately. In particular, this means that waiting for three seconds and then waiting for two seconds, **await** 3 SECONDS; **await** 2 SECONDS, is equivalent to waiting for five seconds, **await** 5 SECONDS [BMR83, BC84], which is not true of all programming languages.

Intuitively, execution stops at an **await** statement until the given signal is present. By default, an **await** statement ignores the status of the given signal at the instant in which its execution begins. There is also an **await immediate** statement that terminates instantaneously when the awaited signal is already present. The difference between the two is obvious in their expansion to primitive statements, refer Table 2.2. The standard version pauses before testing for the signal whereas the immediate version pauses afterward. The **present** statement, used in the expansions, executes one subprogram or another depending on the status of a signal. In this case, when the signal is present, **exit** T is executed and control then jumps to the enclosing **trap** T statement. The **trap** and **exit** statements form a structured goto mechanism. The former introduces a labelled lexical scope and the latter terminates it instantaneously.

In this example, the **emit** LEGS_EXTEND statement causes the interface layer to send a command to the backhoe machinery. The mapping of output signals to external effects is application dependent but all signals are treated identically within the reactive kernel. Signal emission occurs instantaneously. Within a reaction, every signal is given a single, consistent valuation. In particular, subsequent emissions of a signal within a reaction have no additional effect; they do not give rise to distinct events. And all components within the scope of the signal act against the same value. The treatment of the interaction between **emit** and **present** statements is central to the semantics of Esterel and it is discussed in the next subsection.

Once the legs are extended the arm segments are driven outward. The boom, for one, is extended by first setting the direction of the piston, and then driving it until the sensor indicates that it has reached the upper position:

```
emit BOOM_PULL;
abort
  sustain BOOM_DRIVE
when BOOM_OUT.
```

[31] Hardware designers use, respectively, the terms *combinational* and *sequential* [Ber00a, p. 19].

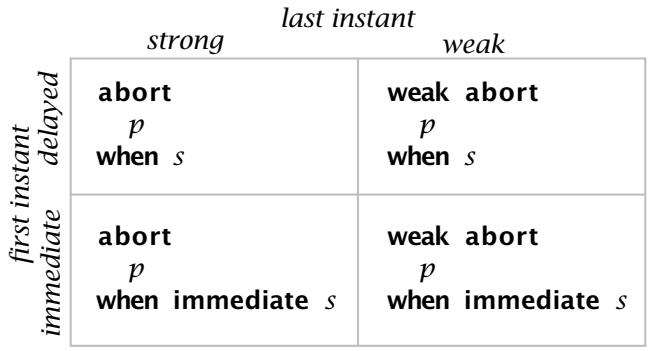


Figure 2.18: Variations of the **abort** statement

Both **sustain** and **abort** are derived statements. A **sustain** expands to a loop that emits the given signal at each reaction. An **abort** expands to two concurrent components inside a **trap**. The first component executes the enclosed statement and the second monitors the watched signal. An **abort** terminates instantaneously when either the enclosed statement terminates or the watched signal is emitted. The enclosed statement is subject to a **suspend** ... **when** *s* statement that, like the inhibition operator of Argos, prevents it from executing in any instant when *s* is present; specifically, it is not given a chance to execute in a reaction in which it is aborted. Abortion in Esterel is thus strong by default. A weak form is also possible. Furthermore, the aborting signal is by default ignored in the first instant, but an immediate form is also possible. The four possible combinations of the weak and immediate modifiers of **abort** are shown in Figure 2.18.

The **suspend** statement is usually considered primitive, but it can be omitted by altering **pause** statements in the primitive expansion of an enclosed statement [TS05]. Essentially, control is only allowed to resume from those **pause** statements when the suspending signal is absent.

The stick and bucket segments are extended identically but for the names of signals. The module system allows the behaviour to be described once, generically:

```
module MOVESEGMENT:
  input STOP, DIR, SECOND;
  output DRIVE;

  emit DIR;
  abort
    sustain DRIVE
  when STOP
end module,
```

and then instantiated multiple times by a **run** statement followed by a list of signal, type, constant, and variable renamings:

```
    run MOVESEGMENT [signal BOOM_PUSH / DIR,
                           BOOM_OUT / STOP,
                           BOOM_DRIVE / DRIVE]
||  run MOVESEGMENT [signal STICK_PULL / DIR,
                           STICK_OUT / STOP,
                           STICK_DRIVE / DRIVE]
||  run MOVESEGMENT [signal BUCKET_PULL / DIR,
                           BUCKET_OUT / STOP,
                           BUCKET_DRIVE / DRIVE].
```

Module instantiation is purely syntactic. As far as the semantics are concerned, each **run** is replaced by the body of the module with parameters renamed appropriately. The three module instantiations are composed with the parallel || operator. They run concurrently in lockstep. The resultant behaviour is relatively simple in this case because the three components do not intercommunicate.

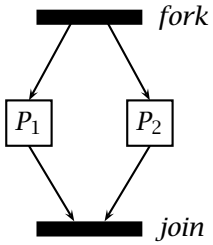


Figure 2.19: Conceptual sketch of synchronization in the statement $P_1 \parallel P_2$

The parallel operator in Esterel is different from those in many other concurrent languages. First, it is completely deterministic; the result of executing subprograms does not depend on how they are interleaved. Second, the parallel operator only terminates when both its components have terminated, its components are thus synchronized initially and at termination, see Figure 2.19. Third, concurrent components execute in a lexical context that determines signal visibility, preemption, and suspension. Fourth, the operator is hierarchical and completely orthogonal. In many languages, there is only a single level of parallel processes. In Esterel, a process effectively evolves as a dynamic tree of active threads.

To conclude the example, suppose that the stop button is to pause and resume the movement of the arm. The composition of the three MOVESEGMENT modules is placed inside a more complicated construction:

```
signal DONE in
  weak abort
    signal STOP_MOVING in
      loop
        await STOP_BUTTON;
        abort
          sustain STOP_MOVING
        when STOP_BUTTON
      end loop
    ||
      suspend
        run ··· || run ··· || run ···
      when STOP_MOVING;
      emit DONE
    end signal
  when DONE
end signal.
```

The innermost **signal** declaration encapsulates the concurrent composition of a component for monitoring the stop button and one for extending the arm segments. The monitoring component has two states: paused and not paused. In the paused state it sustains the local STOP_MOVING signal, which causes the immediate suspension of the extension component. Although the two components run concurrently, the monitoring component must, in every instant but its first, execute before the extension component to determine the status of the local signal and thus whether the three modules are suspended or not. The outermost **signal** declaration, **weak abort**, and **emit** DONE statements are necessary to terminate the monitoring component when the three segments are full extended. Such constructions, though in this case less convenient than a simple **trap** statement, are sometimes useful.[32] Moreover, abortion by a local signal demonstrates another aspect of causality in Esterel. The body of a **weak abort** is executed before the status of the aborting signal is tested, whereas the body of a (strong) **abort** is only executed after the absence of the aborting signal has been determined. The program obtained by removing the **weak** modifier from the example is erroneous

[32] And make for direct comparison with similar constructions in Argos.

because the body of the abort can only be executed if DONE is absent, but it contains an **emit** statement that can cause the signal to be present.

All the primitive Esterel statements and many derived ones have been used in the example. Several features are, however, not used. There are, for instance, variants of the **present**, **trap**, **await**, and **abort** statements with branches listed in order of decreasing priority:

```
present
  case FIRST do ...
  case SECOND do ...
  case THIRD do ...
end present.
```

Furthermore, statements are not limited to testing the presence or absence of a single signal. *Delay expressions* formed from signals, their negations, their previous values, and conjunction and disjunction are permitted [Ber00a, §4.6.3]. They can be prefixed by a *count expression* which is evaluated once when the statement is first executed and then counted down through each reaction where the delay expression is true. The associated statement is only triggered when the count becomes zero.

The example only contains pure signals. Full Esterel allows valued signals, variables, assignments, procedure calls, and branching on variable values. Some of these features are present in the expansion of the **repeat** statement, shown in Table 2.2, which executes an enclosed statement a number of times. The table also shows how an **await** statement with a count expression is expanded using **repeat**. Note that a variable can only be shared between concurrent components if none of them update it.

2.4.3.2 Semantics and compilation

In contrast with many other programming languages, the semantics of Esterel were developed together with the language itself [BC84, BG92, Ber99]. Moreover, rather than just provide a formalization in *Greek letters and funny symbols* [Ber00a, p. 98] or a *26-tuple of sets and relations* [Ber00b, §3], the semantic development aims to provide an intuitive underlying model with mathematical properties.

This subsection contains an overview of the four main contemporary semantic definitions of Esterel and each of their two possible variants. Detailed definitions from the literature [Ber99][PBEB07, Part II] are not repeated, instead the main characteristics of the approaches are compared in broad terms. An early semantics [BG92] is also discussed because it differs from contemporary approaches by being based on primitive statements with a bias toward event-driven execution. The precise semantic definitions makes it easier to discuss and address several issues that are peculiar to Esterel, like instantaneous loops and signal reincarnation, and to develop compilation techniques.

The semantic rules of Esterel are either presented using the concrete syntax shown in the centre column of Table 2.1 [BC84, BG92, PBEB07, TS05] or the concise notation shown in the rightmost column [Ber99]. The concise notation is an additional obstacle to understanding the semantic definitions, but it is easier to write by hand and the encoding of completion codes is particularly natural. Completion codes indicate whether an Esterel statement has terminated instantaneously, 0, paused until the next reaction, 1, or thrown an exception, $k \geq 2$. The **nothing** and **pause** statements are written in the concise notation as 0 and 1 respectively. The names that identify particular **trap** statements in the concrete syntax are replaced by an encoding that assigns an integer to each **exit** which pairs it with an enclosing **trap** declaration.[33] Since outer trap statements have priority over inner ones, priority rules can be resolved using predicates over natural numbers.

The four types of definition and the two variants are listed in Table 2.3. Given a state and input assignment each must determine a new state and output assignment. Internally they track variously the completion status of subprograms, the values of local signals, and the possibilities of signal emissions.

[33] Similarly to de Bruijn indices for bound variables in the lambda calculus.

	logical	**constructive**
rewriting	[BG92, §6] and [Ber99, §6]	[Ber99, §7]
state-marking	[Ber99, §8.3.1]	[PBEB07, §4] and [Ber99, §8.3.2]
operational	[BG92, §8]	[PBEB07, §5]
circuit translation		[PBEB07, §6] and [Ber99, §13]

Table 2.3: Main semantic definitions of Esterel

In the *rewriting* definitions [BG92, §6][Ber99, §§6 and 7], states are encoded as program terms. Each SOS rule defines a transition that maps an input term to an output term and determines a completion code and updated valuation of signals. Rewriting can be described in a small number of relatively abstract rules. It can be implemented directly, but not efficiently and the structure of the original program is rapidly lost.

In the *state-marking* definitions [Ber99, §8.3][PBEB07, §4], states are encoded by annotating terms, usually with a circumflex; for example in the term $1;!o_1;\hat{1};!o_2;1*$ control flow is paused after emitting o_1 but before emitting o_2. Markings change from state to state but the program term does not. More syntax and rules are required compared to the rewriting semantics. The state-marking semantics is adequate for implementing interpreters, but it is not practical for code-generation, particularly for the full language with variables [PBEB07, p. 48].

There are two variations of the rewriting and state-marking semantics: the *logical* and the *constructive*. Each proposes a way of calculating a consistent signal valuation within a reaction, and thereby also a notion of program correctness: a program is only correct if the calculation always succeeds for it. The constructive approach is more restrictive than the logical approach. The two differ only in the rules for handling local and output signals. Within either variation the status of output signals are calculated in the same way as for local signals. The only fundamental difference between output signals and local signals is the restriction of the scope of the latter type.

The *logical* approach is the simplest to define but valid programs can be counter-intuitive and their implementations inefficient. Argos and early versions of Esterel take the logical approach. For a given state and input assignment, valuations of local and output signals are assumed. A valuation is *logically coherent* iff all present signals are actually emitted [Ber99, p. 27]. A state is termed *logically reactive* if for every allowed[34] input valuation, there is at least one logically coherent valuation, and *logically deterministic* if there is at most one logically coherent valuation. A program is *logically correct* if it is both logically deterministic and logically reactive for all reachable states. The logical approach does not respect the flow of control within a reaction. The values of local and output signals cannot necessarily be determined step by step; it is sometimes necessary to speculate and test. For example, consider this program with two parallel components [Ber99, p. 33]:

```
   present O1 then emit O1 end
||
   present O1 then
     present O2 else emit O2 end
   end.
```

By itself, the upper component is not deterministic: assuming the presence of O1 is valid because it will then be emitted, assuming its absence is also valid because it will not then be emitted. On the other hand, the **present** O2 statement in the lower component is not reactive: neither assuming the presence of O2 nor its absence gives a logically coherent valuation. The statements taken together, though, are logically correct because the only logically coherent valuation is O1 and O2 absent.

The *constructive* approach properly respects control flow within a reaction. Sophisticated rules are required for determining the status of local and output signals but the resulting models are nevertheless intuitive and they correspond to an important class of digital circuits [Ber99]. The constructive versions of the rewriting and state-based definitions no longer 'guess' at signal values but instead calculate *Must* and *Cannot*

[34] Only input valuations that respect **relation** declarations are considered.

sets that, respectively, contain all signals that will definitely be emitted and all signals that will definitely not be emitted. Membership in the Must and Cannot sets increases monotonically until a fixed point is reached as more information about signal statuses becomes available. A fixed point is *constructive* if every local and output signal is a member of exactly one of the sets [Ber99, p. 83]. A program is *constructive* iff for every reachable state and allowed input assignment a constructive fixed point can be found. The semantic definitions have the property that these fixed points are unique [Ber99, §7.5]. The previous example is not constructive because O1 cannot be added to either of the Must or Cannot sets; it is not possible to conclude at the outset that it definitely must be emitted, nor that it definitely cannot be emitted, and there is no additional status information that would help.

The rewriting and state-based definitions either involve multiple recursions to determine logical signal valuations or complex mutual recursion to determine Must and Cannot sets. The *operational* definitions [BG92, §8][PBEB07, §5] describe instead the efficient step-by-step computation of a reaction.

The *circuit* semantics describes the translation of a pure Esterel program into a digital circuit [Ber92][Ber99, §13][PBEB07, §6]. It can be considered a semantic definition because the hardware domain comprises simple and well-understood components and can be considered as a set of Boolean relations between state and signal variables.

In contemporary semantic definitions [Ber99, PBEB07], the only non-instantaneous primitive statement is **pause**. The spirit of these definitions is arguably more sample-driven than event-driven because, although not strictly necessary, it is more natural to interpret **pause** as delaying until the next clock tick rather than to the next occurrence of any input signal. In contrast, **halt** is the only non-instantaneous primitive statement in the earlier definition [BG92], and **abort**,[35] rather than polling the status of the watched signal at each reaction, per Table 2.2, is itself a primitive statement. With these primitives, **await** *s* translates to **abort halt when** *s*, rather than to the **loop** with **pause** of Table 2.2. The earlier definition does not include the **suspend** statement, which was introduced later [Ber93].

Besides causality, two other issues arise in the definition and compilation of Esterel: *instantaneous loops* [Ber00a, §§4.7.5–4.7.6][TS05] and *schizophrenia* [Ber99, §12][TS05]. Both involve the instantaneous termination and restarting of loops.

The simplest example of an instantaneous loop is **loop nothing end**. Such divergence within an instant is obviously pathological, but there are more subtle examples:

```
var C : int = 5 in
  loop
    C := C - 1
    if C = 0 then C := 5; pause end;
  end
end.
```

The loop body is executed five times per reaction. Such bounded repetitions cannot be created with pure signals alone in a constructive program, this attempt, for instance, is not constructive:

```
signal S, T
  loop
    present S then
      present T then pause end;
      emit T
    end;
    emit S
  end
end.
```

This program relies on *S* and *T* having different statuses in each of two and a half iterations, which is not allowed. Both unbounded and bounded iterations of loops within a single reaction are forbidden in Esterel. Loop bodies are forbidden from terminating instantaneously, which is assumed in semantic definitions and ensured in practice by

[35] Written **do** *p* **watching** *s* in earlier versions of Esterel.

static analysis [Ber99, §6.6][TS05]. Precise analysis can be exponential in the number of input signals [TS05].

Implementing loops requires care even when their bodies are non-instantaneous because a single syntactic construct may have two different incarnations within a reaction: a situation termed schizophrenia. A single statement may be executed twice, as in the example [TS05]:

```
loop
  present I then pause end; V := V + 1
||
  pause
end loop.
```

If I is present initially, then the program pauses in both parallel branches. In the next reaction, the variable V is incremented and the loop restarts instantaneously. If I is now absent then V will be incremented a second time before the reaction ends. The parallel operator also has two incarnations [Ber99, §12.2]: one where both branches have terminated and another where only the first has. Local signals also cause complications [Ber99, §12.3]:

```
loop
  signal S in
    present S then emit O else nothing end;
    pause;
    emit S
  end signal
end loop.
```

Assume that this program is restarted from the central **pause** statement. The local signal S is then emitted and the loop restarted instantaneously. The first statement of the new loop iteration tests the value of S, but it is not the same signal! Schizophrenic **signal** statements cause local signal *reincarnation*. The reason can be seen more clearly if the loop is unrolled to give the semantically-equivalent program:

```
loop
  signal S in
    present S then emit O else nothing end;
    pause;
    emit S
  end signal
  signal S in
    present S then emit O else nothing end;
    pause;
    emit S
  end signal
end loop.
```

This type of unrolling is necessary prior to translation into a circuit, where for instance specific hardware is needed to resynchronize parallel branches, and it also facilitates other types of compilation where care is otherwise needed to assign or reuse memory locations that track the status of local signals.

Various compilation techniques have been developed to transform Esterel programs into efficient executables [BCE+03, Part III, B]. An important feature of Esterel is that concurrency and communication are resolved statically, that is during compilation, and thus without the overhead of dynamic scheduling. In the earliest approach [BS91] a DFA was generated directly, which gives an executable that is fast but often prohibitively large due to combinatorial explosions of input combinations and parallel statements [BG92]. Another approach is to transform programs into a circuit representation which is then effectively simulated at runtime. This produces compact executables, but they are not efficient because most circuit equations must be evaluated in every reaction even though typically only a few will be relevant to the active control flows. Modern approaches [WBC+00, Edw00, Edw02, PBEB07] transform an Esterel program into a control flow graph and then extract a static schedule from which

compact and fast sequential code is produced. Most Esterel compilers produce C code that can then be compiled for execution on a standard microprocessor. Some, though, target *reactive processors* that are optimised for executing Esterel programs compiled to custom assembly languages [BASRB04, LLB+05].

The accurate detection of non-logical or non-causal Esterel programs within compilers is fundamentally expensive because it depends on the reachable state-space and all combinations of input values. In fact, model-checking techniques, like symbolic reachability analysis using BDDs, are sometimes used. Other compilers insist on the absence of static dependency cycles, which can be checked quickly but which disallows some valid programs.

2.5 Concluding remarks

This chapter has presented a broad introduction to transition systems, TTSs, timed automata, and the synchronous languages Argos and Esterel. Several topics, however, have been omitted. For instance, temporal and real-time logics for specifying systems and properties are not discussed in any detail, nor are specializations and variations of timed automata, like event-clock automata [AFH99] and stop-watch automata [CL00]. While these and similar concepts are important, they are not required directly in the technical chapters. Some additional background topics are included as appendices. Appendix C discusses preorders and equivalences, Appendix A contains an overview of process algebra theory, and Appendix B discusses IOA and related formalisms.

The topics of timed automata and synchronous languages are taken up immediately in the next chapter, where they are applied to together to the simulation of embedded controllers in Simulink.

Chapter 3

Simulating synchronous execution[†]

Computer simulation is an important technique for designing real-time embedded systems. Complex timing characteristics can be analyzed by calculating step-by-step the behaviour of models, which usually comprise both the components being designed and aspects of their intended environment. Simulink [Mat03a] is the de facto industry standard for simulating embedded systems. As its modelling and compilation features have evolved, so has its use as a platform for design and implementation. Recent versions even include a graphical programming language called Stateflow.

Simulink is a practical and powerful tool. It has been developed in response to real-world requirements and warrants academic study if only because the features of the modelling language offer insights into the requirements of industry. Put simply, Simulink is a great source of ideas for modelling and developing embedded systems. Some Simulink features, however, lack the rigorous definitions and attention to fundamentals found in many formal, and often also more theoretical, approaches. Precise semantic definitions are essential for accurate automated analyses, such as model checking, and transformations, such as certified compilation. Recent research has aimed to better define subsets of the Simulink and Stateflow modelling languages through translations to other languages that have a formal basis, such as SMV for model checking [BKB99], hybrid system models for verification [TSR03], Lustre for compilation and analysis [CCM$^+$03, SSC$^+$04, TSCC05], and hybrid automata using graph rewriting [ASK04] and to improve simulation coverage [AKRS08]. The Stateflow language has been studied closely [HR04, Ham05] both because its meaning is particularly intricate, and because it is seen as important to practice.

The proposal in this chapter is not, however, a translation from Simulink/Stateflow to another formalism. It has rather more in common with the co-simulation of controller programs and Simulink models [TNTBS00]. This chapter describes the development of a custom Simulink block that allows the insertion of programs written in the Argos synchronous language [Mar91, MR01] into Simulink models. Argos was chosen because it is an imperative synchronous language with a graphical syntax and because it has a relatively simple definition. Furthermore, the graphical notation enables a direct comparison with Stateflow. While Argos is neither as powerful nor as feature-rich as Stateflow, its underlying model of computation is easier to understand and reason about—a distinct advantage for embedded applications where accuracy and correctness are paramount.

Argos, like all synchronous languages, is based on a discrete model of time derived strictly from the occurrence of events. Simulink, in contrast, executes models in simulated physical time. Embedding Argos programs within Simulink thus requires establishing a relationship between values of the simulation clock t, essentially a finite approximation of a real-valued clock, and a linear sequence of synchronous reactions. In this chapter the mapping is given indirectly by defining a translation from Boolean Mealy Machines (BMMs), the semantic model of Argos, to timed automata; the resulting models thus make for an interesting comparison with the timed automaton models that are developed more directly in Chapter 4. While it is normally assumed that

[†]This chapter is based on two published papers [BS05, BS06].

Argos programs are perfectly synchronous, in the sense that outputs occur simultaneously with inputs within the same simulation step, the translation can account for abstract timing imperfections: specifically non-zero execution times and limits on the frequency of reactions. The mapping accounts for both event- and sample-driven execution schemes, and emphasizes their similarities and differences. There were two aims for developing such a timing model. First, to allow for more detailed simulation results, and second, to gain insight into the effect of implementation limitations on synchronous programs.

The mapping is not only an interesting thought experiment. It also provides a precise specification for implementing the Argos block, which, although conceptually simple, involves many fine details of latching and triggering.

The chapter begins with descriptions of Simulink and Stateflow in §3.1. The details are drawn mostly from experience and user manuals, but much can also be learnt from the research literature. The concepts and algorithms behind Simulink, summarised in §3.1.1, are important to understand the Argos embedding. Details of the Stateflow language are included in §3.1.2 to motivate the embedding and for contrast. The embedding of Stateflow into Simulink, described in §3.1.3, informs the later embedding of Argos into Simulink.

The Argos embedding encodes an idealised execution scheme which is presented in §3.2. The details are first sketched in §3.2.1 and then formalised using timed automata in §3.2.2. The practicalities of implementing the embedding are described in §3.3 and demonstrated in §3.4.

The chapter ends with two reflective sections. In §3.5 the Argos embedding is compared with other similar approaches. In §3.6 the advantages and disadvantages of the approach are evaluated.

3.1 Simulink and Stateflow

Simulink and Stateflow can be used together to create hybrid system models where discrete state changes are combined with continuous dynamics, and to specify designs that combine control logic and discrete dataflow specifications. The Simulink/Stateflow combination is increasingly used as a platform for *model-based design*, where circuits and executable programs for embedded systems are generated directly from simulation models.

The Simulink modelling language is flexible: continuous and discrete elements may be mixed together in a single model. The implementation is designed for performance, allowing the simulation of large and complex models, and also for features like hardware-in-the-loop testing where real hardware is integrated into a model.

The Simulink modelling language and simulation algorithms are described in §3.1.1. Simulink models are built from a set of blocks, of which there are many, and to which new blocks, like the Argos block described later in this chapter, may be added. The Stateflow block is described in §3.1.2, in preparation for a later comparison with models expressed in Argos.

3.1.1 Simulink

Simulink[1] [Mat03a] is a sophisticated realisation of a simple concept: *blocks* are chosen from several libraries and then connected together with lines called *signals* to form a model whose behaviour over a period of simulated time is calculated by repeatedly executing blocks and updating signal values.

Each block provides specific functionality. Blocks are usually parameterised. Parameters range from single values, such as a multiplication factor in a gain block, to complete programming languages, like the Stateflow block. Strictly speaking, libraries contain block classes, each of which may be instantiated multiple times, with differing

[1] Version 5.1.

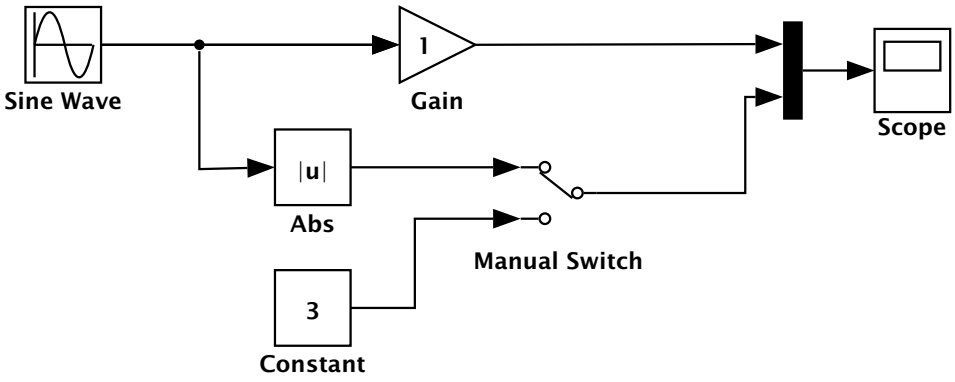


Figure 3.1: Example Simulink model

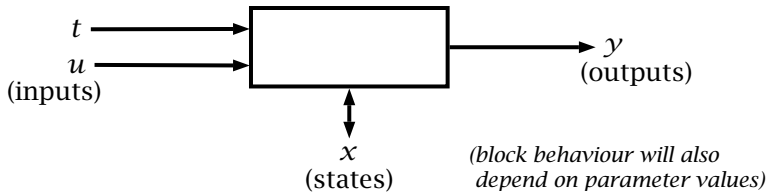


Figure 3.2: Simulink Block

parameter values, in a model. The term 'block' will henceforth refer to a block instance. Some block parameters can be changed while a simulation is running,[2] thereby allowing interactive models. Blocks may be grouped and encapsulated hierarchically.

Blocks are connected together by *signals* which represent quantities that change over time. Signals are typed [Mat03a, §7], primarily for reasons of simulation performance and code generation. Basic values are stored with fixed size and precision in floating point, fixed point, or integer form. There are derived types for matrices and objects. Type checking is performed prior to simulation.

Time is modelled by a single decimal number, nominally seconds, which is incremented by the simulation engine as a model is executed. Conceptually, simulation time mirrors the notion of time typically adopted for physical models: a real number that increases from zero. Simulation on a discrete computer, however, can only calculate signal values at a finite, or at most countable, number of instants. Since such calculations may be non-trivial, various techniques are used to minimise the number of instants where values are calculated while simultaneously trying to maintain simulation accuracy—or at least to allow users to specify the trade-off. Furthermore, the sampling behaviour of implementation models must be taken into account; both for simulation accuracy, as well as for generating implementations.

An example Simulink model is shown in Figure 3.1. The model is simple but serves well enough to illustrate several basic concepts. Blocks are represented by a variety of suggestive shapes: rectangles, triangles, switches, etcetera. Signals are drawn as arrowed lines between blocks. The blocks labelled *Sine Wave* and *Constant* are signal sources with adjustable parameters, for instance the amplitude and frequency of the sine wave or the value of the constant (which is displayed on the block itself). The output value of the sine wave block changes as simulation time passes, and it is passed along the connecting signal lines to the respective inputs of two other blocks, *Gain* and *Abs*. The *Gain* and *Abs* blocks are passive: output values are only updated when input values change. The *Constant* and *Abs* blocks are connected to a block called *Manual Switch* which transmits values from one of its inputs to its output. A parameter chooses which input is transmitted. The parameter value is represented graphically and can be changed during simulation. The *Gain* and *Manual Switch* blocks are connected to a multiplexor block that combines multiple input values into a single output

[2] Parameters that can be changed are termed *tunable*, and those that must remain constant during a simulation are termed *non-tunable* [Mat03a, p. 5-2].

vector.[3] The multiplexor output is connected to the input of a *Scope* block, which is a signal sink that plots incoming values against simulation time. When a simulation is running, a window associated with *Scope* shows the behaviour of the model, which can be influenced by toggling the switch, as time progresses.

3.1.1.1 Block details

A Simulink block, see Figure 3.2, defines the instantaneous relationship between time t, input signals u, state variables x, and output signals y. The relationship may depend on the values of parameters.

States are *discrete* or *continuous*. The values of discrete states change at specific instants, whereas those of continuous states evolve over intervals of time. Blocks that contain only discrete states are termed *discrete blocks*, those that contain only continuous states are termed *continuous blocks*, and those that contain both types are termed *hybrid blocks*. Discrete blocks specify the times when state changes may occur whereas continuous state values are approximated from information on their derivatives and significant values. The states and outputs of discrete blocks are piecewise continuous rather than truly discrete in the sense of possessing only a finite or countable sequence of values [CCM+03]. Controller models, particularly those intended for synthesis, are usually built exclusively from discrete blocks [Mat01].

Conceptually, each block represents three functions of the simulation time t, state vector x, and input vector u [Mat03b, p. 1-6]:

$$\begin{aligned} y &= f_o(t,x,u) \\ x'_d &= f_u(t,x,u) \\ \dot{x} &= f_d(t,x,u) \end{aligned}$$

Output values are determined by f_o, discrete state changes by f_u, and derivatives of continuous state elements by f_d. While in principle the functions are defined over a dense interval of t, since only a finite number of values can be sampled in any simulation run it is assumed that the functions f_o and f_u are constant between sample points, that is given consecutive sample times t_i and t_{i+1}:

$$\begin{aligned} \forall t' .\ t_i \le t' < t_{i+1} : & \\ f_o(t',x,u) &= f_o(t_i,x,u) \\ f_u(t',x,u) &= f_u(t_i,x,u) \end{aligned}$$

In practice, each block is triggered by the simulation engine to compute values for the functions—for given values of the simulation time, inputs, and state. The methods called by the simulation engine are defined per block class. The engine stores the state of each block instance and passes it to the class methods when required. There are three main methods to compute outputs, update discrete states, and calculate derivatives. Other methods handle initialization, termination, and parameter changes.

The simulation engine ultimately decides the simulation times at which to calculate signal and state values, but blocks have an influence in two ways: through *discrete sample times* [Mat03a, pp. 2-8 and 2-28][Mat03b, p. 1-15], which are discussed now, and *zero-crossing detection*, which is discussed in the next subsection.

Discrete sample times are specified as pairs of parameters: a period t_s, which must be greater than zero, and an offset t_0, which is relative to the start of the simulation and whose absolute value must be less than the period. Several discrete sample time pairs may be specified for a single block, which allows, for instance, modelling of the behaviours of distinct periodic tasks within a block. Separate discrete sample time pairs may also be specified for each input of a block.

Instead of discrete sample times, a block, or individual inputs, may be specified as continuous, implicit, or variable. Continuous blocks are triggered at every simulation step.[4] The triggering of a block with an implicit sample time is calculated from the

[3] Multiplexors belong to a class of 'virtual blocks' [Mat03a, p. 5-2]. They are not polled directly by the simulation algorithm, but serve rather to assemble models.

[4] Although triggering can be limited to major steps, described in §3.1.1.2, if desired.

sample times of the other blocks connected to its inputs through a process called sample time propagation which is similar to the clock inference of synchronous dataflow languages [CCM+03]. A block with an implicit sample time will be continuous if any of its inputs are continuous, otherwise it will be discrete with a sample time calculated as the greatest common denominator, with allowance for offsets, of the input sample times [Mat03a, p. 2-8]. Blocks with variable sample times are polled by the simulation engine to determine when to schedule their next triggering.

The decisions of the simulation engine can be influenced by signals within a model through *enabled* and *triggered* subsystems [Mat03a, §4]. The blocks in an enabled subsystem are only executed when the value of a designated control signal is positive. Those in a triggered subsystem are executed when certain value changes occur in a designated control signal. Enabling and triggering may be combined. Blocks may, in addition, be placed within so called function-call subsystems [Mat03b, p. 7-31] where execution is completely controlled by another block rather than by the simulation engine. Execution then occurs when the other block makes a specially designated function call, and it is processed within the calling block's thread of control between its other actions.

Custom Simulink blocks are called *s-functions* [Mat03b]. They can be written in various programming languages. An s-function must export a set of functions to be linked into the simulation engine. The functions are called at various times to query, initialize, and execute block methods. Block methods call functions in the Matlab and Simulink Application Programming Interfaces (APIs) to query and manipulate data values, to modify the visual appearance of a block, and to request configuration parameters.

3.1.1.2 Simulation algorithm

A Simulink model is simulated in two phases: it is first initialized and then it is executed in a simulation loop. The details of the simulation loop are influenced by *solvers*, which make key decisions within the simulation engine. Special techniques identify instants when signals change in significant ways, termed zero-crossing detection, and also loops of instantaneous feedback, termed *algebraic loops*.

During initialization [Mat03a, p. 2-12], models are examined for mismatched signal types and possible algebraic loops. The block hierarchy is flattened. Sample times are collected, propagated, and checked. Data structures are initialized. The simulation engine determines a sequential execution order based on instantaneous signal interdependencies.[5] The blocks of a model are executed in sequence even though they are conceptually concurrent.

Simulink processes a model—a set of blocks and their interconnections—by increasing a global time parameter t in steps from zero. At each step, blocks are polled in sequence to determine new signal and state values. When (if) a stable value is found for every signal, t is increased by an amount determined by the active solver using information from each block. There is a trade-off between accuracy, which, for continuous blocks, means smaller increments of t, and performance, which means larger increments, or equivalently, executing blocks as infrequently as possible. The trade-off is made, in greatest part, by the choice of a solver and its parameter values. The basic simulation algorithm together with a specific solver and parameter values effectively determine the semantics of a model [TSCC05].

The most appropriate solver depends on the particularities of a given model and the requirements of the modeller [Mat03a, pp. 10-9–10-15]. Some solvers are limited to models that only contain certain types of blocks, for instance models where all blocks are discrete or models where no blocks have variable sample times.

For a model containing only discrete blocks, either a *fixed step* or *variable step* discrete solver is most suitable. The former increments t by a fixed value at each simulation step. The fixed value is chosen so as to include all instants where blocks require triggering, according to their offsets and sample times [Mat03a, p. 2-31]. The fixed step solver is the simplest and its results are easy to understand and manipulate, but for some models, depending on the relative ratios between sampling rates, there

[5] Only blocks with direct-feedthrough inputs are considered [Mat03a, p. 2-13].

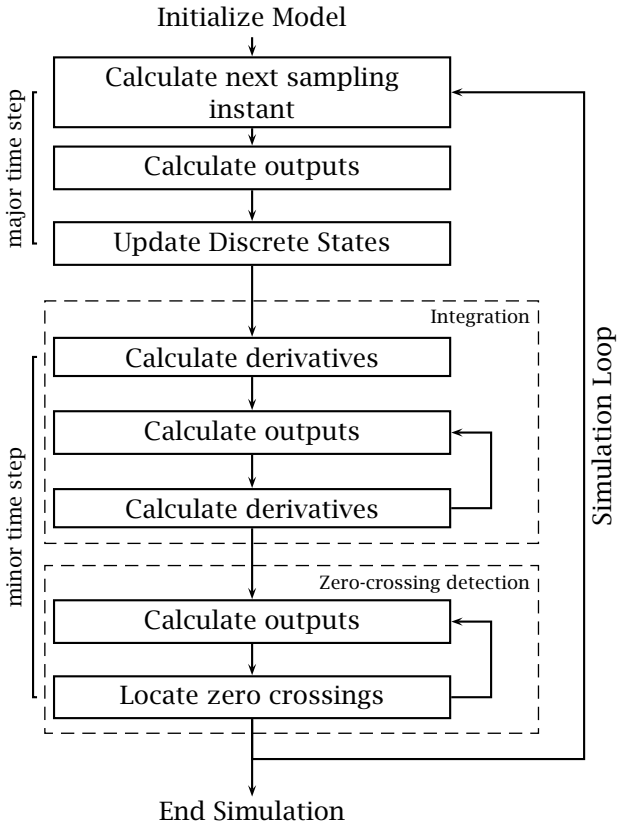


Figure 3.3: Basic simulation steps (Adapted from [Mat03a, p. 14-13][Mat03b, Figure 1-2 and pp. 3-36 and 3-37])

may be many sampled instants where nothing changes. A variable step solver only considers instants where at least one block requires triggering. Less work is done but the results are more complicated because signals are effectively sequences of pairs that combine time and value, rather than just a sequences of values with an implied period.

A model containing only continuous blocks is essentially a set of Ordinary Differential Equations (ODEs). Analytical solutions to ODEs are not possible in general, so Simulink provides several different numerical solvers to find signal values over time. Both fixed-step and variable-step continuous solvers exist; the most appropriate depends on the dynamics of a model and the desired precision [Mat03a, pp. 10-9 and 10-10]. Hybrid system models, containing both discrete and continuous blocks, are simulated by constraining a continuous solver with step-size information calculated from the subset of discrete blocks.

The calculation of, and iteration over sampling instants, as described above, is the chief part of simulation. Each such iteration is known as a *major time step* in contrast with other, subordinate *minor time steps* that some continuous solvers also calculate to increase simulation accuracy. The distinction is clear in Figure 3.3: at each major time step, the value of t is calculated and fixed, blocks are then executed in sequence to determine output signal values, when these have stabilised each block is executed again to update its state. Minor time steps are divided into two parts: integration and zero-crossing detection.

During integration, blocks are polled for derivative values. For increased accuracy, a solver may also iteratively calculate continuous state values for subintervals within the major time step [Mat03a, p. 2-16].

Zero-crossing detection is a technique that aims to identify instants of significant change in continuous state variables, without unnecessarily increasing the number of simulation steps. The need for zero-crossing detection can be understood by comparing two different representations of continuous behaviour.

(a) Solution converges: $z = u/2$ **(b)** No valid solution (divergent): $z = z + 1$

Figure 3.4: Algebraic Loops [Mat03a]

The behaviours of a hybrid system, like those modelled in Simulink, could be thought of as alternating sequences of continuous variable trajectories and discrete actions, that is, essentially as the (A, V)-sequences described in §2.2.2.3. Such a *dense* representation gives a variable valuation for every instant of a dense time interval. Alternatively, a *sampled* representation only gives variable valuations for a countable, monotonically increasing sequence of time instants.[6] Dense representations are a more faithful description of continuous behaviours but sampled representations are closer in spirit to the sequence-based approaches used to model discrete systems [MP93]. Simulink does not represent models analytically, but rather approximates their behaviour numerically: it cannot calculate values for every point in a dense interval and must instead create sampled representations. A sampled representation of a continuous behaviour is adequate if it includes all significant state changes. In some approaches, the significant events in a system are identified by a set of assertions over variable values [MP93]. In Simulink, the timing of significant events is expressed through zero-crossing variables [Mat03a, pp. 2-18–2-23].

Each block can register one or more zero-crossing variables with Simulink. The variable values are calculated similarly to output signals, but rather than being passed to other blocks their values are tracked internally by the simulation engine. The instants when a value changes sign—when its function crosses through zero—are regarded as significant. When the simulation algorithm detects such a sign change it tries to iteratively hone in on the instant when the value would be zero, although it tries to avoid the exact instant since there the value of a state variable may be undefined [Mat03a, p. 2-19]. Zero-crossing variable values are usually calculated as functions of state variables such that discontinuities and other significant events map to zero.

Algebraic loop calculation [Mat03a, pp. 2-23–2-26] is another special feature of the simulation algorithm. The static schedule of block execution that Simulink tries to calculate during initialization is based on signal dependencies. It can happen that the chain of dependencies is circular; that there is a path of direct feedback such that the inputs of a block are directly required to calculate its outputs, which in turn are required to calculate its inputs. These cycles of instantaneous dependency are termed *algebraic loops*. Two simple examples are shown in Figure 3.4. Such cycles may also run through several blocks and there may be several cycles within a model. Some dataflow languages, like Lustre [HCRP91], prohibit paths of instantaneous feedback,[7] but Simulink allows them.

Blocks where there is a delay between changes on input signals and corresponding changes on output signals effectively break paths of instantaneous feedback. Each block declares the input signals that have *direct feedthrough* to output signals so that Simulink can detect algebraic loops. At each instant, the simulation algorithm iterates through the blocks in an algebraic loop trying to determine a fixed point, viz. a set of stable signal values. Some models, like Figure 3.4a, converge in this way, others, like Figure 3.4b, do not.[8]

3.1.2 Stateflow

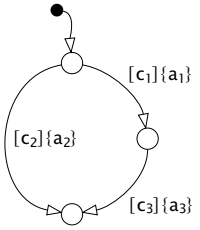
Stateflow[9] [Mat03c] is a Simulink block for modelling and implementing discrete control logic in a graphical notation that is inspired by Statecharts [Har87]. Stateflow is em-

[6] Sampled representations can be thought of as generalisations of timed sequences, see §2.2.2.2.

[7] Unit delays are required: inputs can only depend on previous outputs.

[8] Such issues are common to languages where instantaneous feedback is possible, especially the synchronous languages.

[9] Version 5.1.



(a) Original graphical notation

```
if [c1] {
    a1
    if [c2] {
        a2
    } else if [c3] {
        a3
    }
}
```

(b) Translation to source code

Figure 3.5: Standalone Flow Diagram [Mat03c, p. 2-20]

inently practical, but its semantics is intricate and, arguably, less principled than those of other languages for expressing discrete controllers, most notably the synchronous languages. The outline of Stateflow in this subsection is divided into four parts: the flow diagram language in §3.1.2.1, the details of diagram execution in §3.1.2.2, related issues of event processing in §3.1.2.3, and a consideration of semantic issues in §3.1.2.4.

Stateflow is based on Statecharts [Har87], which embodies a compelling and natural idea: to extend the utility of finite automata with parallelism, hierarchy, and variables, for application to complex reactive controllers. It is not alway clear, however, what such diagrams mean precisely. Despite later proposals [HPSS87, HN96] and vague intuitive expectations there have always been behavioural differences between various Statecharts-like languages [Bee94]. It has been suggested [SSC+04] that the reason for this is the relative immaturity of research into structured state machines, particularly compared to block diagram languages, and because their semantics are both intricate and, to date, seemingly without a complete set of widely-accepted principles.

Stateflow incorporates the main syntactic features of Statecharts, like refinement, orthogonality, multi-level transitions, and entry-by-history, but not some minor features, like asterisked history junctions, the history-clear function, and forking transitions. It also has other features. There are, for example, truth tables for expressing and documenting transition logic [Mat03c, §9][LHHR94] and a flowchart-like programming language called *Flow Diagrams*[10] that extends the basic conditional junctions of Statecharts. Flow diagrams are used for expressing both complex transitions between states and for programming standalone functions.

The three most unique and interesting features of Stateflow are the possibility of compound transitions between states, the handling of input and internal events by queuing and stacking respectively, and the intricate rules for ordering executions. Each of these is discussed in turn in §§3.1.2.1–3.1.2.4.

3.1.2.1 Flow diagrams

A Flow Diagram is a graph of *transition segments*, or just segments, between various types of nodes. *Default markers*, small black dots with only out-going transitions, and states serve as starting points for sequences, branchings, and loops of transition segments between *connective junctions*, that may end in either those junctions, *history junctions*, or other states. States are drawn as rectangles with rounded corners and junctions are drawn as unfilled circles.

For standalone functions [Mat03c, p. 3-40], Flow Diagrams are simply a graphical programming language. An abstract example is shown in Figure 3.5a. Execution begins at the default marker and takes one of two paths through the connective junctions until no further steps are possible. A segment is only eligible for execution when its *condition* is true. Conditions are boolean expressions written between square brackets; c_1, c_2, and c_3 in the figure. Execution of a segment performs the associated *condition actions*—written between braces like a_1, a_2, and a_3 in the figure—and transfers control to the next destination junction. Segments leaving a junction are

[10] Hence the name: 'state' for state machines and 'flow' for flow diagrams.

considered for execution in a deterministic order. Those with conditions take priority over those without [Mat03c, p. 4-9] and otherwise they are ordered by relative angular position [Mat03a, p. 4-10]. Flow Diagram functions can thus be translated directly to sequential source code, as for example in Figure 3.5b.

Transitions between states are expressed as Flow Diagrams which, despite possessing a declarative aspect by virtue of additional priority rules and the possibility of repeated evaluation, remain an essentially imperative programming language. Flow Diagrams that express transitions between states [Mat03c, pp. 3-13–3-24] may have extra *event* and *transition action* labels for, respectively, responding to events and generating new ones. The general transition label format is [Mat03c, p. 2-18]:

$\texttt{event}_1\texttt{|...|event}_n\texttt{ [condition] \{condition-actions\} / transition-actions.}$

Segments labelled with one or more events are only considered for execution when one of those events is being processed. They have priority over segments without event labels. Event labels may contain *temporal operators*, like after(n, E) or every(n, E), where n is an integer expression and E an event [Mat03c, pp. 7-77–7-84].

While condition actions are executed immediately when a segment is taken from one junction or state to another, transition actions are only executed when a complete path to a destination state has been found. Transitions that are evaluated, along a path of enabled segments but not actually taken due to a later lack of enabled segments may yet have an effect on state and output. In addition to variable assignments and function calls, both types of actions may emit new events, which are evaluated recursively against a diagram. Events emitted from transition actions are processed after the source state has been exited, whereas for those emitted from condition actions, the source state remains active. The latter fact introduces the risk of non-termination [Mat03c, pp. 4-43–4-44].

3.1.2.2 Executing diagrams

The execution of transition segments within a state diagram in response to an event is influenced by the relative positions of segments and states. An event is processed down through the hierarchy of states, either from the top of a diagram or from an explicitly named state therein. Active states at higher levels are considered before those at lower levels, and then, from each, segments labelled with the event are considered in order of their destination state (higher states having priority), their label, and, finally, their relative angular position on the source state. If no valid segment with an event label is found then segments without event labels are considered [Mat03c, pp. 4-8–4-10]. Segments are followed from a state until interruption by event emission, or arrival at a junction with no outgoing segments, or at a junction where none of the outgoing segments are enabled, or at a state. For junctions with no outgoing segments and for states, a transition is regarded as complete and no further segments from the source state are considered. But, if evaluation reaches a junction where there are outgoing segments but where none of them are enabled, then the segment next in order from the previous junction is considered, and so on back through to the original state.[11] This unwinding of control back through intermediate junctions is called 'backtracking' [Mat03c, pp. 4-61–4-62], but the effects of condition actions that have already been evaluated are not undone.

Besides transitions between states, other actions may be associated with states using various keywords [Mat03c, pp. 3-9–3-12], and on so called *inner transitions* [Mat03c, p. 3-21, pp. 4-41–4-49]. These features provide flexibility in expression and further complicate the already intricate interpretation algorithm.

Stateflow diagrams may contain parallel components at any level of the state hierarchy. Their behaviours are neither concurrent nor interleaved. Rather, they are evaluated in order of relative graphical position. Stateflow is thus a completely deterministic *sequential imperative language* without concurrency [HR04].

[11] The operational semantics [HR04, p. 8] and the description of the translation to Lustre [SSC+04] are more lucid than the official description [Mat03c, pp. 4-61–4-62].

3.1.2.3 Processing events

When several input events occur simultaneously at a single simulation step, they are queued in order of their position in the input vector [Mat03c, p. 6-8] and processed one by one per the foregoing description. Unlike other versions of Statecharts that process compound events [HN96, HN96] and the synchronous languages, conjunction is not possible in Stateflow event guards. Actions of a Stateflow block may provoke new inputs from other blocks in the same instant, but these are not usually processed until the next simulation step [Mat03c, 6-11]. Such details are not immediately clear from the informal documentation [Mat03c] nor are they always present in detailed semantic treatments [HR04, Ham05], which may, for instance, only describe the response of a diagram to a single event.

Events emitted whilst evaluating a diagram are not added to the queue of pending input events. Rather the original event and the status of the interpreter, that is the current focus within the hierarchy of states and segments, are effectively pushed onto a stack until the new event is processed. Any subsequently emitted events are treated in the same way, and as the processing of each is completed, prior interpretations are resumed one by one from the stack. The state of the diagram, active states, and variable values, is treated as a global variable: changes made within a recursive evaluation persist after its return. In particular, the focal state of a resumed interpretation may have become inactive, in which case interpretation of the affected segments is ended prematurely—so called 'early return logic' [Mat03c, pp. 4-18–4-20].

The stacking of emitted events has both advantages and disadvantages. On one hand, the model of execution is essentially the same as any other sequential programming language and it can thus be interpreted and compiled with standard techniques, whereas, for example, techniques for effectively compiling Esterel have required more novel efforts [PBEB07]. On the other hand, while Stateflow diagrams may look like hierarchical automata, they lack many of the properties, and hence benefits, of more considered languages. In particular, they are not compositional, and, moreover, semantic mappings to more fundamental models, like LTSs, are ineluctably intricate, which renders manual analysis all but impossible, and makes automatic analysis less effective, since the number of states may be unbounded and subcomponents cannot be processed in isolation, and also more complicated, since many corner cases must be considered. Furthermore, the presence of a stack makes it difficult to predict or bound memory usage at compile time, which is important for resource-constrained or critical embedded systems. Correct translation into stringent languages requires fixing an upper bound on recursive processing [SSC+04].

3.1.2.4 Semantics

The behaviour of Stateflow diagrams can be complicated and difficult to determine. Calculating the response of a diagram to a triggering input can require working through intricate execution rules: considering priorities of states and transition segments, tracking which states are active, or about to become active, accounting for backtracking behaviour, and maintaining both a stack of emitted events and a queue of input events [Mat03c, §4]! Flow diagrams, moreover, may involve looping, branching, function calls, and updates to variables.

Although tracking many details and steps presents no fundamental problem for computerized tools, the interpretation algorithm is rather involved, and it must be implemented at least three times: for simulation, for hardware generation, and for software generation. Any misinterpreted or unconsidered detail may cause behavioural differences. Similarly, analysis tools must also encode parts of the interpretation algorithm: any discrepancies risk soundness.

Most formal treatments are limited to subsets of the full language [SSC+04, HR04, Tiw02], often even excluding hierarchy [Spe02] and internal communications [ASK04]. Even industrial bodies recommend that some features are best avoided, namely undirected events [Mat01] and backtracking [Com99], either for reasons of safety or implementation efficiency. Efforts have also been made to simplify and normalise the use of flow diagrams [Com99, BR01].

3.1.3 Executing Stateflow within Simulink

The Stateflow block integrates Stateflow diagrams into Simulink models. The block provides an interface from signal values to events and variables for inputs and outputs and vice versa. It is responsible for triggering a diagram in response to input events, simulation steps, and the passage of time.

The Stateflow block is triggered during simulation, like any other block, to calculate state and output values. Block parameters determine when triggering is required, and whether there will then be an *awakening*, which means repeated executions of the internal diagram until no pending events remain. The distinction between triggering and awakening is only important for *edge-triggered* blocks where input values are polled during triggering to detect transition events and the diagram is only executed when such events actually do occur.[12] Triggering otherwise implies awakening.

In addition to edge-triggering, there are four other types of triggering for Stateflow blocks: *sampled*, *continuous*, *inherited*, and *function-calls* [Mat03c, pp. 8-11–8-21].

A sampled block is triggered at regular intervals of simulation time. The period and offset are either set explicitly for the whole block, as they are for the sample-driven executions of synchronous programs, or inherited from the input signals. In fact, multiple period/offset pairs may be specified for a single block; this feature is useful for modelling the presence of multiple periodic tasks.

Continuous blocks are executed at each simulation step. They are intended for diagrams that model functional relations between input and output signals, or phase changes in physical systems, rather than for specifying discrete control logic, as such they are candidates for zero-crossing detection [Mat03c, pp. 8-14–8-15].

The triggering of inherited blocks is determined by sample-time propagation.

A block triggered by function-call is awakened as part of the execution of another block; possibly another Stateflow block. Execution of the calling block continues, possibly with changed variable and signal values, after the awakening has been processed.

Relations between Simulink signal values and Stateflow variables and events are defined per block. An input variable associates a name with an input port of the block. Reading from such variables returns the current value of the connected signal. For edge-triggered and sampled blocks, input variables are like the sensors of Esterel. For other types of blocks, the connection between input signal rates and awakenings makes input variables closer in nature to the internal signals of block-diagram languages like Lustre and Signal. An edge-triggered block has a distinguished vector of input signals which are polled to detect either, or both, rising and falling transitions [Mat03c, pp. 6-11–6-12]. Detected transitions cause associated events to be added to the input queue of a chart, which is then awakened to process them.

Both Stateflow variables and events can be designated as block outputs. Assigning a value to an output variable changes the latched value of the associated signal. Events may be individually designated as either edge-triggering or function-call outputs. Emissions of edge-triggering output events [Mat03c, pp. 8-19–8-21] cause alternately rising and falling edges on attached signals. These signals can trigger other blocks through the usual Simulink scheduling algorithm; specifically, execution of the emitting block is completed before the execution of triggered blocks commences. Emissions of function-call output events [Mat03c, pp. 8-16–8-19], in contrast, cause immediate execution of connected blocks, and execution of the triggering block continues afterward. Some guidelines [Com99] recommend that control be explicitly propagated through a model using function-call output events, rather than via the Simulink scheduler so that designers have explicit control over execution orderings.

Besides emitting output events, a Stateflow controller can coordinate other activities within a Simulink model through during [Mat03c, pp. 3-9–3-12] and bind [Mat03c, pp. 7-85–7-92] state actions. The combination of Stateflow and Simulink in this way is a natural and powerful implementation of the original Statecharts idea of associating discrete states with continuous activities [Har87, §5], or continuous equations [MMP91].

[12] Somewhat like the event-driven executions of synchronous programs.

3.2 Modelling synchronous execution

Synchronous programs execute over a series of discrete instants. Only the ordering of reactions is relevant and quantitative aspects of time are expressly ignored. But time in Simulink is less abstract and one issue when embedding a synchronous program into Simulink is specifying the relationship between simulation time and logical time.

The relationship is defined in this section by mapping a semantic model for Argos programs, Boolean Mealy Machines (BMMs), into timed automata. The mapping is influenced by both the mode of execution, either sample-driven or event-driven, and two parameters that account for some idealised limitations of implementations. It provides both a specification for the block that will be described in §3.3 and a different way of thinking about synchronous programs implemented in software.

In §3.2.1, the idealised execution parameters are described and discussed. In §3.2.2, the timing model is first described intuitively and then defined formally.

3.2.1 Execution parameters

It will be useful to regard synchrony along two orthogonal dimensions: one *internal*, relating to the semantics of languages and models, the other *external* and concerned with the interaction of implementations and the environment. Different modelling approaches can be classified by whether, for each of the two dimensions, they treat behaviours as instantaneous or delayed. A sketch of this idea is shown in Figure 3.6. Perfect synchrony, Stateflow, and the Argos block are discussed subsequently. The combination that models both internal and external delays, labelled 'other', could be seen as corresponding to programming in assembler where the delays of individual instructions between observable effects are accounted for and exploited.[13]

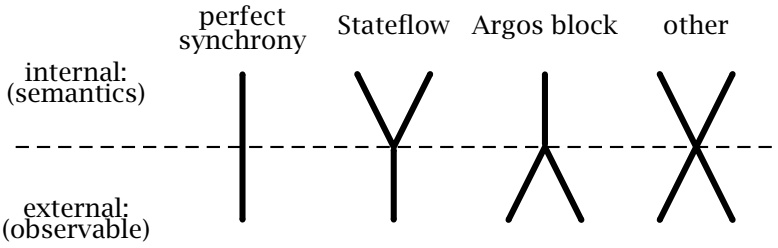


Figure 3.6: Conceptual sketch of internal and external synchrony

The model of *perfect synchrony* shared by languages such as Esterel, Argos, and Lustre, is 'perfect' in both dimensions. Internally, concurrent components communicate and change state simultaneously in a single step. This principle challenged researchers to provide satisfactory semantics despite seeming paradoxes. It yields deterministic behavior with strong mathematical and physical foundations. Externally, the assumption allows a mapping between system function in terms of input/output traces and discrete instants of time. For the synchronous languages the internal and external views are inseparable and interrelated: the single step nature of the internals justifies the assumption of an instantaneous, or at least very fast, transformation of inputs to outputs; instantaneous reactions imply that internal events cannot be further ordered; and each reaction is assigned a single, consistent set of signal valuations. But such a strong coupling between external and internal semantics is no *fait accompli*.

In Statecharts and Stateflow reactions are ideally computed without delay from an external perspective, but the results do depend on internal execution orderings. At each chart awakening, inputs trigger sequences of internal transitions and signal emissions which culminate in outputs and a new chart state. Since events are processed one-by-one and may reoccur in a sense, notions of consistency are less applicable. Final variable values may depend on fine details of the evaluation order. This scheme avoids the difficulties of synchronous language compilation and, arguably, provides a clear sense of cause and effect [Bee94]. Unfortunately, it can also make reasoning

[13] An example of this style is presented in §4.5.

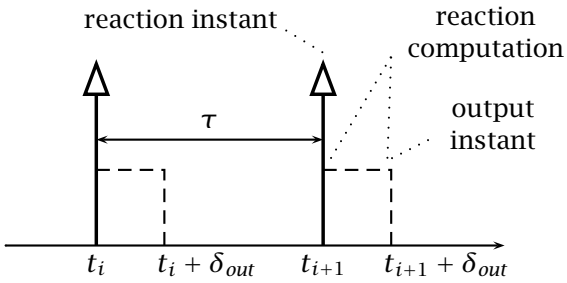


Figure 3.7: External timing parameters

about system behavior complicated and lead to brittle behaviors that depend on the execution target; be it simulation, software, or hardware.

Simulink is used to analyse the dynamic behavior of models containing both continuous and discrete elements. Models may contain liberal mixtures of components and parameters, thanks to the numeric techniques employed, and it is thus feasible to experiment with low-level controller detail and the idiosyncrasies of different external environments. In this spirit, a block for simulating synchronous language programs might remain true to the synchronous semantics internally, but admit more choice when modelling externally observable behavior. The remainder of this chapter pursues this idea in the form of an Argos block.

The Argos block is parameterized by two timing values:

1. (Minimum) Period τ. In sample-driven mode $\tau > 0$ is the time between reactions. In event-driven mode $\tau \geq 0$ is the minimum time between reactions.

2. Output lag δ_{out}. This is the delay between the instant that a reaction is triggered and the instant that the resulting output values may be observed. $0 \leq \delta_{out} \leq \tau$.

The intuition behind the parameters is sketched in Figure 3.7. The vertical open-arrowed lines mark program reactions at t_i and t_{i+1}. Two reactions must be separated by, either a minimum or exactly, τ time units. Furthermore, reaction computation takes time and system outputs do not change instantaneously, but rather after a delay of δ_{out} time units from the previous reaction instant.

An event-driven system with both parameters set to zero would be perfectly synchronous internally and externally, that is reactions would occur precisely when triggered by input events, take zero time, and yield outputs in the same instant.

Fixing δ_{out} at zero and choosing any $\tau > 0$ (still for an event-driven system) implies a finite number of otherwise perfectly synchronous reactions in any interval of time. This blurs the line somewhat between event-driven and sample-driven systems. All of the inputs that occur after one reaction are treated as a single synchronous event at the subsequent reaction. Multiple triggerings of an input, in violation of the assumption of synchrony, reduce to a single observation.

With $\delta_{out} > 0$ reactions have finite duration; the instant of output emission is separated in time from the triggering instant of reaction. To maintain atomicity, inputs that occur during this period are not considered until the subsequent reaction. When $\delta_{out} = \tau$, outputs of the ith reaction may be externally simultaneous with the inputs of the $(i+1)$th reaction. Internally, nothing changes, nor should it; that this behavior is not always desirable is not a sufficient reason for its prohibition. Overlapping reactions are excluded, however, by the constraint $\delta_{out} \leq \tau$, as there seems little practical gain from allowing them and much added complexity for models and implementations.

The timing parameters are deliberate simplifications: δ_{out} encompasses the WCET for computing a reaction and τ captures other inherent limitations of a target platform. Acquiring values for these parameters may not be trivial, though the general assertion that they are relatively easy to calculate for a synchronous program has received more attention recently [BTH07, JHRC08].[14] Furthermore, the model assumes that all reaction computations are of equal duration, and the δ_{out} parameter implies

[14] The commercial SCADE tool from Esterel Technologies also incorporates WCET tools from AbsInt.

that output signals always change value simultaneously at a fixed time after the start of each reaction. These simplifying assumptions could be met by an implementation if necessary.

3.2.2 Timed Automaton Model

3.2.2.1 Intuition

The semantics of the execution parameters τ and δ_{out} with respect to a given synchronous language program are formalized by mapping from a Boolean Mealy Machine (BMM) and the parameters themselves to a timed automaton. This makes the model unambiguous, leads to an accurate Argos block implementation, and provides a precise basis for thinking about how a controller implementation operates over time. A simple example will clarify the basic idea.

Figures 3.8a and 3.8b show the ABRO program [Ber00a] in Argos and Esterel respectively. It has three input signals, a, b and r, and one output signal, o. When both a and b have been received, in any order, or even simultaneously, an o is emitted. The r signal resets the program, prohibiting o from being emitted and forgetting any as or bs that may have already been received. The Argos version uses two local signals, l_1 and l_2, to emulate the behavior of the Esterel parallel construct. Both programs map to the same BMM, shown in Figure 3.8c.

Given parameters τ and δ_{out}, and an execution mode, in this case sample-driven execution, the timed automaton of Figure 3.8d can be produced. The cube-like structures, positioned at each state of the original BMM, represent input latches. A single clock c constrains the timed behavior of the automaton. Latch transitions within the initial BMM state may happen at any time, but those in the other states may only occur when $0 < c \leq \tau$ (for technical reasons discussed in §3.2.2.2). The solid transition lines mark instants of reaction occurring precisely when $c = \tau$, the reaction transitions from the initial state are marked differently, they must occur immediately on startup, that is when $c = 0$. The destination of the reaction transitions depends on the latch contents and the original automaton, but it is always to an empty latch state. The clock c is reset on each reaction transition and thus measures the time elapsed since the last reaction. The right-most original state is split in two: control stays in the left-hand latch until the output o occurs and then it shifts into the right-hand latch. The connecting transitions encode the emission of the output o when $c = \delta_{out}$ and they do not change the input latch contents.

When the event-driven mode is chosen, the translation works differently. Events received while $c \leq \tau$ are latched and a reaction occurs when $c = \tau$, as illustrated by the left-hand side of Figure 3.9. Events received when $c > \tau$ trigger a switch to an urgent latch, the right-hand side of Figure 3.9 (on page 71), where further events may be captured before a reaction occurs in the same instant. Time may not progress between the first such event and a reaction.

3.2.2.2 Details

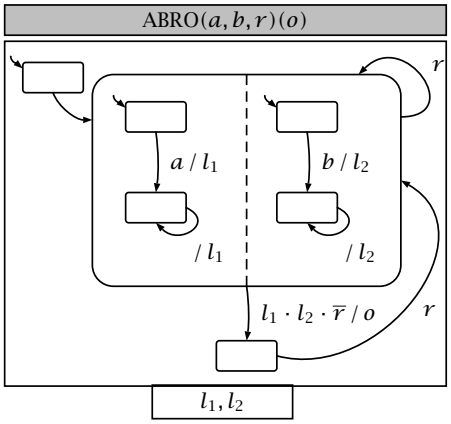
The intricacies of latching and timing in the transformation are made precise by a mapping from BMMs, Definition 2.4.1, given certain timing and execution-mode parameters, into timed automata, Definition 2.2.17. BMMs [MR01] are the basic semantic model for pure Argos programs. Pure Esterel programs may be treated in a similar way. The combination of program and implementation in dense time will be expressed as a timed automaton.

Given a BMM, fixed values for the parameters τ and δ_{out}, and a choice of either sample-driven or event-driven execution, a timed transition system that incorporates the intuitions just described is constructed. In the definition $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$ and $\mathbb{Q}^{\geq 0}$ is the set of positive rationals including zero.

Definition 3.2.1
For a set I, a *valuation* with respect to $J \subseteq I$, written $\nu_J : I \longrightarrow \mathbb{B}$, is defined:

$$\forall i \in I : \ \nu_J(i) = \mathsf{true} \text{ iff } i \in J$$

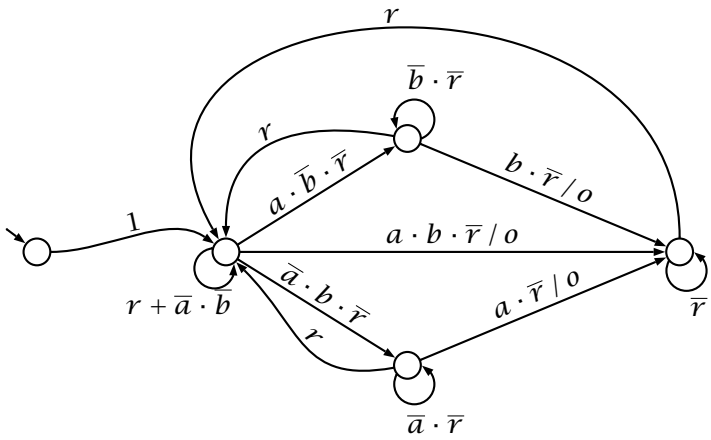
■



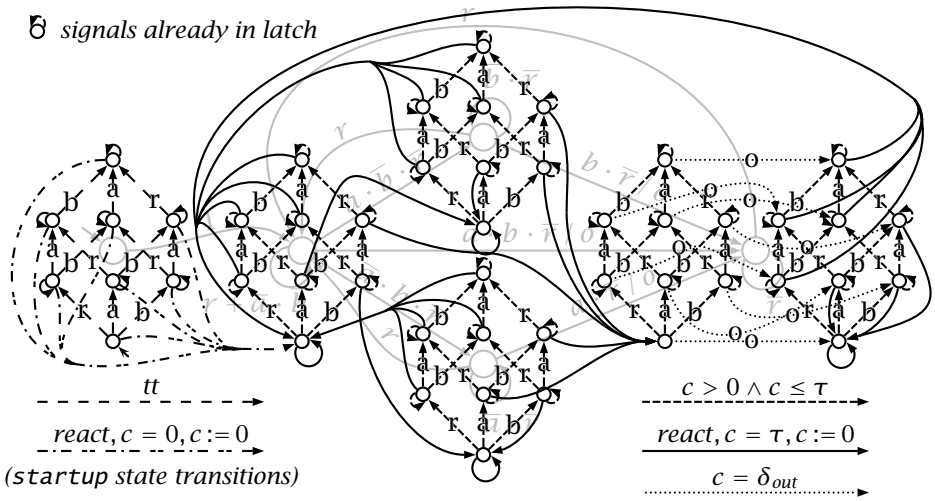
(a) Argos

```
module ABRO:
input a, b, r;
output o;

loop
  [
    await a
  ||
    await b
  ];
  emit o
each r

end module
```

(b) Esterel [Ber00a, §3.1]



(c) Boolean Mealy Machine



(d) Timed Automata (*trigger* = *sample*)

Figure 3.8: Translation of ABRO to a sample-driven system

Definition 3.2.2
A valuation is lifted to an *interpretation* with respect to J over Boolean formulas in I, $\nu_J : \mathrm{AB}(I) \longrightarrow \mathbb{B}$, by structural induction. ■

Definition 3.2.3
Given a BMM $B = (S, s_0, I, O, T_B)$ and parameters $trigger \in \{sample, event\}$, $\tau \in \mathbb{Q}^{\geq 0}$, and $\delta_{out} \in \mathbb{Q}^{\geq 0}$, such that (C1) $\delta_{out} \leq \tau$, and (C2) $trigger = event \vee \tau > 0$, define a timed automaton $TA^{trigger}_{\tau,\delta_{out}} = (L, l_0, K, \mathsf{inv}_K, T)$ on $A = I \dot\cup O \dot\cup \{react\}$:

- $L = (S \dot\cup \{startup\}) \times \mathcal{P}(I) \times \mathcal{P}(O) \times \mathbb{B}$
- $l_0 = (startup, \emptyset, \emptyset, \mathsf{false})$
- $K = \{c\}$
- $\mathsf{inv}_K(s, J, P, u) = \begin{cases} c \leq \delta_{out} & \text{if } P \neq \emptyset, \\ c \leq \tau & \text{if } trigger = sample, \\ c \leq 0 & \text{if } u = \mathsf{true}, \\ \mathsf{true} & \text{otherwise.} \end{cases}$
- T is the smallest set defined by the conjunction of:

  1. $i \in I$
     $\implies (s, J, P, \mathsf{false}) \xrightarrow[i]{c>0 \wedge c \leq \tau \;\; \emptyset}_T (s, J \cup \{i\}, P, \mathsf{false})$
  2. $trigger = event$
     $\implies (s, \emptyset, P, \mathsf{false}) \xrightarrow[i]{c>\tau \;\; \{c\}}_T (s, \{i\}, P, \mathsf{true})$
  3. $i \in I$
     $\implies (s, J, P, \mathsf{true}) \xrightarrow[i]{c=0 \;\; \emptyset}_T (s, J \cup \{i\}, P, \mathsf{true})$
  4. $(\exists m \in \mathrm{AB}(I).$
     $\wedge\ (trigger = sample \vee J \neq \emptyset)$
     $\wedge\ (s, m, P, s') \in T_B$
     $\wedge\ \nu_J(m))$
     $\implies (s, J, \emptyset, \mathsf{false}) \xrightarrow[react]{c=\tau \;\; \{c\}}_T (s', \emptyset, P, \mathsf{false})$
  5. $(\exists m \in \mathrm{AB}(I).$
     $\wedge\ (s, m, P, s') \in T_B$
     $\wedge\ \nu_J(m))$
     $\implies (s, J, \emptyset, \mathsf{true}) \xrightarrow[react]{c=0 \;\; \emptyset}_T (s', \emptyset, P, \mathsf{false})$
  6. $o \notin P$
     $\implies (s, J, P \cup \{o\}, \mathsf{false}) \xrightarrow[o]{c=\delta_{out} \;\; \emptyset}_T (s, J, P, \mathsf{false})$
  7. $trigger = event$
     $\implies (startup, \emptyset, \emptyset, \mathsf{false}) \xrightarrow[i]{\{c\} \;\; \mathsf{true}}_T (s_0, \{i\}, \emptyset, \mathsf{true})$
  8. $trigger = sample$
     $\wedge\ i \in I$
     $\implies (startup, J, \emptyset, \mathsf{false}) \xrightarrow[i]{\mathsf{true} \;\; \emptyset}_T (startup, J \cup \{i\}, \emptyset, \mathsf{false})$
  9. $(\exists m \in \mathrm{AB}(I).$
     $\wedge\ trigger = sample$
     $\wedge\ (s_0, m, P, s) \in T_B$
     $\wedge\ \nu_J(m))$
     $\implies (startup, J, \emptyset, \mathsf{false}) \xrightarrow[react]{c=0 \;\; \{c\}}_T (s, \emptyset, P, \mathsf{false})$ ■

Each state of the resulting timed automaton is a 4-tuple. The first element tracks the state of the original BMM. A distinguished marker, *startup*, indicates that the program is in the initial state s_0 and that a reaction has yet to occur, in which case the timing of transitions is different. The second and third elements are latches for inputs and outputs, respectively. The fourth component is a flag used to model actions that must

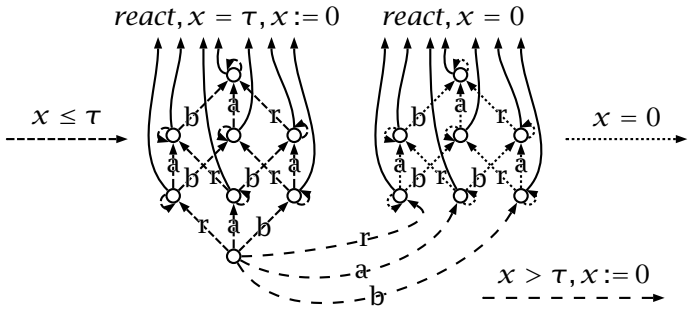


Figure 3.9: Urgent inputs (*trigger* = *event*)

occur urgently. Initially, the underlying state is set to *startup*, both latches are empty, and transitions are not urgent.

A single clock c measures the time elapsed since system startup to the first reaction, and thereafter the time since the last reaction.

The transitions coordinate latching and reacting while respecting the timing and triggering parameters. The end of a reaction is not marked by a special action but a similar effect could be obtained by adding a distinguished output to each transition of the original BMM.

The conjuncts of the definition characterizing T can be understood as follows:

1. Input latching after a reaction in the period $(0, \tau]$.
2. If event-driven, a first input after the period $(0, \tau]$ causes a reaction urgently.
3. Input latching during an urgent reaction.
4. For sample-driven systems, or event-driven systems where at least one input has been latched, reactions occur when $c = \tau$.
5. Urgent reactions must occur before time can progress.
6. Outputs occur one-by-one when $c = \delta_{out}$.
7. An initial input triggers an urgent reaction in an event-driven system.
8. Inputs are allowed at $t = 0$ in sample-driven systems.
9. A reaction occurs at $t = 0$ in sample-driven systems.

The *react* symbol marks the beginning of a reaction. All inputs that have occurred since the last reaction are grouped into a single synchronous event. Inputs that occur afterward are latched for the next reaction, thus maintaining the atomicity of reactions. Multiple input events may occur simultaneously with *react* and will have the same time tag. There are essentially three options for treating such inputs:

1. With a *closed time guard* (conjunct 1, $c \geq 0$) they are latched for the next reaction.
2. With an *open time guard* (conjunct 1, $c > 0$) they occur in the immediate reaction.
3. Alternatively, the inputs at the instant of reaction are split into those that occur before *react* and form part of the immediate reaction, and those that occur afterward and are thus latched for the next reaction.

Option 1 is problematic for event-driven systems where inputs may in fact cause simultaneous reactions. It may be reasonable for sample-driven systems and could be used to add a feedback delay of sorts. For option 3, the meaning of an order on events sharing the same time tag must be considered. At present, this option is rejected, though further consideration may be warranted. Option 2 is adopted in Definition 3.2.3. It

excludes timed words where simultaneous inputs occur after *react*. This choice necessitates special cases—conjuncts 7, 8 and 9—for the initial state where $c = 0$ even though a reaction has yet to occur.

Event-driven systems react as soon as possible once an input event occurs. If an input is received while a reaction is being processed, the corresponding reaction will occur when $c = \tau$ (conjunct 4). Otherwise, a reaction occurs simultaneously with the triggering inputs by switching to an urgent state, the last component of the state tuple is set to true by conjunct 2, where inputs may still be latched (conjunct 3) but time may not advance until *react* has occurred (conjunct 5).

Outputs are emitted δ_{out} units after a reaction. While it would be possible to represent this emission with a single symbol, chosen from $\mathcal{P}(O)$, the outputs occur individually in arbitrary order, and all share the same time tag (conjunct 6). Timed words where some outputs are not emitted are forbidden by the location invariant. Or, if the location invariant were omitted, by constraining *react* transitions to those states where the output buffer component is empty (conjuncts 4 and 5).

The state before any reaction has occurred is marked by the *startup* element. For event-driven systems, conjunct 7 ensures that the first event triggers a simultaneous reaction even if it occurs when $c = 0$. For sample-driven systems, conjunct 8 permits input latching at time $c = 0$ and conjunct 9 mandates that the first sample-driven reaction occurs simultaneously. An additional parameter specifying the time of the first sample-driven reaction could be introduced. Only the clock guard of the transition implied by conjunct 9 need change. A simpler alternative would be to assume that the first reaction occurs at or after $c = \tau$. In this case, conjuncts 7, 8 and 9 would be omitted, L would become $S \times \mathcal{P}(I) \times \mathcal{P}(O) \times \mathbb{B}$ and l_0 would become $(s_0, \varnothing, \varnothing, \mathsf{false})$.

Transitions become urgent due to the location invariant inv_K in three cases. First, when the output latch is not empty and δ_{out} units have passed since the last reaction. Second, for sample-driven systems when τ units have passed since the last reation. And, third, when the urgent flag is set. If the location invariant were omitted, the timed automata resulting from Definition 3.2.3 would only specify safety constraints. Liveness could be introduced by imposing a Büchi condition for the set of accepting states $F = \{(s, \varnothing, \varnothing, \mathsf{false}) \mid s \in S\}$, which would force reactions to occur infinitely often, but for event-driven systems, it would unnaturally exclude timed words where inputs do not occur infinitely often.

Definition 3.2.3 is effectively an interface between a dense-time model where inputs and outputs are interleaved, though simultaneous events may have identical time tags, and the synchronous model where multiple, distinct inputs may be consumed at a reaction as a single event. Implementation details are modelled to some degree. The abstract program is effectively executed by evaluating the guards of potential transitions against the contents of the input latch, that is by $\nu_J(m)$ for each (s, m, O, s') matching the current state.

The mapping defined in Definition 3.2.3 oversimplifies several issues. Inputs are latched instantaneously as soon as they occur. Event detection is idealized; for instance, to detect changes it may be necessary to continuously sample an input value. Perfect timing is assumed; neither clock digitization nor drift are modelled. And all reactions take an exact and equal amount of time to compute, whereas in many programs the time taken will vary depending on which functions are called in a given state. A consistent execution time could be achieved with extra effort, for instance, by double-buffering outputs and updating them on a timer.

3.3 Implementation of an Argos block

The Argos block executes an Argos program inside a Simulink model. The timing details expressed in the transformation to a timed automaton must be realized in terms of the concepts and API of Simulink.

The first part of this section, §3.3.1, describes the structure of the implementation and the timing logic. The second part, §3.3.2, describes practicalities of the software.

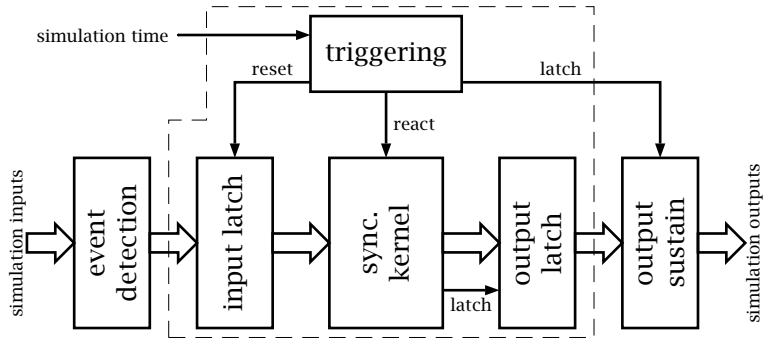


Figure 3.10: Simulation components

3.3.1 Embedding within Simulink

The timed automata produced by the *timing translation* of Definition 3.2.3 must now be related to the operations of Simulink. There are at least three possible approaches:

1. Adopt a semantics for Simulink either in terms of the simulation engine, or of the presumed intent of a model. Timed automata could then be interpreted within the given setting.

2. Relate Simulink models, via informal descriptions of the tool and test simulations, to another better defined model of computation. There have been several such proposals. One of them [TSCC05] shows how to translate some types of Simulink models into the synchronous language Lustre. The present timing translation could be adapted to produce Lustre programs, although questions would remain about how best to encode continuous features.

3. Restrict attention to a single component and view its interaction with Simulink through a mix of conceptual and low-level operations.

The last approach was taken. It provides sufficient guidance to implement a block or model, but does not directly describe interactions with other components nor the functioning of an entire model.

The realization in Simulink of the desired timing behavior will comprise the subcomponents depicted in Figure 3.10. Timed automata produced by the timing translation describe the *synchronous kernel* that implements the original program, the *input latch* and *output latch*, and the *triggering* logic. Some additional *event detection* is necessary to translate Simulink signal changes into discrete events, for example by polling signal values to detect rising and falling edges.

The *output sustain* subcomponent of Figure 3.10 interfaces the pure signals used within synchronous programs to other Simulink components. The external effect of a synchronous signal is implementation-specific. It could, for instance, trigger a Simulink function call, or generate a rising, falling, or pulse signal. Or it could be treated as valued, emitting a zero when absent and a one when present, although it then becomes impossible to distinguish consecutive signal emissions.

While it would be possible to implement each subcomponent of Figure 3.10 separately, using a mix of built-in and custom Simulink blocks, it turns out that implementing most of the subcomponents in a single custom block makes it easier both to encode the subtleties of the timing behaviour and to use the tool in practice. Thus, the components inside the dashed region—input latch, triggering, synchronous kernel, and output latch—are implemented in a single custom Simulink block to the specification of Definition 3.2.3

Implementing the timing behaviour requires care. The clock c of the timed automaton model must be simulated in terms of the current time in Simulink t, which is provided to a block whenever one of its functions is called. The block must calculate when next it should be called and communicate this to Simulink through sample times and zero-crossings. Potential algebraic loops must also be reported.

The clock c of a timed automaton produced according to Definition 3.2.3 is tied to the simulation time t by tracking the clock value and previous sample time t_p as discrete state variables. At any sample instant, $c = x_c + (t - t_p)$, where x_c is the stored clock value. The next clock value to be stored x'_c depends on whether a reaction has occurred,

$$x'_c = \begin{cases} 0 & \text{if } \mathit{is\_reaction} \\ c & \text{otherwise,} \end{cases}$$

where at any sample point, after processing inputs, *is_reaction* is defined as

$$\mathit{is\_reaction} = \begin{cases} c = \tau \vee t = 0 & \text{if } \mathit{trigger} = \mathit{sample} \\ (c \geq \tau \vee s = \mathit{startup}) \wedge J \neq \emptyset & \text{if } \mathit{trigger} = \mathit{event} \end{cases}$$

where $s = \mathit{startup}$ is true only until the first reaction and J represents the input latch.

The next required sample hit t_v is defined as

$$t_v = \begin{cases} t + \delta_{out} - x'_c & \text{if } x'_c < \delta_{out} \\ t + \tau - x'_c & \text{if } x'_c \geq \delta_{out} \wedge (\mathit{trigger} = \mathit{sample} \vee J \neq \emptyset) \\ \infty & \text{otherwise} \end{cases}$$

The block uses port-based inherited sample times to ensure that all input changes are detected and latched.

The technique for scheduling reactions and output emissions depends on the value of *trigger*.

When $\mathit{trigger} = \mathit{sample}$, block-based sample times that depend on the δ_{out} parameter are used:

$$\mathit{blocktimes} = \begin{cases} \{[\tau, 0]\} & \text{if } \delta_{out} = \tau \vee \delta_{out} = 0 \\ \{[\tau, 0], [\tau, \delta_{out}]\} & \text{otherwise} \end{cases}$$

where a sample time is written $[\mathit{period}, \mathit{offset}]$.

When $\mathit{trigger} = \mathit{event}$ and $\tau = \delta_{out} = 0$ inherited sample times are sufficient to schedule output emissions, otherwise the implementation is more involved. Variable sample times are not appropriate because it cannot be known when triggering inputs will occur. Instead two zero crossing functions are defined: $Z_1 = c - \delta_{out}$ and $Z_2 = c - \tau$. These identify the instants when outputs are emitted and those where reactions occurs as significant events.

When $\delta_{out} = 0$ outputs depend instantaneously on inputs and may thus contribute to algebraic loops within a model. The block warns Simulink of this possibility.

3.3.2 Practicalities

A prototype Argos tool chain was designed and implemented to evaluate the ideas and approach just described. The tool chain ultimately provides two of the three layers of a synchronous language implementation: a *reactive kernel* defined by a compiled Argos program and an *interface* between Simulink signals and synchronous signals. External *data handling* is not supported.

The tool chain comprises Simulink, a third-party C compiler, and three custom programs: argostoflow, flowtoc, and syncblock. The way they work together to simulate an Argos program within a Simulink model is shown in Figure 3.11.

The tool chain would ideally include a graphical editor and animator for Argos programs. Such features were not, however, central to this research. Argos programs must instead be provided in the textual format described in Appendix D.

The argostoflow compiler transforms textual Argos programs into Boolean equations using an existing technique [MH96]. It is implemented in C using a generated parser and a BDD library. Reachability analysis is performed to ensure that programs are deterministic and input-enabled for every reachable state and any assignment of inputs. Aside from the calculation of reachable states, constructing BDD transition relations is also sometimes prohibitively expensive. A realistic compiler would include

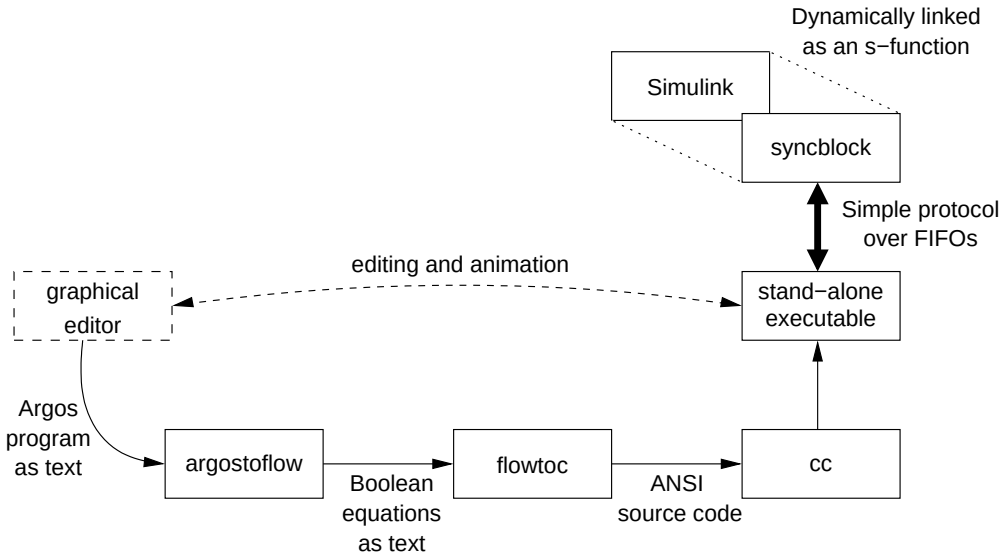


Figure 3.11: Tool chain for Argos block

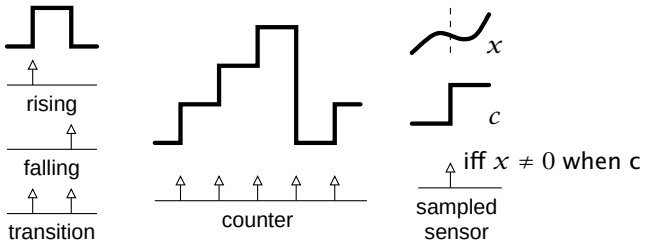


Figure 3.12: Mapping Simulink signals to logical signals

techniques for handling partitioned transition relations, or, more likely, employ less expensive algorithms for approximate analysis.

The Boolean equations are transformed by flowtoc into a C program that can be compiled and linked with a library that implements a simple stream-based interface for querying program properties and computing reactions. The resulting executable provides the reactive kernel.

The syncblock component is a Simulink s-function that provides a bridge between a Simulink model and an Argos program. It launches the compiled Argos program and communicates with it through two FIFO channels. Dynamic linking would be faster, but when it was used in an earlier version, it was found to be less robust and unable to adequately accommodate updates of compiled Argos programs.

The syncblock maps to and from Simulink signal values and pure Argos signals. While any change in the value of a Simulink signal could be regarded as a triggering event, the Argos block only detects the events shown in Figure 3.12. A rising event occurs when an input changes from zero or false to one or true in consecutive simulation steps. Falling and transition events are defined likewise. The counter mode is similar: events are registered whenever the input signal is incremented by at least one or decremented by more than one. In sampled mode an input signal is coupled with a clocking signal. The former is read when an event occurs on the latter. Non-zero sampled values are interpreted as true signals.

Block outputs are set to one when the corresponding signal is present, and zero when it is absent. They are latched between reactions. Program states can be linked to output signals, but given that Argos state names have no semantic value [MR01] this is provided as a debugging technique only.

Instances of the syncblock are configurable through the parameters defined in the timing translation.

3.4 Experience

Simulink is distributed with a number of example models that demonstrate the features of Stateflow. In this section, two of them, sensor failure detection in an automotive fuel controller and a bang-bang temperature controller, are first described and then reimplemented in Argos using the Argos block. The aim is to demonstrate the tool set and to compare its practical aspects with those of Stateflow.

3.4.1 Sensor failure detection

The Mathworks sensor failure detection example models an automotive fuel supply subsystem which determines the rate of fuel supply to an engine based on input from several sensors. Interactive switches allow users to simulate the effect of failed sensors on the subsystem's performance. In this subsection, both the original example and a reimplementation of the Stateflow control logic using the Argos block are described. The analysis focuses on the differences between the Stateflow and Argos models, and on the practicalities of using the Argos block.

3.4.1.1 Original model

The original Mathworks model is shown in Figure 3.13. The upper half, Figure 3.13a, shows the top-level Simulink model and the fuel rate controller subsystem.

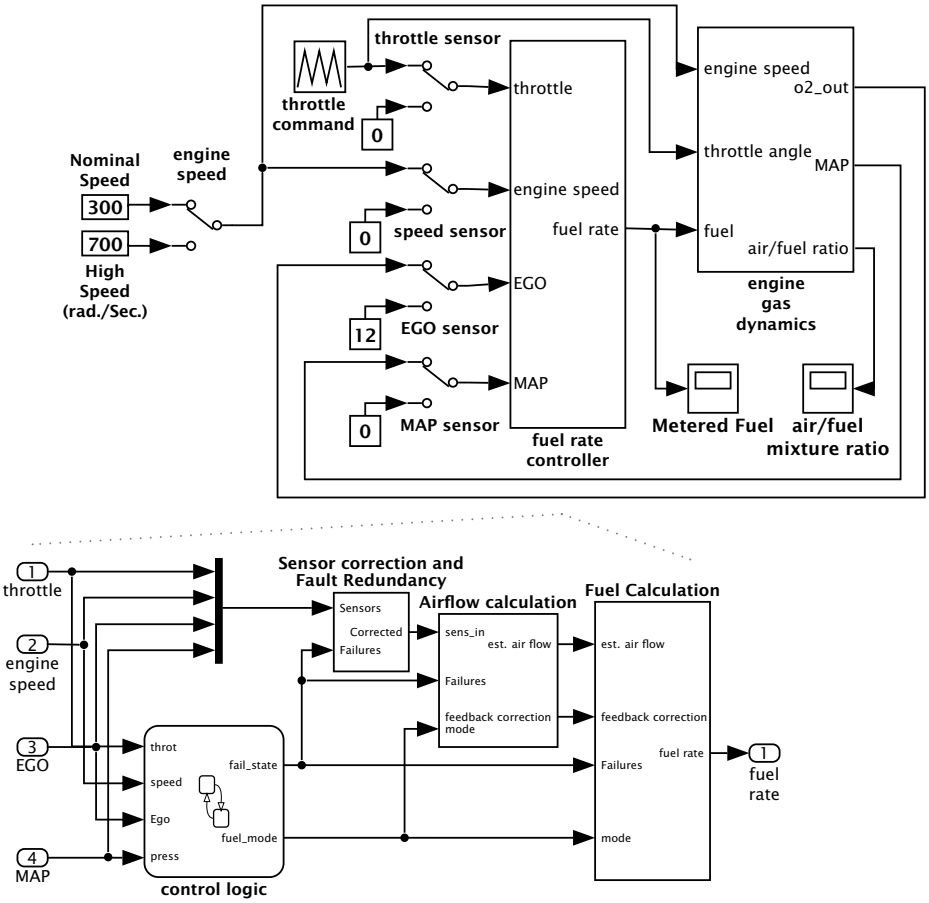
There are three parts to the top-level model: a discrete controller, a model of the environment, and components for manipulating and analysing the simulation. The discrete controller is wholly contained within the *fuel rate controller* block, whose contents are shown underneath. It will subsequently be described in detail. The environment is modelled within the *engine gas dynamics* block, it comprises mathematical models—continuous integration, multipliers, non-linear effects like thresholding, etcetera—of a throttle and manifold.[15] The other smaller blocks at left change the simulation parameters. The *engine speed* switch chooses between two engine speed constants. The *throttle* is modelled by a saw-tooth waveform; as if an accelerator peddle were being alternately pushed in and then released. The four central switches toggle the discrete controller inputs between correct sensor signals and erroneous constant values that indicate sensor failure. There are four sensors: throttle position, engine speed, Exhaust Gas Oxygen (EGO), and Manifold Absolute Pressure (MAP). The two smaller blocks at bottom right chart signal values against time as a simulation progresses: *Metered Fuel* is the control signal produced by the discrete controller, and *air/fuel mixture ratio* is the state variable of interest.

The discrete controller subsystem is shown in the lower part of Figure 3.13a. The sensor inputs are connected at left and the control output at right. There is a Stateflow block and three subsystem blocks. The subsystem blocks contain discrete dataflow networks—unit delays, table lookups, triggered subsystems, etcetera. The *Sensor correction and Fault Redundancy* block tries to compensate for sensor failures, which are detected by and communicated from the Stateflow block, by estimating the missing values. The *Airflow calculation* and *Fuel Calculation* blocks calculate control laws to determine the required rate of fuel supply.

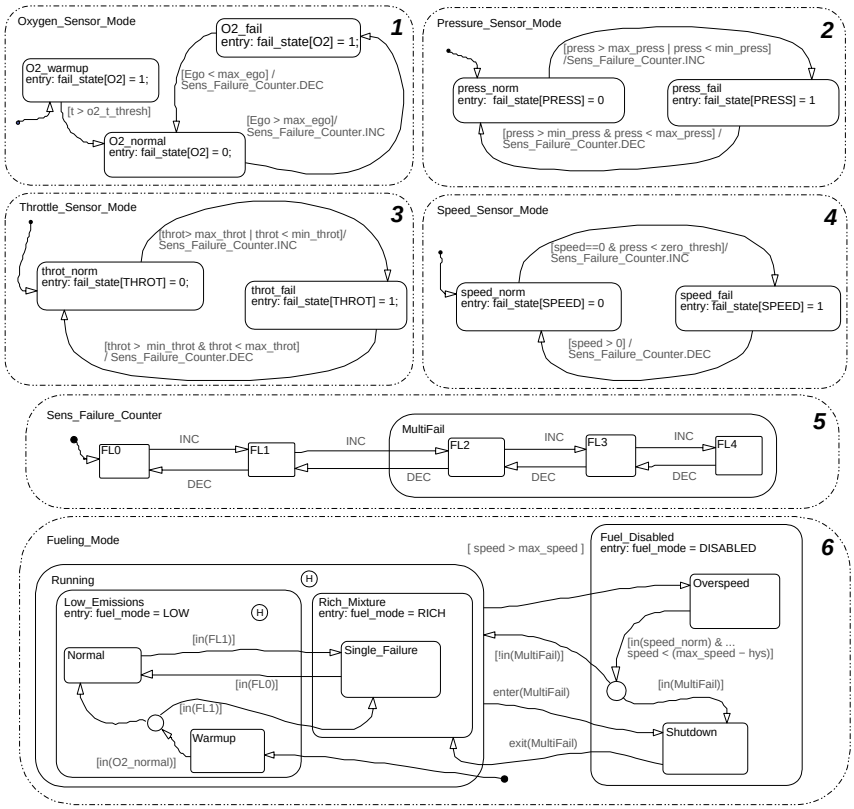
The Stateflow block is sampled and executes every 0.01 units of simulation time. The block contains the diagram shown in Figure 3.13b. It comprises six concurrent state machines. Each is numbered according to a priority derived from its relative position.

Each of the top four machines tracks the state of a sensor input, setting a corresponding element in the `fail_state` output vector to 0 if the sensor value is reliable, and to 1 otherwise. The oxygen sensor monitor begins in the *O2_warmup* state, where the oxygen sensor value is marked unreliable until the simulation time exceeds a constant value. After warming up, the oxygen sensor monitor has behaviour similar to the other three monitors. If the sampled sensor value is not within an expected range, the transition to the *fail* state is taken, sending an *INC* event to the *Sens_Failure_Counter*

[15] A manifold is piping that supplies fuel and air to the combustion cylinders of an engine.



(a) Simulink Model (top-level model, above, and fuel rate controller subsystem, below)



(b) Stateflow Controller

Figure 3.13: Fault-tolerant Fuel Controller [The03b]

state machine, and the sensor is marked as failed in the *fail_state* vector. If the sampled sensor value later returns to the expected range, the transition to the *normal* state is taken, sending a *DEC* event to the *Sens_Failure_Counter* state machine, and the sensor is marked as reliable in the *fail_state* vector. The expected value of the speed sensor also depends on the value of the pressure sensor.

The *Sens_Failure_Counter* state machine counts the number of failed sensors by reacting to *INC* and *DEC* events. The *MultiFail* superstate is entered when two or more sensors have failed. Its status is monitored by the *Fueling_Mode* state machine.

The count of failed sensors is integral to the *Fueling_Mode* state machine which determines whether the *fuel_mode* output is *LOW*, *RICH*, or *DISABLED*. This state machine is the most complicated, having a deep hierarchy of states with multi-level transitions between them. It is normally in the *Running* state, but if the *speed* input exceeds the *max_speed* constant, or if *Sensor_Failure_Counter* enters the *MultiFail* state then it transitions to the *Fuel_Disabled* state.

In the absence of excessive speed values and multiple failures, the *Running* mode begins in the *Warmup* substate until the oxygen sensor comes online. It then alternates between a substate where the *fuel_mode* is low and one where it is rich, depending on whether there are, respectively, no detected sensor failures or a single detected failure. The history junctions ensure that the active substate is remembered if control enters and then returns from the *Fuel_Disabled* state.

If an excessive speed is detected, the *Fueling_Mode* state machine enters the *Overspeed* substate, where it remains until the speed sensor is marked as reliable and its value has dropped below the maximum by a certain margin. If multiple failures are detected, the *Shutdown* substate is entered, but only if—according to the relative priority of transitions—*speed* is less than or equal to *max_speed*. Interestingly, if multiple sensor failures are first detected, and then the speed exceeds the limit, and then the *MultiFail* state becomes inactive, control may return momentarily to the *Running* state before a continued excessive speed value would trigger a transition to *Overspeed*.

3.4.1.2 Argos controller

The Stateflow controller, the block labelled *control logic* in Figure 3.13a, can be replaced with the Argos block running an Argos program, but additional interfacing is required. The replacement subsystem is shown in Figure 3.14a. It has the same inputs and outputs as the original Stateflow block.

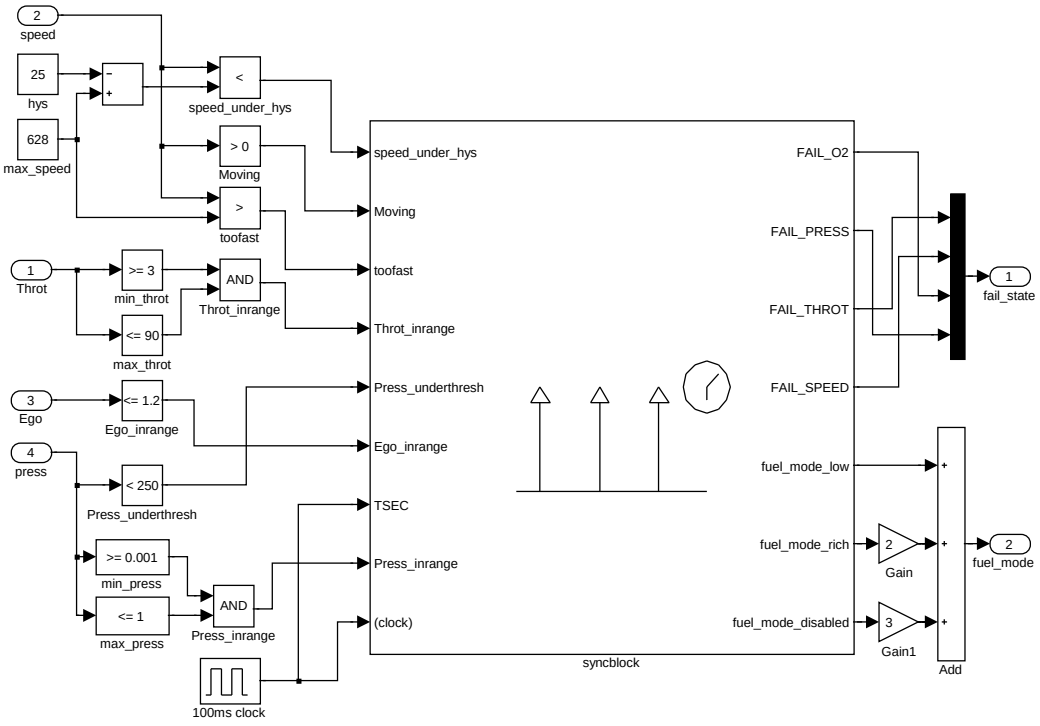
The subsystem inputs are connected through a network of constant, comparison, conjunction, and arithmetic blocks that map integer sensor values into boolean values. For instance, the Stateflow guard `press > min_press & press < max_press` is replaced by comparison blocks for the `min_press` and `max_press` values, and an `AND` block to combine the results into a boolean `Press_inrange` input. This same input is negated to serve for the guard `press > max_press || press < min_press`. The Stateflow and Argos models differ slightly in their treatment of pressure values equal to either of the constants: these values do not satisfy either of the Stateflow guards but they are valid in the Argos model. The other guard expressions are treated similarly.

The external predicates are necessary because the Argos block only executes Pure Argos programs. Such expressions are more conveniently expressed as text, as is done in Stateflow, and as could be done using Argos with variables [MR01, §4.5].

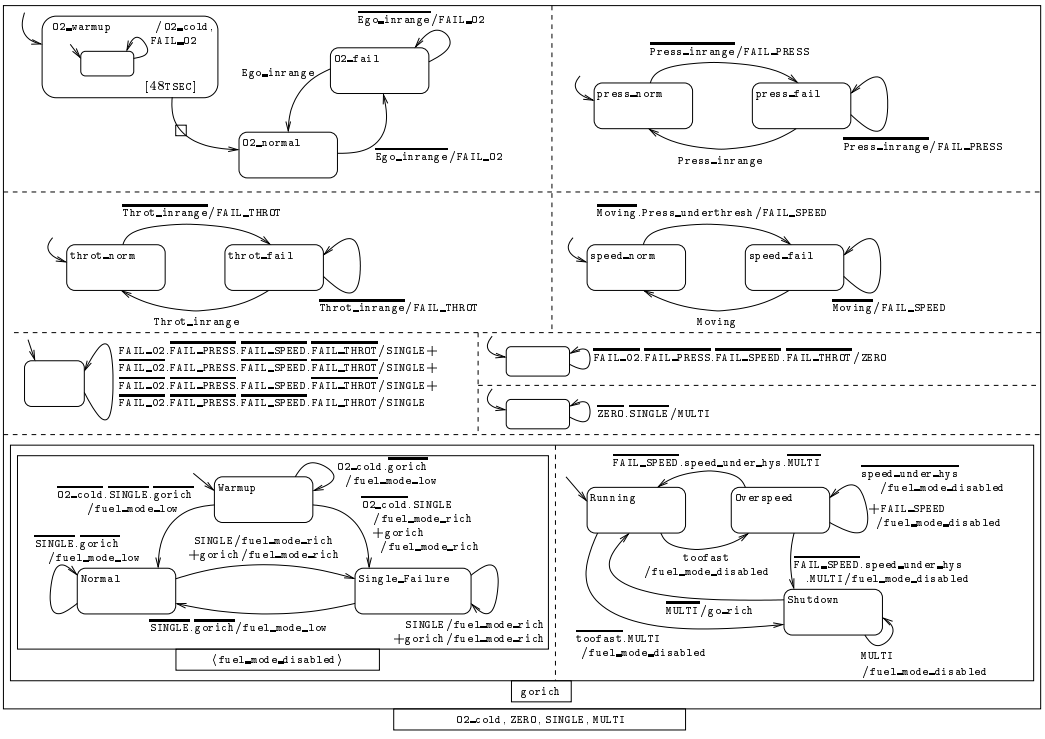
The subsystem outputs are also filtered through external blocks. The sensor fault values are combined into a vector using a multiplexor. The three mutually-exclusive fuel mode outputs are combined into an integer value using a summation block. While the overhead is small for this example, it could be prohibitive for larger systems.

The triggering of the Argos block in the diagram is slightly unusual because it was modelled in a prototype [BS05] that preceded the three-parameter timing model §3.2. The details can, nevertheless, be interpreted in the three-parameter model, and they make little difference to this comparison between modelling styles.

The Argos block is event-driven, *trigger* = *event*. There is a single triggering `(clock)` input, visible at bottom left. A rising edge on this distinguished input triggers the block immediately: $\tau = 0$. Reaction time is not modelled: $\delta_{out} = 0$. The other



(a) Simulink Interface



(b) Argos Controller

Figure 3.14: Fuel control subsystem in Argos

inputs are sampled at the instant of triggering, a signal is marked present only when the corresponding input is not zero. These inputs are thus interfaced like sensors but treated internally as signals.[16] The triggering (clock) input is driven by a square wave signal with a period of 100ms, so the Argos block executes at the same rate as the Stateflow original. Were the block truly sample-driven, that is with *trigger* = *sample* and $\tau = 0.100$, there would be no (clock) input, and, transitions of the other input values would be latched between reactions rather than recorded at the instant of reaction. Such behaviour would be unfaithful to the original model.

The Argos block runs the program shown in Figure 3.14b.[17] The Argos program mimics the basic structure of the Stateflow original, having three component groupings, namely, from top to bottom, four individual sensor monitors, a means for detecting multiple failures, and logic for determining the fuel mode. Within each of the groupings, however, the Argos and Stateflow versions differ in several details.

Each of the four sensor-monitoring components in the Argos program has essentially the same structure as the corresponding component in the Stateflow program. The Argos components differ both in the way timing is expressed, and by using self-loops to sustain status signals instead of INC/DEC signals and array assignments.

The oxygen sensor must remain in a *warmup* state for 4.8 seconds after initialization. This is expressed in the Stateflow original by the guard t > o2_t_thresh, where o2_t_thresh is a workspace variable with value 4.8 and t is the simulation time. Expressions relative to the simulation time may be convenient but they suffer from at least three deficiencies. First, the time is relative to the start of the simulation, not to the initialisation of the controller. Second, there is, in general, a risk of designing systems that rely on implicit synchronisation. Third, and most importantly, the relationship between such timing specifications and program triggering is unclear—there is a blurring of real and logical time.[18]

Timing in Argos, as in all synchronous languages, is strictly logical. The oxygen sensor *warmup* period is measured by counting out 48 TSEC events. The TSEC signal is present at every reaction because it is a sampled input connected to the triggering 100ms clock signal. The reason for this redundancy, since the program could as well count 48 implicit tick events, is the conceptual isolation of triggering and timing.

The sensor-monitoring components in the Stateflow program mark a sensor failure by setting a flag in the fail_state array. In the Argos version, there is a signal for each monitored sensor—FAIL_O2, FAIL_PRESS, FAIL_THROT, and FAIL_SPEED—which is emitted when values of that sensor go out of range and then sustained while necessary by a self-loop transition. These signals are merged into an array by a multiplexor block in the interface subsystem. The FAIL_O2 signal is sustained in the *warmup* state by a self-loop on a refining state so as not to reset the timer at each reaction.

The sensor failure signals are also used in the Argos program to detect multiple failures, instead of counting INC and DEC signals as the Stateflow version does. Such counting is not possible anyway in Argos since signal emissions are conceptually simultaneous; multiple signal emissions cannot be distinguished from a single emission.[19] Instead, three conditions are evaluated at each reaction, by self-loop transitions in parallel state machines, to determine whether zero, one, or many failures have occurred. These failures are communicated by broadcasting one of the signals ZERO, SINGLE, and MULTI. These signals, and O2_cold, are visibly encapsulated as local signals at the top level of the program. In Stateflow, such declarations of scope are also possible, but they are not evident from the state diagrams alone.

The fuel mode component is the most difficult to translate because it uses history junctions and multi-level transitions, and Argos possesses neither. Instead, the *Running* and *Fuel_Disabled* superstates of the Stateflow original are implemented as two parallel state machines in the Argos version, and the former is enclosed in an inhibition operator, the rectangle with label ⟨fuel_mode_disabled⟩, so that its operation is

[16] There is less distinction between boolean sensor inputs and pure events in sample-driven Argos and Lustre programs than there is, for instance, in event-driven Esterel programs.

[17] The textual version of this program in Appendix E was created manually.

[18] This topic will be resumed in Chapter 6.

[19] There is no analogue in Argos for the signal combination functions of Esterel.

suspended in reactions where the latter emits the signal `fuel_mode_disabled`. This standard technique [MR01, §6.1] has the same effect as the combination of refined exclusive states and history junctions: the suspended behaviours are not involved in reactions, but their state is maintained. The 'disabled' component also influences the 'running' component through the `gorich` signal, which is local to the pair.

The restructuring of the fuel mode component eliminates one source of multi-level transitions. The other source is treated by duplicating the emission of `fuel_mode_low`, which, as in the sensor monitoring components, replaces the variable assignment of the original, in the *Warmup* and *Normal* states. This duplication is difficult to avoid and some of the clarity of the Stateflow original is lost.

Finally, it is worth noting that many transitions in the Argos fuel mode component are triggered by conjunctions of signals and negated signals. Such triggers explicitly express transition priority. They are irrelevant in Stateflow, where priority is implicit and signals are processed one by one, but they are needed in Argos programs to ensure deterministic behaviour for all input assignments. For example, of the two transitions leaving the `Running` state it is clear which is taken for any combination of the `toofast` and `MULTI` signals when at least one is present. Guard expressions in Stateflow are certainly less cumbersome, but detecting non-determinism and requiring its explicit elimination ensures that a conscious design choice has been made, and results in programs whose behaviour cannot be accidentally altered by incautious rearrangement.

3.4.1.3 Summary

This section has shown that Argos and the Argos block can sometimes function as a replacement for a Stateflow controller, though extra interfacing components are required to work around the limitations of Pure Argos.

The reduced set of programming constructs available in Argos, as compared to Stateflow, is sufficient to express the failure detection controller but a different style of programming is required. There is some evidence that a more concise program could be written in a synchronous language with more features, like Esterel or Safe State Machines. The failure counting component could, for instance, be simplified using combination functions and valued expressions.

The single timing constraint in the Stateflow controller is stated with respect to simulation time t, which is, at least conceptually, dense. While convenient, this mode of expression has disadvantages, particularly because its relation to program execution is not clear. The same constraint is expressed in the Argos version by counting discrete events, as is usual in synchronous programming. These events are related to the dense time of Simulink through convention, viz. signal names, and the timing parameters of the Argos block. The timing parameters were peripheral to this example. They are more central in the next.

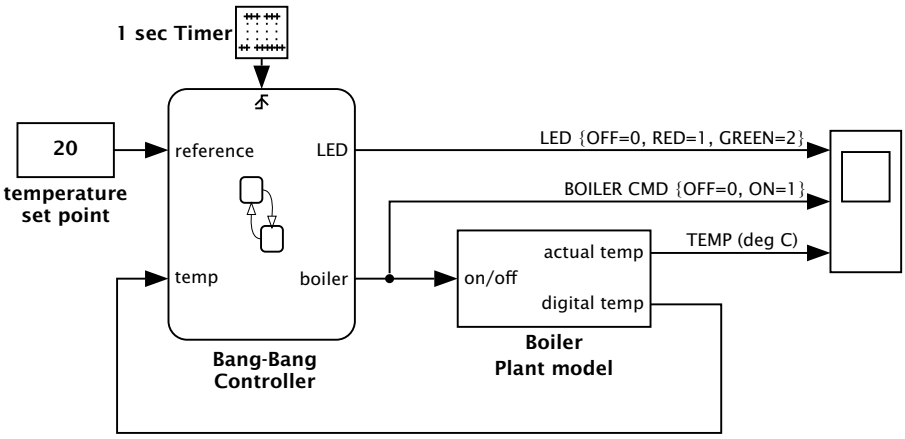
3.4.2 Bang-bang temperature controller

A bang-bang temperature controller tries to maintain a given temperature by switching a heating element on if a measured temperature drops too low, and off again if it becomes too high.

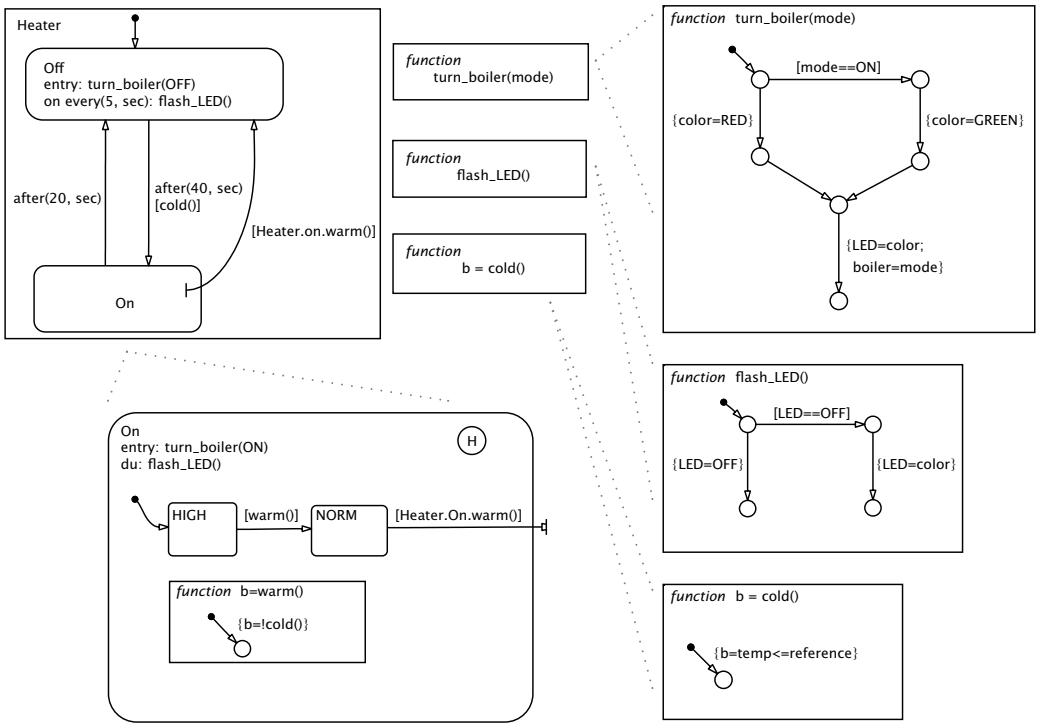
The focus of the temperature controller reimplementation of this subsection is on the generation of simulation results from the two implementation timing parameters.

3.4.2.1 Original model

The original Mathworks Simulink model, Figure 3.15a, has, like the sensor failure model, three parts: a discrete controller, a model of the environment, and components for observing the controller outputs and effect. It is, though, much simpler. The discrete controller, labelled *Bang-Bang Controller*, comprises a Stateflow block, a triggering signal, and a constant parameter. The environment, labelled *Boiler Plant model*, is a subsystem block that encompasses simple models of the heating element and the



(a) Simulink model



(b) Stateflow controller

Figure 3.15: Bang-bang temperature controller [The03a]

actual temperature—a switch and integrator respectively—and a detailed model of a digital thermometer which measures the temperature.

The discrete controller has two inputs, the reference and measured temperatures, and two outputs, a Light Emitting Diode (LED) and a boolean control signal. The LED is off when the system is not operational, or blinking when it is: red and slowly if the heating element is switched off, green and more rapidly if it is switched on. The heating element is switched on while the control signal is true and off while it is false.

The discrete controller program is shown in Figure 3.15b. At the top level there is a state machine with two states and three Flow Diagram functions.

The three Flow Diagram functions are shown to the right of the figure. The first, at top, *turn_boiler*, sets two variables: whether the heating element is on or off and the corresponding LED colour. The second, *flash_LED*, toggles the LED alternately on and off. The third, *cold*, returns true when the measured temperature is less than or equal to the reference temperature.

The state machine begins by entering the *Off* state, which calls *turn_boiler* to switch the heating element off. When the *Off* state is active, the *flash_LED* function is called on every fifth *sec* event—this controller expresses delays relative to input events rather than to the simulation time. The single outgoing transition, to the *On* state, is taken if the *cold* function returns true and at least forty *sec* events have been received.[20] The delay prevents the heating element from being cycled on and off too rapidly.

The *On* state is refined by the two-state machine at the bottom of Figure 3.15b. The heating element is switched on when this state is entered. While the state remains active the *flash_LED* function is called on every triggering, that is, every second. The *On* state is never active for more than twenty seconds. It contains a function called *warm* that simply negates the value returned by the *cold* function. The history junction means that the *HIGH* state is active until the first measurement considered warm is taken, that is until the *On* state is terminated by a reading rather than a timeout. The first warm measurement triggers a transition to the *NORM* state, which becomes the initial state in subsequent activations of *On*. A warm measurement in the *NORM* state triggers the multi-level transition back to *Off*.

3.4.2.2 Argos controller

The bang-bang temperature controller can also be implemented using the Argos block. Like the fuel controller, additional interfacing components and a slightly different programming style are required.

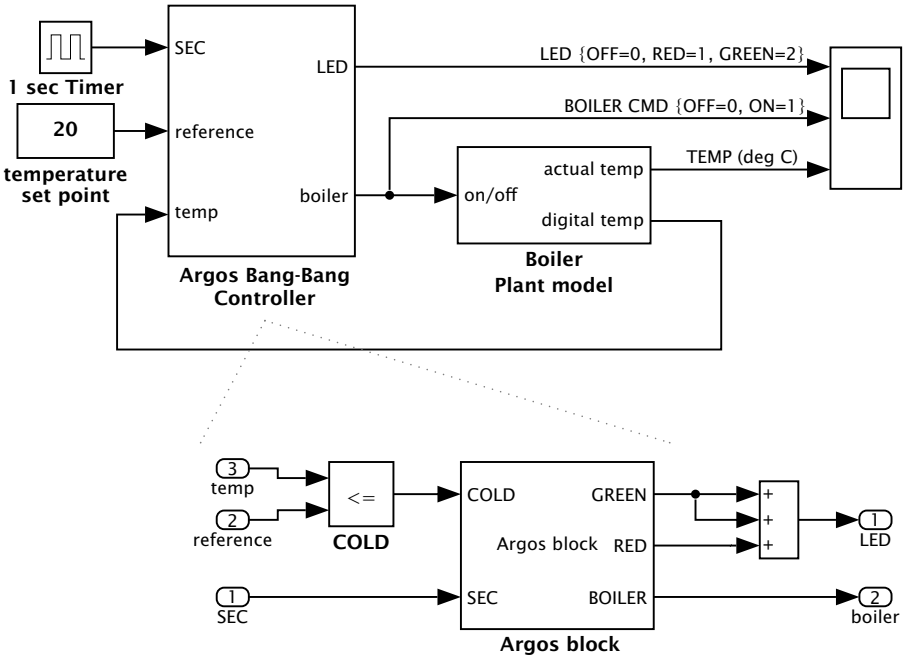
The Simulink model is shown in Figure 3.16a. The original Stateflow block has been replaced by a subsystem block whose contents are shown underneath. The reference and measured temperatures are passed into a comparison block to produce a boolean signal that the Argos block maps to an input signal `COLD`, which replaces the *warm* and *cold* functions of the Stateflow original. Individual outputs for the LED colour, `GREEN` and `RED`, are mapped to a single integer value by a summation block.

The Argos control program is shown in Figure 3.16b (the textual version may be found in Appendix E). It comprises two main states: *On* and *Off*.

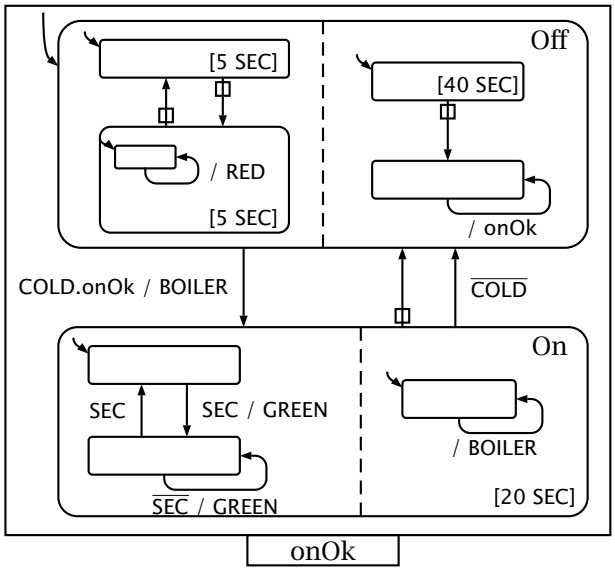
The *Off* state contains two parallel state machines. The one at left causes the LED to blink red every five `SEC` events. The refining state with a self-loop implements a Moore-style output for `RED` (which does not reset the timeout). The state machine at right counts off forty `SEC` inputs and then sustains the local `onOk` signal, which is used in conjunction with `COLD` in the transition from *Off* to *On*. Modelling the LED flashing and minimum delay as refining state machines involves slightly more effort than do the Stateflow equivalents, due to the simpler nature of Pure Argos. This is a disadvantage. On the other hand and more generally, the simplicity allows the interplay of different components to be readily understood in terms of a small number of basic operators, rather than requiring that it be divined through interpretation of user manuals and experimentation with models.

The *On* state is similar. It contains parallel state machines to both blink the `GREEN` signal, and to sustain the `BOILER` output. There are two transitions back to the *Off*

[20] The `after` operator counts events from the last entry into the source state [Mat03c, pp. 7-79–7-80].



(a) Simulink model



(b) Argos controller

Figure 3.16: Bang-bang temperature controller in Argos

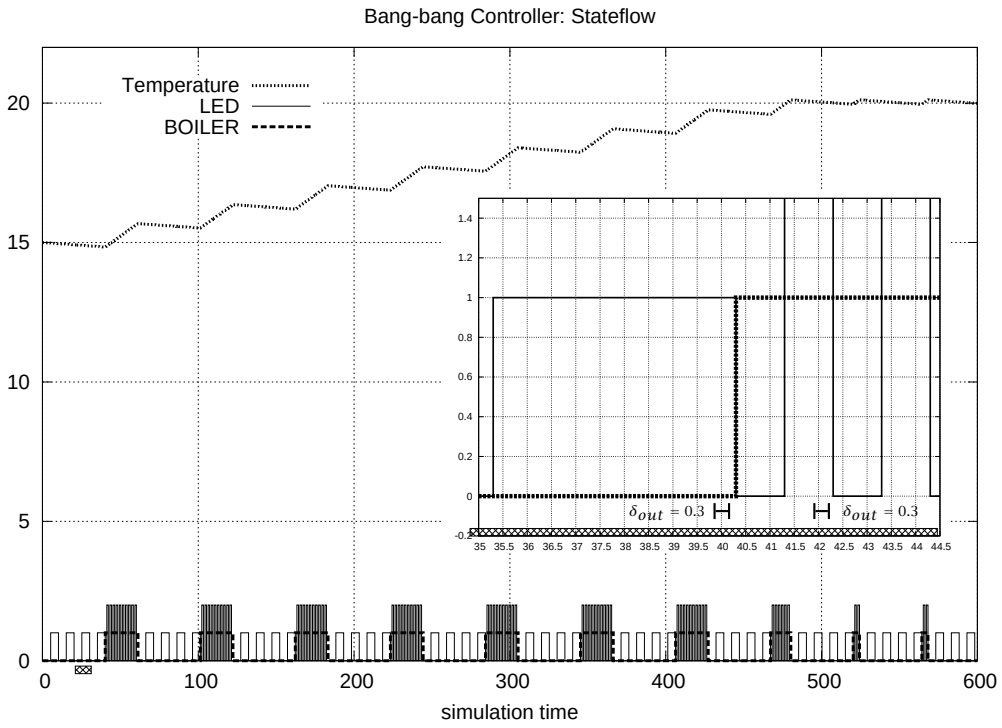


Figure 3.17: Argos simulation results

state: one for a `20 SEC` timeout, the other for when the `COLD` signal is absent. The controller's behaviour differs from the Stateflow original because the first activation of the boiler ends after a single 'warm' reading. Implementing the special case is tedious in Argos. One technique is to add another state machine in parallel at the top level. It would have two states, with a single transition from the initial state to the second state when both `BOILER` and ¬`COLD` are present and a self-loop on the second state to sustain a new local `offOk` signal, which would be required along with ¬`COLD` to make a transition back to *Off*. There seems to be the same trade-off between modelling convenience and language simplicity.

Simulation results for parameters *trigger* = *event*, $\tau = 0.8$, and $\delta_{out} = 0.3$, with `SEC` as a rising edge input and `COLD` as a sampled input, are shown in Figure 3.17. The internal panel shows detail of the signals between 35.0 to 44.5 seconds.

The heater is alternately switched off and then on. When the heater is off, the measured temperature gradually decreases and the LED signal alternates between 0 (off) and 1 (red). When the heater is on, the measured temperature increases and the LED signal alternates more rapidly between 0 (off) and 2 (green). The temperate zig-zags upward until it attains the reference temperature 20°C, which is then maintained.

Note that, due to the controller timing constants, the temperature never increases for more than twenty seconds and always decreases for at least forty seconds. For similar reasons, the shorter LED pulses for the colour red, have a ten second period, while the taller ones for the colour green, more easily seen in the internal panel, have a two second period.

The effect of the δ_{out} parameter is obvious, the outputs from reactions occurring at 40.0 and 42.0 seconds, to choose but two, are not manifest until 40.3 and 42.3 seconds respectively.

3.4.2.3 Summary

The bang-bang temperature controller can also be implemented using the Argos block. Extra interfacing elements and adjustments to programming style are required.

The simulation results show the effect of varying the timing parameters. They also raise questions. How important is it really to capture such small deviations from ideal behaviour? Is an assumption of synchrony sufficient? Certainly, for these idealised examples, the deviations do not seem to be significant. The prototype and examples seem sufficient to propose and demonstrate the approach but they are not convincing in themselves.

3.5 Comparisons with related work

The timing transformation and Argos block were developed with two aims: to account for implementation timing constraints in an abstract model and to integrate a synchronous language into Simulink. In this subsection, four other approaches that share one or both of these aims are discussed.

Various synchronous execution machines [AMP91, AP93, BHHS93] propose structured specifications for the interface layer of synchronous language implementations. They are described in §3.5.1. A translation from Simulink to Lustre [CCM+03, SSC+04] applies synchronous language techniques to Simulink models, as described in §3.5.2. TAXYS [BCP+01, STY03] is a methodology for designing applications in discrete time with Esterel and then showing their correctness with respect to implementations modelled in continuous time with timed automata. It is discussed in §3.5.3. The Almost As Soon As Possible (AASAP) semantics [DWDR04] accounts for idealised implementation parameters in timed automata models; similarities with the present work are discussed in §3.5.4.

3.5.1 Synchronous execution machines

Most synchronous language research focuses on the reactive kernel. The required interface layer is usually considered an unimportant implementation detail. But, not unlike the timing translation of this chapter, there have been attempts [AMP91, AP93, BHHS93] to understand and implement the interface layer systematically.

The interface layer can be understood as a link between an asynchronous environment and a synchronous automaton [AMP91]. It can be described abstractly as an execution machine that is to be implemented for simulation, as on a workstation [AMP91, AP93], or on a Real Time Operating System (RTOS) [AMP91] or microkernel [BHHS93].

The abstract execution machines have distinct components for mapping system events, like interrupts or value changes detected by polling, to signals, for bundling these signals into events, for deciding when to execute the reactive kernel, for doing so, and for collecting and dispatching output signals. Triggering decisions may involve event queueing [AMP91], or custom strategies that account for signal relations, resource management, and quality of service issues [BHHS93]. Additional signals that mark, for instance, the beginning and end of reactions, may be introduced. The abstract execution machine may even itself be specified in Esterel [BHHS93]. Implementations of these machines may involve multiple processes and care is required to preserve the atomicity of reactions and other characteristics of the synchronous semantics. The machines usually support access to asynchronous aspects of the implementation through the **exec** statement of Esterel.

The timing translation and Argos block confront the same issues of embedding a synchronous program into a non-synchronous environment. And similar solutions are proposed: different components for latching inputs and outputs, and explicit logic for deciding when to trigger a reaction. The timing translation goes beyond the asynchronous environments previously considered, and adds timing detail. The formal model and details of its integration into Simulink are also more specific and concrete than previous approaches. The similarities and differences between sample-driven and event-driven execution schemes are better clarified.

3.5.2 Simulink to Lustre

Simulink was originally used only to design and simulate discrete controllers. But its newer code generation features make it possible to compile models into executable code. The correctness of these features and the quality of their results, though, can be questioned.

A translation from subnetworks of discrete time Simulink blocks to Lustre programs has been proposed [CCM+03] as part of an alternative compilation technique. An extension [SSC+04] handles a subset of Stateflow programs. In this approach, controllers are designed and simulated in Simulink together with continuous models of their environments. The controller components are then automatically translated into Lustre whence they can be verified using model checking and compiled into executables using a certified compiler.

The transformation assumes a time-triggered execution platform. The timing characteristics of the platform are expressed as annotations that state lower and upper bounds on task execution times. The annotations are used to perform timing analyses and to constrain algorithms for synthesis and scheduling.

A different perspective is taken in the Argos block proposal. The most obvious difference is that rather than convert a Simulink model into a synchronous program, the block allows synchronous programs to be integrated into Simulink models. It is conceivable that the Argos block could be incorporated into the Simulink to Lustre translation. The two execution parameters, τ and δ_{out}, might then be passed into the timing analysis and compilation stages. Care would be required to adapt the triggering scheme declared for an Argos block to the scheme used after translation to Lustre. In particular, the event-driven model would likely add complications.

3.5.3 TAXYS framework

In the TAXYS methodology [BCP+01, STY03], application software is specified in logical time, as an Esterel program, and then compiled to an implementation in continuous time, modelled as a timed automaton. A formal notion of correctness between two such models is defined.

The execution times of external functions called from the Esterel program are annotated with lower and upper timing bounds. The running time of the control skeleton is ignored. Systems are analyzed in closed-loop with an environment that is specified as an Esterel program with special clock annotations. The system is verified by compiling the controller and environment models into timed automata, and then incorporating routines from the Kronos model checker [Yov97] to produce a verification engine.

TAXYS specifies an execution platform involving multiple asynchronous tasks, a scheduler, and an event-handler. These are comparable with the synchronous execution machines described earlier, and the Argos block components described in §3.3.1. The TAXYS event-handler bundles individual inputs into synchronous inputs based on separator events, others events are added for interrupts and sampled signals.

The focus of TAXYS is the verification of system properties, and, in particular, that an abstract model is correctly implemented, whereas the Argos block focuses on Simulation with automatic adjustments for simple implementation parameters. The timing parameters of the Argos block are provided separately, rather than as internal program annotations. It may be advantageous to simulate annotated TAXYS programs. The timing behaviours could then be more dynamic and potentially more accurate. The environment could also be modelled in Simulink rather than with Esterel. A mechanism for selecting specific values from between the timing bound annotations would be necessary because Simulink only traces single paths through models.

3.5.4 AASAP Semantics

In the AASAP semantics [DWDR04] controllers are directly expressed as timed automata. Two platform timing characteristics are considered: delays between the occurrence of an event and its detection as an input, and the time taken to compute a

response. Both are expressed as a single parameter Δ. The delay between occurrence and detection necessitates latching, which is expressed by tracking the ages of events.

AASAP programs are executed in three steps:

1. Read the time from a digital clock.
2. Update input latches.
3. If possible, take a transition that updates the state and may emit an output.

The Δ parameter used for modelling and verifying controllers is refined into two implementation parameters: Δ_L for the execution loop delay and Δ_P for the digital clock precision.

There are several differences between the AASAP approach and that of synchronous languages. Transitions in AASAP models may be non-deterministic, whereas determinism is central to synchronous languages. The AASAP execution scheme is not necessarily either sample-driven, since the period is only bounded by Δ_L, or event-driven, since programs may act spontaneously by emitting outputs and taking internal steps. The explicit referrals to a digital clock in an AASAP controller contrast with the more abstract idea of multi-form time espoused for synchronous languages. The biggest difference between the AASAP semantics and synchronous languages is the treatment of events. In AASAP, input events are queued and processed one-by-one and individual output events occur separately from one another, whereas in a synchronous program individual inputs and outputs are bundled together into simultaneous events.

The parameterization of delays in the timing translation of this chapter is similar to that of the AASAP semantics. It differs, though, in both intent and detail. The timing translation focuses on generating simulation traces and not on verification issues. The timing translation models the classical synchronous language execution schemes. It addresses signal bundling explicitly, and also the fact that multiple occurrences of an input event between two reactions are ignored. Multiple event occurrences in the AASAP semantics would be classed as a receptiveness problem for the given environment. Finally, in the AASAP semantics, the delay before an action becomes urgent depends on when individual input events are latched, whereas in the timing translation it is measured from the previous reaction.

A recent report [AT05] presents a technique for modelling programs embedded in system implementations. It could be seen as combining elements of synchronous execution machines with aspects of the AASAP approach. The proposal describes a timed automata construction with five components: an *execution model* for triggering computations; a *digital clock model* that provides discrete timing signals; a *digital controller model* that incorporates an abstract program; an *input and output interface model* which is, in effect, a generalisation of the latching in the timing translation; and a *plant/environment* model.

3.6 Reflections and conclusions

An approach for modelling and simulating controllers expressed as synchronous programs has been presented in this chapter. Certain implementation details are accounted for by mapping discrete semantic models into timed automaton models.

The details were implemented and evaluated using Argos. There are two advantages to using Argos. First, the graphical syntax of Argos makes possible a direct comparison with Stateflow programs; the two languages share the fundamentals of their Statecharts heritage. Second, Argos is simple and clarifies the key concepts of the approach; particularly the translation to timed automata, which can be visualised directly from BMMs. Argos is easier to understand and to compile than a more feature-rich language, like Esterel. Esterel though, would likely compare more strongly with Stateflow as it is ultimately a more practical language for expressing reactive controllers.

The three-parameter model is a novel proposal for integrating synchronous language programs into a dense time model. It can be judged from two perspectives: the practical and the semantic.

From a practical perspective, the model promises increased simulation accuracy. On the positive side, it is able to account for certain realities of synchronous language execution, particularly as the assumption of synchrony breaks down. It allows for the evaluation of slower, relative to the occurrence of inputs, execution speeds. It is not certain, however, that having the extra detail is really a compelling advantage. Any systems whose correctness rests on such fine timing details is perhaps too brittle, and, when this is unavoidable, an even more concrete model may be required. Otherwise, the basic assumption of synchrony may be sufficient. Furthermore, it has been suggested that latencies and jitter between input occurrence and action may be more important to control stability than delays through to output actuation.[21]

From a semantic perspective, the model formalizes aspects of synchronous language interfacing that are usually assumed away. By doing so, it reveals issues of latching and coordination that, while not conceptually difficult, are intricate and subject to choices during design and implementation. The model emphasizes the similarities and differences between the realities of event-driven and sample-driven execution. These benefits aside, the model is deficient in two main ways. First, it does not really make the design of embedded systems any easier; no one would reason directly on the timed automata produced by the transformation. For systems where an assumption of synchrony is suitable, engineers would continue to reason against the model of perfect synchrony, and, at best, employ the three-parameter model to experiment with implementation choices. For systems where synchronous languages are inadequate or awkward, the more detailed model provides no extra assistance. Second, the combination of synchronous programming and real-time parameters is shallow. The transformation is defined on semantic models of Argos programs; their internal structure, wherein the designer's intent is encoded, is ignored.

The next chapter is concerned with modelling rather than simulation. The example described therein requires a deeper embedding of timing parameters. The timing details are no longer mere implementation inconveniences, but rather essential to understanding and meeting an application specification. And, the eventual implementation relies on the concrete intricacies of the execution platform.

[21] Paul Caspi made this observation at EMSOFT 2006.

Chapter 4

An infrared sensor

4.1 Introduction

This chapter contains an examination of a relatively simple sensor component in more detail than is customary. It differs in some obvious ways from Chapter 3. Exhaustive analysis in Uppaal takes the place of simulation in Simulink.[1]. Models are created manually, or from assembly language programs, rather than associated with programs in synchronous languages like Argos or Esterel. Yet the two chapters share the same underlying themes. They both address the real-time behaviour of embedded controllers, particularly as modelled with timed automata. They are both concerned with the implementation issues of platform limitations and triggering, and their interaction with the more abstract concerns of higher-level programming and specification. The examples in both typify a specific sort of embedded system, although the system in this chapter is smaller in scale and studied in more detail. The observations made in this case study, and the idea of working from a specification sheet to a model, bear on the approach to programming, triggering, and timing of Chapter 6.

The sensor, typically for such components, is described by a data sheet that includes a timing diagram. The timing diagram is interpreted and then modelled as a timed automaton. The timed automaton model is refined into a *split model* of cooperating timed automata where the roles of driver and sensor are separately modelled. A testing automaton is constructed from the original model and analyzed together with the split model in Uppaal to show timed trace inclusion. Furthermore, a transmission correctness property is formulated and verified, also by reachability analysis. The driver component of the split model can serve as a specification for actual implementations. One such implementation, an assembly language driver, is modelled with timed automata and validated against the specification via timed trace inclusion testing.

The timed automaton model of the timing diagram is created manually. An alternative approach is to transcribe the diagram into one or more of the formal notations proposed in the literature. An example of this alternative is presented, along with a comparison to the present approach, in Appendix F.

The sensor was chosen because it is simple, time is essential to faithfully describe its behaviour, and it is a concrete example of the type of embedded systems development of interest. One of the original motivations for pursuing this case study was to experiment with different real-time modelling approaches, thereby revealing their relative strengths and weaknesses. Although some progress was made with a timed process algebra [BM02], it turned out that much experience could be gained using timed automata alone. Experiments relating one timed automata model to another led directly to the techniques and software described in Chapter 5. Another original motivation was to compare different ways of specifying delays in programming languages, and related issues of implementation. This theme is reprised in Chapter 6.

The sensor is a small-scale case study: the specification sheet has only four pages and a driver can be written in around twenty lines of assembly language. It suits the intent of looking in detail at an example taken from practice rather than one contrived,

[1] Uppaal is described in §2.3

or adjusted, to exhibit points of theoretical interest. Regrettably, some characteristics of the example limit the possibility of generalisation; they are listed, after the timed automaton model has been presented, in §4.3.5. The discussion of possible interpretations and modelling choices would be less manageable with a larger example.

Quantitative time is integral to understanding and directing the sensor. Mostly because the timing diagram prescribes a way for two asynchronous components to communicate with minimal interconnections: two wires in fact. Both the description of these timing constraints and the way they may be met by implementations are examined. Although the meaning of most signal changes depends more on protocol state than on precise time of occurrence, an open-loop version of the driver is also considered; its operation relies on the passage of time rather than the receipt of events.

Integrating specialist components into larger systems is fundamental to most embedded engineering projects. Engineers must select, understand, and interface with, components and subsystems supplied by third parties. While the selection process may involve sales and technical representatives, the latter two tasks are often performed in isolation working from data sheets, application notes, and manuals alone. The models in this chapter are thus intentionally developed under similar constraints.

The models are expressed with timed (safety) automata [HNSY94], which are well-suited for describing the time-constrained sequential behaviour at hand, and have the considerable advantage that many relations and properties can be checked automatically in Uppaal [LPW97],[2] which is described in §2.3.

The sensor and its timing diagram are explained in §4.2. A timed automaton model of the timing diagram is then presented and explained in detail in §4.3. In §4.4, a different, two component version of the original model is described and verified. Part of this model is carried over into §4.5 where it becomes the specification for an assembly language program that is modelled with timed automata and shown to be a correct implementation. Finally, the strengths and weaknesses of the whole approach are discussed in §4.6.

4.2 The Sharp GP2D02 range sensor

The Sharp GP2D02 range sensor exemplifies the sort of specialist component which an engineer would integrate into an embedded system. The sensor contains electro-optical components that exploit physical principles to produce a specific result that is useful in the context of a larger system.

The choice of investigating the GP2D02 is somewhat arbitrary. Significantly, however, the device specification is relatively simple, includes timing constraints, and mixes event-driven responses with sampling. An awareness of the example's strengths and weaknesses could guide future experiments at the same scale. The guiding philosophy is to examine realistic examples for opportunities where they may be better understood or developed using rigorous approaches, rather than to choose examples which suit one or another particular approach.

Several features of the GP2D02 limit the possibilities of generalising from it. They are not listed in this section, but rather in §4.3.5 after the timing diagram model has been presented, where they may be understood more readily.

4.2.1 Overview

The sensor, refer Figure 4.1, is a small (14 by 29 by 14mm) box. Visible features include two mounting slots, an infrared light emitting diode, a lens that covers a detecting surface, and four electrical terminals: voltage, ground, input (*vin*), and output (*vout*). The sensor measures the distance between itself and another object by emitting infrared beams from the diode, these are reflected by the object and returned to the detecting surface. The distance is estimated by measuring the position of the reflected beams

[2] Discussion in this chapter refers to version 4.0.6.

along the detecting surface. Measurement cycles are triggered and 8-bit distance estimates read from the output terminal by providing a suitable signal on the input terminal. This signal and other details are described in a datasheet [Sha97], which is typical of those made available to engineers for integrating components. The behavioural descriptions are informal and thus subject to interpretation against a background of engineering practice.

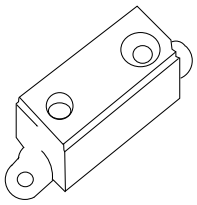


Figure 4.1: Physical appearance of sensor

More abstractly, the sensor is a unit that returns a data value when prompted. Communications with the unit are subject to timing constraints: range estimation is not instantaneous, the eight bits of data must be transferred serially between two otherwise unsynchronized components, and the sensor cannot be re-triggered or powered down immediately after producing a reading.

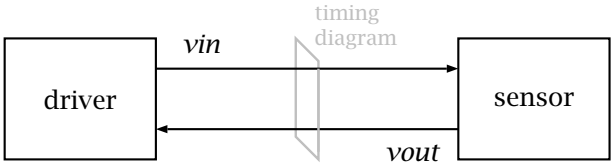


Figure 4.2: Using the sensor

In practice a sensor is connected to another device, which will be termed the *driver*, as shown in Figure 4.2. The driver has control over the signal level on *vin*, and the sensor over the signal level on *vout*. The first model §4.3 will describe the combined protocol—the various causal relations and timing constraints between voltage levels and voltage level changes on the two wires—as represented by the sensor timing diagram. The sensor will be treated as a black box that guarantees the stated behaviours on *vout* provided the driver conforms to the expected behaviours on *vin*. Later models will describe the individual roles of driver and sensor §4.4, and also detailed implementations of the driver §4.5.

4.2.2 Timing Diagram

As is typical, signal values and constraints on when they may change are communicated by a timing diagram. Figure 4.3 is copied directly from the sensor datasheet. The upper signal *vin* specifies the input sequences that may be applied to the unit. The lower signal, labelled *output* but henceforth called *vout*, specifies the expected response. Both signals are high in the quiescent state.[3]

The first falling edge on *vin* triggers a range reading which may take at most 70ms. A series of pulses are then applied to clock data out of the sensor, and, finally, at least 1.5ms must elapse before either repeating or terminating the process.

The choice between stopping and continuing is the only point where control behaviours, as opposed to timing or data valuations, branch. Both possible scenarios are shown at right in the diagram, in each a pulse on *vin* is labelled with *1.5ms or more*. The first pulse on *vin* is interpreted as the beginning of another range reading. The second, which is labelled with *Power OFF* is interpreted as the decision to switch the sensor off. Between these two alternative behaviours, the signal lines are broken by a dashed horizontal section. Clearly, representing branching in a timing diagram is awkward. Specifications often resort to such ad hoc depictions of multiple scenarios, or to multiple figures and explanatory notes.

[3]The datasheet states that *vin* is *open drain* and that an internal pull-up resistor is connected to *vout*.

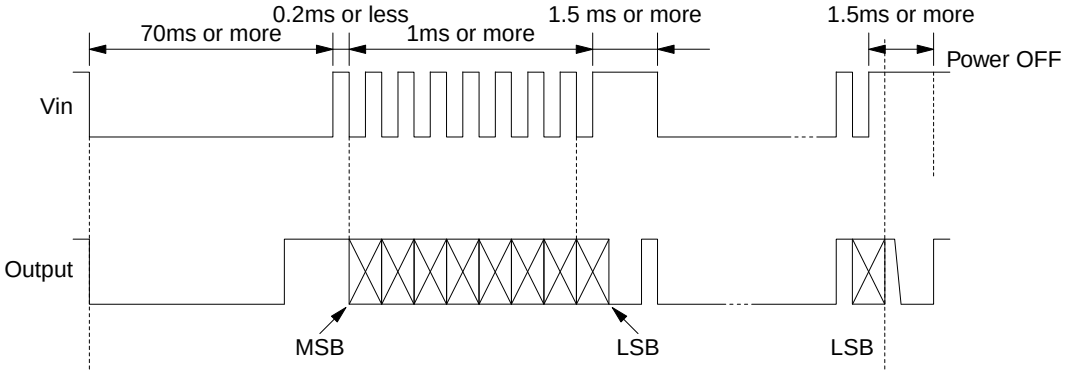


Figure 4.3: Sensor timing diagram [Sha97]

There are four dashed vertical lines between the two signal waveforms. These seem to indicate synchronizing events, from left: commencing a range reading, triggering a change in *vout* for the most significant bit (MSB), similarly for the least significant bit (LSB), and returning to a low *vout* level after an unspecified, though bounded, delay. There is also a fifth vertical line that does not extend all the way to the lower waveform. Its meaning is the most difficult to decipher, it seems to imply that *vout* will be returned to the high level before 1.5ms has elapsed.

The diagram indicates that the *vin* signal should remain constant for at least 70ms after the falling transition that triggers a range reading, during which time the *vout* signal will change. Some implementations [Gri99, Ram01] ignore the timing constraint and act instead as soon as *vout* becomes high.

The crossed boxes on the *vout* signal, between most and least significant bits, indicate data non-determinism. The signal value, either high or low, will depend on the range reading itself and will usually be sampled since there will not necessarily be a detectable event (consider, for example, the eight bits representing 255). For accurate sampling, it is necessary to know precisely when, relative to other signal events, the *vout* level will be stable. The timing diagram could be more explicit, but it seems that changes in *vout* are usually triggered by falling transitions on *vin*. The exact behaviour, however, of *vout* after the least-significant bit has been sampled is not clear. If the last bit is zero, *vout* must return to the high level before 1.5ms elapses. If it is one, it seems that *vout* must go to a low level first before returning to a high level. An engineer could clarify such unclear details, should they prove important, by running experiments with an instance of the device. This is an effective approach, but, at least in principle, such observed behaviours may change between different versions of the specified device. It is assumed that a rising *vin* transition after the last sample triggers *vout* to fall if necessary and then rise again.

According to the diagram, the sensor requires at least 1.5ms between the end of one complete range reading and the beginning of the next. During this period there is a rising transition on *vout* which could indicate that the sensor is ready to make another reading before the whole 1.5ms has elapsed. The specification is not clear.

The *70ms or more* and *1.5ms or more* constraints are readily justifiable: it takes time for the device to make measurements and to recover afterward. Less so the *0.2ms or less* constraint. Rather than seek motives the constraints will simply be accepted as given. Also, rather than interpret *0.2ms or less* as a constraint for all the other positive pulses ⎍ and perhaps also the negative pulses ⎎, it will be assumed to pertain only to the first.

The pulses, though, must definitely have some minimum value, as suggested by the *1ms or more* constraint. The minimum width of a positive pulse will be represented by minmark, and that of a negative pulse by minspace. The *1ms or more* constraint is not otherwise further interpreted. The values of minmark and minspace will depend on the sensor's (unspecified) internal electronics and properties of the interconnection (such as wire capacitance). They are assumed to be equal to zero in the rest of the chapter.

The timing diagram defines a large class of acceptable signals, one of which, Fig-

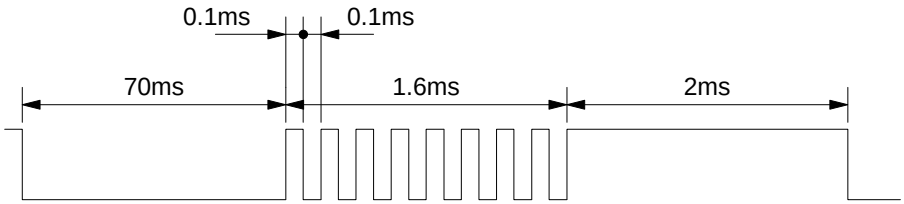


Figure 4.4: Sensor test signal [Sha97]

ure 4.4, is given as part of a test circuit in the specification. This signal shows how a device might trigger the sensor and acquire a reading without any event-based feedback from the *vout* line.

Although not perfect, the timing diagram is adequate for interfacing with the sensor once its ambiguities have been resolved. The inadequacies that may annoy engineers prove to be points of interest in the formalisation of the next section, where they demand more interpretation than transcription. Converting such an informal description into a precise notation quickly reveals what is clear and what is not, which is an important benefit for applications where accuracy is important.

4.3 Timing diagram model

In this section a timed automaton model of the timing diagram is described. The sensor timing diagram is ordinarily read as the specification for a device driver—a circuit or program for triggering the sensor and extracting a reading. It could also be taken as a template for creating different types of compatible sensors. In either case, one would classify events and constraints within the diagram from the perspective of one side or the other, as inputs or outputs, or as assumptions or guarantees. In this section sensor is not distinguished from driver. The focus is instead on the timing diagram as an artefact in itself. It is modelled as a timed automaton, thus providing a precise interpretation. Alternative modelling choices and possible variations are discussed.

There are four subsections. In the first §4.3.1, some benefits of producing such a formal model and some of the philosophies that guide its creation are discussed. In the second §4.3.2, the choice of alphabet for the model is described. The third subsection is the longest and contains a detailed description of the timing diagram model. Lastly, in §4.3.4, liveness requirements and the reasons they are not modelled are described.

4.3.1 Rationale and guiding philosophy

Timing diagrams are usually understood through convention, culture, and practical experimentation rather than in formal terms, although much research in this direction exists (refer to Appendix F for details). There are at least four reasons for translating a timing diagram into a formal framework:

1. Detailed questions are asked of the specification. Its meaning is clarified and ambiguities or omissions may be discovered and noted.
2. A formal specification defines a notion of correctness against which other artefacts, such as sensor and driver implementations, might be validated.
3. Tools for validating and transforming some types of models exist, synthesis being a special case of transformation into an executable form.
4. It helps to distill and better understand features of a system in a precise way.

Implementation in a specific programming language is effectively a translation to a formal, or at least semi-formal, notation. There are, however, important differences:

1. While implementation also requires attention to technical details, certain language features may behave differently across compilers and platforms. Significantly, issues of timing detail may become difficult to properly express.

2. At best a program is a *reference implementation*. It gives one specific notion of correctness. It may be a practical aid to development, but usually cannot reflect the full range of permitted behaviours.

3. Automatic synthesis is the key advantage of using a specific programming language. But the choice of language may limit the range of application. For example, compiled programs may be too large and features like garbage collection may be too slow or unpredictable. Programming language abstractions need not come at the cost of performance and predictability though. For instance, those embodied in the synchronous languages explicitly address these issues of embedded systems programming.

4. There may be confusion between which features of the program are properties of the object under consideration and which are necessitated by the chosen language. Mathematical notations tend to seek a principled blend of property and notation, and ideally the two are interlocked, whereas programming languages often involve more pragmatic compromises.

The key difference between a model and an implementation is one of abstraction. Models will usually ignore details essential to implementations, and conversely, implementations will typically be too constraining to act as models for all purposes. When specifying timing, and other behaviours, it is particularly desirable to ignore distracting implementation details.[4] Ideally though, the models and implementations of a system are interrelated in a precise manner.

The present desire to address the timing diagram in isolation would be obstructed by a programming language intended to address only one side of the described interaction. Essentially, the focus will be on the modelling process and details in themselves.

The timing diagram, Figure 4.3, defines a partial ordering and relative timing constraints on a set of events. The aim is to capture precisely this information, and no more, in a timed automaton model. Justification will be offered for all compromises.

4.3.2 Choosing an alphabet

Sometimes the alphabet of a formal model is presented as an inevitability, but there is a choice and it is important because the rest of a model follows from it.

The transitions from one signal level to another are of most importance in this particular timing diagram. They are the events the model must address.

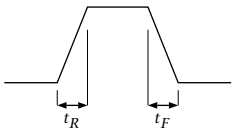


Figure 4.5: Non-instantaneous signal changes

The *vin* and *vout* signals can be described in terms of *edges* and *levels*. Edges are characterised by direction of change, rising or falling. Ideally they are instantaneous with a definite time of occurrence. Levels are characterised by a state, usually high or low, that persists over an interval between two edges. The sensor timing diagram depicts ideal signal edges. Other timing diagrams acknowledge the non-instantaneous nature of edges by using vertical lines with a slight slant. Allowable or expected rise t_R and fall times t_F are sometimes given, as in Figure 4.5, when relevant or necessary.

There are at least three different ways to associate events with transitions. The first associates a distinct event with every change in the value of a signal, and each event has a unique label: for example, $e_1, e_2, \ldots, e_n$. Such detail is tedious and, for the timing diagram model, unnecessary. The second way is to associate an action with each signal, for example vin and vout. Different occurrences of the actions are distinguished from one another by order of relative occurrence. The third way is similar, but associates two actions with each signal: one for a rising transition on the signal, the other for a falling

[4] Although, conversely, most real-time programming is characterised by careful attention to such details.

transition. Fundamentally, this extra detail is not necessary, since rising and falling actions must alternate strictly, but differentiating the two types of transition clarifies the relationship between the model and the timing diagram, which makes them easier to think about and to describe. This third choice is thus adopted. Transitions labelled vinL and vinH correspond to falling and rising transitions, respectively, on *vin*. Likewise for voutL and voutH on *vout*. Little distinction is made between actions, categories of what can happen, and events, specific occurrences of actions.

The rising and falling transitions on *vin* and *vout* are explicit in the timing diagram, but the diagram also constrains two other events: powering off and sampling.

A powerOff action is introduced to represent the act of turning the sensor off.

The timing diagram is not explicit about when the level on *vout* can be sampled, even though it is an important feature of the protocol. The timing diagram model will be more precise: a sample action is introduced to represent instants when readings can be accurately taken from the *vout* line. This new action is controversial since it is not explicit in the timing diagram. But, arguably, it would be inferred by engineers anyway, and should perhaps have been included.

4.3.3 An explanation of the model

The model in Figure 4.6 is the result of many gradual refinements. It is thought to be an accurate interpretation of the timing diagram of Figure 4.3, which is reproduced in gray to aid comparison. One small improvement could perhaps be made. The model, as it stands, only admits powerOff after at least one range-reading cycle has finished. Adding an extra transition, from s_0 to s_{11} labelled with powerOff, would allow termination even when no reading has been taken.

The model is expressed as an Uppaal timed automaton so that its relationships with other models can be verified automatically. As a consequence, transitions must be labelled as inputs or outputs despite the neutral stance taken on the issue. They have all been made outputs, simply because this makes validation in §4.4.3 easier. The model is to be interpreted as an open system, one that defines a set of timed sequences of allowed actions, rather than as part of a closed system that can only act when another component is willing to synchronise.

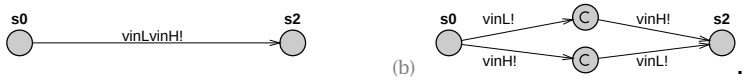
In a valid Uppaal model all timing constants must be integers, thus one unit of model time is equivalent to 0.1ms in the timing diagram.

There are three main phases in the timing diagram protocol: the initial triggering of a range reading §4.3.3.1, transferring the resulting value bit-by-bit §4.3.3.2, and finally deciding whether to power off or to repeat the process §4.3.3.3.

4.3.3.1 Initial triggering

A range reading cycle is triggered from the initial location, s_0, by a falling transition on *vin*: vinL. According to the timing diagram voutL occurs simultaneously. Conceptually, vinL causes, or at least precedes, voutL. There are at least four ways to model the relationship between the two events:

1. A single event composed of two others: vinL · voutL, as in process algebras like SCCS [Mil83] or ACP [BW90] (refer Appendix A). Such a composite event is the result of the synchronization of two components, but in this model the event would later be decomposed into the separate actions of driver and sensor. There are at least two possible expressions in Uppaal,



(a) (b) (Figure 4.7)

   The one at left is direct but makes decomposition clumsy. The one at right tries to capture the commutativity and atomicity of the event combination operator. Atomicity is not necessarily guaranteed by the committed locations, marked Ⓒ, in models where there are other such locations in parallel. An urgent location would also be suitable since the automaton is interpreted as an open system.

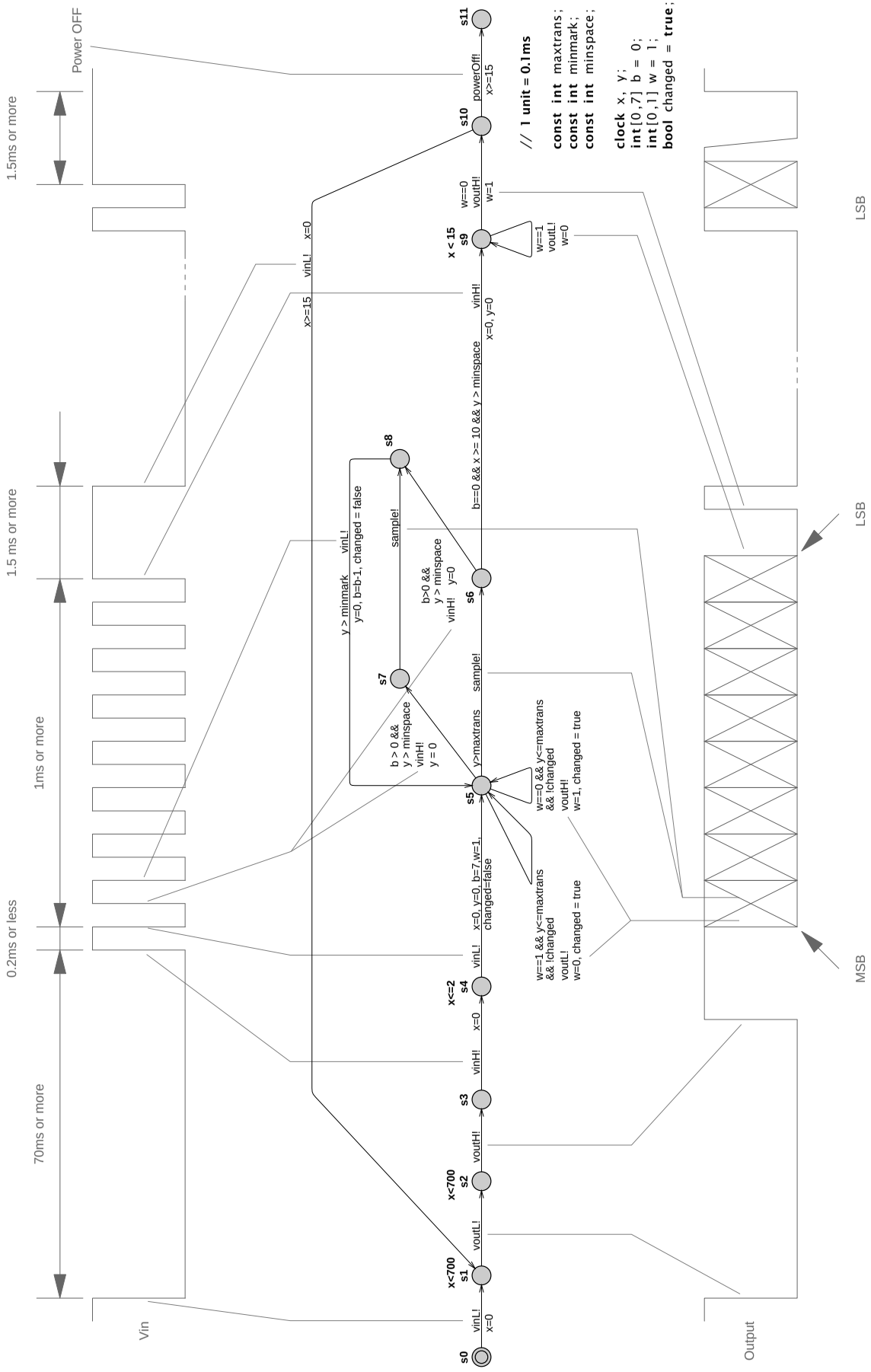
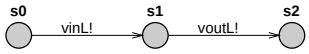


Figure 4.6: Timing diagram model

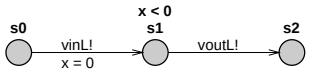
2. One event simply following the other; capturing the causal dependency but not the implied synchronization.



(Figure 4.8)

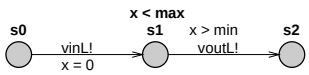
This choice is adopted for the timing diagram model.

3. One event immediately following the other, that is, like the previous option, but with an invariant on the middle location, or equivalently with the location marked urgent, to express simultaneity and necessity of occurrence. Simultaneous but ordered events are often termed *micro-steps* in state-diagram languages [HPSS87], or *delta steps* in discrete-event simulation languages, like VHDL.



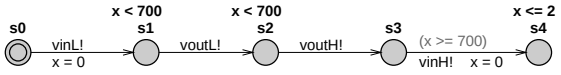
(Figure 4.9)

4. One event following the other within a fixed period, that is, the upper bound on the second event is relaxed, and a lower bound may be added, giving a model that treats the physical characteristics of an implementation more accurately.



(Figure 4.10)

The *70ms or more* constraint is given between initial vinL and vinH events, but causal dependencies between vinL and voutL, and voutL and voutH, and voutH and vinH mean that the intermediate voutL and voutH events are also constrained. The first of the causal dependencies has been discussed. The second arises because signal levels must alternate. The third is less explicit, it is divined from some extra knowledge of the sensor: when it has finished taking a reading it raises the output signal. Thus both locations s_1 and s_2 have an invariant label $x < 700$, which guarantees the occurrence of voutL and voutH before 70ms passes:



(Figure 4.11)

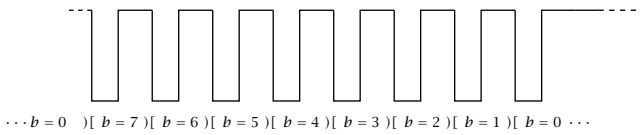
Strict compliance with the timing diagram would require an $x \geq 700$ guard on the vinH transition between s_3 and s_4. The model varies on this point: the driver may proceed as soon as voutH is detected—as, in fact, do some implementations [Gri99, Ram01].

The vinH event must be followed by vinL within *0.2ms or less*. This constraint is modelled by resetting the clock x when the former event occurs and adding an $x \leq 2$ invariant to s_4, which occurs between the two. The vinL transition sets the clocks x and y, and the variables b, w, and changed:

x models the *1ms or more* constraint, extending from this occurrence of vinL to the last vinH in the reading cycle,

y times each of the 15 alternating pulses,

b counts down eight transmitted bits through locations s_5, s_6, s_7, and s_8,

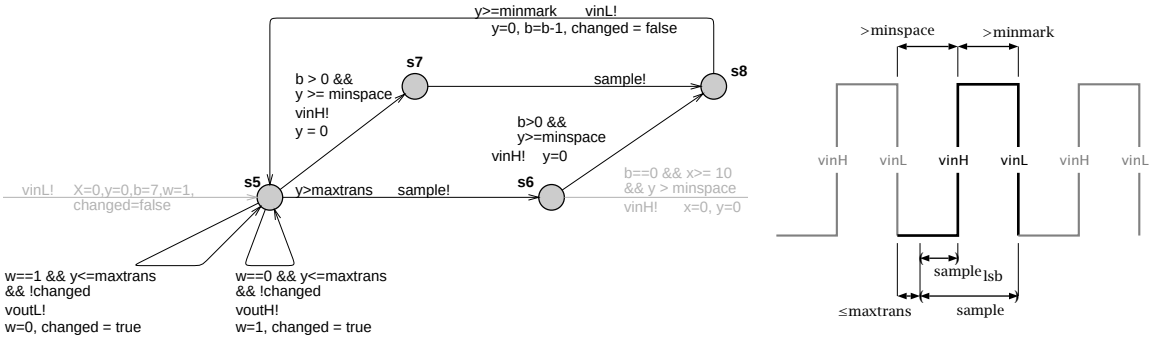


(Figure 4.12)

w tracks *vout* to ensure strict alternation of voutL and voutH events,

changed ensures that at most one output event occurs for each sampled bit.

The effect of the binary-valued w and changed variables, and even the bounded integer variable b, could be instead made explicit in the structure of the automaton, but doing so would obscure the essential feature of the model.



(a) Sampling loop from Figure 4.6 **(b)** Sampling parameters

Figure 4.13: Sampling detail

4.3.3.2 Data transfer

The loop that models serial data transfer is duplicated in Figure 4.13 alongside the corresponding extract from the timing diagram.

The transmission cycle begins after a falling edge on *vin*, vinL, prompts the sensor to transmit the next bit of the range reading. Both the initial vinL, from s_4, and the looping vinL, from s_8, lead to s_5.

The level of *vout* is only allowed to change while s_5 is active and within maxtrans of the triggering vinL. The self-loops on s_5 express possible changes on *vout*; they are discussed in more detail later. The timing constraint is measured by the clock y which is reset, within the loop, whenever there is a change on *vin*. The constant maxtrans combines an assumption on the maximum time the sensor will take to change the vout level after being triggered by vinL, and the time required to transmit any change through the wiring and interface electronics. It is not explicit in the timing diagram but the sampling period is not well defined without it.

From s_5 the driver must both sample the *vout* level and return *vin* to a high level.[5] Both actions, sample and vinH, must happen after the maxtrans delay and before the next vinL. They are causally-independent for all but the least significant bit, that is while b > 0. Such a relationship is naturally modelled with parallelism; but such small-scale parallelism cannot be expressed directly in Uppaal—extra synchronizations would be needed. For such a small number of actions it is practical to explicitly model all of their possible interleavings, of which there are two: the path s_5–s_6–s_8 that samples first and then raises *vin*, and the path s_5–s_7–s_8 that raises *vin* first and then samples. Such an approach quickly becomes untenable as the number of mutually-independent actions increases. For the case of the least significant bit, when b $= 0$, only one of the interleavings is allowed. The sample action occurs first because it is assumed that vinH signals to the sensor that sampling is complete.

The alternation of vinL and vinH actions within the transmission loop, Figure 4.13a, gives alternating negative and positive pulses on *vin*, Figure 4.13b. Both types of pulse have a minimum width: minspace for the negative pulses and minmark for the positive ones. This is expressed in the timed automaton by guard expressions on clock y. The timing diagram is not explicit about minimum pulse widths. It states only that the eight negative pulses and seven positive pulses that comprise each cycle must take *1ms or more*. Rather than assume minmark = minspace = $1/15ms$, the model is validated with minmark = minspace = 0, which is the most permissive choice. The *1ms or more* lower bound is enforced separately by the $x > 10$ guard on the vinH transition that exits the loop.

The current model assumes that maxtrans ≤ minspace, but rather than add an extra clause to the guards on edges s_5–s_7 and s_6–s_8, b>0 && y>minspace && y>maxtrans, this constraint on the constants is stated separately. Setting minmark $= 0$ thus implies that maxtrans $= 0$. An alternative approach would be to duplicate the voutL and voutH

[5] Although one could also imagine a model where sampling is optional.

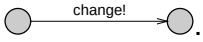
self-loops on s_7 and add the guard y>maxtrans to the transition between s_7 and s_8; but it seems less natural to allow the driver to act before the sensor value has stabilised.

The *vin* signal generated by the sampling loop is shown in Figure 4.13b, where it is annotated with several parameters. The negative pulse is marked with the constant minspace; it ends with a vinH action. The positive pulse is marked with the constant minmark; it ends with a vinL action. There is a period of maxtrans units after the initial vinL before sampling may occur. Sampling is thereafter allowed until a subsequent vinL action, but for sampling of the LSB which may not occur after a subsequent vinH.

In the main timing diagram, Figure 4.3, there are several crossed boxes in the *vout* signal between MSB and LSB annotations. They represent data non-determinism; the signal may change or remain constant from one bit to the next depending on the value being transmitted. The crossed boxes thus abstract over 2^8 possible data signals, not counting variations in timing. There are at least five ways to model them:

1. By not modelling them explicitly. This results in a simpler model because there are fewer transitions and the behaviour of *vout* after the least significant bit is easier to model. Whilst sufficient, from the driver's perspective, to only specify when sampling may occur, the timing diagram model tries to avoid bias toward either role.

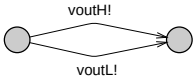
2. By marking the event with a change label, thereby ignoring the specific value.

   (Figure 4.14)

   This, however, would complicate the verification of data transmission in the split model of §4.4. It is also misleading because the level of *vout* will not change if adjacent bits are identical.

3. By non-deterministic choice between both possible transitions.

   (Figure 4.15)

   This technique refines the change action into two different actions: rising and falling transitions. It is more in the spirit of the timing diagram, as a partially ordered set of transition events, but it incorrectly infers that an update of the output level is always observable. That depends, rather, on the values of adjacent bits in a data reading.

4. By modelling both types of transition, but also including the level of *vout* in the automaton state. This treats the observability of events more accurately—no event occurs if the level does not change—and also facilitates the verification of data transmission (§4.4.4).

   This technique was chosen for the timing diagram model of Figure 4.6. Variable w encodes the level status and variable changed ensures that at most one event occurs per data bit. In principle, one could write an event-triggered driver that responds to the presence or absence, in a given period, of transitions on *vout*.

5. By modelling the possibility that *vout* may change several times before settling, within maxtrans units, to a constant value. This approach [VG04] is closer to physical reality where a signal may, after a change in level, oscillate unpredictably before stabilising. In this situation a driver triggered by events, rather than one that samples the signal level, is impractical.

   The timing diagram model can be adjusted to use this technique by removing all references to the changed variable.

After eight transmission cycles, when b is zero, a transition leaves the loop from s_6 on vinH. If the last *vout* level was high (the least significant bit (LSB) was one, $w = 1$), it must now be returned to a low level via a voutL event. The timing diagram is not precise about the occurrence of this event, but it must precede the voutH event, which in turn must precede the vinH event that exits the sampling loop within 1.5ms. Thus both the voutL self-loop on s_9 and the voutH transition to s_{10} are constrained by the invariant $x < 15$.

4.3.3.3 Power off or repeat

After triggering a range reading and sampling the resulting eight bits, there is a choice of terminating, the powerOff action to s_{11}, or of requesting another reading, the vinL action back to s_1. Neither can happen until the sensor is ready. The timing diagram states that the choice may only be made *1.5ms or more* after the vinH event that ends sampling. This constraint is reflected in the model by $x \geq 15$ guards on the transitions leaving s_{10}. An alternative would be to interpret the voutH within this period as an indication that the sensor is ready; as when waiting for completion of a range reading, where a driver can either wait for a safe amount of time, *70ms or more*, or instead just until the occurrence of voutH.

4.3.4 Liveness and progress

The location invariants in the timing diagram model specify when actions are necessitated; or equivalently, when unbounded delay is forbidden. Where there is no invariant the protocol may pause indefinitely. The vinL, vinH, sample, and powerOff actions from s_0, s_3, s_6, s_7, s_8, and s_{10} need never occur. Instead, the protocol may simply stop. Similarly for s_5 where voutL and voutH may possibly, but need not, occur within bounded time, and s_{11} where only unbounded delay is allowed.

The timing diagram contains bounded liveness guarantees on all sensor responses. These upper timing bounds are modelled as location invariants, on s_1, s_2, and s_4, or transition guards, on the loops at s_5. The sensor must always respond to the driver within a fixed period of time. A time-bounded response from the driver is only expected at s_4—and even this *0.2ms or less* constraint is somewhat dubious.

It is sometimes useful to specify the necessity of progress without stating an explicit time bound. Some of these more abstract liveness properties may be expressed via an acceptance criterion, like Büchi or Muller conditions, and a subset of accepting states, refer §2.2.3.4. Assuming Büchi conditions, liveness constraints could be added to the timing diagram model in several ways:

- *Require eventual termination*: make s_{11} accepting and add a τ-transition self-loop. Only a finite number of range readings could then be performed before powerOff inevitably occurs.

- *Require complete protocol cycles*: make both s_{10} and s_{11} accepting and add a τ-transition self-loop to s_{11}. Range reading cycles must then run to completion—reading taken and transmitted—once a triggering vinL has occurred. Cycles must be triggered continually forever, or a finite number of times until powerOff occurs.

- *Allow a finite number of complete readings without mandating* powerOff : make s_0, s_{10}, and s_{11} accepting and add τ-transition self-loops to all three. Cycles must then run to completion once triggered, but neither endless cycles nor eventual powerOff are required: nothing need occur initially or between readings.

There are a few ways of stating in Uppaal that a model does not delay indefinitely: by giving a specific upper bound in a location safety invariant or equivalently using an urgent location,[6] committed location, or urgent channels, though these are less appropriate for an open model since their behaviour in arbitrary composition is awkward. Uppaal cannot, however, model more abstract liveness requirements. It does not, for instance, allow states to be marked as accepting.

Additional liveness constraints will not be specified for the timing diagram model. They are not really justified by the sensor specification alone, and it would not anyway be possible to express them with Uppaal. Some liveness properties will be checked, however, of a later implementation model §4.5.3.

[6] Urgent locations have an upper bound of zero

4.3.5 Limitations

At least seven factors limit generalizations from this example:

1. The timing diagram represents communications between driver and sensor where the former is essentially master and interaction is limited. In particular, it is possible for the driver to operate without any direct feedback from the sensor.

2. Although the communications involve two components running concurrently, the protocol is essentially sequential. That is, neither component performs many independent actions between synchronizations with the other.

3. There is almost no branching in the control structure. In particular, when one component waits for the other, it always expects a specific signal—its actions do not depend on which signal occurs, they are only delayed until the awaited signal.

4. The driver and sensor essentially form a closed system. That the environment affects transmitted infrared beams has no bearing on the integration task.

5. Data values do not affect the observable behaviour of either component in a significant way.

6. Timing constraints are stated solely between driver events, but never between sensor and driver events, or between two sensor events, and although there are sometimes implications for sensor events no constraints are stated directly. The constraints do not overlap, nor do they depend on the particular values of earlier delays.

7. Component behaviour does not depend on the measured length of delays.

Limitations 5 and 7 could be removed, but only by considering an unusual version of the driver that reads data by detecting event occurrence and absence rather than sampling signal levels.

4.4 Driver/sensor split model

Whereas the different roles of driver and sensor were ignored in the timing diagram model they are the focus of this section where, by first assigning responsibility for individual signals in §4.4.1, two variations of a *split model* are constructed in §4.4.2. The split models distinguish the contributing roles of driver and sensor. They effectively add behavioural detail to the structural diagram of Figure 4.2.

The timing diagram model restricts the synchronizations between driver and sensor. These details are made more explicit in the split model. The timing diagram and split models are related formally, after some abstraction, by timed trace inclusion. This relation will be shown in §4.4.3 by transforming the timing diagram model into a testing automaton, composing it by turns with augmented versions of the split models and performing reachability analysis with Uppaal.

Constructing a split model and confirming its relation to the timing diagram model increases confidence in both. Each corroborates the other. In fact, working with the split model did result in improvements to earlier versions of the timing diagram model.

In the split model, the separate behaviours of the driver and sensor are isolated from one another. It becomes possible to express, using an auxiliary automaton and reachability property, and then to verify transmission correctness—that sensor readings are always transmitted accurately to a driver. Uppaal is again used for the verification, §4.4.4. In addition, the submodels are useful for further verifications. This topic is pursued in the next section, §4.5, where the driver component of the split model becomes a specification for an implementation in assembly language.

Six models are presented in this section. There are two variations on the split model: one with paired synchronisations,[7] the other with broadcast synchronizations. Trace

[7] The term ‘rendezvous’ is expressly avoided because it is somewhat ambiguous.

signal	driver	sensor
vinL	output	input
vinH	output	input
voutL	input	output
voutH	input	output
sample	output	not applicable
powerOff	output	not applicable

Table 4.1: Action directions relative to components.

inclusion with the timing diagram model and transmission correctness are verified for each. The six constructions are presented over four figures:

	paired (pair)	**broadcast** (bcast)	
split model	Figure 4.16 *(upper)*	Figure 4.16 *(lower)*	§4.4.2
trace inclusion verification	Figure 4.19	Figure 4.20	§4.4.3
transmission verification	Figure 4.21 *(upper)*	Figure 4.21 *(lower)*	§4.4.4

4.4.1 Action direction and synchronization

The driver and sensor components are extracted from the timing diagram model by deciding which actions each will control as outputs, and which will serve as inputs.

Action direction was immaterial in the timing diagram model and outputs were used solely for convenience. But in the split model each of the six actions has a meaningful direction at each component. An action is an output at a component if its occurrence is determined by that component. An action is an input at a component if its occurrence may influence that component's behaviour. Actions vinL and vinH are sensor inputs and driver outputs. Actions voutL and voutH are sensor outputs and driver inputs. These assignments reflect both the reality of connections between driver and sensor and a desired division of responsibility. The events sample and powerOff are different in nature. They do not influence the behaviour of the sensor, and are thus driver outputs but not sensor inputs. The direction assignments are summarised in Table 4.1.

The most direct way of creating a driver model from the timing diagram model is to change action directions to match the stated assignment. A sensor model could be created similarly, but the sample and powerOff actions would be dropped, as would state s_{11}. Additionally, s_7 would be merged with s_8 and s_5 with s_6. Such a split model would faithfully implement the range-reading protocol, but the components would be coupled too tightly: each would synchronise with every action of the other.

In the interpretation of the original specification, in §4.3, the sensor only synchronises with the driver at the triggering vinL, at the vinL for reading each bit, and at vinH after the least-significant bit. These synchronizations must be inferred from background knowledge about the sensor and its operation. Though the predominantly sequential behaviour expressed by the timing diagram and the strict alternation of falling and rising transitions on a signal are already quite limiting.

The driver need not synchronise with the sensor at all: the timing guarantees make open-loop control possible. Alternatively, the driver may synchronise on the voutH that indicates a completed range reading. Both possibilities will be modelled.

4.4.2 The split models

A principle of the split model is that the driver alone is responsible for determining when events on *vin* occur, and similarly for the sensor on *vout*.[8] That is, output actions should be *non-refusable*; they may not be constrained by other components and the controlling component alone should determine their occurrence. But in Uppaal, by default, communication on a channel requires the participation of two processes: one offering an input action, the other an output action. An output from a component is

[8] The possibility of a component forcing its input line to a low or high level is ignored.

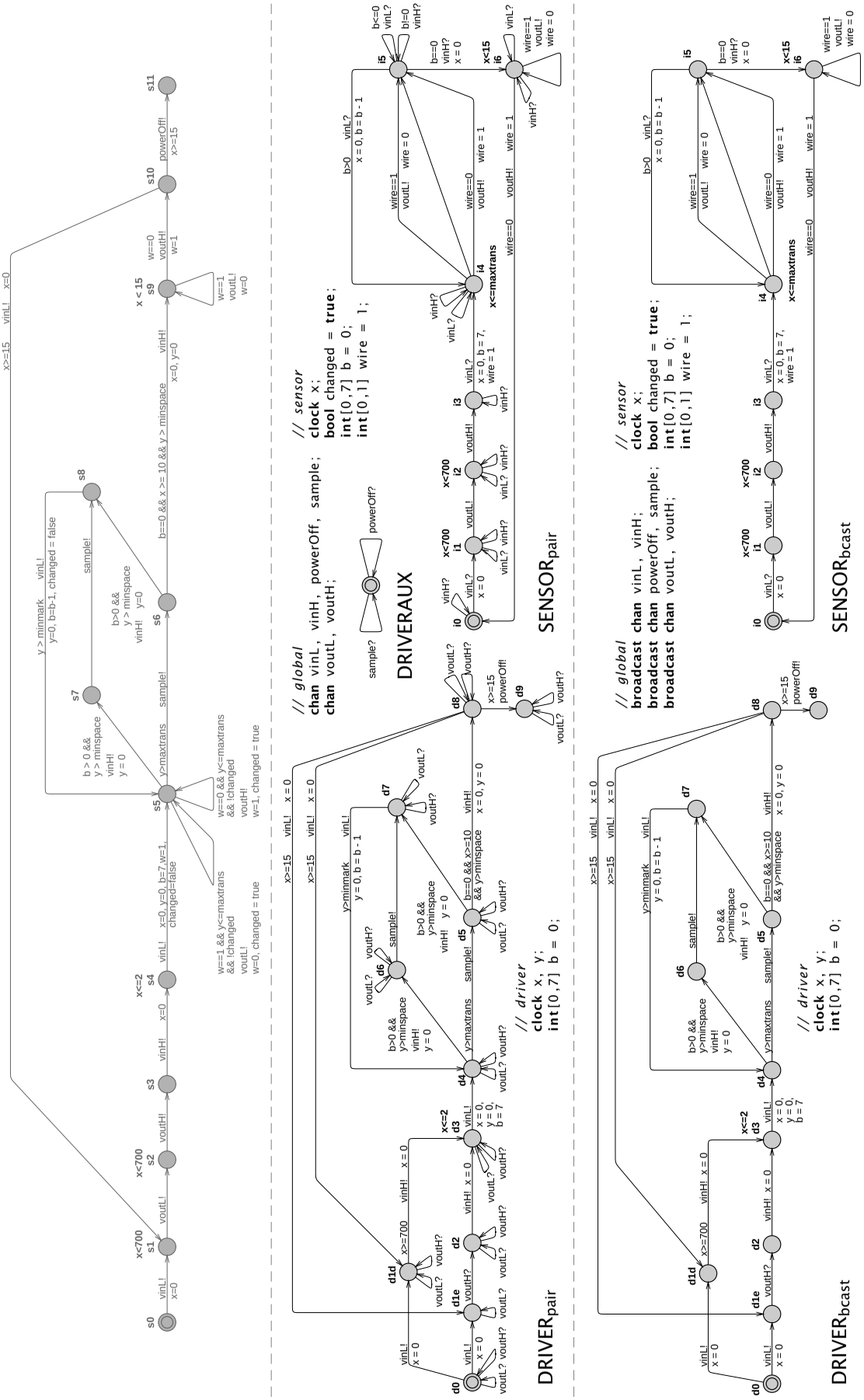


Figure 4.16: Split models

blocked if no other component is ready to accept it as an input. When each action represents communication between a pair of processes, it is sufficient to ensure that each process is *input-enabled*: in every state, every input action of the process is enabled.[9]

Timed automata that are free of τ-transitions are made input-enabled, with respect to a set of actions, by adding self-loops for actions that would otherwise be ignored.

Definition 4.4.1
Given a timed automaton $\mathcal{A} = (L, l_0, K, \mathsf{inv}_K, T)$ on actions in A and a set of input actions $I \subseteq A$, let $\mathcal{A}_{ie} = (L, l_0, K, \mathsf{inv}_K, T_{ie})$, be a version of $\mathcal{A}$ that is input-enabled with respect to I, where T_{ie} is the least set such that $T \subseteq T_{ie}$ and,

$$\frac{l \in L \quad \mathsf{c}? \in I \quad \{g_1, \ldots, g_n\} = \left\{ g \mid l \xrightarrow[c?\ T]{g\ \cdot} \cdot \right\}}{l \xrightarrow[\mathsf{c}?\qquad T_{ie}]{\neg(g_1 \vee \ldots \vee g_n)\ \ \varnothing} l}\ \mathsf{ie} \quad .$$

■

Timed automata with τ-transitions cannot be treated so easily: input actions may be possible from states reachable by τ-transitions, and delay transitions may need to be accommodated. Fortunately, the simple definition suffices for this chapter.

A split model based on rendezvous communication is shown in the middle of Figure 4.16. The model consists of three components in parallel:

$$\mathsf{DRIVER}_{\mathsf{pair}} \parallel \mathsf{DRIVERAUX} \parallel \mathsf{SENSOR}_{\mathsf{pair}}.$$

The timing diagram model is repeated in gray at the top of the figure for comparison.

The driver model was created directly from the timing diagram model. All transitions on voutL were removed, as were all but the first, between $\mathsf{d}_{1\mathsf{e}}$ and d_2, for voutH, whose direction was changed from output to input; states were merged where necessary. The initial triggering sequence was duplicated through $\mathsf{d}_{1\mathsf{d}}$, the $x < 700$ location invariants were removed and replaced with $x \geq 700$ guards. All references to the variables wire and changed were removed. Finally, the model was input-enabled for voutL and voutH, which introduced two self-loops on all states except $\mathsf{d}_{1\mathsf{e}}$, where the model reacts to voutH.

The sensor model was also created directly from the timing diagram model. All transitions on vinH but the one between s_6 and s_9 were removed; the transition guard only depends on the value of b, not those of clocks x and y. All transitions on sample and powerOff were also removed. States were merged where necessary. The variable w was renamed to wire to avoid confusion. The variable changed was removed, its effect is represented in the control structure of the sensor model by states i_4 (changed = 0) and i_5 (changed = 1). The voutL and voutH self-loops on s_5 become transitions between the two states, and a τ-transition is added for the case where there is no change. Clock y is no longer needed. The purpose of clock x is changed, it appears in invariants on i_4 and i_6 that force the sensor model to meet timing assumptions that are mandated by the protocol and required by the driver model. Finally, the model was input-enabled for vinL and vinH, which introduced two self-loops on most states, but not i_0 and i_3 where the model reacts to vinL. The guard expressions of input-enabling transitions from i_5 depend on b.

Much of the transformation from timing diagram model to each of the driver and sensor component models was automated using features of the tool described in §5.4 of Chapter 5. Automation was particularly advantageous because several iterations were required during the development of the timing diagram model.

The DRIVERAUX process is introduced to ensure that sample and powerOff outputs are never refused. There would have been no technical difference had the sensor model also been input-enabled for these two actions, but that would have violated the spirit of the model and also have made it awkward to reuse the model separately.

Most of the input-enabling loop transitions are redundant, they never synchronise with an output in any trace of the complete model. They are included so that the

[9] Appendix B discusses a formalism based on this idea.

components are self-contained and may be used alone in different contexts, and also so that the property of output non-refusal can be verified statically, that is without considering the reachable state space of the complete (closed) model.

Input-enabling is essential to the paired synchronization split model. But it requires extra effort to maintain and the extra transitions distract from more important features of the model; these problems only worsen as the number of inputs increase. Furthermore, input-enabling does not scale well beyond pairs of communicating processes. Its two main advantages are that only the default, and relatively simple, paired communication semantic rules are needed to understand models, and that the timed trace inclusion validation is readily applicable for verifying an implementation model, as in §4.5.3.

A split model based on broadcast communication is shown at the bottom of Figure 4.16, underneath the paired model. It consists of two components in parallel:

$$\mathsf{DRIVER}_{\mathsf{bcast}} \parallel \mathsf{SENSOR}_{\mathsf{bcast}}.$$

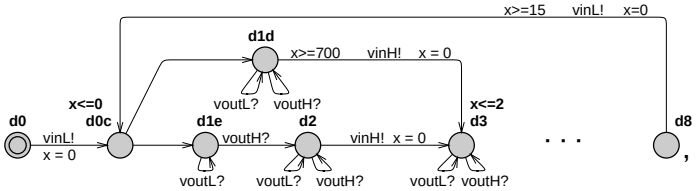
The DRIVERAUX process is unnecessary. Removing the input-enabling loops from the pair components gives the respective bcast components.

Since an output action on a broadcast channel may always occur, non-refusability is ensured by the semantics. All processes that can react to an output still will.

The broadcast model would seem ideal: no extra effort is required to ensure the output non-refusal or input enabled properties, the model is less cluttered and more easily examined and modified, and communications need not be limited to pairs of processes. It will be shown in subsequent sections, however, that timed trace inclusion validation of most broadcast models is not currently possible with Uppaal.

The rest of this section describes the progressive interaction between driver and sensor components in the self-loop split model.

The driver initiates a range-reading from d_0 by emitting a vinL output, which the sensor synchronises on from i_0. The driver non-deterministically enters either $\mathsf{d}_{1\mathsf{d}}$, where it waits for *70ms or more* before continuing, or $\mathsf{d}_{1\mathsf{e}}$, where it can continue as soon as the sensor emits voutH. A similar arrangement exists from d_8. The choice could be modelled using τ-transitions and an extra state, $\mathsf{d}_{0\mathsf{c}}$,



(Figure 4.17)

which avoids duplicating the vinL transitions, but which makes output non-refusal difficult to handle—how should $\mathsf{d}_{0\mathsf{c}}$ respond to voutH?. Furthermore, such a model would complicate time trace inclusion testing, where τ-transitions are forbidden.

In the original model, under the assumption that the sensor will respond to vinL with voutH within 70ms, the trace fragments accepted by the top path $(\mathsf{d}_0/\mathsf{d}_8)$–$\mathsf{d}_{1\mathsf{d}}$–$\mathsf{d}_3$ are also accepted by the bottom path $(\mathsf{d}_0/\mathsf{d}_8)$–$\mathsf{d}_{1\mathsf{e}}$–$\mathsf{d}_2$–$\mathsf{d}_3$. But both paths are included to make the driver model self-contained when used in isolation from the sensor model and to make implementation choices explicit; either branch of the choice can be eliminated to produce a more specific specification.

The driver sampling loop d_4–$(\mathsf{d}_5/\mathsf{d}_6)$–$\mathsf{d}_7$–$(\mathsf{d}_4/\mathsf{d}_8)$ is taken directly from the timing diagram model. The sensor sampling loop is simpler. After receiving a vinL input, the sensor acts within maxtrans units to transition between i_4 and i_5, either signalling a level change by synchronising on voutL or voutH, depending on the local variable wire, or by not signalling any change, but instead changing state with a τ-transition. Input-enabling loops can safely be added to both the source location of the τ-transition and to its destination location, as, according to the protocol, neither vinL nor vinH may occur until after a delay of maxtrans. There is thus no risk of losing an event that is required for the protocol to proceed.[10]

[10]These kind of intricacies around τ-transitions recommend the benefits of the maximal progress as-

Location invariants are present at d_3, i_1, i_2, i_4, and i_6. From each of these locations, there is at least one outgoing transition labelled with an output action. This is obligatory in the split model because otherwise one component could influence the behaviour of another by 'stopping' time, either indefinitely, giving a zeno trace, or until an awaited input action is forced to occur.

Finally, the driver component has, from d_8, the choice of delaying indefinitely, triggering another range reading, or powering the sensor off.

4.4.3 Verifying implementation

Both the two split models and the timing diagram model are proposed as formalisations of the sensor reading and transmission protocol. The split models only differ from one another in the technicalities of ensuring input-enabledness, but their relation to the timing diagram model is less certain, even though they were derived from it. It is, in fact, possible to show that each of the split models implements, in a precise sense, the protocol described by the timing diagram model. Timed trace inclusion is proposed as a suitable implementation relation in §4.4.3.1. Its verification between the timing diagram and each of the split models is described in §4.4.3.2.

4.4.3.1 Which relation?

There are many ways to relate transition systems in general and timed automata in particular, refer Appendix C. Timed trace inclusion has the advantages of simplicity and of allowing the inference of safety properties. It can, moreover, often be verified in Uppaal using a simple construction. Its main drawback is that, by disregarding branching and completion, liveness properties and deadlock, are ignored and must be separately addressed. There are also good reasons for requiring the split model to be as permissive as the timing diagram model, but timed trace equivalence would be, in this case, more difficult to verify.

Timed trace inclusion is a relatively simple notion of implementation. One timed automaton implements another timed automaton if the set of all timed traces of the former are a subset of the set of all timed traces of the latter. To claim that a split model implements the timing diagram model is to claim that any timed trace of the split model is also a valid timed trace of the timing diagram model. There are some technicalities because the split model is a network of timed automata rather than a single timed automaton. They are discussed later. A more complicated relation, like timed ready simulation (see §5.3.4), would have been required had the models contained committed locations or interactions through shared variables.

Showing an implementation relation between split and timing diagram models encourages faith in the accuracy of both. The correspondence between the pairs of models is like a form of double-entry bookkeeping. Several modelling discrepancies were actually found and corrected through repeated comparisons. Alternatively, if the model of the timing diagram is considered faithful, the relation shows that the split model is also correct. More technically, the relation means that any safety properties proved of the timing diagram model may be immediately inferred of the split model.

While safety properties can be inferred from timed trace inclusion, liveness properties cannot; sets of traces express only what may happen, not what must. This fact is less limiting for timed models than for untimed models, because bounded liveness, the necessity of action within a fixed, finite time, is a safety property. The requirements that the sensor respond within a given period are adequately preserved. Abstract liveness properties, like those proposed in §4.3.4, would have to be verified separately.

Timed trace inclusion also ignores deadlock. Again the limitation affects timed models less than untimed models, provided upper bounds are specified where progress is important. Moreover, the split models are input-enabled and thus rather than deadlock states, which cannot occur in the split model, quiescent states are more relevant (see §C.3).

---

sumption made in formalisms like Real Time Calculus of Communicating Systems (RTCCS), see §2.2.3.3. In the present framework, τ-transitions from urgent locations could be permitted.

The separate verification of liveness and deadlock properties is sufficient for the models of this chapter, since their branching structures are uncomplicated. It will not be necessary to introduce a relation that can distinguish delay in quiescent states from delay in states where further output is still possible. Similarly, separate checks will be made, using Uppaal, to ensure that time deadlock is not possible.

Timed trace inclusion is undecidable in general [AD94], but when the specification model is deterministic, which also effectively means free of τ-transitions, there are constructions for deciding it via reachability analysis. The basic idea behind one such construction is outlined in the next section §4.4.3.2, but a detailed explanation is delayed until Chapter 5, where extended techniques and a software implementation for manipulating Uppaal models are described. The extensions are not required for the models in this chapter, but the implementation proved invaluable since both the timing diagram and split models were improved over several iterations of timed trace inclusion testing.

Timed trace inclusion is an asymmetric relation. Using it to verify that the split model implements the timing diagram model only shows that the former never exceeds the behaviours permitted by the latter. Showing the same relation in the other direction would imply timed trace equivalence of the models. In fact, there are traces of the timing diagram model that are not traces of the split model, for example, from s_5 the former allows any number of alternating voutL! actions before the maxtrans limit, whereas the latter, from i_4, allows at most one. Detecting and resolving all such differences would give a stronger relation between the two models, in particular, that no valid behaviours are excluded from the split model. The models would be interchangeable.[11] There are two reasons why such a validation has not been attempted. First, the current development works from a protocol specification through to a model of a driver implementation. Working in one direction is sufficient to verify the driver against the protocol. Second, the testing transformation would have to be extended from automaton specifications to those involving a network of automata, which are inherently non-deterministic due to component interleaving. Although there are techniques that can sometimes allow non-determinism to be handled by the testing construction [Sto02, §A.1.5], and, there cannot anyway be much interleaving in the split model since its timed traces are included in those of the timing diagram model which only has one 'diamond' of causally-independent actions.

4.4.3.2 Verification

The procedure for testing timed trace inclusion[12] within a specification, namely the timing diagram, has two steps: transforming the specification into a testing automaton, then performing reachability analysis of the result in parallel with the implementation automaton, that is with one of the split models. There are slight complications because each split model is not an open automaton, but rather a closed network of automata. Different adjustments are required for each of the paired and broadcast split models but both can be verified against the timing diagram model.

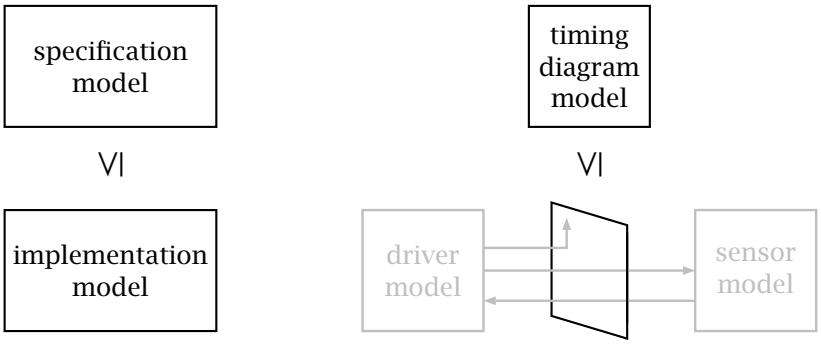
There is nothing fundamentally difficult about transforming the specification automaton for testing. A new location, called Err, is added. Inputs are made outputs and vice versa. Then from each location new transitions to Err are added for every action that cannot occur in that location. Input and output actions are considered as distinct even when on the same channel. Whether an action can occur or not may depend on the value of clocks and variables, so the guards of outgoing transitions must be considered. Furthermore, each location invariant is replaced by a τ-transition to Err that has as a guard the negated invariant. The result is a testing automaton that can 'observe' another automaton by running in parallel with it and synchronizing on each of its actions, hence the inversion of action directions, and decide, based on location, clocks, and state variables, whether an action is allowed, in which case observation continues, or forbidden, in which case a transition to the Err state is taken. Should the

[11] Modulo the limitations of timed trace equivalence.

[12] Technically, the procedure verifies a timed simulation relation, but timed trace inclusion can then be inferred because it is a coarser relation.

observed automaton fail to act in sufficient time, thus violating a location invariant of the original specification, a τ-transition to Err will become enabled. The Err state will not be reachable if the timed traces of the observed automaton are a subset of those of the specification automaton, since each action of the former will synchronise with a valid action of the latter. Reachability analysis, in Uppaal, can determine whether Err is reachable and hence whether two timed automata are related by timed trace inclusion.

The result of applying the testing transformation to the timing diagram model is shown at the top in Figure 4.19. The original structure is still present in the nodes and transitions with black lines, but the actions are now inputs rather than outputs. The new state, Err, is the hub of a nest of new gray transitions sourced at each of the other locations. While individual transitions are not legible, the general structure is clear. Guards, where present, and actions on the error transitions are tabulated above source locations s_7 and s_8, in two columns for each, and below the others, in a single column for each. For instance, the only action permitted from s_0 in the timing diagram automaton is vinL!, so in the testing automaton vinL? leads to s_1, and all other actions lead to Err; none of them have guards. The situation is similar from s_1, but since there is a location invariant $x < 700$ in the timing diagram model, a τ-transition with guard $x \geq 700$, shown in the figure as an expression with no associated action, is added to the testing automaton, and the guards of the other transitions are augmented with the invariant, thus ensuring determinism. The guards on the error transitions are more complicated at s_5. For instance, vinH? is allowed when $b > 0$ but forbidden otherwise, similarly, sample? is forbidden when $y \leq$ maxtrans.



(a) Normal scenario **(b)** The split models

Figure 4.18: Deviation from the usual verification of timed trace inclusion

The testing technique is intended for comparing one timed automaton to another, per Figure 4.18a, but in the present case the implementation model is a network of timed automata and it is their interactions that are to be verified, per Figure 4.18b. The two approaches differ from each other, and from Uppaal itself, in their interpretation of a model as a set of timed traces. The differences have ramifications for verifying timed trace inclusion.

An Uppaal model $\mathcal{U}$ is a parallel composition of N timed automata $\mathcal{A}_i$ where all actions A are restricted at the top-level,

$$\mathcal{U} = (\mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_N) \setminus A.$$

Some details were discussed earlier, in §2.3, and even more precision is required later, in §5.2.4, but only the broad ideas are necessary at present. The key point is that, due to the restriction, Uppaal models are *closed*, they only perform τ-transitions and delay transitions. Input actions cannot occur during simulation and reachability analysis unless corresponding outputs are also able to occur, and vice versa for non-broadcast outputs. In typical timed trace inclusion testing, each automaton, whether specification or implementation, is given an *open* interpretation where any action can occur at any time. An automaton is thus assigned a maximal set of timed traces that subsume its behaviours in any specific composition. To verify the split models, the timing diagram model is given an open interpretation but the split models themselves must be

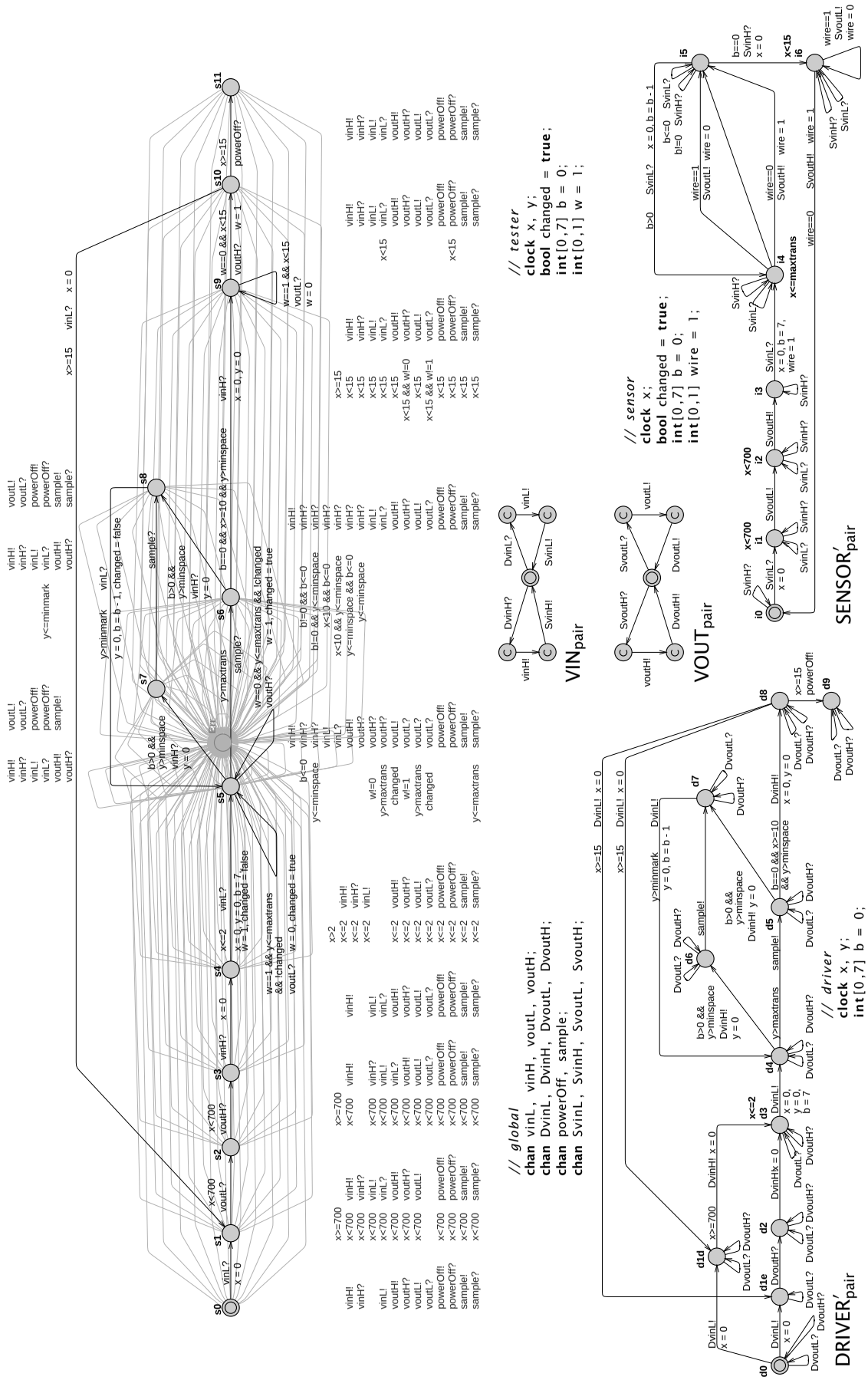


Figure 4.19: Trace inclusion testing in the paired synchronization model

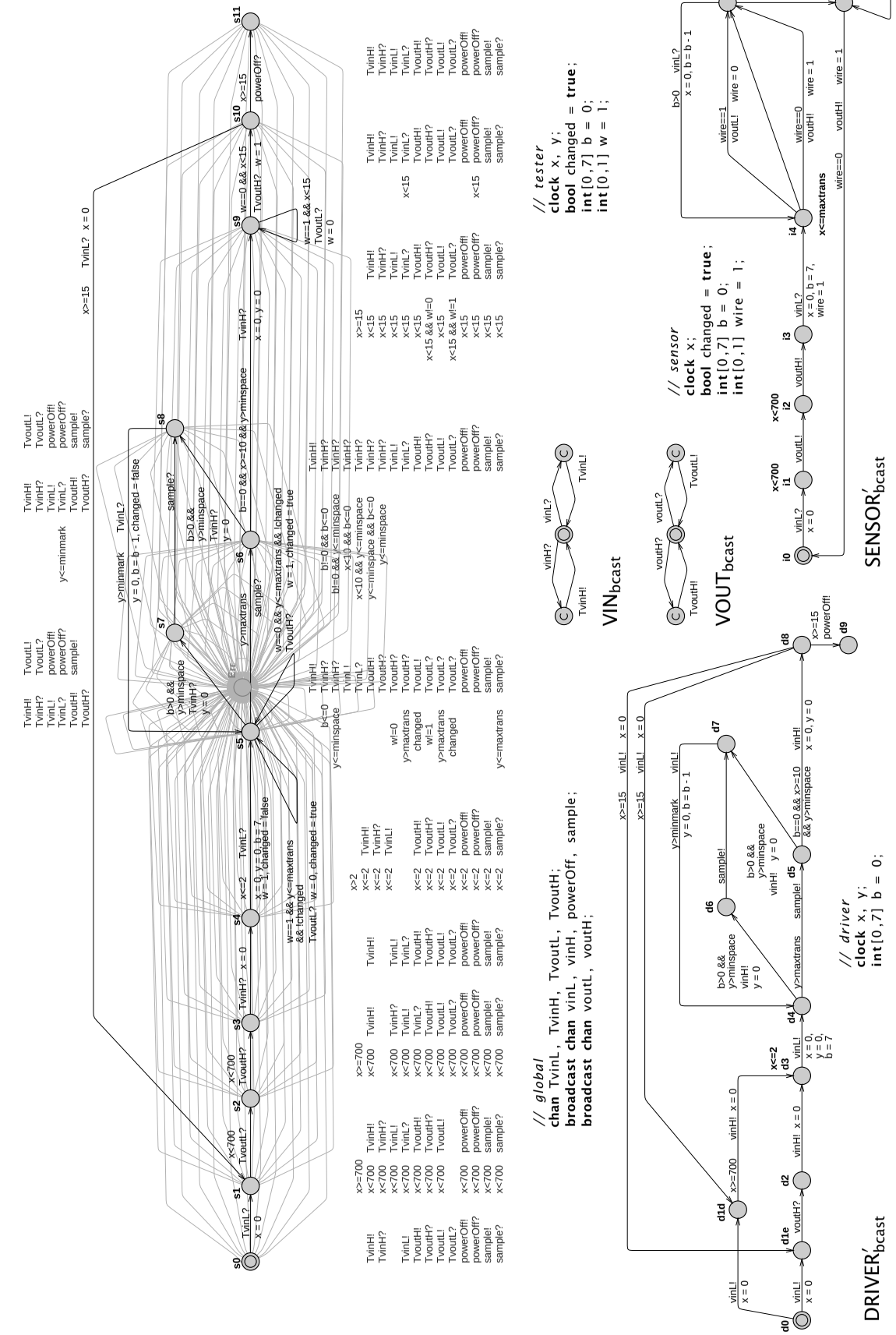


Figure 4.20: Trace inclusion testing in the broadcast model

interpreted differently. The paired synchronization version is assigned the intersection of the open trace sets of the driver and sensor components, after mapping all input actions to corresponding outputs and adjusting the open trace set of the sensor component so that sample and powerOff actions may occur at any time. The broadcast version is interpreted as open so far as output actions are concerned and any input actions they trigger are internalised. The split models are thus assigned the set of timed traces resulting from their interactions; that is, from the mutual resolution of their respective constraints.

This interpretation of the split model has two immediate ramifications. First, extra efforts are required to verify timed trace inclusion using the testing technique. The paired split model must be augmented to reveal actions exchanged between driver and sensor, which are normally hidden as τ-steps; the sample and powerOff actions can be handled normally. The broadcast model can be verified directly, but transitions labelled with output actions must be pruned from the testing automaton. Second, the correctness, with respect to the timing diagram model, of individual implementation components is only guaranteed in composition with a partner that is also correct. This is particularly important in §4.5.3 when the driver component is used as a specification.

The Uppaal model for verifying timed trace inclusion of the paired split model against the timing diagram model is shown in Figure 4.19. Several actions in the driver component are renamed:

$$\begin{array}{rcl} \mathsf{vinL!} & \longrightarrow & \mathsf{DvinL!} \\ \mathsf{vinH!} & \longrightarrow & \mathsf{DvinH!} \\ \mathsf{voutL?} & \longrightarrow & \mathsf{DvoutL?} \\ \mathsf{voutH?} & \longrightarrow & \mathsf{DvoutH?} \end{array}$$

As are several actions in the sensor component:

$$\begin{array}{rcl} \mathsf{vinL?} & \longrightarrow & \mathsf{SvinL?} \\ \mathsf{vinH?} & \longrightarrow & \mathsf{SvinH?} \\ \mathsf{voutL!} & \longrightarrow & \mathsf{SvoutL!} \\ \mathsf{voutH!} & \longrightarrow & \mathsf{SvoutH!} \end{array}$$

Two new channel automata, $\mathsf{VIN}_{\mathsf{pair}}$ and $\mathsf{VOUT}_{\mathsf{pair}}$, are added. They share the same basic structure—a central location surrounded by two three-transition circuits through committed locations. They could as well be combined into an equivalent automaton with four three-transition circuits. When either of the sensor or driver components performs an output on, respectively, *vout* or *vin*, it synchronises with a channel automaton and then the only possibility, for the whole model, is an output to the testing automaton followed by an output for the other component. This inevitability is due both to the use of committed locations, of which only one will ever be active at any time, and the relative input-enabledness of the components. Certain actions thereby always occur as part of a fixed subsequence within a timed trace of the model. For instance, the action DvinL! always occurs in the subsequence DvinL!–vinL!–SvinL!.

The model resulting from these modifications, that is the composition

$$\mathsf{DRIVER}'_{\mathsf{pair}} \parallel \mathsf{VIN}_{\mathsf{pair}} \parallel \mathsf{VOUT}_{\mathsf{pair}} \parallel \mathsf{SENSOR}'_{\mathsf{pair}},$$

can be given an open interpretation and tested against the timing diagram model.

The Uppaal model for verifying timed trace inclusion of the broadcast split model against the timing diagram model is shown in Figure 4.20. The split model in the figure is identical to the one presented earlier (Figure 4.16). No modifications are required. The actions of the testing automaton, though, have been renamed, and two channel automata have been added. The need for these complications is best understood by first considering a more ideal version and the reasons why it is not presently acceptable.

Ideally, the broadcast testing automaton could simply be derived from the standard version by removing all transitions labelled with output actions. In other words, by forming it from the timing diagram model with respect to an alphabet consisting solely of output actions (since when inverted they become input actions). This would be possible because none of the transitions in the timing diagram model are labelled with input actions. It would be necessary because a broadcast output is not constrained

by the readiness or otherwise of other components to synchronise on the corresponding input, and a testing automaton could thus transition to its error state as soon as a transition labelled with a broadcast output became active, irrespective of what the model being tested could actually do at that instant. Unfortunately, however, in Uppaal[13] such an automaton would be rejected because *clock guards are not allowed on broadcast receivers*. The restriction seems arbitrary, or at best influenced by implementation concerns, since clock guards on transitions with broadcast inputs would seem a natural generalisation of guards without clocks—when the guard is not satisfied the model behaves as if the transition does not exist.

The model of Figure 4.20 works around the limitation on broadcast inputs by renaming signals in the testing automaton,

$$\begin{array}{rcl} \mathsf{vinL} & \longrightarrow & \mathsf{TvinL} \\ \mathsf{vinH} & \longrightarrow & \mathsf{TvinH} \\ \mathsf{voutL} & \longrightarrow & \mathsf{TvoutL} \\ \mathsf{voutH} & \longrightarrow & \mathsf{TvoutH}, \end{array}$$

and adding components that catch broadcast actions and then force immediate synchronisations, using committed locations, with the renamed signals. An alternative solution[14] would be to modify the testing automaton to receive broadcast inputs into a committed location from which outgoing τ-transitions would choose a suitable destination location according to the guards containing clocks. While this alternative is more complicated than the present solution, it would be easier to implement automatically for the general case.

Figures 4.19 and 4.20 were verified with Uppaal and their respective error locations are not reachable. This indictes that they are correct implementations of the timing diagram model of Figure 4.6.

4.4.4 Verifying transmission correctness

The driver and sensor components of the split model interact to request and perform range-readings, and then to transfer the resulting value bit-by-bit through repeated signalling and sampling over the *vin* and *vout* wires. It has been established in the previous section that the split model implements the timing diagram model. In this section, the correctness of data transmission is verified by augmenting the split model with extra details, which do not affect the communication protocol, then expressing the transmission property as a separate automaton, and, finally, performing reachability analysis in Uppaal.

The models for verifying transmission correctness are shown in Figure 4.21. There are two: one for the paired version,

$$\mathsf{DRIVER}''_{\mathsf{pair}} \parallel \mathsf{SENSOR}''_{\mathsf{pair}} \parallel \mathsf{DRIVERAUX} \parallel \mathsf{TESTER},$$

and another for the broadcast version,

$$\mathsf{DRIVER}''_{\mathsf{bcast}} \parallel \mathsf{SENSOR}''_{\mathsf{bcast}} \parallel \mathsf{TESTER}.$$

A single TESTER automaton serves for both. The DRIVERAUX automaton exists to allow $\mathsf{DRIVER}''_{\mathsf{pair}}$ to perform sample and powerOff actions at any time. No such provision is necessary when the actions have broadcast semantics.

The paired and broadcast models are augmented in the same way. An array Sbyte[8] is added at the sensor component to store the bits representing the most recent range-reading. The choice at i_4 between changing the *vout* level or leaving it stable is no longer non-deterministic, but rather depends on the bit value to be sent, that is on Sbyte[b] where b is successively decremented by the transmission loop. An array Dbyte[8] is also added at the driver component to store the bits sampled from the

[13] Both versions 4.0.6 and 4.0.7.

[14] Suggested by Marius Mikučionis on the Uppaal mailing list (12 January 2008).

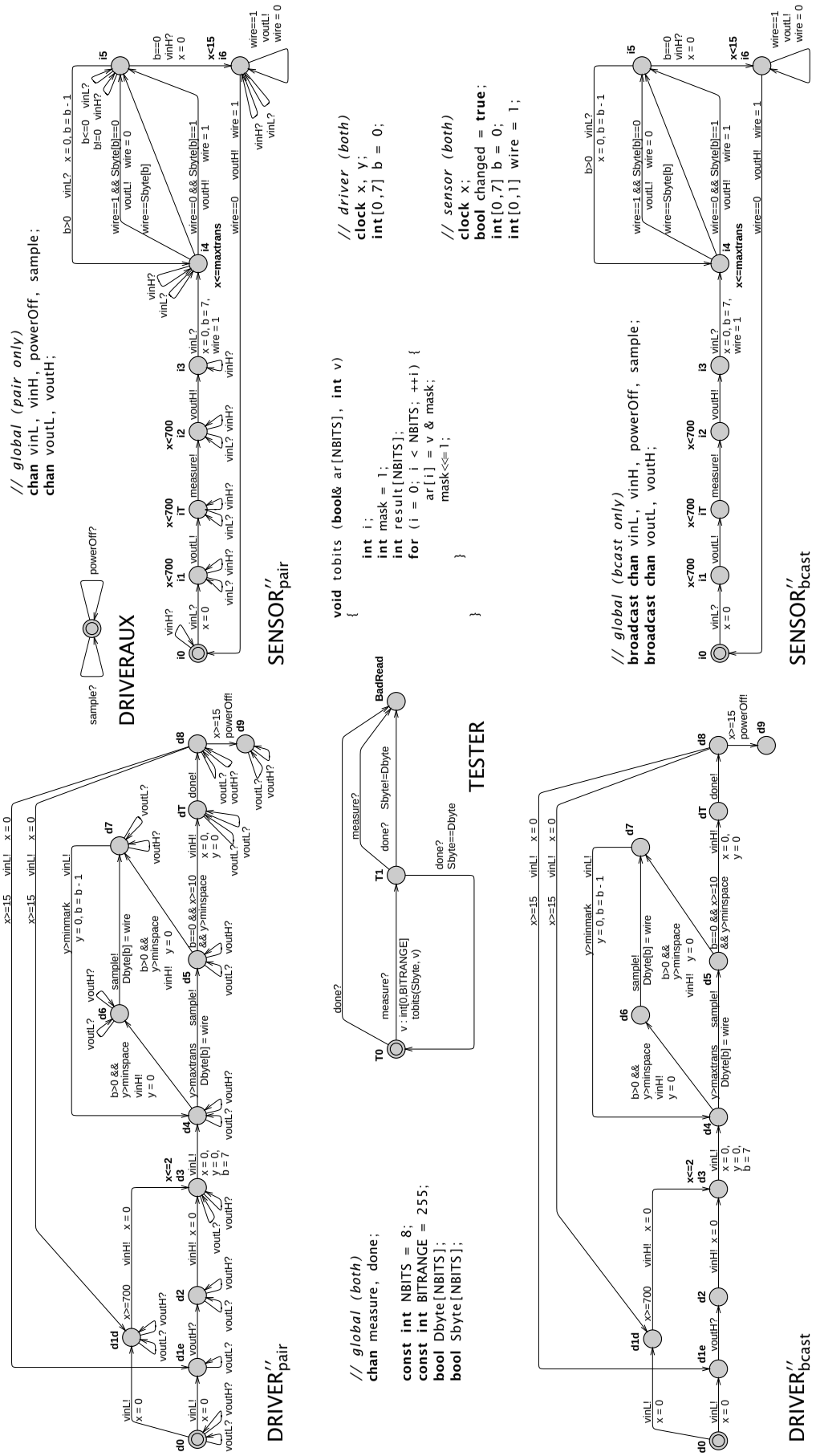


Figure 4.21: Validating transmission

vout line. The value on the shared wire variable, which was previously used only to restrict level changes in the sensor component, is copied into Dbyte[b][15] when a sample action occurs.

The value of Sbyte must be set when a reading occurs and compared with that of Dbyte when a transmission is complete. These tasks are performed by the TESTER automaton, but it is important that the sensor itself decides when a reading has been made and, likewise, that the driver decides when data has been received. These judgements are part of the interaction. New transitions are thus added to both component models. In the sensor model, a transition labelled measure! is added between locations i_1 and i_2. The new location i_T has an invariant to preserve the timing properties of the protocol. In the driver model, a transition labelled with done! is added between d_5 and d_8. The altered model has a different timed trace set to the original, but the protocol remains fundamentally unchanged: dropping measure! and done! actions from every timed trace recovers the original set. The actions could also have been renamed into τs to give a set that would be equivalent to the original modulo τ-transitions.

The TESTER component synchronises on the measure and done events. It assigns a value to Sbyte when the former occurs and compares it with the value in Dbyte when the latter occurs, entering BadRead if there is a discrepancy. A measure or done action at the wrong time also causes TESTER to enter BadRead. All possible assignments to *SByte* are tested using a selection binding to non-deterministically choose a value in the range $[0, 2^8)$, before the tobits function converts it into eight separate bits. The same idea could be modelled using eight separate one-bit selection bindings, which obviates the need for a conversion function but makes parameterisation, over the number of bits to transmit, impossible. Reachability analysis is performed by Uppaal to verify $A\Box\,(\neg$TESTER.BadRead$)$ and hence transmission correctness.

Range readings are correctly transmitted in the split model. Unfortunately, this fact cannot be inferred of the timing diagram model because timed trace inclusion has only been verified in one direction. The ability to make such inferences would be another advantage of showing timed trace equivalence.

4.5 Assembly language implementation

Despite the neutral attitude adopted in earlier sections toward the roles of sensor and driver, the timing diagram typically serves as a specification for writing drivers to integrate the sensor's functionality into larger designs. In this section, an implementation in assembly language (MCS51) is first described in §4.5.1 and then modelled using timed automata in §4.5.2. The model is shown to be an implementation of the driver component of the split model in §4.5.3. The timing diagram, split, and program models can thus be related by a short chain of timed trace inclusions:

TIMEDIAG
⊵ (§4.4.3)
DRIVER ‖ SENSOR
⊵ (§4.5.3)
MCS51.

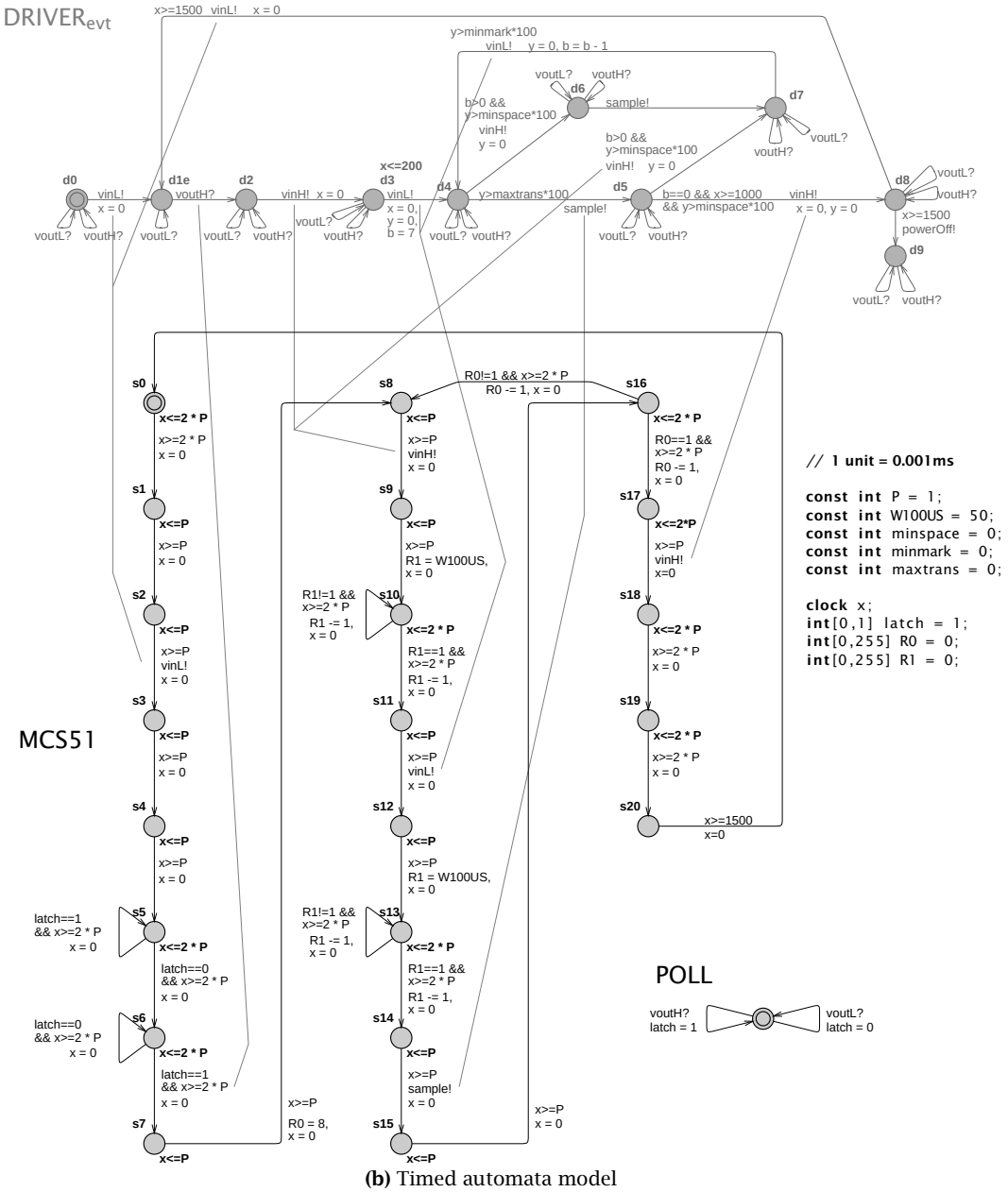
The assembly language model and verification is interesting for two reasons. First, it demonstrates the verification of a simple program against a timing diagram specification by model checking timed trace inclusion relations. Second, the assembly-language techniques for meeting timing requirements provide for contrasts with the abstract approach taken by the synchronous languages and with the expression of behaviour in timed automata.

The limitations of the assembly program model and the verification are discussed in §4.5.4 along with some suggestions for further research.

[15] Note that the b variables are local to the components and hence distinct.

```
1   VIN EQU P1.0
2   VOUT EQU P1.1
3   W100US EQU 50
4
5   RREAD:  PUSH IE
6           CLR EA
7           CLR VIN
8           NOP
9           NOP
10          JB VOUT, *
11          JNB VOUT, *
12          MOV R0, #8
13  LOOP:   SETB VIN
14          MOV R1, #W100US
15          DJNZ R1, *
16          CLR VIN
17          MOV R1, #W100US
18          DJNZ R1, *
19          MOV VOUT, C
20          RLC A
21          DJNZ R0, LOOP
22          SETB VIN
23          POP IE
24          RET
```

(a) MCS51 program



(b) Timed automata model

Figure 4.22: Assembly language driver implementation

4.5.1 MCS51 Program

A driver for the infrared sensor, implemented in MCS51 [Int94] assembly language, is presented in Figure 4.22a. It was adapted from a version written for a 68HC12 microcontroller [Gri99]. The MCS51 architecture (8051) is older technology, but it is relatively simple, well-understood, and typical for the application domain of low-level embedded controllers. The main reason for targeting the MCS51, however, is because instruction timing is simple and easy to predict. Instruction cycle counts can be added together without having to consider instruction pipelining, cache effects, and other such mechanisms that are often present in modern microcontrollers. While this characteristic simplifies the timed automaton model of the next section, it also limits generalization. In any case, the aim is to examine how timed behaviour is realised by the program, not to propose a general methodology for verifying assembly language programs.

The assembly program is designed to be called as a subroutine from within a larger program. When called, it interacts with the sensor before eventually returning with a range reading in the accumulator register.

```
1  VIN EQU P1.0
2  VOUT EQU P1.1
3  W100US EQU 50
```

Lines 1–3 are assembler directives that declare constants. The first two associate the *vin* and *vout* signals with the two lowest pins of I/O port 0. The third defines a constant `W100US` for the number of iterations required to delay for 100 microseconds later in the program. The value depends on the number of cycles taken by certain decrement and jump instructions and the clock frequency of the target platform.

```
5  RREAD:  PUSH IE
6          CLR EA
           ⋮
23         POP IE
```

The code is intended to execute without interruption on the target device, so it begins by pushing the Interrupt Enable (IE) register onto the stack and then disabling all interrupts by clearing the Enable All (EA) bit flag. Interrupt handling is restored, prior to returning from the subroutine, by popping the original value of the IE register which contains the EA bit. Disabling interrupts greatly simplifies modelling and reasoning about program timing, but, again, limits generalization.

```
7          CLR VIN
8          NOP
9          NOP
```

Line 7 sets the *vin* signal to low. The following two `NOP`, *no operation*, instructions cause a brief delay before the status of *vout* is checked. They have absolutely no other effect. Pausing in this way between Input/Output (IO) actions is characteristic of assembly language programming and significant with respect to timing behaviour.

```
10         JB VOUT, *
11         JNB VOUT, *
```

The instruction at line 10 either jumps back to itself if *vout* has a high value, the assembler replaces the asterisk with a relative offset, or otherwise continues to the next instruction. It continuously polls the signal state until the desired value is observed. The next line is similar, it waits for a rising transition on *vout* by jumping back to itself while the signal value is low. The two instructions together await the occurrence of two events in sequence: voutL–voutH.

```
12         MOV R0, #8
13 LOOP:   SETB VIN
           ⋮
21         DJNZ R0, LOOP
```

Lines 12–21 implement a loop for receiving a range reading bit-by-bit. The R0 register acts as the loop counter, it is initialized with 8 at line 12. Line 13 is labelled LOOP, so that the instruction at line 21 can jump back to it—the DJNZ, *Decrement and Jump if Not Zero*, instruction decrements the given register by one, then jumps to another location if the new value is not zero, and otherwise continues to the next instruction. The instruction on line 13 sets *vin* high, thereby causing a rising transition: vinH.

```
14          MOV R1, #W100US
15          DJNZ R1, *
```

Between the rising transition on *vin* and a subsequent falling transition there is a delay of approximately 100 microseconds, which is implemented by counting down from the W100US value in the R1 register. The delay is equal to the amount of time it takes to execute the DJNZ instruction multiplied by the initial value of the counter register. Such loops, containing no internal statements, serve only to delay program execution.

```
16          CLR VIN
17          MOV R1, #W100US
18          DJNZ R1, *
```

These lines set *vin* to low, giving a vinL action, and then delay for another 100 microseconds.

```
19          MOV VOUT, C
20          RLC A
21          DJNZ R0, LOOP
```

Lines 19 and 20 together sample the level of *vout* into the received range reading. The former copies the bit named VOUT, which tracks the external voltage level, into the carry flag. The latter shifts the contents of the accumulator to the left, effectively multiplying the stored value by two, and sets its LSB to the carry flag value.

```
22          SETB VIN
23          POP IE
24          RET
```

Finally the driver sets *vin* high, restores the interrupt enable register, and returns to the calling program.

4.5.2 Program model

The assembly language program has been modelled as the composition of two timed automata,

$$\text{MCS51} \parallel \text{POLL}.$$

The model is presented in Figure 4.22b below a copy of the driver specification, whose details, apart from the change in time scale, are not discussed until the next subsection. Most of the program model was generated automatically from the source program. In particular, the timing constraints were derived directly from the individual assembly instructions. They capture some aspects of the execution platform, but perhaps oversimplify others. The automatically-generated automaton was adjusted to model input and output and an assumption on the minimum delay between repeated executions.

Since only integer timing constants are allowed in Uppaal, the timing diagram and split models assumed a scale of one unit of model time to 0.1ms of real time (refer §4.3.3). It turns out that even older microcontrollers, like those of the MCS51 family, are much faster and thus require a smaller time scale. One unit of model time in the program model is equivalent to 0.001ms of real time. The number was chosen for a device clocked at 12MHz with a machine cycle every 12 clock periods and running at one cycle per microsecond [Int94, p. 1-18]. All of the constants in the driver model, shown at the top of Figure 4.22b in gray, have been multiplied by 100 to account for the increased timing resolution.

The final program model is based on a direct translation from the source program.[16] The translation effectively gives a semantics for programs written in the MCS51 instruction set in terms of timed automata. Close structural similarity between the source program and its model is important because results obtained for the latter are inferred of the former. In other words, that the model is a faithful formalisation of the program is determined solely by informal argument: a larger gap between them gives more chances for error. A location s_n in the model of Figure 4.22b corresponds to the instruction at line $n - 5$ of the program in Figure 4.22a.

Each MCS51 instruction takes a fixed number of cycles to execute. These delays and the propulsion of execution are expressed in the model by location invariants and transition guards. The constant P is the time taken for a single execution cycle. It varies across different MCS51 implementations and oscillator frequencies. The mapping, though, from instructions to required number of execution cycles is fixed for the MCS51 instruction set [Int94, Table 10]—the informal semantics thus describe not only how instructions transform microcontroller states but also how long those transformations take, at least nominally. Lower and upper bounds on instruction execution times in the model are expressed as inequalities between the clock x, which is reset on every transition, and integer multiples of P. For instance, the ⟨**PUSH** IE⟩ instruction takes two cycles to execute, thus location s_0 has the invariant $\mathsf{x} \leq 2{*}\mathsf{P}$, which forces progress, and the outgoing transition has the guard $\mathsf{x} \geq 2{*}\mathsf{P}$, which expresses the required execution time. The invariant on s_1, $\mathsf{x} \leq \mathsf{P}$, and the guard on the outgoing transition, $\mathsf{x} \geq \mathsf{P}$, reflect the fact that it only takes one cycle to execute the ⟨**CLR** EA⟩ instruction.

According to the translation, instructions begin and end at precise multiples of P. While this simplification is adequate for present purposes, since the timing constants in the program model are two orders of magnitude smaller than those in the timing diagram model, it ignores two potentially important aspects: oscillator inaccuracies and processor states within machine cycles.

No oscillator is perfect. At the very least such 'clocks' drift with respect to one another and against an ideal notion of real time [Kop97, Chapter 3]. Such inaccuracies, although usually small, can be important in certain applications, and, more fundamentally, their existence makes the decision to conflate processor time and real time in the program model doubtful. They could be incorporated into the program model by increasing the resolution of model time and widening both invariants, for instance to $\mathsf{x} \leq \mathsf{P} + \epsilon$, and guards, $\mathsf{x} \geq \mathsf{P} - \epsilon$. In this way, the (idealised) behaviour of the execution platform is encoded in the program semantics, and any approximations, even just $\epsilon = 0$, are clearly stated. Rather than blend the different aspects of platform and program so implicitly, execution platforms could be modelled separately and composed with program models; the models are then not only individually reusable, but each is likely more comprehensible, the intricacies of their interrelationships being left to the mechanics of the modelling language. In one approach [VG04], the cycle clock is modelled as a separate automaton that emits a tick action at intervals. The disadvantage, compared to simply widening the invariants, is that synchronizing with tick on each instruction transition makes it awkward in Uppaal to also synchronize with other actions, as will be done for inputs and outputs in the program model.

Aside from oscillator inaccuracies, the program model also abstracts from the detailed timing behaviour of an MCS51 device [Int94, pp. 1-17–1-20]. Each oscillator period corresponds to a *phase*, two phases make a *state*, and six states make a *machine cycle*. Instructions are fetched and executed at specific phases within a cycle. Significantly, values are written or sampled from ports in specific phases [Int94, pp. 3-33–3-35]. While many applications depend on the time taken to execute individual instructions, it is doubtful whether the correctness of a system should further depend on the finer timing details within a cycle; in any case, that is the approach adopted in this chapter. But such judgements are perhaps best made by engineers for each specific application. There is definitely a point, though, when abstractions must be made. The program model stops at the instruction level rather than the circuit level. Another model may stop at the circuit level rather than the level of classical or quantum physics. It is nonetheless important to justify abstractions, to ensure they are

[16] It was implemented using libraries developed for the testing construction, see §5.4.

congruent in some sense,[17] and to understand what approximations have been made.

The program model contains several other interesting features, listed in order of their appearance: input events, registers, output events, loops for delaying, and assumptions on how frequently range-readings are requested.

The program treats *vout* as an input line. Sometimes it reacts to changes in the signal level. Sometimes it samples the signal level. It would not be accurate to label transitions that represent instructions with voutL? or voutH? actions because their occurrence is restricted to particular instants of time, whereas not only may events of both types occur at any time, but, for the model to be input-enabled, they must be allowed to occur at any time. Instead, a separate automaton POLL is introduced. It effectively models the hardware latches of the IO port connected to *vout*. It is always ready to synchronise with voutL?, setting a variable latch to zero, and voutH!, setting latch to one.[18] The assembler instructions that poll *vout*, for either a high level ⟨**JB** VOUT, ∗⟩ or low level ⟨**JNB** VOUT, ∗⟩, are modelled as loops, at s_5 and s_6 respectively, that poll the latch variable within the timing constraints of instruction execution. The assembly program waits for a rising transition on *vout* by first blocking until the latched value is zero and then polling again until it is one.

Processor registers are modelled as variables. Instructions involving registers were handled by the initial automatic translation, including updates to registers, like the initialisation of R0 between s_7 and s_8, and comparisons against them, like the branching from s_{16} to either s_8 or s_{17}.

Output actions, in contrast to inputs, are justifiably constrained by the program model. Thus, and because it is assumed that outputs are never refused, the transitions corresponding to certain instructions can simply be labelled with output actions:

⟨**SETB** VIN⟩	⟶	vinH!
⟨**CLR** VIN⟩	⟶	vinL!
⟨**MOV** VOUT, C⟩	⟶	sample!

The loops at s_9/s_{10} and s_{12}/s_{13} cause the program to pause between changes to *vin*. The register R1 is first initialised and then decremented until it reaches zero, which accumulates individual 2∗P delays giving a total delay of W100US∗2∗P. The loops do not change the microcontroller state in any significant way, but yet they cannot be removed without changing the models observable behaviour. A contrast can thus be made with program language semantics where such instructions could be shown equivalent to an instruction that does nothing, for instance to **skip** or ⟨**NOP**⟩.

The last transition from s_{20} back to s_0 was added manually. The guard $x \geq 1500$ expresses a required minimum delay between calls to the subroutine, which is necessary to meet the requirements of the timing diagram. As there is no location invariant on s_{20}, the model allows any finite non-zero number, or even an infinite number, of repeat executions. There is also a case for modelling the possibility that the routine is never called. This can be done either by removing the location invariant on s_0, or adding a new initial state with no location invariant.

4.5.3 Verifying the program model

Not only does modelling the program offer insight into the assembly program, execution platform, and instruction set semantics, but the model itself can be validated against the timing diagram and checked for other desired properties. Validation against the timing diagram occurs indirectly by verifying that the program model is timed trace included in the driver component of the split model, but some adjustments are first required before the testing construction can be applied and, ultimately, correctness is contingent on assumptions on the environment. Both data transmission and liveness properties require extra effort.

[17] An example, in this regard, is the constructive causality of Esterel, see §2.4.3, where there is a relationship between the class of valid programs and the class of deterministic, stable circuits.

[18] There are similarities with the execution model of Chapter 3, where input events could occur at any time despite timing constraints on the occurrence of synchronous reactions.

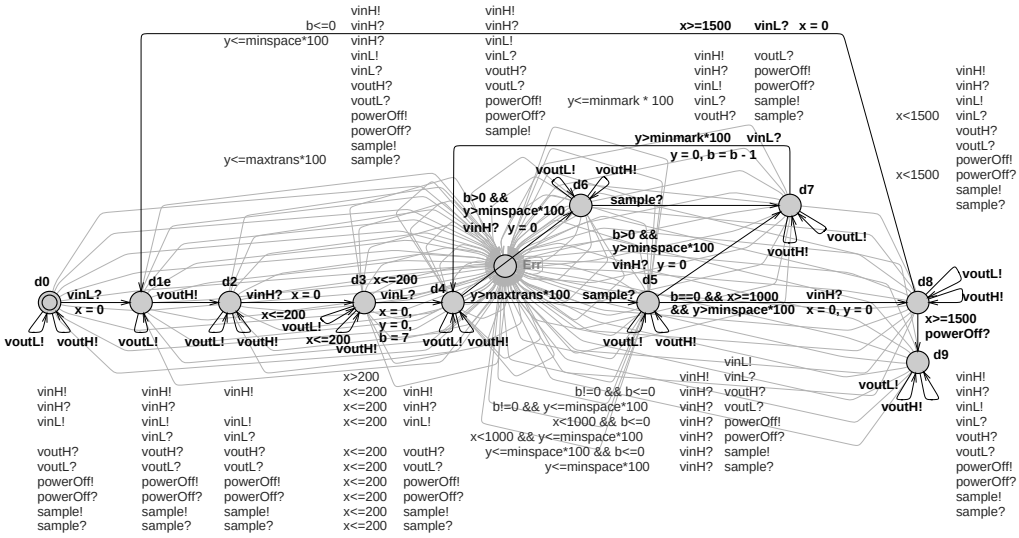


Figure 4.23: $\mathsf{DRIVER}_{\mathsf{test}}$: timed trace inclusion tester for $\mathsf{DRIVER}_{\mathsf{evt}}$

The driver specification is shown in gray above the program model in Figure 4.22b. It is essentially the driver component of the split model with constants multiplied by 100, to account for the increased timing resolution, and the top path through $\mathsf{d}_{1\mathsf{d}}$ removed to make the model deterministic. Determinism is required by the timed trace inclusion testing construction. Since the assembler program does not wait for 70ms or more after requesting a range reading but rather begins clocking data in response to an initial voutL, the unused possibility is simply removed from the model. This is sound for timed trace inclusion because it reduces the set of timed traces of the specification. The driver component from the split model is chosen over that of the broadcast model due to the issues with guards on broadcast inputs discussed in §4.4.3.2.

The gray lines in Figure 4.22b show the relationship between corresponding events in the driver specification and in the program model. A formal relation also exists between the two models. It can be verified by first constructing a testing automaton for the driver specification, like that shown in Figure 4.23 which was created automatically using the tool described in Chapter 5, and then model checking it in parallel with MCS51 and POLL to validate the property: $\mathsf{A}\Box\ (\neg \mathsf{DRIVER}_{\mathsf{test}}.\mathsf{Err})$. Uppaal shows that this property holds, and thus that the timed traces of the program model are included in those of the driver model. But what about the original timing diagram? Importantly, it is the composition of driver and sensor models that was verified against the timing diagram: there is a possibility that the driver model has behaviours which violate the timing diagram model, but which do not occur in composition with the sensor model. The verification thus gives only that the program model is correct against the timing diagram model, and by inference the assembly program against the specification, provided that it is used with a correct sensor, namely one whose timed traces are a subset of those of the sensor component of the split model.

The timed trace inclusion verification has at least two inadequacies: it says nothing about data transmission nor about liveness.

Data transmission is verified with a similar approach to §4.4.4, but some modifications to MCS51 are first required. A new variable to model the carry flag C must be updated, C = latch, on the transition between s_{14} and s_{15}. And a new variable to model the accumulator A must be updated, A = (A << 1) + C, on the transition between s_{15} and s_{16}. Finally, both a done! label and the update Dbyte = A are required on the transition between s_{20} and s_0. The resulting automaton is analysed in parallel with $\mathsf{SENSOR}'_{\mathsf{pair}}$, DRIVERAUX, and TESTER of Figure 4.21. In a larger program, the effects of instructions on processor flags would have to be modelled more systematically, possibly as part of the automatic translation.

Timed trace inclusion gives no guarantees about liveness, and such properties must be checked separately. Both deadlock freedom, $\mathsf{A}\Box\ (\neg \mathsf{deadlock})$, and that the routine

always has the possibility of running to completion, $\mathsf{E}\Diamond(\mathsf{MCS51.s}_{20})$, hold when the program model is placed in parallel with the sensor component,

$$\mathsf{MCS51} \parallel \mathsf{POLL} \parallel \mathsf{SENSOR}_{\mathsf{pair}}.$$

As does the stronger property that the routine always completes once it has been called, $\mathsf{MCS51.s}_0 \leadsto \mathsf{MCS51.s}_{20}$.

4.5.4 Discussion

In this section, an implementation of a driver to control the infrared sensor was presented, modelled, and verified. The choice of such a relatively uncomplicated execution platform restricts the number of issues that must be addressed and simplifies both the approach and exposition. But it also ignores several important issues, notably interrupts and the complex behaviour of more recent processors. Furthermore, the model itself does not capture all the timing intricacies of the MCS51, and the question of when such complexities can be ignored completely, or distilled to more abstract principles, and whether they must ever be modelled in complete detail is worthy of further thought. The verification of the program model, especially the assumptions made of the environment, suggest some avenues for future research. The program model itself reveals an unexpected feature of the infrared sensor example: the processor is too fast relative to the timing constants of the specification.

'MCS51' is Intel's name for a family of microcontrollers that implement the 8051 instruction set and memory model. Similar devices are available from several other manufacturers. They all faithfully preserve both the register behaviour and the cycles per instruction of the 8051 instruction set. The period of a cycle varies, but only in a consistent and predictable way. MCS51 assembler programs have, effectively, a tractable real-time semantics which can be exploited to validate programs against specifications that include quantitative timing constraints.

The MCS51 architecture is simple and predictable. Rough timing calculations based on counting instruction clock cycles, for instance on paper or in a spreadsheet, give reasonably accurate results that do not depend on recent execution paths or the contents of caches. Such advantages have inspired recent proposals for more predictable embedded microprocessors [EL07] and this line of research may result in a compromise between the predictability of simple processors and the performance offered by more sophisticated processors. The modelling approach of this section is not adequate for more sophisticated processors. Another limitation of the presented program model is that it ignores interrupts and their concomitant timing complexities. While interrupts are also forbidden in other approaches, like the synchronous languages and the TTA [KB03], they are still used in many applications, either as a lightweight means of scheduling, as a dynamic and flexible architecture that permits task prioritisation, or as a way of reducing system latencies.

Detailed timing models of processors and peripherals are possible, and can be made either through painstaking reconstruction from specification sheets or semi-automatic translation from actual hardware descriptions [The06]. In the same way that cycle accurate simulations are used for performance measurements and estimation in hardware design, detailed system timings are one way of arriving at tight WCET bounds. Their applicability to programming and design is, arguably, not completely clear. Too much detail can be overwhelming. It obstructs both verification and portability. WCET requirements are useful abstractions, but they could perhaps be better integrated into the programming model for applications like the sensor driver, rather than stated in pragmas, comments, or separate documentation. The sensor driver makes clear, moreover, that lower timing bounds are in some cases just as important. Chapter 6 contains further discussion on the topic of specifying timing in programming languages.

The verification in §4.5.3 suggests taking a model, like the program model, and a liveness property, like any of those stated in that subsection, and then generating a timed automaton that captures the minimum assumptions required of the environment to ensure that the properties hold of the model. To, essentially, calculate the largest timed trace set whose intersection with the timed trace set of the model gives

a third set that satisfies the desired property. Presumably, in the present case, the calculated set would include the timed traces of the more explicit sensor model. It is possible that existing research is either readily applicable or can be suitably adapted.

The program model is two orders of magnitude faster than the timing constraints in the timing diagram. Its essence, therefore, is about creating delays rather than meeting deadlines, which underscores the importance of programming with time—a theme reprised in Chapter 6—but also makes meeting the real-time constraints too easy and avoids the difficulties of using economical hardware to implement challenging specifications. There is thus less to be gained from rigorous verification. Future case studies should consider this issue.

4.6 Discussion and future work

The discussion in this section contains a summary and reflection on the results of the sensor case study in §4.6.1 and a discussion of how the approach could be improved and extended in §4.6.2.

4.6.1 Summary and contributions

The detailed study presented in this chapter has three main benefits. The case study is a concrete example of an application of rigorous modelling and analysis methods to a realistic, if simple, embedded component. The detailed examination exposes the peculiarities of one specific problem and also offers more general insights into the application domain. While the chapter makes no pretence of proposing a methodology for verifying similar embedded systems, it does contribute a specific example to the growing collection of applied timed verifications.

The infrared sensor exemplifies a particular type of embedded component. Constructing a rigorous model clarifies intricate and ambiguous details. It reveals that there is a surprising amount of detail in even such a simple artefact. The strengths and weaknesses of formal approaches are, arguably, best revealed by attempting to apply them to such arbitrary examples from engineering practice. The features of these examples are induced by both technical and nontechnical forces. They are often absent in idealised examples; perhaps because they are difficult to concoct artificially, or because they are deemed unnecessary distractions. Yet these peculiarities are central to the current practice of embedded development.

It is challenging, however, to extract general insights from the study of such arbitrary examples. Some peculiarities may be truly idiosyncratic and others may occur in many different applications. On the other hand, a device like the infrared sensor may be studied simply because it exists and research can try to influence the development of future creations.

Some questions cannot be answered from a single case study, but it does seem reasonable to make four general observations. First, time is integral to some behavioural specifications and not simply a nonfunctional requirement to be considered in a later design stage. While timing constraints can rule out sequential behaviours, as exemplified in the correctness argument for Fischer's Protocol [AL94, §3.4], this is not their only purpose. Second, behaviour in time is important, and not always just that deadlines be met but also to determine precisely when an action must occur. Some commands are only given for their effect on the timings of events. Third, timing behaviour results from an interaction of programming language semantics and platform limitations and inaccuracies. Creating models involves making choices about abstraction and separating concerns. Last, even a seemingly simple timing diagram can express quite complex relations between events and present difficulties of interpretation.

The case study is a concrete example of modelling a timing diagram and assembly language program, and applying techniques for timed trace inclusion to verify a formal relation between the two models in Uppaal. Techniques for modelling timing diagrams exist in the literature, though usually at the bus-level rather than for component interfacing. Nor is the verification technique new, other applications have been

verified with it [JLS00, Sto02]. The sensor example, though, has its own peculiarities and its details have been treated with more attention than is perhaps usual. A specific example of embedded software verification has been presented rather than a general methodology for verifying assembly language programs against timing diagrams. Verifying assembly language programs of realistic size requires specific abstractions and techniques [Sch08], even when real-time behaviours are ignored.

4.6.2 Possible extensions and improvements

The work described in this chapter could be improved or extended in several ways. Full equivalence of the timing diagram model with each of the split models could be achieved and verified. It is also worth questioning whether a single automaton timing diagram model is really necessary. Moreover, whether and when it is better to manually model a timing diagram or to transcribe it into a formalized notation is not clear: more comparison and exploration of the various formal notations is required, as is better tool support. There are many ways of implementing drivers for the infrared sensor, but only one has been examined in this chapter. Modelling different implementations can offer insights into programming techniques and several other issues that are central to the design and implementation of embedded systems.

The timing diagram has been thoroughly formalised as a timed automaton. One or two small adjustments would be possible but the basic model is complete. The split models could be improved by ensuring that their timed traces include those of the timing diagram model; that is, inclusion in the other direction and hence timed trace equivalence. This would require, for a general solution, that the testing technique be adjusted to handle specifications comprising more than one automaton, which would mean coping with non-determinism, when possible, using step refinements [Sto02, §A.1.5] and possibly other techniques to handle non-determinism from interleavings.

It could be said that the timing diagram model is unnecessary: modelling and verification could just start from a split model. There is some force to this argument. The biggest advantage of a single automaton model, at least for the sensor example, is simplicity: it is easier to argue that it correctly models the timing diagram. A second advantage is that several flaws were discovered and corrected by the process of constructing two models and validating one against the other.

Several ways of improving the assembly language model have already been suggested in §4.5.4, but perhaps the greatest omission is the lack of alternative implementations and associated models. For instance, interrupt-driven and timer-driven assembly language routines, similar versions in C, and a circuit or Field Programmable Gate Array (FPGA) implementation. More importantly, the driver could be implemented in Esterel and given real-time semantics using the two-parameter approach of Chapter 3. Each implementation would present new modelling problems. Together they would catalog some of the different ways that engineers use to express timing behaviours.

More generally, studying different implementation approaches and corresponding models offers insight into the mix of ideal semantics, platform imperfections, and details of triggering that are an important, but sometimes overlooked facet of rigorous approaches to embedded systems development. These issues are important, and particularly challenging because several timing models are involved. They have already been approached from one perspective in Chapter 3, and they will be taken up again in Chapter 6. But first the timed trace inclusion testing technique used in this chapter is described in more detail and extended in the next.

Chapter 5

Validating timed trace inclusion in Uppaal[†]

5.1 Introduction

Models help to design, simulate, verify, and synthesise embedded systems. Multiple models may be constructed for a system describing it from different perspectives and at different abstraction levels. Details may be added to specify an implementation, or removed to make analysis practicable. The correctness of a design may depend on the relations between models, which in turn may require validation.

In Chapter 4, multiple models were constructed to analyse a simple embedded systems application from different perspectives and at differing levels of detail. Two models were considered related when the timed traces of one were included in the timed traces of the other. Each relationship was verified by constructing a validation automaton and performing reachability analysis.

Constructing a validation automaton manually is straightforward but can be tedious and error prone. There is thus motivation for developing a tool to do it automatically. Automation is particularly beneficial because insights gained while experimenting with timed trace inclusion can lead to improved specifications which in turn require new constructions for repeated validations.

The Uppaal[1] modelling language [LPW97, BDL04, BW04] contains several features for creating succinct models: channel arrays, selection bindings, and quantifiers within expressions. It turns out that these features make model transformations, such as the validation construction, more challenging, even though they are not fundamentally more expressive. Many of these extended features are demonstrated in §5.1.1 where an extension to the standard railway controller example is presented. Unlike the models of the previous chapter, this example cannot be addressed directly by the standard validation technique.

In §5.2, many of the extended Uppaal features are formalised in a *process* model. Processes are transformed into timed automata by expanding the extended features. The formalisation is sufficient to explain and justify the extended construction. Unlike other formalisations [BV08], the aim is not to study semantic issues.

The main contribution of this chapter is contained in §5.3. It shows how, when possible, the existing validation construction can be extended to incorporate advanced Uppaal features while preserving concision and parameterisation. The extensions render timed trace inclusion techniques applicable for a wide class of embedded systems models, including the extended railway controller. The explanation and justification of the extended technique necessitates a formalisation of Uppaal that is more concrete than usual. The technique provides insights into the automatic manipulation of Uppaal models that may apply to other types of transformations.

[†]This chapter is based on a published paper [BS08b].

[1](Classic) version 4.0.6

An implementation of the extended technique is described in §5.4. The result of applying it to the railway case study is presented in §5.5.

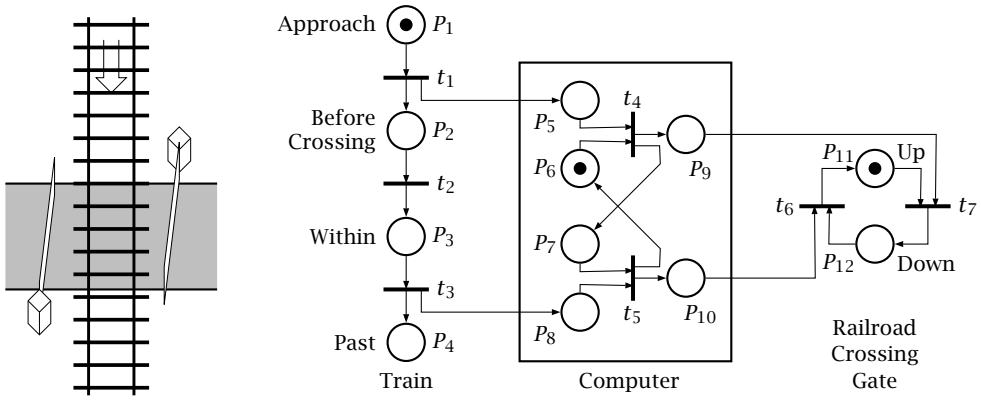
5.1.1 A flexible railway controller

The case study presented in the previous chapter involved details typical of embedded systems development. There the aim was to preserve and treat such concrete characteristics accurately. In this section an existing and more abstract case study is adapted. Here the aim is to illustrate features of Uppaal that were not needed in the previous chapter, and to motivate extensions to the standard validation technique.

Many formal methods case-studies are idealised versions of industrial systems. Examples include a mine pump system [KMSL83], a manufacturing production cell [LL95], and the railway crossing controller [LS85] discussed in this section. These case-studies have the flavour of real-world scenarios but typically ignore the sort of small details that were considered for the infrared sensor. They aim instead to embody features that researchers believe are important and common to a class of systems, or otherwise necessary for formalisms and methodologies to address.

5.1.1.1 Background

Railroad crossings are frequently used to demonstrate timed and hybrid system modelling and verification techniques.



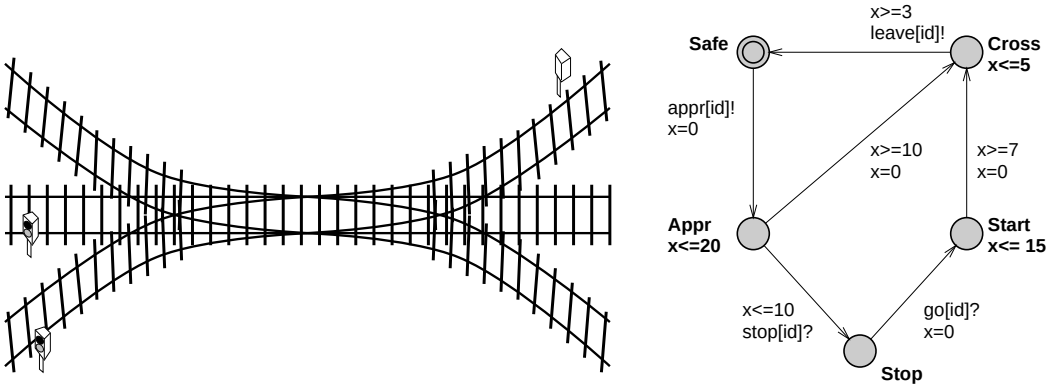
(a) Scenario **(b)** Petri net model [LS85]

Figure 5.1: Basic railway crossing

The original version [LS85] models a single one-way track, a controller, and a crossing gate as a Petri net, Figure 5.1.[2] An approaching train triggers transition t_1 which signals the controller. The controller lowers the gates by triggering t_7. Meanwhile the train enters, t_2, then leaves, t_3, the crossing. The computer lifts the gates, t_5, when the train is gone. The gates must be down when the train is in the crossing; if P_3 is marked then P_{12} must be too. This is only true of the model in Figure 5.1b if its reachable state-space is restricted by assigning minimum and maximum firing times to transitions such that $min(t_2) > max(t_4)$ and $min(t_2) > max(t_7) + max(_4)$. Time is used in this example to constrain events that are otherwise causally independent.

This example has been adapted to demonstrate other approaches for analysing timed models [JS88]. It has also been generalised [HLJ93] to a version where the crossing spans multiple tracks that carry trains in both directions. Solutions must satisfy both a safety property, that the gate is down whenever one or more trains are within

[2] Petri nets are directed bipartite graphs; nodes are either *places*, drawn as unfilled circles, or *transitions*, drawn as bars. Places may contain tokens, each indicated by a filled circle drawn within the place. A transition may be the destination of arcs from *input places*, and the source of arcs to *output places*. A transition can *fire* when each of its input places contains at least one token; doing so removes a token from each input place and adds a token to each output place. Multiple transitions may fire together if sufficient tokens are available. Several references [Pet81, Rei85] describe Petri Nets.



(a) Scenario for $N = 3$ **(b)** Model of train [WPD94]

Figure 5.2: Simple railway control system

the crossing, and a utility property: the gate is up as often as possible. One such solution is given in terms of timed automata [HL94].[3]

Uppaal is distributed with a different railway control model [WPD94] consisting of a junction where several lines converge temporarily to a single track, as, for example, at a bridge or station. The case of three convergent tracks is illustrated in Figure 5.2a. Rather than guarding a path that crosses the railway by raising and lowering a gate, a controller commands trains to stop and start as they approach the junction.

Trains are modelled as timed automata, Figure 5.2b. Each is assigned a unique id from the type id_t which contains all integers between 0 and $N - 1$ inclusive. Trains communicate with the controller by synchronising on four channel arrays:

appr[N]	signals when trains approach
leave[N]	signals as trains leave
stop[N]	commands trains to stop
go[N]	commands trains to resume

The id value selects a particular channel from an array. An approaching train sends appr[id]!, or triggers a sensor with the same effect, and resets a local clock x. It will then begin to cross the junction between 10 and 20 time units later unless the controller synchronises on stop[id]?, in which case the train stops and waits for another synchronisation on go[id]? before continuing through the junction. Trains take between 3 and 5 units to cross and they synchronise on leave[id]! when they have.

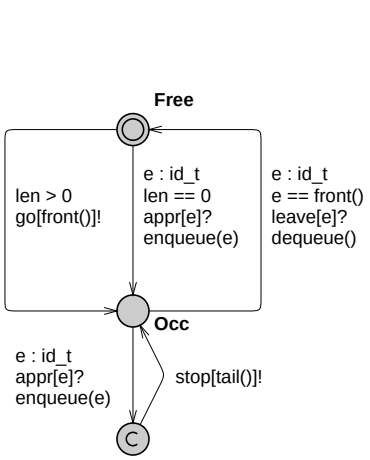
Controllers must coordinate N trains and satisfy two properties. Safety: only one train tries to cross at a time, and liveness: all trains that approach eventually leave.

An example controller is provided with the Uppaal distribution. The automaton is shown in Figure 5.3a and its subroutines, in Uppaal's C-like description language, in Figure 5.3c. The controller begins in the Free state. The middle transition from Free to Occ is labelled with all four types of transition label:

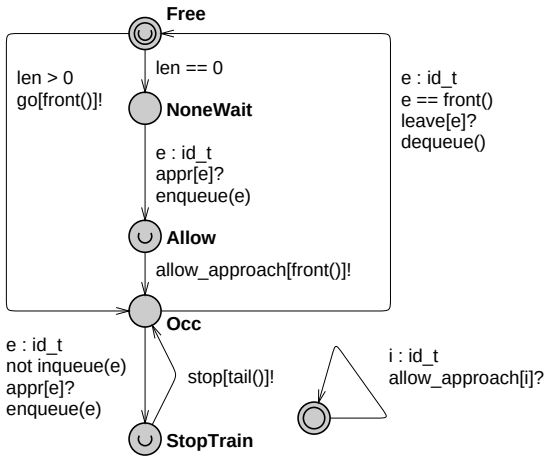
e : id_t	*selection bindings*: e may take any value from the type id_t.
len==0	*transition guard*: a condition on the local variable len.
appr[e]?	*synchronising action*: an input action on the eth channel of appr.
enqueue(e)	*update statement*: this procedure is called when the transition fires.

Selection bindings are effectively a macro notation for parameterized transitions, each valuation giving a separate transition between the source and destination states. In this case the controller is willing to accept an approach signal from any of the train models, and when it does the id of that model is queued in a local array. When the train passes through the crossing it triggers the transition from Occ back to Free and the id is removed from the queue. Any trains that arrive before an earlier one has left the crossing are added to the queue and commanded to stop. The two transitions, to queue

[3] There are also other timed automaton railway gate models [AD94, p. 42].



(a) Original controller [WPD94]



(b) Modified controller

```
id_t list[N+1];
int[0,N] len;

void enqueue(id_t ele)
{
  list[len++] = ele;
}

id_t front()
{
  return list[0];
}
```

```
void dequeue()
{
  int i = 0;
  len -= 1;
  while (i < len)
  {
    list[i] = list[i+1];
    i++;
  }
  list[i] = 0;
}
```

```
id_t tail()
{
  return list[len-1];
}

//additional function
bool inqueue(id_t i)
{
    return exists (j:id_t)
            j < len &&
            list[j] == i;
}
```

(c) Controller procedures and functions [WPD94]

Figure 5.3: Original and modified controllers

the id and issue the stop command, are linked through a committed location which ensures that neither may time pass, nor transitions from non-committed locations in parallel occur, between them.

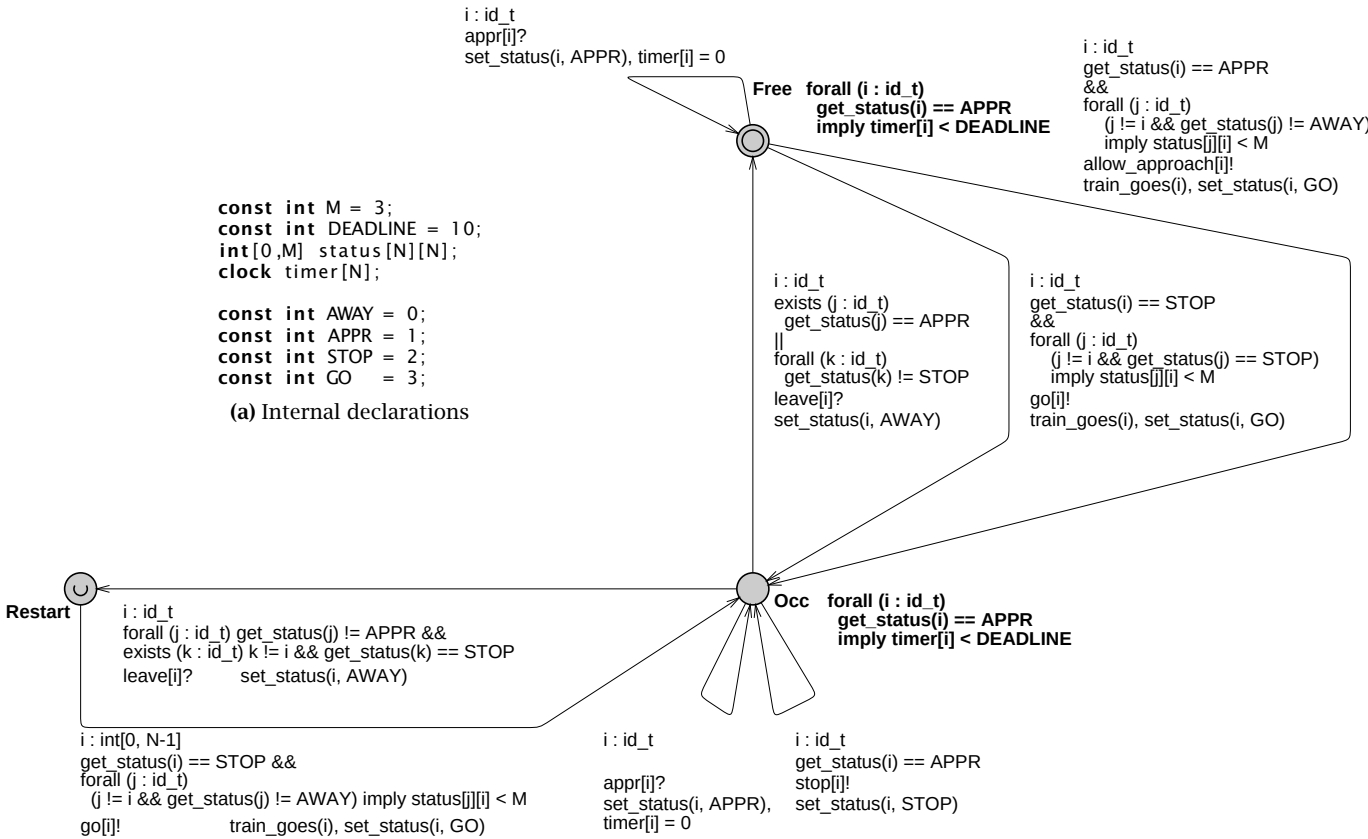
In the original controller the go channel is marked urgent. Since time may not pass when a synchronisation is enabled on an urgent channel, the next queued train will be commanded to go as soon as the last train has left the junction.

The original controller is now altered to remove features that cannot be addressed by the technique described in this chapter. The resulting model is presented in Figure 5.3b, where four changes have been made:

1. The go channel is no longer urgent. Instead the Free location is made urgent to ensure that go[front()]! may only occur without delay. This relies on the train model always being ready to synchronise on go[id] after having been stopped. It necessitates the addition of an extra state NoneWait because inputs on appr cannot be rushed. Although the extended validation technique can handle urgent channels, combining them with invariants, as will be required, can give validation automata with guards containing clocks in a form that Uppaal rejects, see §5.3.4.

2. Committed locations are not addressed by the tool and have been replaced by urgent locations. This does not affect the model because no other transitions can occur anyway since the trains only communicate with the controller and not with one another.

3. The allow_approach[front()]! action was added to make observable the decision to let an approaching train pass. The reason for this is discussed in the next section. A single location automaton is added in parallel to ensure that this action is never blocked. A broadcast channel cannot be used for reasons that will be discussed in §5.2. The source state of this transition, Allow, is urgent to ensure that the action happens without delay.



(a) Internal declarations

(b) Automaton

```
int[0,3] get_status(id_t t)
{
  return status[t][t];
}

void set_status(id_t t,
     int[0,3] s)
{
  if (s==APPR imply
       status[t][t]==AWAY)
    status[t][t] = s;
}

void train_goes(id_t t) {
  int i;
  for (i=0; i<N; ++i)
  {
    if (i != t)
    {
      if (get_status(i)==STOP
          && status[i][t]<M)
        status[i][t]++;

      status[t][i] = 0;
    }
  }
}
```

(c) Functions and procedures

Figure 5.4: Flexible railway controller

4. In the closed model, no train may ever send two consecutive approach signals. This assumption is made explicit by adding an extra guard on the appr[e]? transition. A self-loop transition is added to Occ to preserve the property that synchronisation on appr is never delayed by the controller.

5.1.1.2 A flexible controller

The original railway controller manages the junction in a rigid way. Trains are only allowed to pass in the same order that they arrive. It is, nevertheless, a good implementation because it is deterministic and straightforward. Queueing trains is effective and fair. It also ensures the required liveness property. But suppose that a more flexible specification is desired, one that allows controllers to favour trains already in motion, or to prioritise certain tracks, or to delay decisions for longer.

A candidate solution is presented in Figure 5.4. The controller there maintains a two-dimensional array, Figure 5.5, with, for each train, a status—AWAY, APPR, STOP, or GO—and, if it is stopped, the number of times trains on other tracks have gone through ahead of it. The routines in Figure 5.4c read and update the array. The set_status() procedure only changes a status value to APPR if it is currently AWAY; while this behaviour is inherent to the train model, it is made explicit because timed trace inclusion will be validated without making any assumptions on the occurrence of inputs. When a train crosses the junction, train_goes() is called to increment the counters of all stopped trains and to clear the counters for the departing train.

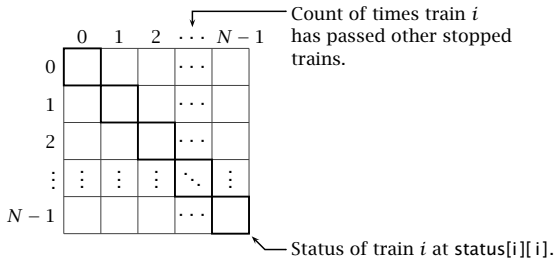


Figure 5.5: Status array

The controller also maintains an array of timers, one for each train. A timer is reset when the corresponding train approaches and monitors the deadline for sending a stop signal. By not sending stop signals immediately, the controller may wait for another train to leave the junction, or for a train with higher priority to arrive. But it also increases dependence on train speed and reduces safety margins.

The controller begins in the Free location, Figure 5.4b, which has an invariant to force an outgoing transition when any of the approaching trains require action before it is too late to request a stop. The invariant contains a universal quantifier binding, **forall** (i:id_t), over a clock variable, timer[i].

The self-loop on Free synchronises with notifications from approaching trains; it sets the appropriate internal status variable and timer. Both of the two other outgoing transitions let a train go through the junction. The innermost signals a stopped train to resume its journey, the other grants the crossing to a moving train—that train will not receive a stop signal. Both transitions have a nesting of selection and universal bindings to ensure that trains on one track cannot pass before a train stopped on another more than M consecutive times. This mechanism ensures the liveness property with finite control state. The transition that grants the crossing to a moving train would normally be a τ-transition since it is a decision internal to the controller, but the validation construction can only be applied to a deterministic controller; hence the need to differentiate which train is allowed to approach. The original controller was also modified so that the validation automaton would know when the same decision had been made.

The controller is in the Occ location when it considers that a train has been scheduled to cross. Trains that subsequently approach are treated as in the Free location; the relevant status is set to APPR and the corresponding timer is reset. Trains with a status

of APPR may be told to stop at any time, via the other self-loop transition. A location invariant again ensures that this happens quickly enough.

One of two actions occur when a train leaves the crossing. If another train is moving, tested with existential quantification **exists** (j:id_t) get_status(j)==APPR, or no trains have been stopped, the controller returns to Free. This guard removes the possibility of a train being stopped forever if no other trains arrive. Alternatively, a stopped train is resumed without delay by taking the transition to the urgent Restart location. Removing the first conjunct of the guard on the transition between Occ and Restart, **forall** (j:id_t) get_status(j) != APPR, gives a controller that satisfies the safety and liveness properties but that has a non-deterministic choice between entering Free or Restart on leave[i]? when there are both stopped and approaching trains. Non-determinism, however, complicates or precludes the validation technique.

Model checking succeeds for values of N between 1 and 4 inclusive, larger values exceeded the available computing capacity. It can be shown that the flexible controller satisfies the same properties as the original and modified controllers,[4] including the main safety property:

```
A□ forall (i : id_t)
     forall (j : id_t)
       Train(i).Cross and Train(j).Cross imply i == j,
```

and a liveness property for every i in id_t: Train(i).Appr −−> Train(i).Cross

Are the behaviours of the original controller included in those of the more liberal controller? To apply the basic validation technique, the channel arrays, selection bindings, and quantifiers would have to be unravelled for each fixed value of N. The resulting model would likely be difficult to examine, due to the loss of original structure and the increased number of transitions, reducing confidence in the result and making counter-example traces difficult to follow. The underlying structure would also be obscured and difficult for a model-checking engine to exploit. Instead, the remainder of this chapter will describe how the validation construction can be extended to address more Uppaal features. The technique is applied to the liberal controller model in §5.5.

5.2 Uppaal

An Uppaal model comprises multiple communicating components. Each component is typically presented as a (timed safety) automaton, but a more concrete formalisation is required to describe and understand the extended validation technique. This section contains the presentation of a novel process model for Uppaal components with explicit variable valuations, selection bindings, and channel arrays.

Several Uppaal features are excluded from the process model.

Urgent nodes are those where delays cannot occur. A model without urgent nodes that satisfies precisely the same properties can be produced by adding a clock x_u, which is reset on every edge. Urgent nodes are then replaced with normal nodes whose invariants include the additional conjunct $x_u \leq 0$, see Figure 2.8.

Urgent channels and shared variables are not formalised in the process model but they are treated in §5.3.4 and addressed by the implementation.

Committed nodes are used to express the atomicity of action sequences. The edges leaving committed nodes have priority over those leaving other nodes. They are not addressed.

Broadcast channels provide one-to-many synchronisations. An output action on a broadcast channel triggers all input actions that are enabled on the same channel. A broadcast output can occur even when no inputs are enabled on its channel.

Models with broadcast inputs cannot be transformed into validation automata because they become broadcast outputs that may always occur irrespective of how

[4] These properties are taken directly from the original example.

the model being validated behaves. Broadcast outputs, however, become inputs in the validation automaton and may be treated normally, at least in principle.

Broadcast outputs are potentially useful for validating whether a pattern of synchronisations between two input-enabled automata, each relying on assumptions embodied in the other, is timed trace included in a protocol model; as was done in §4.4. Unfortunately, as is there discussed, Uppaal rejects transitions on broadcast inputs when they have clock variables in their guards. This occurs when clock variables are used in the guards of transitions on broadcast outputs, which is allowed, or in location invariants, since the transformation includes them in validation automaton transition guards. The technique is thus of limited use until Uppaal is modified to lift the restriction.

The standard validation construction for a model with no broadcast inputs may have transitions to the error state on broadcast outputs. They can be removed provided no model containing reachable transitions on broadcast inputs is ever tested; it would not anyway be timed trace included in the original model.

Priority ordering gives preference to some edges over others when a model is validated or simulated. The enabled edges of a process, respectively on a channel, will occur before those of lower priority processes, respectively channels. Priority orderings effectively prune transitions from the timed transition system implicitly defined by a model. They are not addressed.

5.2.1 Variables, expressions, and valuations

Some preliminary definitions and notation are required before an Uppaal process can be defined. They are presented in this subsection.

Uppaal models may contain clock variables, which express timing constraints, and data variables, which facilitate concise and flexible process models.

Most data types can be ignored, without loss of generality, including boolean variables, record types, and arrays of anything but channels. The set of all variables Vars encompasses the two remaining categories of types: bounded sets of integers, written $[l,u]$ where $l \leq u$ and $\forall i \in \mathbb{Z}.\, l \leq i \leq u$ implies $i \in [l,u]$, and finite scalar sets, written $[\mathrm{S}]$. Both integer and scalar types can index arrays, and be bound by quantifiers in expressions and by selection bindings over process edges. Scalar variables may only be directly assigned to one another and compared for equality, so as to permit optimisation during model checking. Each scalar declaration, for example, **scalar**[5] ids, gives a new disjoint set. Type aliases, stated with **typedef**, preserve set identity.

Expressions are built from variables, constants, function calls, operators, relations, and quantifiers. They are treated as members of a set Exprs. The set of unbound variables in $e \in \mathsf{Exprs}$ is written $\mathsf{freevars}(e)$. Each expression e denotes either a truth value, an integer, or a value from a scalar set. Its value depends on the values assigned to variables in $\mathsf{freevars}(e)$. A valuation val_V gives a value $\mathsf{val}_V(v)$ of appropriate type for each variable v in the finite set V. The set of all valuations for a given set V is written Vals_V. Valuations may be composed:

$$\mathsf{val}_{V_1} \triangleright \mathsf{val}_{V_2}(v) = \mathsf{val}_{V_2}(v) \text{ if } v \in V_2, \text{ and otherwise } \mathsf{val}_{V_1}(v).$$

The value of expression e with respect to valuation val_V is written $[\![e]\!]_{\mathsf{val}_V}$. It is only defined if $\mathsf{freevars}(e) \subseteq V$.

Given a set of clock variables K, let val_K^0 be the valuation that maps each $k \in K$ to 0, and val_K^{+d} for the valuation mapping each $k \in K$ to $\mathsf{val}_K(k) + d$.

An update function $\delta_V \in \Delta_V$ maps one valuation val_V to another val'_V. Clock variables must either remain unchanged, or be set to 0; arbitrary values cannot be assigned. The set of clock variables reset by an update δ_V is written $\mathsf{resets}(\delta_V)$.

5.2.2 Channels and actions

Channel arrays are convenient for specifying a parameterisable number of channels; a sequence of expressions can select a specific channel for synchronisation. The con-

trollers of Figure 5.3 use channel arrays to communicate with N trains. A precise notation is necessary to define transformations involving channel arrays.

Let Chansets be a finite collection of *channel sets*. Every $C \in \mathsf{Chansets}$ is associated with a sequence of n_C types, each either of the form $[l,u]$, or [S]. A single element $C_{\langle i_1,\ldots,i_{n_C}\rangle}$ of the set C is designated by a sequence of values $\langle i_1,\ldots,i_{n_C}\rangle$ of appropriate types. Non-array channels are denoted by singleton channel sets, for example C_ϵ. Two *actions* are associated with each channel. To each channel set C are associated sets of input and output actions,

$$\begin{aligned} C? &= \{\, C_{\langle i_1\ldots i_{n_C}\rangle}? \mid \text{for all } \langle i_1,\ldots,i_{n_C}\rangle \,\}, \text{ and,} \\ C! &= \{\, C_{\langle i_1\ldots i_{n_C}\rangle}! \mid \text{for all } \langle i_1,\ldots,i_{n_C}\rangle \,\} \end{aligned}$$

Let $^{?}/_{!}$ stand for either direction, used consistently in a rule, and $^{!}/_{?}$ for its complement.

For $\mathbb{C} \subseteq \mathsf{Chansets}$, let $\mathbb{C}? = \bigcup_{C\in\mathbb{C}} C?$ and $\mathbb{C}! = \bigcup_{C\in\mathbb{C}} C!$.

Let $\mathsf{Acts} = \mathsf{Chansets}? \dot\cup \mathsf{Chansets}!$ be the set of possible actions on Chansets. The set $\mathsf{Acts}_\tau = \mathsf{Acts} \dot\cup \{\tau\}$ also includes the silent action τ. The complement of an action $a \neq \tau$, is written $\overline{a}$, that is if $a = c?$ then $\overline{a} = c!$ and vice versa, and similarly for action sets.

Subsets of a channel set C are designated by a sequence of expressions $\langle e_1,\ldots,e_{n_C}\rangle$. Evaluation is lifted to such designations to specify a single channel from the subset,

$$[\![\, C[e_1,\ldots,e_{n_C}] \,]\!]_{\mathsf{val}_V} = C_{\langle [\![e_1]\!]_{\mathsf{val}_V},\ldots,[\![e_{n_C}]\!]_{\mathsf{val}_V}\rangle}.$$

5.2.3 Processes

Guard expressions in Uppaal are restricted in form for efficient manipulation as symbolic zones, refer §2.2.3.2. They are built from clock terms.

Definition 5.2.1
For predicates $p_{nclk} \in P_{nclk}$, the set of *clock terms* $T_{clk}(K,V)$, where $K \cap V = \emptyset$, is the smallest containing

1. p_{nclk} where $\mathsf{freevars}(p_{nclk}) \subseteq V$,
2. $k\ R\ e$ where $k \in K$, $R \in \{<,\leq,=,\geq,>\}$, $\mathsf{freevars}(e) \subseteq V$, and,
3. $k_1 - k_2\ R\ e$ where $k_1,k_2 \in K$ and $\mathsf{freevars}(e) \subseteq V$. ∎

Form 1 permits boolean-valued, clock-free expressions. In form 2 a clock variable is compared with an integer-valued, clock-free expression, in form 3 the difference of two clock variables is compared similarly. P_{nclk} is a set of predicates over terms that do not contain clock variables.

Definition 5.2.2
The *guard expressions* on K and V, $E_g(K,V)$, where K and V are disjoint sets of clock and non-clock variables respectively, is the smallest set allowed by the rules:

$$\frac{p \in T_{clk}(K,V)}{p \in E_g(K,V)} \qquad \frac{p,q \in E_g(K,V)}{p \wedge q \in E_g(K,V)} \qquad \frac{p \in E_g(K,V) \qquad v \in \mathsf{Vars}}{(\forall v.\, p) \in E_g(K,V)}$$

Uninterpreted clock-free expressions p_{nclk} may contain existential quantifiers and disjunctive sub-terms, but guard expressions may not. Invariant expressions $E_{\mathsf{inv}}(K,V)$ are the subset of guard expressions where comparisons in clock terms are restricted to $R \in \{<,\leq\}$.

There is now sufficient background to formalise processes as they are modelled within Uppaal.

Definition 5.2.3
A process $\mathcal{P} = (N, n_0, K, V, \mathsf{val}_V^{\mathrm{init}}, \mathsf{inv}_{V\cup K}, E)$ over Acts_τ comprises finite sets of nodes N, clocks K, variables V, and edges E; an initial node n_0 and valuation $\mathsf{val}_V^{\mathrm{init}}$; and $\mathsf{inv}_{V\cup K}$, a mapping from N to expressions in $E_{\mathsf{inv}}(K,V)$. The labelled edges connect pairs of nodes, where $S \subseteq \mathsf{Vars}$,

$$E \subseteq N \times S \times E_g(K, V \cup S) \times 2^{\mathsf{Acts}} \times \Delta_{V\cup K}(V,S) \times N.$$

An edge $(n, S, e, C[e_1, \ldots, e_{n_C}]^?_!, \delta^{(V,S)}_{V \cup K}, n') \in E$ is written

$$n \xrightarrow[C[e_1,\ldots,e_{n_C}]^?_!]{S \;\; e \;\; \delta^{(V,S)}_{V\cup K}}_E n',$$

(and similarly for τ-transitions) where n and n' are, respectively, source and destination nodes; S is a finite set of *selection bindings*, $S \cap V = \emptyset$; $\mathsf{freevars}(e) \subseteq V \cup S$; $\delta^{(V,S)}_{V\cup K}$ is an update for valuations over $V$and K that depends on a valuation of $V$and S; and the action set is either $\{\tau\}$, or consistent in name and direction, that is, it has either of the forms $C[e_1, \ldots, e_{n_C}]?$ or $C[e_1, \ldots, e_{n_C}]!$ where $\mathsf{freevars}(C[e_1, \ldots, e_{n_C}]) \subseteq V \cup S$. ■

5.2.4 Automata

Timed automata were presented in §2.2.3. Definition 2.2.17 is adopted in this chapter with alphabet $A = \mathsf{Acts}_\tau$, guard expressions $E_K = E_g(K, \emptyset)$, and invariant expressions $I_K = E_{\mathsf{inv}}(K, \emptyset)$. In contrast to processes, the only variables in timed automata are clock variables.

The semantics of processes are given by defining a function Γ_V that maps processes to automata by expanding references to non-clock variables. The notation $[\![e]\!]_{\mathsf{val}_V}$ stands for a partial evaluation that maps one expression to another such that

$$[\![\, [\![e]\!]_{\mathsf{val}_V}]\!]_{\mathsf{val}_{V'}} = [\![e]\!]_{\mathsf{val}_{V'} \triangleright \mathsf{val}_V} \qquad \text{if } \mathsf{freevars}(e) \setminus V \subseteq V'.$$

Definition 5.2.4
Given a process $\mathcal{P} = (N, n_0, K, V, \mathsf{val}^{\mathrm{init}}_V, \mathsf{inv}_{V\cup K}, E)$, let $\Gamma_V(\mathcal{P}) = (L, l_0, K, \mathsf{inv}_K, T)$, where $L = N \times \mathsf{Vals}_V$, $l_0 = \left(n_0, \mathsf{val}^{\mathrm{init}}_V\right)$, $\mathsf{inv}_K\left((n, \mathsf{val}_V)\right) = [\![\, \mathsf{inv}_{V\cup K}(n) \,]\!]_{\mathsf{val}_V}$, and T is the smallest relation satisfying:

$$\frac{n \xrightarrow[C[e_1,\ldots,e_2]^?_!]{S \;\; e \;\; \delta^{(V,S)}_{V\cup K}}_E n' \quad \mathsf{val}_V \in \mathsf{Vals}_V \quad \mathsf{val}_S \in \mathsf{Vals}_S}{(n, \mathsf{val}_V) \xrightarrow[[\![C[e_1,\ldots,e_{n_C}]^?_!]\!]_{\mathsf{val}_V \triangleright \mathsf{val}_S}]{[\![e]\!]_{\mathsf{val}_V \triangleright \mathsf{val}_S} \;\; \mathsf{resets}\left(\delta^{(V,S)}_{V\cup K}\right)}_T \left(n', \delta^{(V,S)}_{V\cup K}(\mathsf{val}_V)\right)} \;?,!$$

$$\frac{n \xrightarrow[\tau]{S \;\; e \;\; \delta^{(V,S)}_{V\cup K}}_E n' \quad \mathsf{val}_V \in \mathsf{Vals}_V \quad \mathsf{val}_S \in \mathsf{Vals}_S}{(n, \mathsf{val}_V) \xrightarrow[\tau]{[\![e]\!]_{\mathsf{val}_V \triangleright \mathsf{val}_S} \;\; \mathsf{resets}\left(\delta^{(V,S)}_{V\cup K}\right)}_T \left(n', \delta^{(V,S)}_{V\cup K}(\mathsf{val}_V)\right)} \;\tau$$

■

The three rules in Definition 5.2.4 differ from each other only in action type. Automata locations are formed of process nodes paired with non-clock variable valuations. After fixing a variable valuation, each process edge may still expand to multiple transitions, one for every valuation val_S of the selection bindings S. The original guard is partially evaluated against binding and variable values, the resulting guard may still depend on clock variables. The combined binding and variable values affect the update of the destination valuation, and, in the rules for ? and !, select an element from the channel set. Updates to clocks remain on the result transitions.

Later manipulations work from the observation that every edge in the automaton corresponds to a set of transitions in the process determined by the combination of control node and data valuation, and all possible assignments to S.

An Uppaal model of n processes $\mathcal{P}_1, \ldots, \mathcal{P}_n$ over Acts_τ and variables V can be mapped to a *closed system* automaton containing only τ-transitions,

$$\mathcal{A}_\tau = (\Gamma_V(\mathcal{P}_1) \parallel \ldots \parallel \Gamma_V(\mathcal{P}_n)) \setminus \mathsf{Acts}.$$

Definitions 2.3.1 and 2.3.2 describe, respectively, parallel composition and restriction.

5.2.5 Validation automata

The standard definition of the validation construction [Sto02, §A.1.5] for automata is included here. It will be lifted to processes in §5.3.

Definition 5.2.5
For deterministic $\mathcal{A} = (L, l_0, K, \mathsf{inv}_K, T)$ over Acts let $\mathcal{A}^{\mathsf{Err}} = (L \dot\cup \{\mathsf{Err}\}, l_0, K, \mathsf{inv}_K^{\mathsf{Err}}, T^{\mathsf{Err}})$ where $\mathsf{inv}_K^{\mathsf{Err}}(l \in L) = \mathsf{true}$, and T^{Err} is the least relation such that

$$\frac{}{l \xrightarrow[\tau]{\neg \mathsf{inv}_K(l)\ \emptyset}_{T^{\mathsf{Err}}} \mathsf{Err}}\ 1 \qquad \frac{a \in \mathsf{Acts}}{\mathsf{Err} \xrightarrow[a]{\mathsf{true}\ \emptyset}_{T^{\mathsf{Err}}} \mathsf{Err}}\ 2$$

$$\frac{l \xrightarrow[a]{g\ R}_T l'}{l \xrightarrow[\overline{a}]{(g\ \wedge\ \mathsf{inv}_K(l))\ R}_{T^{\mathsf{Err}}} l'}\ 3 \qquad \frac{g_{(a,l)} = \neg \bigvee \{\, g \mid \exists l'.\, l \xrightarrow[a]{g\ R}_T l' \,\}}{l \xrightarrow[\overline{a}]{(g_{(a,l)}\ \wedge\ \mathsf{inv}_K(l))\ \emptyset}_{T^{\mathsf{Err}}} \mathsf{Err}}\ 4$$

■

If there are no transitions for a pairing of action and location (a, l) the upper part of rule 4 becomes $\neg \bigvee \emptyset = \mathsf{true}$, giving a transition directly to the Err state.

The validation construction only applies to deterministic automata, which implies the absence of τ-transitions. Timed trace inclusion is undecidable in general [AD94, Corollary 5.3], but PSPACE-complete for deterministic specification automata [AD94, Theorem 6.6].[5]

Although timed trace inclusion of a non-deterministic specification S and an implementation $\mathcal{I}$ cannot be verified directly using the validation technique, an alternative approach is sometimes possible. It involves showing timed trace inclusion of two proxies: a deterministic specification S_p and a proposed implementation $\mathcal{I}_p$, where channel names can be renamed or changed to τ such that relabelling actions in S_p gives S and relabelling those in $\mathcal{I}_p$ gives $\mathcal{I}$ [Sto02, Appendix A]. Appropriate relabellings do not always exist, even when timed trace inclusion holds.

As an example of the relabelling technique, the allow_approach[i]! actions in the railway controller of Figure 5.3b and the specification of Figure 5.4b could be replaced with τ actions to give more natural models. Showing timed trace inclusion of the original models implies timed trace inclusion of the relabelled ones. Direct verification without the allow_approach[i]! actions would fail because a gate being validated could silently enter Occ—while the validation automaton remained in Free—and then perform an action that triggered a transition to Err. Even a single allow_approach! action is not sufficient because the model would still be nondeterministic. Although initially both it and the validation automaton would be synchronized, they could assign different values to their variables when synchronizing on allow_approach!, and thereby permit later transitions into Err.

5.3 Transforming Uppaal models

The basic validation construction for automata was given in Definition 5.2.5. Given a process $\mathcal{P}$ the aim is now to construct another process $\mathcal{P}^{\mathsf{Err}}$ such that

$$\Gamma_V(\mathcal{P}^{\mathsf{Err}}) \approx \Gamma_V(\mathcal{P})^{\mathsf{Err}}, \qquad \begin{array}{ccc} \mathcal{P} & \xrightarrow{\text{Definition 5.3.1}} & \mathcal{P}^{\mathsf{Err}} \\ \big\downarrow \Gamma_V & & \big\downarrow \Gamma_V \text{ (Definition 5.2.4)} \\ \mathcal{A} & \xrightarrow{\text{Definition 5.2.5}} & [\mathcal{A}^{\mathsf{Err}}] \end{array}$$

where $\mathcal{A}_1 \approx \mathcal{A}_2$ iff 1) $\mathcal{A}_1$ contains a location Err, 2) $\mathcal{A}_2$ contains a location Err, and 3) $\forall \mathcal{A}\,.\,((\mathcal{A}_1 \parallel \mathcal{A} \vDash \mathsf{A}\Box\, \neg \mathcal{A}_1.\mathsf{Err}) \Leftrightarrow (\mathcal{A}_2 \parallel \mathcal{A} \vDash \mathsf{A}\Box\, \neg \mathcal{A}_2.\mathsf{Err}))$. Commutativity only holds to within an equivalence class because the transformation of $\mathcal{P}$ into $\mathcal{P}^{\mathsf{Err}}$ requires additional states and transitions to handle unexpanded process features. In pragmatic terms, a correct extension of the original error construction allows direct validation of more models, leaving explicit expansion, Γ_V, to Uppaal.

[5] The undecidability and complexity results are obtained for timed language inclusion of timed automata without location invariants.

Definition 5.3.1
Let $\mathcal{P} = (N, n_0, K, V, \mathsf{val}_V^{\mathsf{init}}, \mathsf{inv}_{V\cup K}, E)$ where the underlying automaton $\Gamma_V(\mathcal{P})$ is deterministic and τ-free, then

$$\mathcal{P}^{\mathsf{Err}} = (N \dot\cup \{\mathsf{Err}\}, n_0, K, V, \mathsf{val}_V^{\mathsf{init}}, \mathsf{inv}_{V\cup K}^{\mathsf{Err}}, E^{\mathsf{Err}}),$$

where $\mathsf{inv}_{V\cup K}^{\mathsf{Err}}(l \in L) = \mathsf{true}$ and E^{Err} is the least relation such that,

$$\frac{}{n \xrightarrow[\tau]{\varnothing\ \neg \mathsf{inv}_{V\cup K}(n)\ \cdot}_{E^{\mathsf{Err}}} \mathsf{Err}}\ 1 \qquad \frac{a \in \mathsf{Acts}}{\mathsf{Err} \xrightarrow[a]{\varnothing\ \mathsf{true}\ \cdot}_{E^{\mathsf{Err}}} \mathsf{Err}}\ 2 \qquad \frac{n \xrightarrow[C[e_1,\ldots,e_{n_C}]^{?}/_{!}]{S\ g\ \lambda}_{E} n'}{n \xrightarrow[C[e_1,\ldots,e_{n_C}]^{!}/_{?}]{S\ (g \wedge \mathsf{inv}_{V\cup K}(n))\ \lambda}_{E^{\mathsf{Err}}} n'}\ 3$$

$$\frac{\begin{array}{c} n \in N \quad C \in \mathsf{Chansets} \quad (S', g', \langle e'_1, \ldots, e'_{n_C}\rangle) \in \mathsf{flip}(C, T) \\ \text{where } T = \left\{(S, g, \langle e_1, \ldots, e_{n_C}\rangle) \mid n \xrightarrow[C[e_1,\ldots,e_{n_C}]^{?}/_{!}]{S\ g\ \cdot}_{E} \cdot\right\} \end{array}}{n \xrightarrow[C[e'_1,\ldots,e'_{n_C}]^{!}/_{?}]{S'\ (g' \wedge \mathsf{inv}_{V\cup K}(n))\ \cdot}_{E^{\mathsf{Err}}} \mathsf{Err}}\ 4$$

The function flip maps a set of triples—selection bindings, guards, and subscript expressions—of edges for a fixed location l and channel set C, to another set of triples so as to satisfy $\Gamma_V(\mathcal{P}^{\mathsf{Err}}) \approx \Gamma_V(\mathcal{P})^{\mathsf{Err}}$. ■

Each of the four rules for constructing validation processes corresponds with one of those for constructing validation automata in Definition 5.2.5. Rule 4 applies to node/action set pairs, mapping, via the flip function, all associated edges to another set of edges to the Err state. The flip function encapsulates the treatment of selection bindings and channel arrays. It is presented incrementally in the following, beginning with singleton channel sets in §5.3.1, the simplest form first in §5.3.1.1, then with selection bindings in §5.3.1.2, and then with quantifiers in §5.3.1.3. More general channel sets are addressed in §5.3.2 using two different techniques. A partitioning technique works for subscripts over state variables in §5.3.2.1 and limited combinations of selection binding subscripts in §5.3.2.2. The insights gained lead to an improved and more general technique in §5.3.2.3. The presentation includes details that, while not required for the abstract definition, are important to implement the transformation.

5.3.1 Elementary channels

Only edges labelled with singleton channel sets are considered in this section; selection bindings are thus limited to guards. Channel arrays are treated in the next subsection.

5.3.1.1 No selection bindings or quantifiers

In the absence of selection bindings and quantifiers over expressions containing clocks, the challenge is to produce guard expressions that meet the syntactic restrictions of Uppaal. Ideally, expressions must be simplified when possible.

Clock expressions represent the intermediate results of manipulations. They must be converted to guard expressions, $E_g(K, V) \subseteq E_{clk}(K, V)$, for acceptance by Uppaal.

Definition 5.3.2
The set of *clock expressions* $E_{clk}(K, V)$ over clock K and non-clock variables V is the smallest such that

$$\frac{p \in T_{clk}(K, V)}{p \in E_{clk}(K, V)}\ 1 \qquad \frac{p, q \in E_{clk}(K, V)}{p \wedge q \in E_{clk}(K, V)}\ 2 \qquad \frac{p, q \in E_{clk}(K, V)}{p \vee q \in E_{clk}(K, V)}\ 3$$

$$\frac{p \in E_{clk}(K,V) \quad v \in \mathsf{Vars}}{(\forall v.\, p) \in E_{clk}(K, V)}\ 4 \qquad \frac{p \in E_{clk}(K,V) \quad v \in \mathsf{Vars}}{(\exists v.\, p) \in E_{clk}(K, V)}\ 5$$

■

Proposition 2
For any sets of clocks K and variables V, $E_g(K,V) \subseteq E_{clk}(K,V)$.[6] ∎

For now, the focus is limited to *quantifier-free clock expressions*, that is to those formed without using rules 4 and 5. Quantifiers may nonetheless appear in clock terms $p \in T_{clk}(K,V)$ where they do not encompass clock variables.

Quantifier-free clock expressions are closed under negation. The function neg is defined over the structure of expressions, after the pattern:

$$\mathsf{neg}(c < e) = c \geq e, \qquad \mathsf{neg}(c = e) = c < e \vee c > e,$$
$$\mathsf{neg}(p_{nclk}) = \mathsf{neg}_{nclk}(p_{nclk}), \qquad \mathsf{neg}(p \vee q) = \mathsf{neg}(p) \wedge \mathsf{neg}(q).$$

where $\mathsf{neg}_{nclk}(p_{nclk})$ negates clock-free expressions p_{nclk}.

Proposition 3
The function neg is closed over the set of quantifier-free clock expressions. ∎

Proposition 4
For a quantifier-free clock expression e and $\mathsf{val}_V \in \mathsf{Vals}_V$, $\neg [\![e]\!]_{\mathsf{val}_V} = [\![\mathsf{neg}\, e]\!]_{\mathsf{val}_V}$. ∎

The set of m edges to flip can be written as $E = \{g_1, \ldots, g_m\}$, since each edge has the same source node, none have selection bindings, each synchronises on the same action, and neither updates nor destination nodes are relevant.

The set of edges $\overline{E} = \mathsf{flip}(E)$ should contain guards such that at least one is true when all of the guards in E are false. The premise of rule 4 in Definition 5.2.5 is mimicked by forming $\mathsf{neg}(g_1 \vee \cdots \vee g_m)$. To ensure that the resulting expression conforms to the syntactic restrictions of Uppaal, it must first be converted into DNF, $\overline{g}_1 \vee \cdots \vee \overline{g}_{m'}$, before separating the clauses to give $\overline{E} = \{\overline{g}_1, \ldots, \overline{g}_{m'}\}$.

In practice it is often possible to simplify the resulting guard terms. For example, $c > 2 \wedge c \leq 2$ may be omitted completely, and $c < 2 \wedge c < 4$ may be replaced with $c < 2$. Simplification is not strictly necessary, but it improves readability which, in turn, increases confidence in the results, and makes counter-example traces easier to follow. The current implementation uses simple syntactic criteria to assess, for a pair of terms in a conjunctive clause, whether one implies or contradicts the other. A possible improvement would be to exploit a heavy-duty simplifier, like those employed in theorem proving tools.

Figure 5.6 shows both a process $\mathcal{P}_{5.6}$, which has one clock x and synchronises on channels c and d, and the corresponding validation process $\mathcal{P}^{\mathsf{Err}}_{5.6}$. The two transitions from s_1 of $\mathcal{P}^{\mathsf{Err}}_{5.6}$ to Err are labelled with complements of the actions that do not leave that state. Transitions from s_2 of $\mathcal{P}^{\mathsf{Err}}_{5.6}$ have more complicated guards

$c?, c!, d?$	$x < 4$	the node invariant
τ	$x \geq 4$	the negated invariant
$d!$	$x > 1 \wedge x < 3$	the original guard
$d!$	$x < 4 \wedge x \geq 3$	invariant and half of the negated guard
$d!$	$x \leq 1$	the other half of the negated guard

In the third and fifth lines above, simplification has removed the redundant invariant conjunct. The negated guard of $d!$ is split over two transitions to avoid disjunction over clock variables.

5.3.1.2 With selection bindings

A selection binding pairs a variable name with a bounded integer or scalar type. A set of selection bindings may be associated with an edge.[7] The bound variables on an edge may occur in the guard expression, update statement, and action subscript expressions. This last possibility is ignored until §§5.3.2.2 and 5.3.2.3 where arrays of actions are considered.

[6] Clock terms and guard expressions are defined in Definitions 5.2.1 and 5.2.2 respectively.
[7] It is a list in Uppaal, but the last of multiple identical names overrides the others.

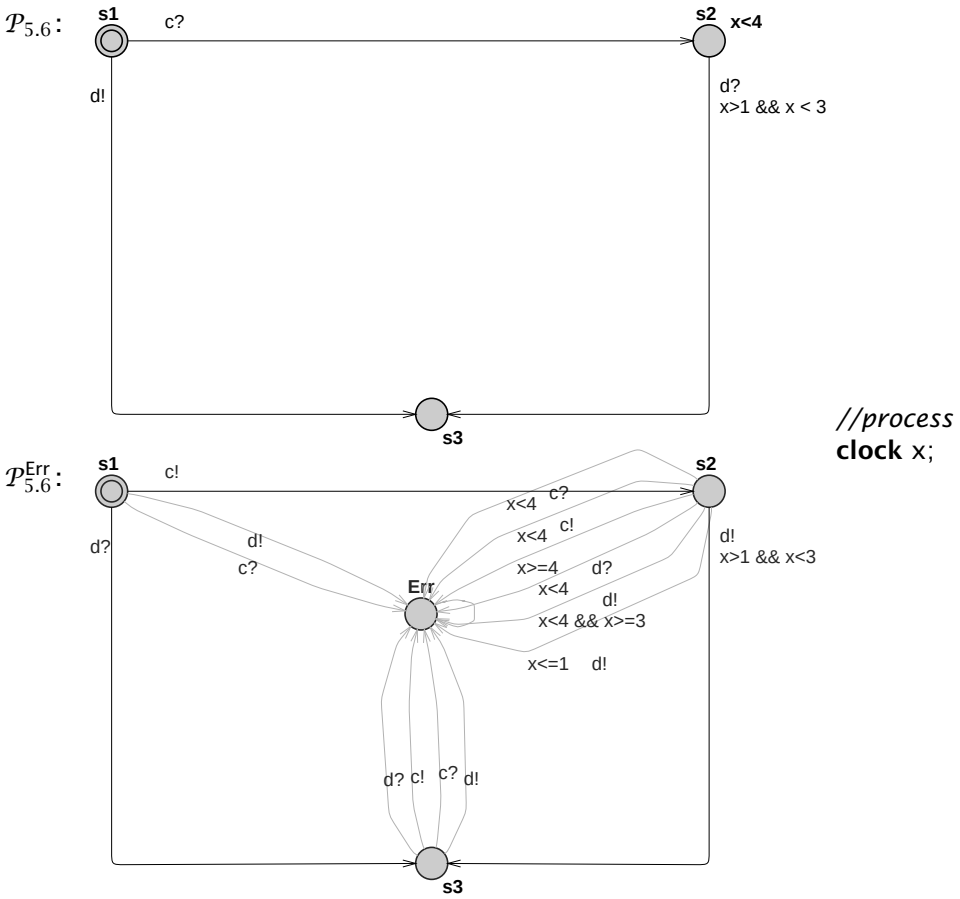


Figure 5.6: No selection bindings or quantifiers

An edge with selection bindings may represent multiple transitions in the underlying automaton, even after fixing the values of state variables, as is apparent in Definition 5.2.4. Any choice of values for the selection bindings that satisfies the guard represents a possible transition.

The set of m edges to be flipped is now written $E = \{(S_1, g_1), \ldots, (S_m, g_m)\}$, where each S_i is a set of selection variables, bound over the corresponding guard g_i. There is no loss of generality in assuming that the selection sets are pairwise disjoint since elements may be renamed if necessary. Only quantifier-free g_i are considered for now.

A transition for a fixed location and action is enabled when

$$(\exists s_{11}, \ldots, s_{1n_1}.\, g_1) \vee \cdots \vee (\exists s_{m1}, \ldots, s_{mn_m}.\, g_m),$$

where $S_i = \{s_{i1}, \ldots, s_{in_i}\}$, which can be rewritten

$$\exists s_{11}, \ldots, s_{1n_1}, \ldots, s_{m1}, \ldots, s_{mn_m}.\, g_1 \vee \cdots \vee g_m.$$

Thus, no transitions are enabled when

$$\forall s_{11}, \ldots, s_{1n_1}, \ldots, s_{m1}, \ldots, s_{mn_m}.\, \mathsf{neg}(g_1 \vee \cdots \vee g_m),$$

that is, when no guard is satisfied for any valuation of the selection bindings. The result of $\mathsf{neg}(g_0 \vee \cdots \vee g_m)$ can be converted to DNF, $\overline{g}_0 \vee \cdots \vee \overline{g}_m$, but it is only possible to assign each clause to a separate transition if the scope of each universally bound variable can be reduced to a single disjunct. One solution is to eliminate problematic quantified variables by creating a new edge for each of their possible values and every disjunct in scope. This is not possible for variables that take values from a scalar set, or from bounded integers where either of the bounds is an expression that cannot be reduced to a concrete value, for instance those involving template arguments.

The construction in Figure 5.7 shows another solution, applicable when all the selection bindings are of bounded integer type. A local meta variable s_i is introduced for each binding, taking care to avoid naming conflicts. A τ-transition from the original location to a new committed location resets each variable s_i to its lower bound L_i. The self-loop transitions on the committed location loop through all possible values of the variables. Other τ-transitions return to the original location if any of the guards are true for a variable valuation. If this is not the case for any valuation, one of the τ-transitions to Err will be taken.

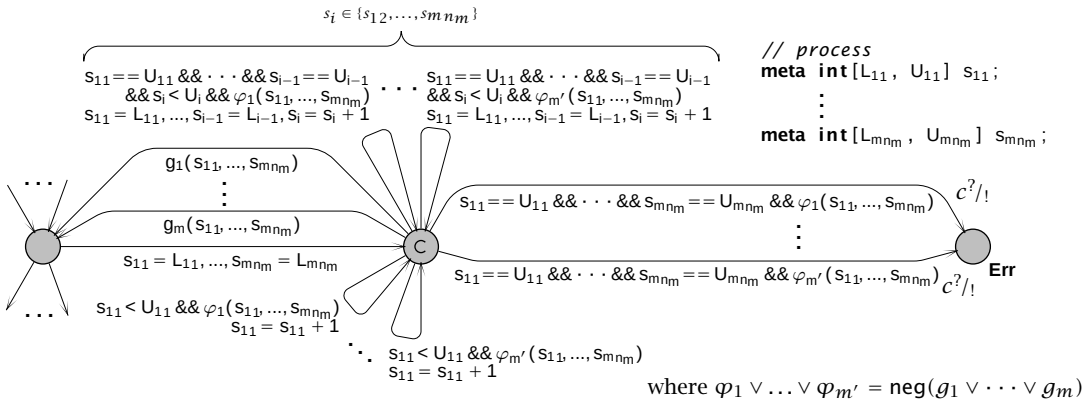


Figure 5.7: Construction for $\forall s_{11}, \ldots, s_{1n_1}, \ldots, s_{m1}, \ldots, s_{mn_m}.\, \mathsf{neg}(g_1 \vee \cdots \vee g_m)$

The current implementation does not attempt the construction of Figure 5.7, rather it simply warns when a negated expression cannot be split into separate transitions and is thus likely to be rejected by Uppaal.

Two examples where selection bindings are addressed by introducing universal quantification into the validation process are shown in Figures 5.8 and 5.9. The first gives a valid Uppaal process. The second does not.

In $\mathcal{P}^{\mathsf{Err}}_{5.8}$ of Figure 5.8 a transition from s_0 to Err occurs on $c?$ when the negated guard is true for all possible values of i.

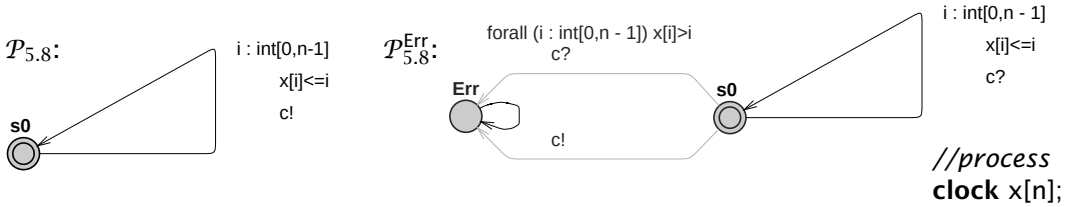


Figure 5.8: Selection bindings but no quantifiers

Another process and corresponding validation process are shown in Figure 5.9. The guard disjuncts of $\mathcal{P}^{\mathsf{Err}}_{5.9}$ cannot be split into separate transitions because of the **forall** (without using the construction defined above); this process is rejected by Uppaal.

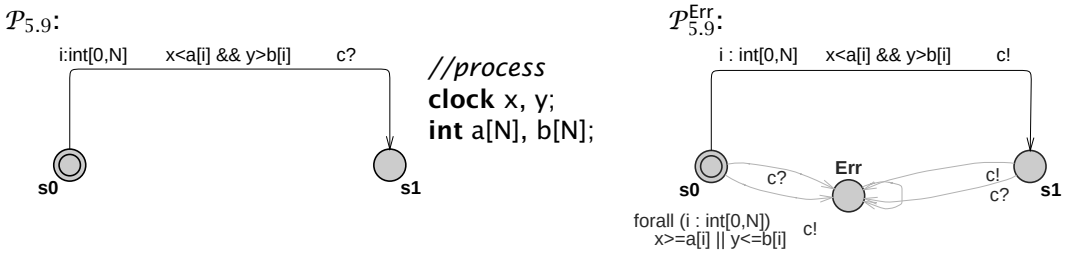


Figure 5.9: Selection bindings/negated guard clash

5.3.1.3 With universal quantifiers

The previous section showed how universal quantifiers are introduced when negating transitions with selection bindings. Now guards that already contain universal quantifiers (rule 5 of Definition 5.3.2) are considered. Guards with existential quantifiers are rejected by Uppaal,[8] and hence they are not considered.

Quantifier bindings, like selection bindings, bind a name and finite type over a subexpression. A universally quantified expression $\forall i \in [l,u].e(i)$ is equivalent to a sequence of conjunctions $e(l) \wedge \ldots \wedge e(u)$, and similarly for scalar types.

As it is possible to convert guard expressions into prenex normal form where all the quantifiers are universal, and thus order is irrelevant, a set of m edges is now written

$$E = \{(S_1, A_1, g_1), \ldots, (S_m, A_m, g_m)\},$$

where each A_i is a set of universally quantified variables binding over the corresponding g_i. Assume that all selection and quantifier sets are pairwise disjoint and further that quantified variables only occur in corresponding guard expressions, that is, for all $1 \leq i, j \leq m$, $S_i \cap A_j = \emptyset$, and if $i \neq j$ then $S_i \cap S_j = \emptyset$, $A_i \cap A_j = \emptyset$, $A_i \cap \mathsf{freevars}(g_j) = \emptyset$, and $S_i \cap \mathsf{freevars}(g_j) = \emptyset$. These assumptions can be met by renaming as required.

A transition for a given action is enabled whenever

$$\exists s_{11}, \ldots, s_{mn_m}. \forall a_{11}, \ldots, a_{mn'_m}. g_1 \vee \cdots \vee g_m.$$

Thus, there are no transitions enabled for the action when

$$\forall s_{11}, \ldots, s_{mn_m}. \exists a_{11}, \ldots, a_{mn'_m}. \mathsf{neg}(g_1 \vee \ldots \vee g_m), \tag{ψ_1}$$

which is problematic because Uppaal will reject it if any of the guards contain clock variables. One solution is to rearrange the expression, if possible, into the form

$$\exists a_{11}, \ldots, a_{mn'_m}. \forall s_{11}, \ldots, s_{mn_m}. \mathsf{neg}(g_1 \vee \cdots \vee g_m), \tag{ψ_2}$$

where existential bindings in the prefix could then be converted into selection bindings, and the remainder of the expression would have the form discussed in §5.3.1.2 and would be subject to the same limitations and treatment.

Doing so requires some $\mathsf{condition}(\psi_1)$ such that

$$\mathsf{condition}(\psi_1) \implies \forall\, \mathsf{val}_V \in \mathsf{Vals}_V.\ [\![\psi_1]\!]_{\mathsf{val}_V} = [\![\psi_2]\!]_{\mathsf{val}_V}.$$

The minimal condition is false which rejects all processes with guards that mix selections and quantifiers over clock variables. Logical equivalence of ψ_1 and ψ_2 is the most accepting. It would, for example, be possible to emit constraints and proof obligations for treatment in a theorem prover or a model checker. The current implementation uses an approximate condition.

Definition 5.3.3
The canswap predicate is defined on formulas of the form

$$\forall a_1, \ldots, a_n. \exists e_1, \ldots, e_m. \varphi_1 \vee \ldots \vee \varphi_l,$$

where each φ_i is a conjunction of clock terms, $p_i^1 \wedge \ldots \wedge p_i^{n_i}$. Let

$$A = \{a_1, \ldots, a_n\}, \qquad E = \{e_1, \ldots, e_m\},$$
$$A_i = \mathsf{freevars}(\varphi_i) \cap A, \text{ and,} \qquad E_i = \mathsf{freevars}(\varphi_i) \cap E.$$

Then canswap is true iff for all $1 \leq i \leq l$ either **1.** $A_i = \emptyset \vee E_i = \emptyset$, or **2.** For $1 \leq j \neq i \leq l$, $A_i \cap A_j = \emptyset$, $E_i \cap E_j = \emptyset$, and for all $1 < k < n_i$, $\mathsf{freevars}(p_i^k) \cap A_i = \emptyset$ or $\mathsf{freevars}(p_i^k) \cap E_i = \emptyset$. ∎

[8] The disjunctive quality of existential quantification splits clock zones.

Proposition 5
For a quantifier-free formula ψ in DNF,

$$\mathsf{canswap}(\forall a_1,\ldots,a_n.\,\exists e_1,\ldots,e_m.\,\psi) \implies \forall a_1,\ldots,a_n.\,\exists e_1,\ldots,e_m.\,\psi \equiv \exists e_1,\ldots,e_m.\,\forall a_1,\ldots,a_n.\,\psi \quad \blacksquare$$

Proof Assume that ψ is in disjunctive normal form. It is possible, through associativity and commutativity of disjunction, to juxtapose clauses that do not contain any universally bound variables. The scope of each existential quantifier can be reduced to either the juxtaposed group or a single clause; the clauses of canswap guarantee the side condition of $\exists x.\,(\phi_1 \vee \phi_2) = (\exists x.\,\phi_1) \vee \phi_2 \quad (x \notin \mathsf{freevars}(\phi_2))$. The scope of the universal quantifiers can be similarly reduced.

The scope of existential quantifiers in clauses of the form $\forall A_i.\,\exists E_i.\,p_i^1 \wedge \ldots \wedge p_i^{n_i}$ can be reduced to a subset of the terms. Likewise for the scopes of universal quantifiers. The second clause of canswap guarantees that no existential quantifier overlaps with any universal quantifier on a term.

The scope of existential bindings in a formula with reduced scopes can be widened to cover all clauses, and similarly for universal bindings. □

The canswap predicate is no panacea, but it does address several useful cases, for example, sets of transitions where no single transition employs both selection bindings and universal quantifiers and where each guard is a single term.

The construction of Figure 5.7 can be generalized to provide an alternative to rearranging formulas, if none of the universal quantifiers are of scalar type, as shown in Figure 5.10. The basic idea is the same, looping through all possible valuations of the universal quantifiers, but the existential quantifiers are accounted for as selection bindings on loop and error transitions, and as **forall** bindings on the transitions back to the original location.

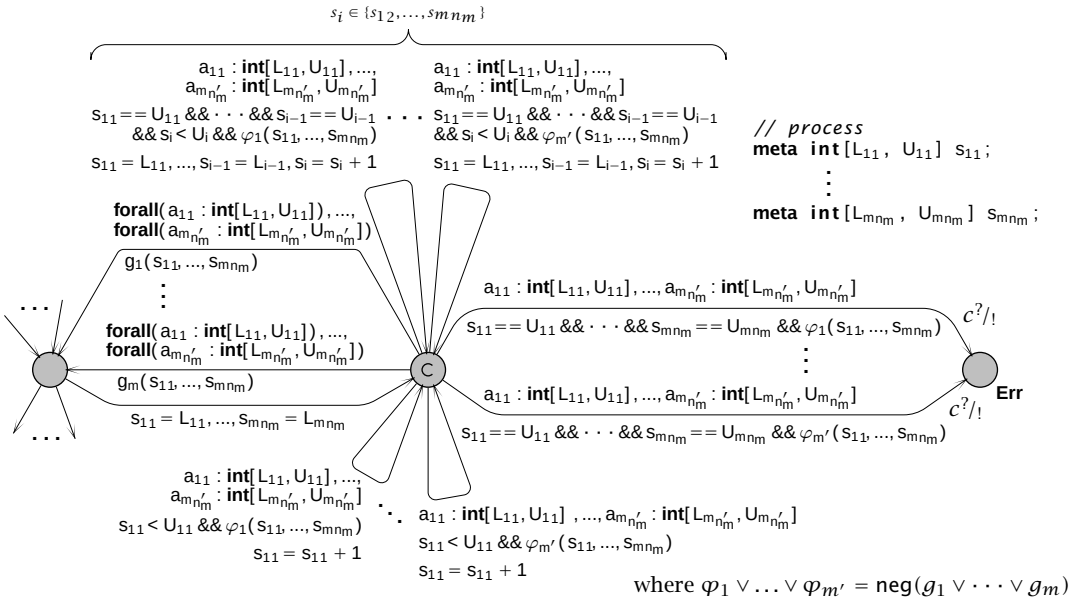


Figure 5.10: Construction for $\forall s_{11},\ldots,s_{mn_m}.\,\exists a_{11},\ldots,a_{mn'_m}.\,\mathsf{neg}(g_1 \vee \ldots \vee g_m)$

5.3.2 Channel arrays

The techniques of the previous section are now generalised to edges labelled with actions on elements of channel arrays.

For basic automata actions were grouped by name and direction, and then the negated disjunction of their guards was formed, see rule 4 of Definition 5.2.5. The

guards of processes with elementary channels, that is singleton channel sets, are effectively grouped together in the same way. Channel arrays introduce additional complexity. Transitions may be grouped by channel array name and direction, rule 4 of Definition 5.3.1, but whether two transitions refer to the same element within the array, that is the same element within the channel set, depends on the valuation of their respective index sequences relative to the process state and selection bindings.

Initially only array index expressions that do not contain selection bindings are considered, that is all variables in these expressions are state variables. Two techniques to group channels are developed. The first was used in an earlier version of the implementation. It offers insight into the problem, but the number of transitions required in the validation automaton grows exponentially and it is difficult to incorporate selection bindings. The second technique, used in the current implementation, is more effective. It is readily extended to incorporate selection bindings in a restricted way.

In Uppaal, channel arrays may be passed by reference as template parameters. Aliasing of such parameters is not detectable because reference values cannot be compared for equality. The implementation detects this possibility and warns about it; but the burden of correctness rests with modellers.

5.3.2.1 Partitioning: state expressions

Rather than collecting edges on a single action, they must now be grouped by channel set and direction.

The set of m edges to be flipped is now written

$$E \quad = \quad \left\{ \left(S_1, A_1, g_1, \langle e_1^1, \ldots, e_{n_C}^1 \rangle\right), \ldots, \left(S_m, A_m, g_m, \langle e_1^m, \ldots, e_{n_C}^m \rangle\right) \right\},$$

where the added expression sequences specify an element of the channel set. Disjointness assumptions are made as in the previous section, with the additional assumption that quantifier bindings are restricted to guards, and likewise, until the next section, so are selection bindings, that is,

$$\forall 1 \le i, j \le m, 1 \le k \le n_C.\ (A_i \cup S_i) \cap \mathsf{freevars}(e_k^j) = \emptyset.$$

The edges within E must be grouped by channel and allowance made for channels not represented by any edge. For example, given a set of two channels $C = \{c[1], c[2]\}$, and,

$$E = \{(S_1, A_1, g_1, e^1), (S_2, A_2, g_2, e^2)\},$$

there are two possibilities, for a fixed valuation val_V, if $[\![e^1]\!]_{\mathsf{val}_V} = [\![e^2]\!]_{\mathsf{val}_V}$ the previous techniques can be applied to the edges $(S_1 \cup S_2, A_1 \cup A_2, g_1 \vee g_2)$ on action $c[\,[\![e^1]\!]_{\mathsf{val}_V}\,]$ and $(\emptyset, \emptyset, \mathsf{false})$ on action $c[i]$ where $i \neq [\![e^1]\!]_{\mathsf{val}_V}$. Otherwise, if $[\![e^1]\!]_{\mathsf{val}_V} \neq [\![e^2]\!]_{\mathsf{val}_V}$ there is one edge (S_1, A_1, g_1) on $c[\,[\![e^1]\!]_{\mathsf{val}_V}\,]$, and another (S_2, A_2, g_2) on $c[\,[\![e^2]\!]_{\mathsf{val}_V}\,]$.

In general, every possible partitioning of the m edges must be considered. Unfortunately, this (Bell) number B_i grows very quickly: $B_1 = 1$, $B_2 = 2$, $B_3 = 5$, $B_4 = 15$, $B_5 = 52$, $B_6 = 203$, $B_7 = 877$, ..., which limits the size of models that can be addressed, based on the maximum number of edges leaving a single node, on the same channel array, in the same direction, and with unique index expressions. Most models that can be effectively checked in Uppaal, however, would not test this limit. Before partitioning, the implementation merges edges that have sequences of syntactically identical index expressions by forming unions of their selection binding sets and disjunctions of their guards.

Each partition is characterised throughout by a predicate p_b that equates index expressions in the same block and differentiates those in other blocks. For example, for three edge labels on a two dimensional array: $\langle e_1^1, e_2^1 \rangle$, $\langle e_1^2, e_2^2 \rangle$, and $\langle e_1^3, e_2^3 \rangle$, there are five partitions and associated predicates:

[1, 2, 3]	$(e_1^1 = e_1^2 \wedge e_2^1 = e_2^2 \wedge e_1^1 = e_1^3 \wedge e_2^1 = e_2^3)$
[1, 2] [3]	$(e_1^1 = e_1^2 \wedge e_2^1 = e_2^2) \wedge (e_1^1 \neq e_1^3 \vee e_2^1 \neq e_2^3)$
[1] [2, 3]	$(e_1^2 = e_1^3 \wedge e_2^2 = e_2^3) \wedge (e_1^1 \neq e_1^2 \vee e_2^1 \neq e_2^2)$
[1, 3] [2]	$(e_1^1 = e_1^3 \wedge e_2^1 = e_2^3) \wedge (e_1^1 \neq e_1^2 \vee e_2^1 \neq e_2^2)$
[1] [2] [3]	$(e_1^1 \neq e_1^2 \vee e_2^1 \neq e_2^2) \wedge (e_1^1 \neq e_1^3 \vee e_2^1 \neq e_2^3) \wedge (e_1^2 \neq e_1^3 \vee e_2^2 \neq e_2^3)$

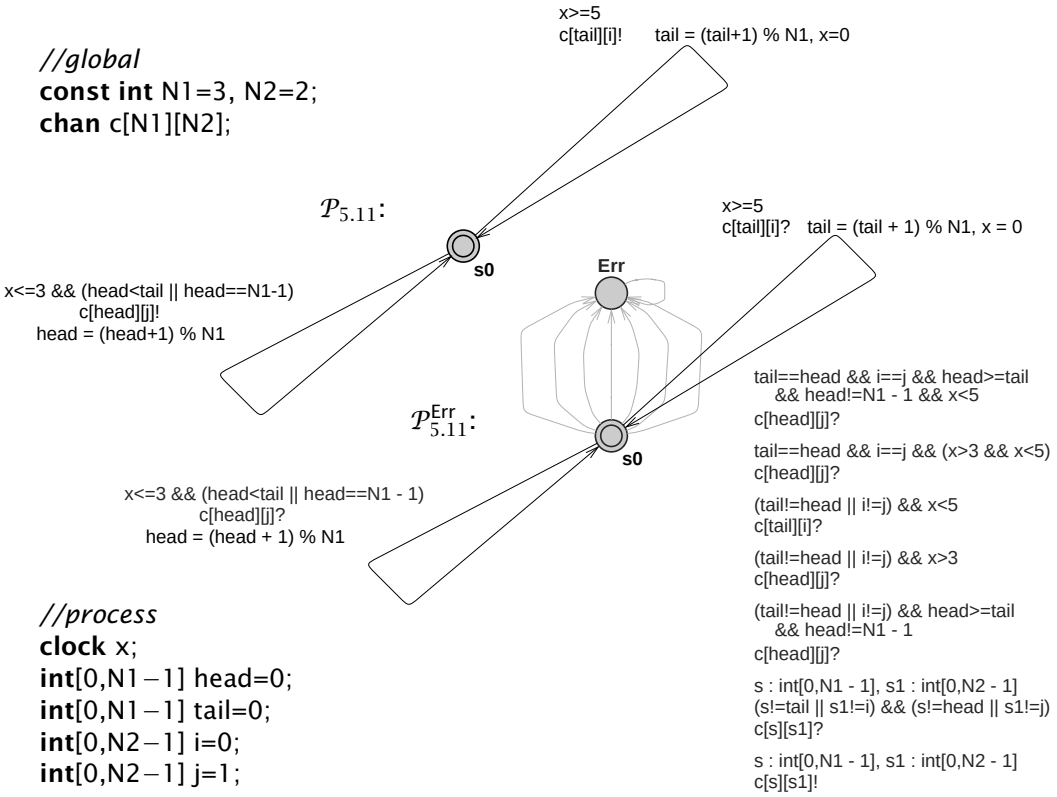


Figure 5.11: Channel selections without bindings (partitioned)

Any given valuation will satisfy exactly one predicate. In practice, predicates may, like guard terms, be simplified. Some subexpressions can be replaced with true, for example when a variable is compared with itself $x = x$, others with false, as when comparing two different constants $1 = 2$. Such changes may in turn lead to further simplifications.

The techniques of previous sections apply to edges within a single partition provided the predicate is added after negating the guard: $\forall \ldots \exists \ldots p_b \wedge \mathsf{neg}(g \vee \ldots \vee g)$. After partitioning, channel subscripts are irrelevant: while the exact action groupings may depend on variable values, the predicates guarantee that all possibilities are properly taken into account.

It is possible that some channel set elements are never enabled. These are directed to the Err state by creating an edge with a selection binding for each dimension and a guard that is true whenever at least one of the selection bindings differs from all other expressions in the same dimension. For the previous three transitions this *sweep edge* would be:

$$\left(\{i_1, i_2\}, \varnothing, \left((i_1 \neq e_1^1 \vee i_2 \neq e_2^1) \wedge (i_1 \neq e_1^2 \vee i_2 \neq e_2^2) \wedge (i_1 \neq e_1^3 \vee i_2 \neq e_2^3)\right)\right).$$

In the $\mathcal{P}^{\mathsf{Err}}_{5.11}$ process of Figure 5.11 seven transitions connect the original state to the Err state, from top to bottom: two for state valuations where both original transitions have the same input action, three for when both transitions have different input actions, another to cover any outputs on c not present on the other edges, and one to cover all inputs on channels in c. The constants N_1 and N_2 are declared globally, but they could be template parameters. The validation automaton is correct regardless of their exact values.

5.3.2.2 Partitioning: selection bindings

The restriction that selection bindings may not occur in channel array subscripts is now partially relaxed. The set of m edges to be flipped is written as in the previous

subsection, and disjointness per §5.3.1.3 is assumed. Quantifiers are restricted to guards, $\forall 1 \le i, j \le m, 1 \le k \le n_C.\ A_i \cap \mathsf{freevars}(e^j_k) = \emptyset$. Subscript indices are partitioned into two classes: free I_f and bound I_b. These classes are used to limit the use of selection bindings in channel expressions. For each e^i_k, of each edge i in the set, either $k \in I_\mathrm{f}$ or $k \in I_\mathrm{b}$. There are two cases for each index expression. Either it has no selection variables, $\forall 1 \le i \le m, k \in I_\mathrm{f}.\ S_i \cap \mathsf{freevars}(e^i_k) = \emptyset$; or it is a selection variable that spans the array dimension, $\forall 1 \le i \le m, k \in I_\mathrm{b}.\ \exists s \in S_i.\ e^i_k = s$. No selection variable is allowed in two different positions, if $\forall 1 \le i \le m\ \ k, l \in I_\mathrm{b}.\ e^i_k = e^i_l$ then $k = l$, and it is assumed, renaming as necessary, that the same variable name is used consistently across such bound positions, $\forall 1 \le i, j \le m\ \ k \in I_\mathrm{b}.\ e^i_k = e^j_k = s_k$, writing $\mathsf{sel}(i \in I_\mathrm{b})$ for that variable. Let $S_\mathrm{sub} = \{\mathsf{sel}(i) \mid i \in I_\mathrm{b}\}$.

As an example, consider $I_\mathrm{f} = \{1, 3\}$, $I_\mathrm{b} = \{2, 4\}$, and,

$$\begin{aligned} E \quad = \quad & \{(\{s,t\},\ \emptyset,\ l > 0,\ \langle 2, s, 2l-1, t\rangle), \\ & (\{s,t,u\},\ \emptyset,\ u \neq s \wedge o_u > 0,\ \langle 1, s, 2l, t\rangle)\}, \end{aligned}$$

where s and t range over their entire respective dimensions. The guard expressions contain a free variable l, which is not bound by the selection set. The variables s and t appear in the same index position. The second transition has a third selection binding, but it is only used in the guard.

The restrictions ensure that no two assignments to the selection variables specify the same channel. The transition set is changed slightly before applying the basic technique:

$$\begin{aligned} E' = \Big\{ & \Big(S_1 \setminus S_\mathrm{sub},\ A_1,\ g_1,\ \langle e^1_1, \ldots, e^1_{n_C}\rangle \restriction_{I_\mathrm{f}}\Big), \ldots, \\ & \Big(S_m \setminus S_\mathrm{sub},\ A_m,\ g_m,\ \langle e^m_1, \ldots, e^m_{n_C}\rangle \restriction_{I_\mathrm{f}}\Big)\Big\}. \end{aligned}$$

where $\langle i_1, \ldots, i_n\rangle \restriction_{I_\mathrm{f}}$ is a new sequence of, in the same order, those elements occurring at indices in I_f in the original $\langle i_1, \ldots, i_n\rangle$. As E' satisfies the assumptions of §5.3.2.1, the technique of that section applies to give m' transitions,

$$\begin{aligned} \overline{E'} = \Big\{ & \Big(\overline{S_1}, \overline{A_1}, \overline{g_1}, \langle e^1_1, \ldots, e^1_{|I_\mathrm{f}|}\rangle\Big), \\ & \ldots, \Big(\overline{S_{m'}}, \overline{A_{m'}}, \overline{g_{m'}}, \langle e^{m'}_1, \ldots, e^{m'}_{|I_\mathrm{f}|}\rangle\Big)\Big\}, \end{aligned}$$

to which the elements of S_sub may be returned,

$$\begin{aligned} \overline{E} = \Big\{ & \Big(\overline{S_1} \cup S_\mathrm{sub}, \overline{A_1}, \overline{g_1}, \mathsf{inject}_{\mathsf{sel}}\Big(\langle e^1_1, \ldots, e^1_{|I_\mathrm{f}|}\rangle\Big)\Big), \ldots, \\ & \Big(\overline{S_{m'}} \cup S_\mathrm{sub}, \overline{A_{m'}}, \overline{g_{m'}}, \mathsf{inject}_{\mathsf{sel}}\Big(\langle e^{m'}_1, \ldots, e^{m'}_{|I_\mathrm{f}|}\rangle\Big)\Big)\Big\}, \end{aligned}$$

where the inject function inserts variables from S_sub back into their original positions within the sequence.

The assumptions that a selection variable is used in at most one subscript dimension and that its type spans the entire array dimension are easily circumvented by augmenting guards with additional constraints; as will be shown in the next section.

The restriction that subscript dimensions be partitioned into those that may only contain selection bindings and those that may not is more difficult to dispatch. In fact, this challenge motivated the development of the more general technique presented in the next subsection.

5.3.2.3 Generalised technique

The partitioning technique was implemented, but the potential for generating a great many partitions and the difficulty in extending it to selection bindings prompted the creation of an improved technique, which turns out to be simpler and more effective.

Beginning with the sweep edge from the previous section, a new selection binding is introduced for each array dimension; each having a type that spans the entire dimension. Each valuation of these selection bindings specifies a different element of the channel set. All elements of the channel set are considered.

For a given valuation of the new selection bindings, the aim is to form the negated disjunction of all applicable guards, as done explicitly by the partitioning method. So, to each sequence of index expressions is associated a clause, satisfied only when every sweep binding matches the corresponding expression. The conjunctions of each clause and guard may then be combined in disjunction, and the entirety negated as required. The clauses ensure that guards are partitioned appropriately by the model checking algorithm as it evaluates the entire expression for every possible valuation.

The set of m edges to be flipped is written as in §5.3.2.1, and with the same assumptions, selection bindings as channel array subscripts are not yet considered,

$$E \;=\; \left\{ \left(S_1, A_1, g_1, \langle e_1^1, \ldots, e_{n_C}^1 \rangle \right), \ldots, \left(S_m, A_m, g_m, \langle e_1^m, \ldots, e_{n_C}^m \rangle \right) \right\}.$$

Let $S_w = \{z_1, \ldots, z_{n_C}\}$ be a fresh set of n_C *sweep bindings*. Each corresponds with the channel array index of the same subscript and has the same type. They are disjoint from one another and from the other selection and quantifier bindings.

The m edges can then be merged into a single edge:

$$\begin{aligned}
\epsilon = \Big(& S_w \cup S_1 \cup \cdots \cup S_m, A_1 \cup \cdots \cup A_m, \\
& \quad (z_1 = e_1^1 \;\wedge \cdots \wedge\; z_{n_C} = e_{n_C}^1 \;\wedge\; g_1) \\
& \vee\; (z_1 = e_1^2 \;\wedge \cdots \wedge\; z_{n_C} = e_{n_C}^2 \;\wedge\; g_2) \\
& \vdots \\
& \vee\; (z_1 = e_1^m \;\wedge \cdots \wedge\; z_{n_C} = e_{n_C}^m \;\wedge\; g_m),\; \langle z_1, \ldots, z_{n_C} \rangle \Big).
\end{aligned}$$

For a valuation of the sweep bindings, each conjunct $(z_1 = e_1 \wedge \cdots \wedge z_{n_C} = e_{n_C} \wedge g)$ is either equivalent to g and remains in the disjunction, or equivalent to false and drops from the disjunction, depending on whether the guard applies to an edge on a channel set element identified by $\langle z_1, \ldots, z_{n_C} \rangle$. The sets of universal bindings A_i may be grouped into the union because the scope of each applies only to a single guard expression g_i and is disjoint from the set of free variables in the array index expressions $e_1^i, \ldots, e_{n_C}^i$. Similarly for the selection bindings S_i. The $(z_1 = e_1 \wedge \cdots \wedge z_{n_C} = e_{n_C})$ subexpressions do not contain clocks since they are formed by equating selection bindings and array index expressions. Every assignment to S_w selects a different channel, and a separate sweep edge is not necessary since all channels are taken into account.

The edge ϵ represents multiple transitions, one for each valuation of S_w. Thus, there are no transitions enabled for an action when

$$\begin{aligned}
\forall s_{11}, \ldots, s_{m n_m}. \exists a_{11}, \ldots, a_{m n'_m}. \quad & (z_1 \neq e_1^1 \;\vee \cdots \vee\; z_{n_C} \neq e_{n_C}^1 \;\vee\; \mathsf{neg}(g_1)) \qquad (\psi_3) \\
\wedge\; & (z_1 \neq e_1^2 \;\vee \cdots \vee\; z_{n_C} \neq e_{n_C}^2 \;\vee\; \mathsf{neg}(g_2)) \\
& \vdots \\
\wedge\; & (z_1 \neq e_1^m \;\vee \cdots \vee\; z_{n_C} \neq e_{n_C}^m \;\vee\; \mathsf{neg}(g_m)).
\end{aligned}$$

The sweep bindings are not negated into universal quantifiers since their role is only to choose elements from the channel set. For an element the quantifier free part is equivalent to a conjunction of l negated guards $g_{i_1} \wedge \cdots \wedge g_{i_l}$, which is equivalent to the corresponding form for elementary channels, §5.3.1.3.

Formula ψ_3 must be manipulated to form a set of valid Uppaal transitions. One approach is to swap the universal and existential quantifiers and transform the quantifier-free subexpression into DNF before splitting the resulting clauses over multiple transitions. The previously presented techniques remain applicable. The sweep variables are treated as state variables during the manipulations and then reintroduced on each separate transition afterward. The set of transitions to the error state can be considered as a set of sets, each element corresponding to the transitions for a valuation of the sweep bindings, and, equivalently, to a distinct action from the channel set. In the construction for formulas with unswappable quantifiers, Figure 5.10, the sweep binding

values must remain constant while testing the range of universal quantifier valuations. Thus a meta variable must also be introduced for each sweep binding. Variable values are selected non-deterministically and assigned on initial entry to the committed state. The variables select a specific channel on the transitions into Err. They may also occur in guard expressions.

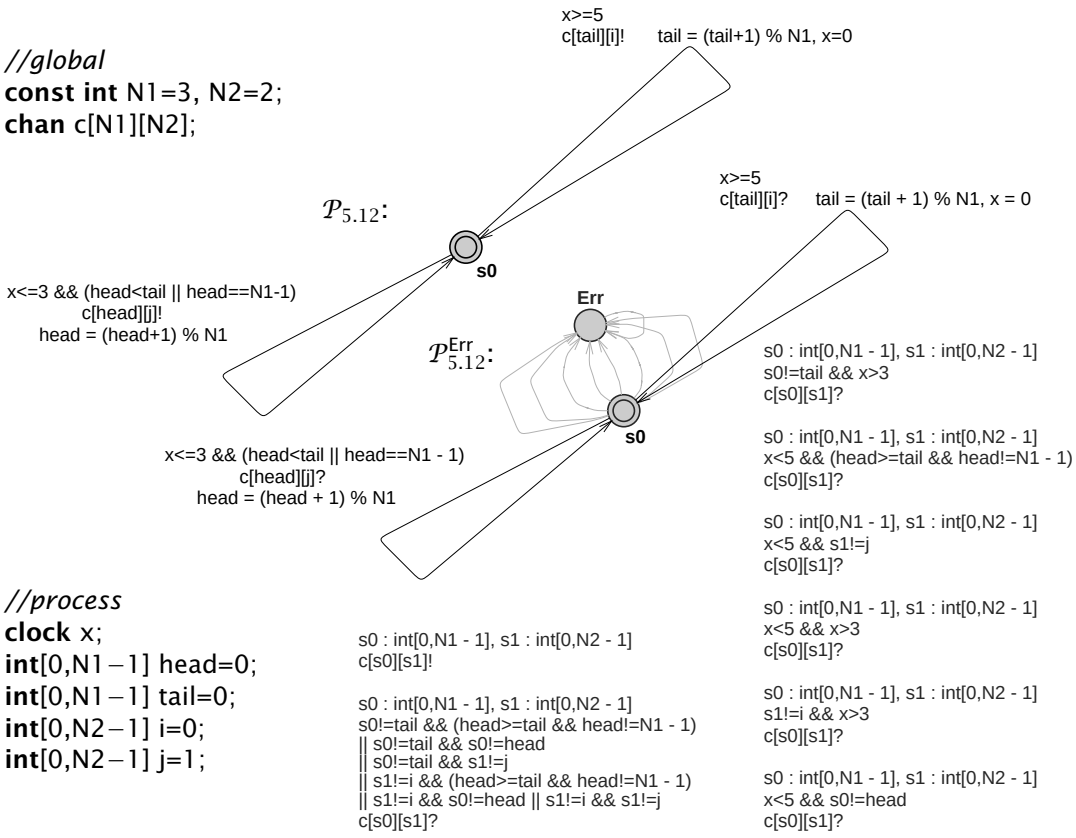


Figure 5.12: Channel selections without bindings (generalised)

Process $\mathcal{P}_{5.12}$ of Figure 5.12, is identical to $\mathcal{P}_{5.11}$ of Figure 5.11. There are two transitions on a two dimensional channel array. Both use state variables to choose a channel from the array: one uses tail and i, the other uses head and j. In this instance the generalised technique introduces seven transitions to check output actions on c, which compares well with the six generated by the partitioning technique. But an extra simplification step is required to avoid the twelve transitions that would result from a naive conversion to DNF, since the negated combination of the two original guards is

$$(s_0 \neq \text{tail} \vee s_1 \neq i \vee x < 5) \\ \wedge (s_0 \neq \text{head} \vee s_1 \neq j \vee x > 3 \vee (\text{head} \geq \text{tail} \wedge \text{head} \neq N - 1)) .$$

And, in the conversion to DNF each of the three upper disjunctive subclauses is combined with each of the four lower disjunctive subclauses. The number of conjunctive clauses is in proportion to the number of array dimensions. But it is evident that the conjuncts that do not contain clocks can be collected together onto a single transition. The partitioning algorithm, on the other hand, gives three transitions for the case when the subscript variables are equal, two for the case when they are not, and another to sweep other elements. The results are proportional to the number of source transitions that differ in at least one subscript state expression.

The earlier exclusion of selection bindings from subscript expressions can now be relaxed to allow subscript expressions that consist of a simple selection binding, that is the expression may not contain operators. It is briefly assumed that each selection binding indexes at most one dimension of a channel array.

Ideally each selection binding has the same type as the channel array dimension where it appears. Scalar types always meet this requirement, but it is possible for an integer binding s of type $[l,u]$ to index a dimension of greater extent.[9] In such cases, the type of the selection binding can safely be made to match the array dimension if the guard is further constrained by the clause $l \le s \le u$. It can thus be assumed, without loss of generality, that all selection bindings that index a channel array dimension span all possible values.

When merging transitions a selection binding s that indexes array dimension i will be replaced by the sweep binding z_i, as for a state expression. But rather than include the restriction $z_i = s$ in the new guard conjunct, observe that of all the possible valuations that include z_i and s, only those where the two are equal need actually be considered. Since both variables have the same type, s can be replaced with z_i in both the set of selection bindings, effectively removing s, and the guard, where the restriction $z_i = z_i$ would have anyway become redundant.

This optimisation results in simpler edges. It reduces the number of clauses and potentially, as discussed, the number of transitions after conversion to DNF.

The restriction of selection variables to a single dimension, which, for example, excludes the expression $(\{s\}, A, g, \langle s, s\rangle)$, is not limiting; additional constraints can always be added to the guard, for example, given $S_w = \{z_1, z_2\}$, the example would become $(S_w, A, z_2 = z_1 \wedge g, \langle z_1, z_2\rangle)$. This manipulation can be combined with the one for matching types. Care is required, but there are no fundamental problems.

Figure 5.13 is an example of channel selection involving selection bindings. The two edges leaving s_0 are combined and negated to give an edge to Err. The selection bindings in the first dimension are conflated to a sweep binding. The state variables, m and n, are replaced by a sweep binding and new guard constraints. Note that integer subscript types must be written as inclusive intervals, for example **int**[0,4], whereas array declarations may specify the size of an interval, **chan** c[N][5]. The two edges leaving s_1 on c are handled similarly, even though one specifies the first dimension with a state variable curr and the other with a selection binding s. The resulting guards are not converted to DNF because they do not contain clock variables.

Another example is given in Figure 5.14. The selection binding on the lower transition is of type **int**[2,N], whereas the channel array dimension is of type **int**[0,9]. The merged guard thus becomes $(x > 1 \wedge i \neq 3) \vee (x < 8 \wedge i \neq 4 \wedge i \ge 2 \wedge i \le N)$, which is negated, then converted to the five transitions in the figure. The current simplifier does not eliminate the transition labelled $i_1 = 3 \ \wedge \ i_1 = 4$.

Allowing selection variables in state variable expressions makes it difficult to determine which valuations resolve to the same channel and thus how to group edges for negation. Certain expression forms preserve the property that each valuation gives a different channel, for instance addition and multiplication shift the range, whereas others, like the modulo and shift operators, division, and function calls (for example, $f(x) = 1$) do not. While those expressions where the mapping is injective could be accomodated by a more sophisticated grouping algorithm, the additional complexity does not seem to be justified.

5.3.3 Inverting invariants

The validation construction produces τ-transitions for locations with non-trivial invariants (rule 1 of Definition 5.3.1). It is sometimes necessary to split negated invariants over transitions per §5.3.1.1, and to treat quantifiers per §5.3.1.3. Invariants are otherwise treated without any especial problems.

5.3.4 Urgent Channels and shared variables

A process is not usually obliged to synchronise on enabled channels if its location invariant permits further delay, but synchronisation on urgent channels has priority over delay.

[9] Indexing dimensions of smaller extent gives a range error.

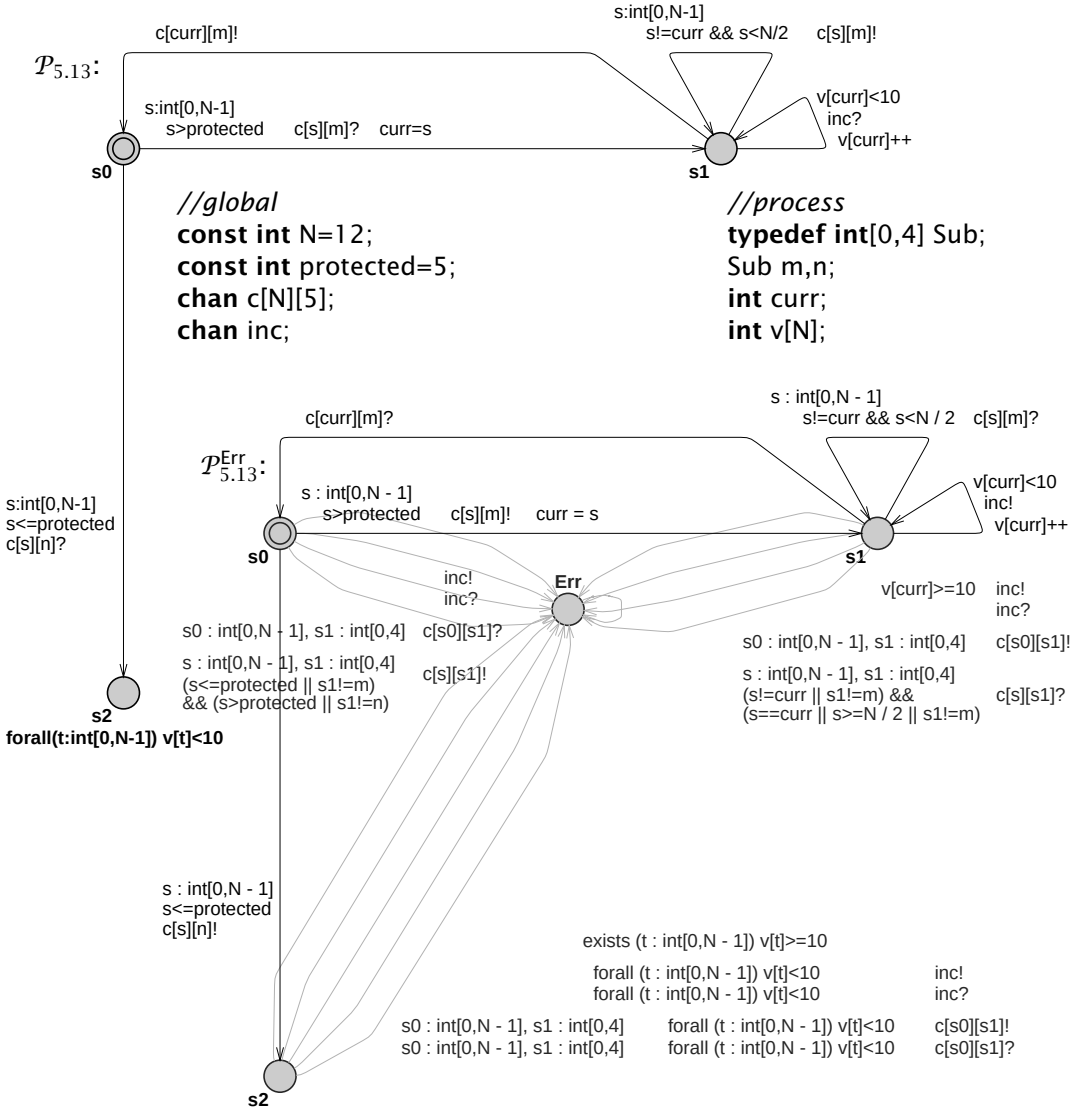


Figure 5.13: Channel selections with selection bindings

Basic timed trace inclusion does not suffice for models with urgent channels, instead *timed ready simulation* and an extended validation construction, that can also handle shared variables, have been proposed [JLS00]. This technique can be extended and implemented for the processes considered here. Arrays of urgent channels are handled by storing index values in a variable across the two-transition check for immediate synchronisation, as in the example of Figure 5.15. Uppaal supports direct array comparison, so shared variable arrays present no special challenges.

Uppaal does not allow clock variables in the guards of transitions that synchronise on urgent channels. But they may be introduced when transforming models where clock variables are used in location invariants[10] (and Uppaal rejects such models).

5.4 Implementation

The techniques described in this chapter have been implemented in a tool called *urpal* that parses Uppaal models and generates validation automata when possible. It implements the standard techniques [JLS00, Sto02] and the extensions described in this chapter.

[10] Unless the negated disjunction of guards simplifies to false.

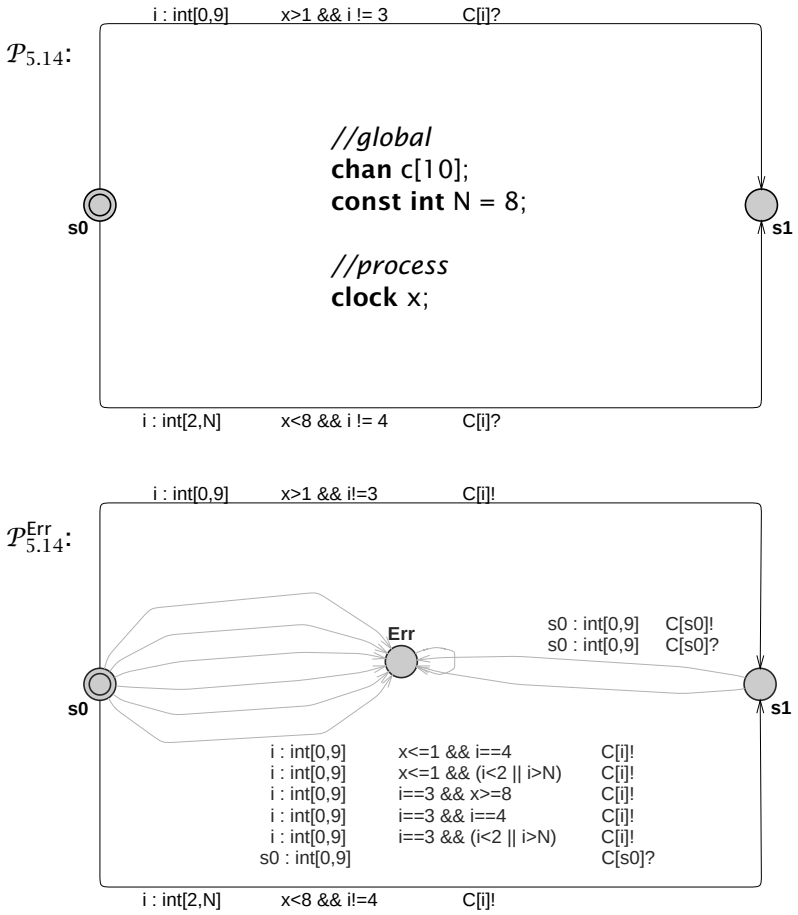


Figure 5.14: Channel selections with differing ranges

Urpal is written in SML, which is ideal for complex symbol manipulation. The experience of implementing urpal in SML, and some alternative implementation choices are discussed in §5.4.1.

The five major subsystems of the tool are shown in Figure 5.16: a parser for the Uppaal Extensible Markup Language (XML) format; a parser for the description language used in declarations, expressions and actions; algorithms for transforming models; an interface to Graphviz for placing nodes and routing edges; and a pretty printer back to an Uppaal file.

Uppaal is distributed with a C++ library (libutap) for parsing the file format and description language, and for performing type checking. Using the library potentially saves implementation effort and provides some insulation from changes to the XML file format, but it was decided that integrating the object-oriented library into SML would involve as much work as writing a custom parser. It would also make the tool dependent on updates from the Uppaal developers, and it would complicate its compilation and installation.

Thus a custom parser was implemented. It allows access to type information, but otherwise assumes that input files have already been validated by Uppaal. Uppaal XML files are first parsed by FXP [Neu99], giving unparsed declarations and templates which are then processed by an ml-lex/ml-yacc [AMT94, TA00] parser for the description language. The result is an Uppaal model expressed as an SML data type. Extensive use is made of the SML Basis [GR04] and SML/NJ libraries.

The manipulations performed by the tool can introduce many new transitions. A custom interface to the Graphviz [GN00] `fdp` and `neato` tools attempts to untangle introduced states and transitions while preserving the original layout of a model. Producing readable models is essential for manual inspection, which increases confidence in the results and aids debugging of both the tool and models, and for understanding

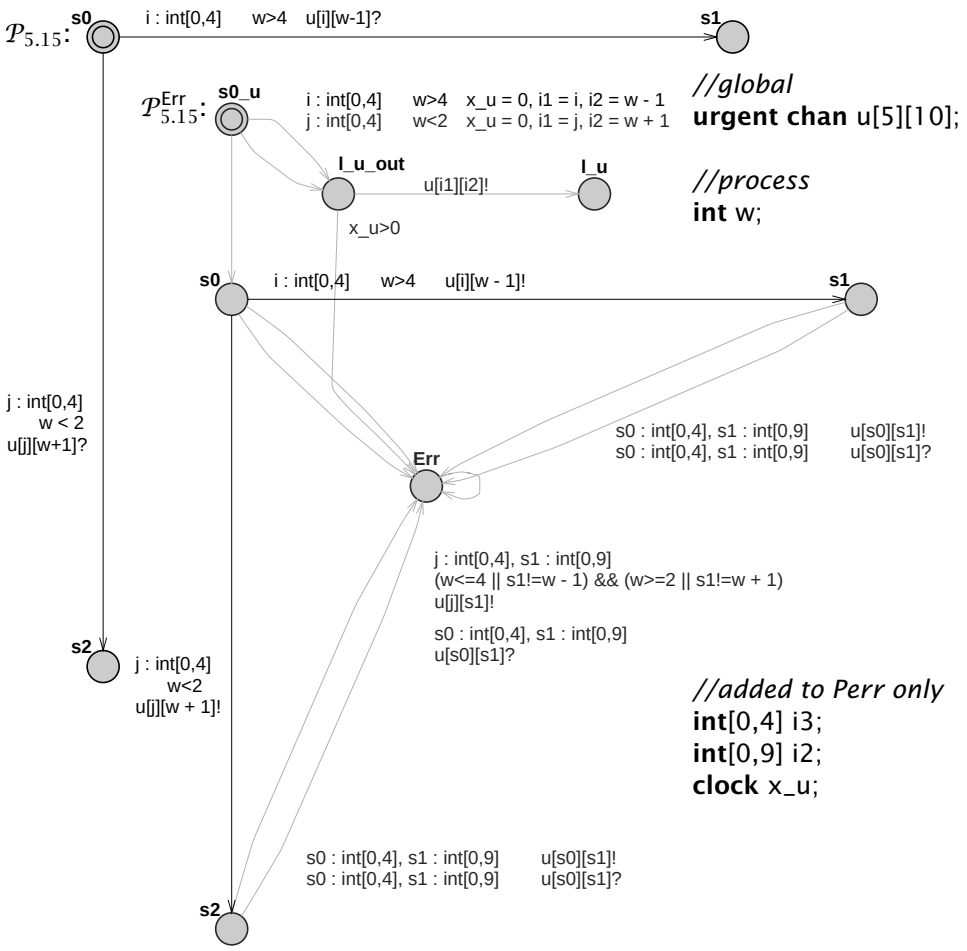


Figure 5.15: Validating an array of urgent channels

Figure 5.16: High-level structure of Urpal

counter-example traces when timed trace inclusion validation fails. The SML/NJ pretty printing library is used, for similar reasons, to output updated declarations and expressions. Most of the validation automata figures in this paper were produced directly by the tool, though edge labels were occasionally adjusted to improve readability. Some of the nodes and transitions in Figure 5.15 were placed manually.

Although the tool focuses on the validation construction, its subsystems are suitable for implementing other model transformations. It includes, for instance, an expression language for pruning transitions, and merging, dropping, and renaming nodes. The driver model of §4.5 was initially generated directly from the assembler source using the tool.

5.4.1 Experience and implementation language

The implementation has two main tasks: to parse and represent Uppaal models, and to perform symbolic manipulations on transitions and expressions. Performance and efficiency are not major concerns as the tool need only transform models of a size for which model checking is feasible. Correctness and robustness are more important.

SML was chosen because it is garbage-collected, statically typed, and mostly functional. The module system, which includes functors and support for abstract data

types, helped increasingly as the implementation grew in size.

There is no reason not to use garbage collection and it greatly simplifies the implementation.

The static typing did not cause any major problems or require any significant compromises. On the contrary, the type system made it easier to write and understand the required higher-order functions, and the SML type checking found many potential errors at compile time. Type checking was especially invaluable in finding errors introduced as data structures were revised during development. Programs written in SML never have segmentation faults although they may still raise exceptions and suffer from programming flaws and incorrect algorithms. The Uppaal expression language was modelled with algebraic data types; writing pattern-matched functions over these types helped to understand many subtleties of manipulating Uppaal models.

Most of the implementation is purely functional. Preferring recursive functions and avoiding mutable state encouraged a focus on correctness rather than speed and seemed to reduce the time spent debugging, relative to previous experiences with other programming languages.

The formal semantics of Standard ML [MTHM97, MT90] were not required directly. But having a stable definition means that programs are unambiguous, barring flaws in the definition, and likely to be usable in the future, subject to tool-chain availability. It also aids portability between compilers and interpreters. The program has been tested with SML/NJ,[11] MLton,[12] and Poly/ML.[13] The biggest challenges are the differences between build systems, which are not standardised, and missing support for features of the SML Basis Library [GR04]. Different build files are required for each. A lack of functional I/O routines prevented compilation with SML.NET[14] and Moscow ML[15] (whose build system is also obstructive). Builds with the ML Kit,[16] Alice,[17] TILT,[18] HaMLet,[19] SML#[20] or Poplog[21] were not attempted, but the availability of so many compilers is encouraging.

Several alternative languages were considered:

Lisp/Scheme: The flexibility of dynamic types was not required and there was no advantage to using the minimalist s-expression syntax.

OCaml: Active development of this language means that more libraries are available, but they are arguably less stable than those of SML. There are fewer compilers and interpreters. But, for the most part, OCaml would also have been a suitable choice for implementing the tool.

Haskell: The more modern syntax, including list comprehensions, `where` clauses, and indentation for nesting, is attractive. Haskell cannot yet match the module system of SML, but its type classes have their own advantages. Many libraries are available for Haskell but the interfaces are subject to change, and many, specifically the XML parsers, require a mastery of concepts like monads and arrows. There are fewer compilers for Haskell. Lazy evaluation was not especially advantageous for this implementation.

Java, C++: An object-oriented style for this type of program would likely decompose the transformation algorithm into separate parts such that the function of the whole would be difficult to reason about, without offering many advantages. Neither language has strong static typing or algebraic data types.

[11] `http://www.smlnj.org`
[12] `http://mlton.org`
[13] `http://www.polyml.org`
[14] `http://www.cl.cam.ac.uk/research/tsg/SMLNET`
[15] `http://www.itu.dk/~sestoft/mosml.html`
[16] `http://www.it-c.dk/research/mlkit/index.php/Main_Page`
[17] `http://www.ps.uni-sb.de/alice`
[18] `http://www.tilt.cs.cmu.edu`
[19] `http://www.mpi-sws.org/~rossberg/hamlet`
[20] `http://www.pllab.riec.tohoku.ac.jp/smlsharp`
[21] `http://www.cs.bham.ac.uk/research/projects/poplog/freepoplog.html`

Python, Perl: Dynamic languages offer few specific advantages for this task. The class of errors that are only detectable at runtime is larger in these languages than for statically typed languages.

Nearly all of these languages have more comprehensive libraries than does SML. Fortunately, the SML Basis Library [GR04], SML/NJ Library,[22] and FXP parsing library [Neu99] covered most needs. In other languages, however, it may have been possible to avoid writing an interface to Graphviz.

The transformation discussed in this chapter is implemented directly. Many of the minor details in the description require careful programming to implement. For example, routines are required to validate and maintain the assumption of disjoint variable binding names. Care is needed to avoid unintended variable capture and to handle type abbreviations (**typedef**) and the identities of scalar types, since each declaration gives a new set.

5.4.2 Ensuring determinism

The tool does not provide any support for validating the assumption of determinism. To decide whether a process is deterministic would involve checking at each location l, for all reachable valuations of state variables val_V, whether, given a pair $i = \{1,2\}$ of edges on the same channel set and direction $(S_i, A_i, g_i, \langle e_1^i, \ldots, e_{n_C}^i \rangle)$,

$$\forall S_1, S_2. \left(\left(\mathsf{inv}(l) \wedge e_1^1 = e_1^2 \wedge \ldots \wedge e_{n_C}^1 = e_{n_C}^2 \right) \Longrightarrow \neg \left(g_1 \wedge g_2 \right) \right).$$

An over approximation could be used to warn if a model might be non-deterministic.

Fortunately there is a better way. Since timed trace inclusion is reflexive, verification of $\mathcal{P} \parallel \mathcal{P}' \vDash \mathsf{A}\Box\, \neg \mathsf{Err}$ should succeed. A failure indicates a model that does not satisfy the assumption of determinism, or that there is a fault in the transformation software or in Uppaal.

Unfortunately the converse is not necessarily true. For example, a $\mathcal{P}'$ consisting of a single state with unconstrained self-loop transitions for every action will always satisfy the formula in parallel with a more restricted $\mathcal{P}$, and in fact with any other process, including those having traces that $\mathcal{P}$ does not.

It should be possible to augment the tool with features to help find suitable renaming functions [Sto02, Appendix A], for making some automata deterministic for the purposes of testing timed trace inclusion. Such a feature could require automatically executing the Uppaal verification engine and analysing its counter-example traces.

5.5 Validation automaton for the railway controller

The extended timed trace inclusion validation technique can be applied to the railway controller of Figure 5.4, here called $\mathcal{P}$. The required validation automaton, here called $\mathcal{P}'$, is presented in Figure 5.17.

There are four nodes in $\mathcal{P}'$. A new Err' node, and three others transferred with changes from the original automaton: Free', Occ', and Restart'.

$\mathcal{P}$ contains an urgent node Restart, which is expanded before constructing $\mathcal{P}'$ by adding both a new clock c_u, which is reset on all incoming edges, c_u = 0, and the invariant c_u $\leq$ 0. This invariant and those on Free and Occ are not carried over onto the nodes of $\mathcal{P}'$, rather they influence the edges at each node in two ways. First, the original invariant is added as a conjunct to the guards of all outgoing edges. Second, a τ-transition to Err' is added with the negated invariant as a guard. These new transitions are listed at the bottom of each of the three grey clusters from the original state. For instance, the τ-transition from Restart to Err has the guard c_u > 0. The negated invariant for Occ would be $\exists i \in id_t.\, \mathsf{get\_status}(i) = \textsc{appr} \wedge \mathsf{timer}[i] \geq \textsc{deadline}$, but since this expression would be rejected by Uppaal, because timer is an array of clocks, a selection binding is used instead of an existential one. And similarly for Free.

[22] http://www.smlnj.org/doc/smlnj-lib

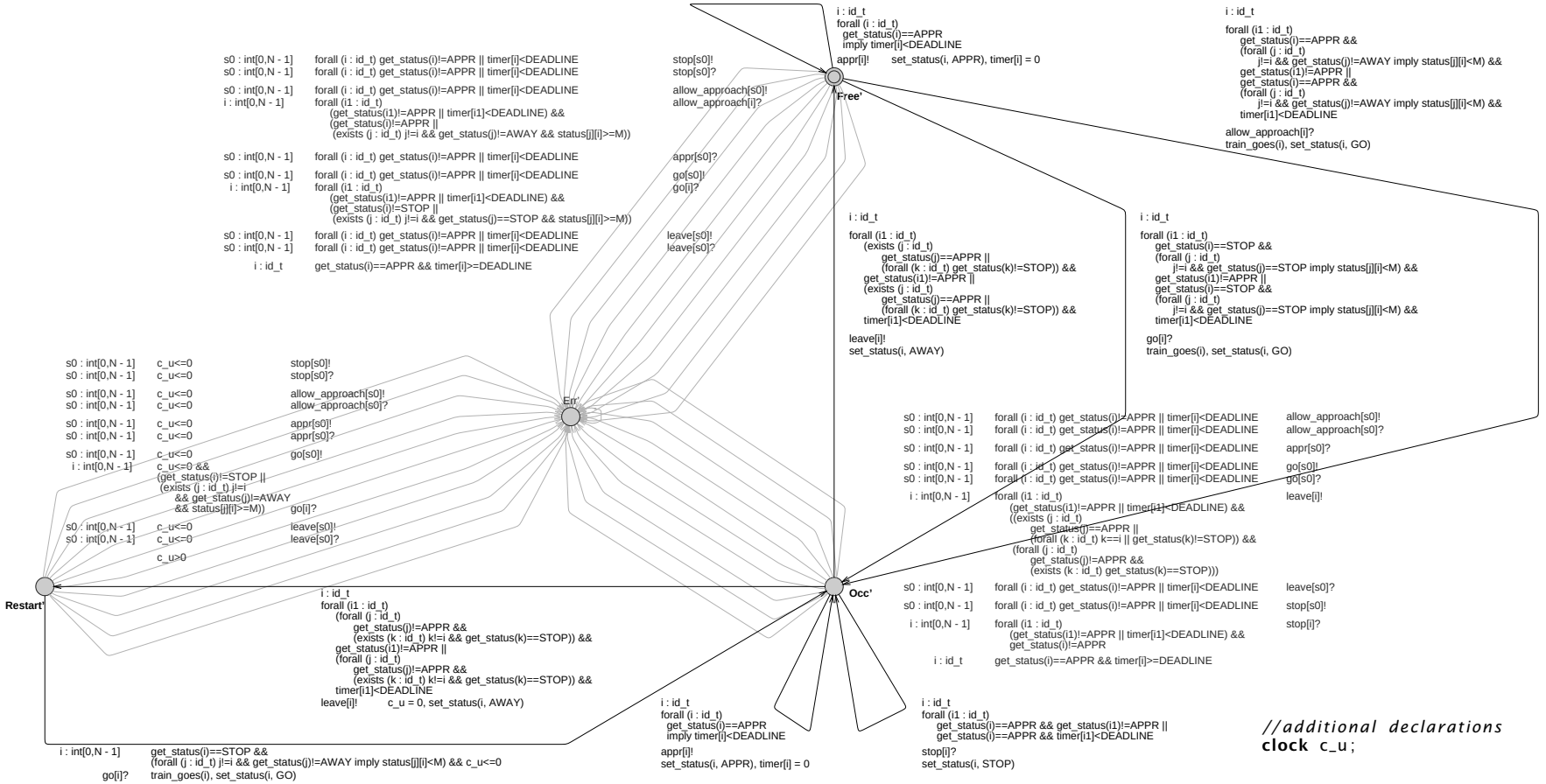


Figure 5.17: Validation automaton for Railway controller

Besides the addition of invariant conjuncts and the inversion of the direction of synchronisation on channels, the edges replicated from the original, and coloured black in Figure 5.17, keep the same assignment expressions and essentially the same guards (though they are rearranged into DNF).

The new edges to the Err' state are coloured gray in Figure 5.17. Their labels are tabulated near their respective source nodes.

Most of the edges in $\mathcal{P}'$ from Free' to Err' are on the complements[23] of action sets that cannot occur from Free in $\mathcal{P}$: stop$^{?}/_{!}$, allow_approach!, appr?, go!, leave$^{?}/_{!}$. Each of them employs a selection binding to cover all elements, and has the original invariant of Free as the guard expression. Since actions in appr? are always enabled in $\mathcal{P}$, there are no corresponding edges to Err' on appr!. The guards of the complements of the other two action sets that can occur from Free, allow_approach? and go?, have the same form: a conjunction of the original invariant and the negation of the original guard. Quantifier nesting is permitted in these guards because the bound clauses do not contain clock variables.

The edges from Occ' and Restart' are similar to those from Free'. The only additional complication is that in $\mathcal{P}$ there are two edges on leave? from Occ. So, their guards are first combined with disjunction:

$$\begin{aligned} & \forall j \in id_t.\, \mathsf{get\_status}(j) \neq \textsc{appr} \wedge \exists k \in id_T.\, (k \neq i \wedge \mathsf{get\_status}(k) = \textsc{stop}) \\ \vee\ & \exists j \in id_t.\, \mathsf{get\_status}(j) = \textsc{appr} \\ \vee\ & \forall k \in id_t.\, \mathsf{get\_status}(k) \neq \textsc{stop}, \end{aligned}$$

before the complement is formed:

$$\begin{aligned} & \exists j \in id_t.\, \mathsf{get\_status}(j) = \textsc{appr} \vee \forall k \in id_T.\, (k = i \vee \mathsf{get\_status}(k) \neq \textsc{stop}) \\ \wedge\ & \forall j \in id_t.\, \mathsf{get\_status}(j) \neq \textsc{appr} \\ \wedge\ & \exists k \in id_t.\, \mathsf{get\_status}(k) = \textsc{stop}, \end{aligned}$$

which, along with the node invariant, forms the guard on an edge from Occ' to Err'.[24]

Uppaal verifies that the modified original controller $\mathcal{O}$, Figure 5.3b, is timed trace included in the flexible version $\mathcal{P}$, by model checking: $\mathcal{O} \parallel \mathcal{P}' \vDash \mathsf{A}\Box\, \neg \mathsf{Err'}$.

While the railway example contains several of the features addressed by the extended validation technique, it does not require handling of mixed quantifiers over clock variables, splitting to eliminate disjunctions with clocks, or channel array subscripts with state variables, duplicated selection bindings, or bindings that do not span the entire subscript.

5.6 Discussion

Uppaal contains features that increase both modelling convenience and the complexity of model transformations. Many of these features have been formalised in this chapter, and a standard technique for timed trace inclusion testing has been extended to address them.

The implementation works well but would be improved by more sophisticated term rewriting, (semi-)automatic validation of the assumption of determinism, features for relabelling to eliminate non-determinism, and better support for multi-process specifications. It allows validation automata to be created automatically for a large class of Uppaal models without having to expand Uppaal features to lower level primitives. The efficacy of the tool has been demonstrated on an example adapted from the literature.

The model of Uppaal processes of §5.2 was developed with the sole intent of extending and describing the validation construction. A semantics for Uppaal with a careful treatment of shared variables and committed locations has been developed and published contemporaneously [BV08]. It does not explicitly address selection bindings, quantifiers, or channel arrays.

[23] The directions are reversed.

[24] The expression in Figure 5.17 is slightly different but equivalent.

There are a number of motivations for developing a theory of Uppaal models in a theorem prover like Isabelle [NPW02]; possibly based on the aforementioned semantics with the extensions proposed in this chapter. First, the correctness of the extended validation construction has only been justified informally. Its intricacies would be better tracked and maintained in a machine-checked development. Second, the implementation could generate the validation construction simultaneously with a justification of its correctness in a form that a theorem prover could validate. Third, an implementation embedded in a theorem prover [WW07] could exploit a more sophisticated and extensible term rewriting engine, and possibly also present some proof obligations to users for interactive solution.

The extensions presented in this chapter build on a known technique [Sto02, JLS00] for testing timed trace inclusion. It seems that an alternative approach using timed games is also possible [BV08, §§4 and 5] and supported by newer versions of Uppaal [BCD+07].[25] The essential challenge in extending the original technique to handle selection bindings, quantifiers, and channel arrays is the need to bundle transitions and negate guards while respecting the syntactic restrictions of Uppaal. These particular problems may be irrelevant in such alternative approaches, but the individual manipulations are still of intrinsic interest and much of the implementation would anyway be reusable.

This chapter grew out of the needs of the last; constructing validation automata for the sensor models was too tedious and delicate to do manually. The challenges of addressing various Uppaal features became apparent as the process was automated. Even though time is not really central to the validation construction, the formalization and techniques provide insight into timed automata modelling in Uppaal and the challenges and practicalities of implementing model transformations for a non-trivial language. The next chapter returns to the core theme of programming embedded controllers and the focus moves from timed automata back to synchronous languages.

[25] This possibility was also independently suggested by Frank Cassez.

Chapter 6

Delays in Esterel

6.1 Introduction

From the Uppaal-specific technicalities of the previous chapter, the focus returns in this chapter to the problem of specifying and implementing embedded controllers with complex timing behaviours. The central premise of this thesis is reprised: time is both a behavioural dimension and a resource.

Time is an integral behavioural dimension in many embedded systems; timing details cannot always be treated as requirements to be validated independently of other design stages. They may rather be so intertwined with other behavioural aspects as to be inseparable from them. For instance, the meaning of an event may depend on the relative time of its occurrence, and the absence of an event relative to another, or within a certain period, may have significance to a system.

Time is also a resource; a physical constraint on system design that introduces limitations and costs. Balancing timing requirements and timing limitations is central to the design of many embedded systems. Accuracy and simplicity of design must be weighed and traded against the cost, complexity, and energy use of eventual implementations. Design and implementation choices are often explored and decided simultaneously, complicating both tasks and encouraging platform-specific programs which may later be difficult to adapt or to reuse. Behavioural timing details often become tightly bound with the mechanisms of their implementation, making them harder to later understand and to modify.

The Esterel programming language was designed as a language for real-time programming [BMR83, Ber89b]. It is thus well suited for describing timing behaviours. But, although the synchronous model of discrete time isolates the logic of programs from many details of their realisation, timing behaviours still cannot be expressed without making significant implementation choices at early stages of specification and design. Such early choices can make it difficult to strike a balance between timing requirements and timing constraints, and they may result in platform-specific programs.

These perceived limitations of Esterel are specific to certain applications and quite subtle. Discussing them with reference to a specific example will serve both to clarify the issues and to underscore their practical importance. They do not adversely affect high-level designs, like the controllers of Chapter 3 or the railway example of Chapter 5, nor are they especially limiting when programming to specifications with relatively liberal timing constraints, like the infrared sensor of Chapter 4. Rather they arise when a program must be designed to meet strict and intricate behavioural timing requirements and when the implementation platform has not yet been chosen; possibly because the minimum platform requirements cannot be known until after the program has been written. A good example is to be found in controllers for the microprinters that print cash register dockets and other transaction logs. The devices themselves are fairly simple which only makes the required controller more complicated. In addition, the controller must usually be executed on as inexpensive a microcontroller as possible.

One simple solution, for addressing applications like the microprinter controller, is to express delays using a macro statement whose expansion into standard Esterel is

determined by an abstract model of an intended implementation platform. This would allow designers to state delays directly during specification and then later to tailor programs to the limitations of particular platforms as more details become available. While program models are often given in discrete time and implementation models in continuous time [STY03], the macro statement implies the opposite approach: programs are stated in continuous time and their implementations in discrete time. Abstract programs are stated in the same terms used in descriptions of the physical hardware to be controlled. Concrete programs are then derived in the form necessary for implementation as a digital system. This approach is familiar in traditional control system design where analytical models are constructed in continuous time and then later discretized for implementation.

While the motivations and basic idea behind the macro delay statement appear sound, the solution presented in this chapter is not completely satisfactory. There remain unresolved questions about the practical utility of the presented transformations and also about the relation between programs with physical time delays and the discrete controllers generated from them. Any proposal for the latter would have to account for the kind of approximations and compromises usually employed when engineering such systems.

The main body of this chapter comprises four sections. In §6.2, the microprinter example is presented. It is both a motivating, realistic application and a concrete illustration of the issues under discussion. Extracts from the example are used throughout §6.3 to illustrate deficiencies in the standard techniques for expressing delays in Esterel. It is argued that each of these techniques either forces engineers to make implementation choices too early in the design process or otherwise adulterates the expression of requirements with the mechanisms of their realisation. A possible solution is presented in §6.4 in the form of a macro statement and its expansion to statements of standard Esterel. Some problems and unfinished aspects of these ideas are discussed in §6.5.

6.2 Motivating example: a microprinter controller

The microprinter controller example typifies a certain type of embedded systems design where engineers integrate specialised electro-mechanical devices from various suppliers into a useful whole. Engineers can sometimes work with suppliers to specify the components, as in the automotive industry or for highly-specialized applications like medical devices, but at other times they must select from a range of Commercial Off The Shelf (COTS) components and then work to constraints set out on data sheets and in timing diagrams. The microprinter specification, like that of the infrared sensor of Chapter 4, is full of idiosyncratic detail originating in its particular physical characteristics. Proposals for improved programming and modelling must confront such detail without compromise, at least if they aim to provide realistic solutions. The microprinter differs from the infrared sensor in its scale—its interface is much more complicated—and in the strictness of its timing requirements.

A program for controlling a device like the microprinter is often loosely called a ‘driver’. But this term is best preserved for the glue code that links OSs with peripherals where the challenge is to interface OS data structures and concurrency with high-level register-based coordination of a specific type of device or class of devices. On the contrary, software like the microprinter controller orchestrates the low-level details of bare components in real-time. It must be much more responsive, and its sequential and timed behaviour will usually be much more involved. There will usually be more internal control state.

Microprinters are electro-mechanical components for producing monochrome images on paper. They are often used in cash registers for printing receipts. A typical example is sketched in Figure 6.1. The actual device, from which the following details and delay values are taken, is not named due to licensing sensitivities. Thermal paper is drawn into the printer from a roll (not shown) by a rubber drum that is rotated by a stepper motor. The paper passes under a print head comprising a row of tens of

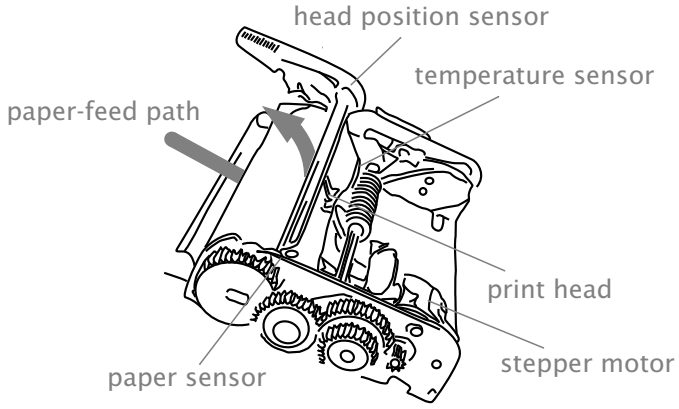


Figure 6.1: Physical structure of the microprinter example

resistors. Current is applied to the resistors to generate heat which marks the paper; individual resistors are enabled and disabled through latched transistors. Images are formed line-by-line by carefully coordinating the movement of the paper, the contents of the latches, and the application of current to the resistors. There are also microprinters that mark normal paper with an impact mechanism, like, for instance, tiny hammers driven through an ink-impregnated ribbon. The microprinter has sensors that give the temperature of the print head, whether it is open or closed, and whether there is paper under it.

The sequential logic required to interface directly with the microprinter is intricate. A controller must produce a signal pattern for the stepper motor, retrieve then serially transmit the next line of pixels, apply current to the resistors, and respond to no-paper and print-head-open events. It must respect the microprinter's physical and electrical characteristics. For instance, when the number of active pixels in a line exceeds a certain threshold, that line must be printed over several phases to avoid drawing too much current; when paper feeding is temporarily stalled, the stepper motor must be switched on and off to reduce the average power needed, and thereby reduce the risk of damaging hardware or circuits.

Furthermore, the relative timing of actions is both important and intricate. The duration of motor steps changes depending on the number of pixels in the line being printed, the duration of the previous step, and the operating phase: starting, feeding, printing, or stopping. The duration of current pulses through the print head depends on feedback from the temperature sensor, the recent print history, and the battery level. The lengths of various delays are given in the microprinter specification in physical time, seconds and milliseconds, not as counts of a digital clock or multiples of a base period. They are integral to the behavioural specification and as much a part of the controller requirements as are the discrete events. It is unnatural to consider the timing constraints and discrete events in isolation from each other.

Expressing the required sequential logic and timing patterns in software is only part of the problem. A microcontroller must also be chosen and interfaced to the microprinter, to a power supply, and to the rest of the system. The choice of microcontroller is critical to implementing, and—as will be seen—usually even to stating, the timing behaviour. Platform selection may thus occur simultaneously with initial design. To give one scenario, an engineer might identify the tightest timing requirements in the specification and then sketch a preliminary implementation in assembly language from which the minimum required processor speed can be estimated. A suitable platform could then be chosen, allowing timing behaviours to be expressed in terms of its characteristics and features. Porting the program to a different platform may require considerable effort. Any detailed verification would have to consider the program and the platform as a single unit.

Esterel is intended for applications like the microprinter controller. It is certainly easier to express the sequential logic in Esterel than in assembler, but it is still difficult to untangle the application timing details from the implementation choices and

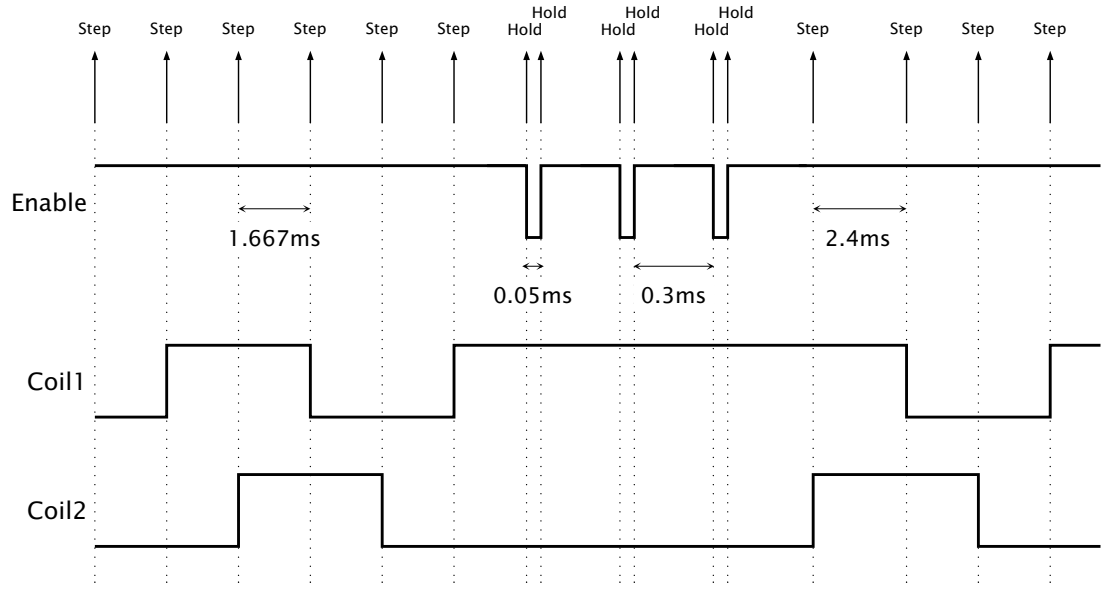


Figure 6.2: Typical microprinter motor control signals

constraints, and this has implications on both design flow and portability. As the microprinter controller is too complicated to present in full, only a subcomponent will be considered, namely the one responsible for energizing the coils of the stepper motor to make it rotate.

A sample trace of the motor control signals is presented in Figure 6.2. There are three outputs: MotorEna, MotorCoil1, and MotorCoil2. The MotorENA signal is asserted to allow current into the stepper motor coils. The MotorCoil1 and MotorCoil2 signals determine the direction of current in each of two coils within the stepper motor. At the lowest level, the coils must be energized according to the pattern of steps in the bottom half of Figure 6.2. At a higher level, the length of each step and whether current should flow or not is determined by the length of the previous step and whether the motor paper must be held in place for one reason or another. The latter condition will be represented by an input signal Hold, which, it can be assumed, will be emitted by other system components as required. When printing, each step is normally energized for 1.667ms, but if the motor is held for more than 0.733ms in one step then the next step must be energized for 2.4ms. The coil directions are not changed while the motor is being held in place. Since this requires less energy, the coil current must be repeatedly switched off for 0.05ms and on for 0.3ms until movement restarts. This 'chopping' reduces the risk of overheating. Other complications relating to starting the motor, stopping it, and feeding paper when not printing will be ignored.

Thanks to the synchronous semantics of Esterel, the motor control logic is readily expressed as two concurrent modules: PrintSteps and Stepper. They are both shown in Figure 6.3. The PrintSteps module emits a Step signal when the coil energisation pattern is to change. The Stepper module responds simultaneously to each emission of Step by changing the direction of current in one of the coils. Other concurrent components for sequencing feed and print cycles, clocking data into the print head, and handling exceptional conditions can readily be imagined. For the most part, the domain specific constructs of Esterel give a convenient and natural specification that can be simulated, analyzed and compiled into software or hardware. There is, however, a problem.

How should the various delays in Stepper be stated? At present, they are given as comments in terms of timing constants from the specification, but the resulting program is neither correct nor executable. Several techniques for expressing the delay are evaluated in the next section, but none of them are ideal. Just as in the assembly language scenario, each technique requires early decisions about the eventual implementation platform, or confuses specifications of delay with their implementation.

```
module PrintSteps:

input Hold;
output Step, MotorENA : boolean;

signal LongStep in
  loop
    emit Step;
    present LongStep
      then % delay 2.4ms
      else % delay 1.667ms
    end present;

    present Hold then
      trap Stalling in
        loop
          emit MotorENA(false);
          % delay 0.05ms;
          present Hold else
            exit Stalling
          end;

          emit MotorENA(true);
          % delay 0.3ms;
          present Hold
            else exit Stalling
          end
        end loop
      ||
        % delay 0.733ms;
        sustain LongStep
      end trap
    end present
  end loop
end signal

end module
```

(a) PrintSteps module

```
module STEPPER:
input Step;
output
  Coil1 := false : boolean;
  Coil2 := false : boolean;

loop
  await Step;
  emit Coil1(true);

  await Step;
  emit Coil2(true);

  await Step;
  emit Coil1(false);

  await Step
  emit Coil2(false)
end loop

end module
```

(b) Stepper module

Figure 6.3: Stepper motor controller in Esterel

6.3 Expressing delays in Esterel

Timing delays can be expressed variously in Esterel. Several techniques from the literature are reviewed in this section. It will be argued that all of them constrain eventual implementations, at least if naively compiled, and that several of them either emphasize mechanism over effect or interact imperfectly with other constructs.

Techniques from published Esterel programs are presented in the first three subsections: **pause** statements in §6.3.1, signal counting in §6.3.2, and external timers in §6.3.3. A seemingly novel variation using the **exec** statement is presented in §6.3.4. Two techniques proposed in the research literature are discussed: non-deterministic pause statements in §6.3.5, and quantitative watchdog timers in §6.3.6. Two techniques from other programming languages are described in §6.3.7: instruction delays and library functions. These techniques are included for completeness and contrast; they are never applied in orthodox Esterel programs.

6.3.1 Pause statements

In the latest semantics of Esterel [Ber99, PBEB07], **pause** is the only non-instantaneous statement. Its meaning in the discrete semantics is clear: it delays execution until the next reaction. The complication for expressing quantitative delays is that the time of the next reaction depends on the execution mode and parameters.

In the event-driven execution mode, the physical duration of a **pause** depends on external stimuli. For a set of inputs $\{i_1, \ldots, i_n\}$, a **pause** could be replaced with:

await [i_1 **or** ... **or** i_n].

Although, the replacement would have to be adjusted were other inputs added; if, for instance, other modules were placed in parallel. Any relation between abstract delays and physical delays must account for times of input occurrence, which is not feasible in general. In event-driven systems, unadorned **pause** statements alone are not suitable for specifying precise physical delays.

The arguments against unadorned **pause** statements for expressing delay, are similar to those against allowing the next operator in temporal logics. In temporal logic, *stutter invariance* is an important property because *the number of steps required to complete an operation is not a meaningful concept when one gives an abstract, high-level specification* [Lam83]. A more delicate balance is required in a synchronous language like Esterel where semantics and programmer intuitions are based on the precise interrelationships of program steps. The challenge is rather to treat physical delays at a higher, more abstract level, without completely losing the character of the language.

It seems that these issues were considered in earlier semantic definitions of Esterel [BG92] where the only non-instantaneous statement is not **pause** but rather **halt**, which delays indefinitely, or, more usually, until aborted by an input event. The ability to detect any reaction is recovered by providing a distinguished tick signal that is present at every reaction; **pause** can then be encoded as a macro statement: **await** tick. Even with a primitive **pause** statement, the tick signal is still useful for expressing n consecutive **pause**s:

await n tick.

In the sample-driven mode of execution, a **pause** statement specifies a precise physical delay: the length of one execution cycle. There is thus a direct, though implicit, relation between the discrete semantics of a program and its physical behaviour. For a given sample time the relation can be defined precisely; as, for example, the model in Chapter 3 does. In applications where behaviour in physical time is important, modules could be specified together with their intended execution period. It is not clear, however, how modules with different execution periods would be composed. Furthermore, designers would be forced to choose a period before writing a program. They must make an implementation choice before even beginning a precise specification!

Deciding on an execution period involves compromises between the application requirements and the execution platform. The timing requirements of the micro-printer controller example can be summarized by the list of delays: 2.400ms, 1.667ms,

0.050ms, 0.300ms, and 0.733ms. A designer could decide to round 1.667ms down to 1.650ms and 0.733ms down to 0.750ms before choosing 0.05ms, the greatest common divisor and, in this case, also the smallest delay, as the execution period. The first part of the program could then be written:

```
present LongStep
then await 48 tick
else await 32 tick
end present.
```

This technique is effective but not ideal. The program has strayed from the original specification. If the execution period is changed—for instance, a different microcontroller is used, or a faster module is put in parallel—the program must be rewritten. The original delay values are obscured and the execution period is implicit. Moreover, a complete list of delays may only become clear as the program is written: the specification and important details of the implementation must be decided in tandem. A fixed execution period limits potential platforms, since the whole program must run at the speed required for the smallest delay, even though in this case the next smallest delay is an order of magnitude greater. There is little scope for the sort of optimisations often applied to embedded controllers; for example, a timer-interrupt-driven routine for motor chopping that permits the rest of the program to be executed less frequently.

For some applications, for example bus protocols and computer hardware, cycles are an integral part of the design. The Turbochannel bus controller (tcint) that is widely used for benchmarking Esterel is a typical example. The class of applications exemplified by the microprinter controller is different in this respect.

6.3.2 Timing inputs

A natural generalisation of the **await** tick form of the **pause** statement is to allow signals other than tick. The semantic expansion of **await** *s* is:

```
trap T in
  loop
    pause;
    present s then exit T end
  end loop
end trap.
```

It may be seen as a conditional **pause** that delays until the next reaction where the given signal is present. The **await** *n* tick form of the **pause** statement effectively counts reactions. The generalisation **await** *n s* counts the occurrence of particular signals; its semantic expansion is given in Table 2.2.

Counting specific signals instead of reactions is a partial remedy for some limitations of **pause** statements: the event being counted is stated explicitly, and it need not be present at every reaction. Executing signal counting programs more frequently does not change the fundamental relation between their behaviour and the occurrence of external events, although the order of external events may be discerned more finely and the actual discrete traces may vary.

Additional information is still required to relate a signal counting statement to a physical time delay. Rather than assign an execution period to a program or module, as for **pause** statements, certain *timing inputs* are distinguished and assigned fixed delay values. The delay values are usually relative to the initial reaction or to system startup. They are, in other words, absolute. Timing inputs must be provided by the interface or run-time layer at regular intervals. They are invariably given suggestive names, for example SECOND or MSEC, and implication relations between them are often declared.

Returning to the microprinter, a controller program could commence with declarations of two timing inputs, TMS for ‘tenths of milliseconds’ and HMS for ‘hundredths of milliseconds’:

```
input TMS,   % ms/10
      HMS;   % ms/100
relation TMS => HMS; .
```

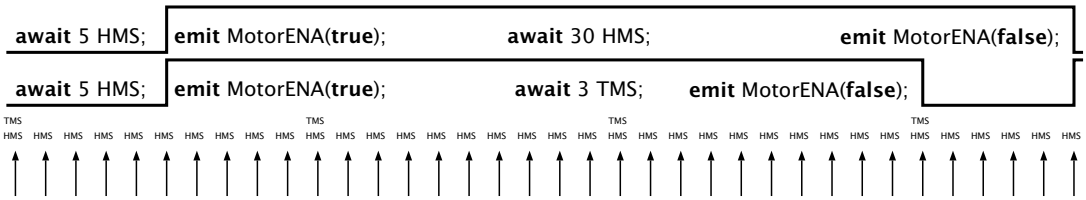


Figure 6.4: Granularity of timing inputs

Longer delays would be specified in terms of TMS:

```
present LongStep
  then await 24 TMS
  else await 17 TMS
end present.
```

and shorter ones in terms of HMS:

```
loop
  emit MotorENA(false);
  await 5 HMS;
  emit MotorENA(true);
  await 30 HMS
end loop.
```

Timing inputs are employed in several examples [Ber89a, BG89]. They fit superbly with the idea of multiform time and the abstract synchronous model. They work well with other Esterel constructs like **suspend** and **abort**.

There are, however, at least three disadvantages to counting timing inputs. First, although the relation between timing inputs and physical time seems intuitive, there are some subtleties related to granularity and relativity. Second, although signal counting programs are relatively unaffected by changes to execution mode and period, the choice of signal granularity is effectively an implementation choice and trading accuracy for economy afterward may not be trivial. Third, the structure of the state space of signal counting programs may be difficult for debugging and model checking tools to exploit.

Regarding granularity and relativity, a signal counting statement synchronizes with timing inputs foremost and creates a delay in physical time only as a byproduct. Timing inputs are not relative to the commencement or termination of statements within a program. For instance, consider these changes to signal granularity in the motor chopping loop:

```
loop
  emit MotorENA(false);
  await 5 HMS;
  emit MotorENA(true);
  await 3 TMS                % ⇐ was 30 HMS
end loop.
```

The two fragments are not equivalent but one might naively expect that replacing **await** 30 HMS with **await** 3 TMS would preserve the physical time delays. This is not so, as evidenced by Figure 6.4. The statement **await** 3 TMS always gives a logical delay of 3 TMS events but, in principle, the associated delay in milliseconds could be anywhere in the interval $(0.2, 0.3]$. The precise delay depends on when the statement receives control and thus on the system execution period and, in the event-driven mode, when other inputs occur. Consider, for instance, this statement:

```
await I; await immediate 2 S.
```

The start of the second **await** depends on when the I signal occurs. It effects a delay greater than one second but strictly less than two seconds, that is, a delay in the interval $[1, 2)$—assuming that S has a period of one second.

Does it really matter? After all, engineering involves tolerances: perfect measurements are never possible. The point is, rather, to delay such decisions for as long as

possible; to model in ideal terms and then only later to make and evaluate various compromises. Fixing timing inputs at an early stage in the specification either renounces accuracy too soon, perhaps even before the ramifications can be properly understood, or risks imposing unnecessarily strict demands on eventual implementations.

There is another conflict between abstract specifications and concrete implementations. In applications like the microprinter controller, data sheets and abstract designs describe physical models as functions of an ideal t in seconds. But oscillation and execution periods in implementation platforms are often determined by characteristics of the application and hardware. For instance, an oscillator frequency of 11.0592MHz may be ideal for serial communication but not for counting milliseconds [Pon01, pp. 367–369]. The execution periods of early arcade game systems were often based around the update frequency of the attached video hardware [ET05, §4]. Moreover, the timing inputs in early stages of a design may be in multiples of seconds, but those in later versions may not. Discretization is ultimately implementation issue.

Naturally, the standard tools for simulation and verification can handle programs that count timing inputs. But they do not usually exploit the specific structure of these programs: the long chains of counting transitions. When debugging, for instance, it may be necessary to cycle through long runs of timing events before anything interesting happens. In tools like Uppaal, by contrast, such traces can be explored symbolically. Verification tools might also be more effective were they to identify and exploit the additional structure. For instance, by supporting the expression and validation of quantitative timing properties in terms of physical rather than discrete time, and by summarising information about delays in counter-example traces.

6.3.3 External timers

One-shot timers are commonly used in embedded programs to implement delays and timeouts. The same idea is readily expressed in Esterel, as demonstrated by several published examples [CDO96, JPO95, MS92].

Such programs initiate a delay by emitting an event that starts a timer, for instance **emit** START_TIMER. The event may be parameterised by the required number of ticks, for instance **emit** START_TIMER(100). The program then waits for an event that indicates timer expiry: for instance, **await** TIMER.

Timers need not necessarily be provided by an implementation platform. They may themselves be implemented in Esterel, as, for example, in the POLIS seatbelt alarm controller [BCG+97, §1.3.2], shown in Figure 6.5, where a timer module counts timing inputs. The POLIS approach is special because the two modules may be executed on different asynchronous processes, each with a different execution period. The timer module may even later be refined to a hardware timer.

There are several advantages to using timers. They give relative rather than absolute delays. They separate, at least to some degree, issues of behavioural delay from those of program execution. Timers can, for instance, run at a finer granularity than the rest of the program; though any benefit is lost if the ratio between the timer and execution periods is too great. They are perhaps most appropriate for event-driven implementations where reactions can be triggered by timer interrupts.

There are four main disadvantages to using timers: a sacrifice of program concision and clarity, an early introduction of implementation detail, an imperfect interaction with other Esterel constructs, and a lack of support in simulation and analysis tools.

The loss of concision and clarity is evident in this fragment of the microprinter controller example, now expressed with timers:

```
loop
  emit MotorENA(false);
  emit START_HMSTIMER01(5); await HMSTIMER01;
  emit MotorENA(true);
  emit START_HMSTIMER01(30); await HMSTIMER01
end loop.
```

Not only are two instructions required to express each delay, but the emphasis has shifted from meaning to mechanism. Nothing prevents the emission that starts the

```
module belt_control:

input reset, belt_on,
      key_on, key_off,
      end_5, end_10;
output alarm(boolean),
       start_timer;

loop
  abort
    emit alarm(false);
    every key_on do
      abort
        emit start_timer;
        await end_5;
        emit alarm(true);
        await end_10
      when [key_off or belt_on];
      emit alarm(false)
    end every
  when reset
end loop
```

(a) belt_control module

```
module timer:
constant count_5,
         count_10 : integer;
input start_timer, msec;
output end_5, end_10;

every start_timer do
  await count_5 msec;
  emit end_5;
  await count_10 msec;
  emit end_10
end every
```

(b) timer module

Figure 6.5: POLIS seatbelt alarm controller [BCG+97, §1.3.2]

delay being placed apart from the statement that detects its end. This may sometimes be an advantage, but it surely also complicates potential analysis and compilation techniques.

Timers introduce implementation details. Each has a granularity and a maximum value. Timers must be allocated and named. An implementation platform must either provide enough timers, or provide extra routines for queuing and managing timer requests. Care must be taken when interfacing timers to ensure that they react appropriately with features of Esterel. Consider this program fragment for example:

```
abort
  emit START_TIMER(10); await TIMER;
  emit O1
when I;
emit START_TIMER(20); await TIMER;
emit O2.
```

Assume that it is executed in the event-driven mode and that it has been waiting at the first **await** TIMER statement for almost 10 units when the I input triggers a reaction. The I input will abort the first delay and start the second one. But if the timer expires while the reaction is being processed it may set an interrupt flag or other latch, and, if the latch is not properly cleared by the interface layer, the second delay may be terminated prematurely in the next reaction. Such bad interactions with abortion can be avoided, but only with care.

Interactions with the **suspend** statement are not as easily solved. The problem is that timers essentially sit apart from the lexical scope of the statements that start and await them. Two examples will illustrate the issues.

First, indefinite delays are easily introduced when timers are combined with suspension, as in this program fragment:

```
suspend
  emit START_TIMER(10); await TIMER;
  emit O1
when HOLD.
```

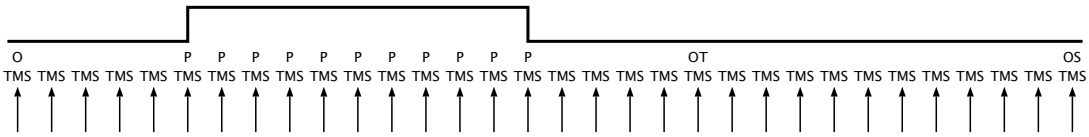


Figure 6.6: Effect of suspend on delays

If the HOLD and TIMER signals occur simultaneously, the **suspend** prevents termination of the **await**, and, if the timer is not restarted elsewhere, the O1 signal will never be emitted. One alternative to accepting this behaviour is to declare a conflict relation between the two signals:

```
relation TIMER # HOLD;
```

But this really only shifts the burden to the interface layer. In any case, the combination of timers with suspension requires care.

The second example compares the effect of suspension on a delay expressed with an external timer and one expressed by counting timer inputs:

```
emit O;
suspend
  emit START_TMSTIMER01(20); await TMSTIMER01;
  emit OT
||
  await 20 TMS;
  emit OS
when P.
```

Suppose the P signal indicates when a certain button is held. A run of the system with the button held for 0.1ms is shown in Figure 6.6. The signal OT is emitted when the timer in the top branch expires. This emission is completely unaffected by the suspension because the timer is external to the program, even though the statements that trigger and wait for it are within the scope of the **suspend**. In contrast, the emission of the signal OS, which occurs after counting the timing inputs, is delayed by the length of the suspension, modulo sampling effects. The latter behaviour is the more powerful but it is not easy to achieve outside the Esterel kernel.

As far as the semantics of Esterel are concerned there is nothing special about the **emit** and **await** statements that comprise a timer delay. This means that standard simulation and analysis tools will not usually exploit the implied timing constraints. To verify quantitative timing properties or to eliminate spurious counter-examples, the timers themselves would have to be modelled or otherwise taken into account.

6.3.4 External intervals

Esterel is extended to Communicating Reactive Processes (CRP) [BRS93] through the addition of an **exec** statement, which starts an asynchronous process and waits until it terminates. This gives another way to implement external timers, for instance:

```
loop
  emit MotorENA(false);
  exec HMSTIMER01(5);
  emit MotorENA(true);
  exec HMSTIMER01(30);
end loop.
```

The HMSTIMER01 process is assumed to sleep for the given number of hundredths of milliseconds and then terminate.

There do not seem to be any published examples that use this technique. Perhaps because the **exec** statement is not always included as a core statement of Esterel.

There are three main advantages over the timers described in the previous subsection. The delay is expressed as a single statement, which makes programs easier to

read and simpler to analyze. The semantics of **exec** precisely defines its interaction with **abort**. The semantics also accounts for issues of naming and reincarnation.

Otherwise, timers expressed with **exec** have similar disadvantages to those expressed with **emit** and **await**. Their use involves the early introduction of implementation detail: timer names, quantities, granularities, and maximum values. They do not interact well with suspension, which was introduced contemporaneously [Ber93]. The additional timing information is not explicit in the semantics of programs and it is difficult to exploit in simulation and analysis tools.

6.3.5 Non-deterministic pause

The TAXYS [BCP+01, STY03] system analyzes the timing behaviour of Esterel programs with timed automata model-checking. Two techniques account for timing properties.

The first technique addresses deviations from the ideal of instantaneous execution. External function calls within an Esterel program are annotated with lower and upper bounds on their execution times. As these are delays that occur within a reaction, this approach is not suitable for expressing delays in the microprinter controller.

The second technique is for expressing the physical timing properties of a system's environment. Environments are modelled in a version of Esterel that includes a *non-deterministic choice* statement, for example:

```
npause %{ 65 ≤ X ≤ 70, X := 0 }%.
```

The **npause** keyword is followed by a clock guard expression and a list of clock reset commands that are transferred to transitions when the model is compiled into a timed automaton. In the TAXYS methodology, *this statement is not to be used for programming the application software* [STY03, Section IV-A]. But **npause** statements with deterministic clock expressions could be used to express delays in physical time:

```
% clocks are specified in ms/100
loop
  emit MotorENA(false);
  npause %{ X = 5, X := 0 }%;
  emit MotorENA(true);
  npause %{ X = 30, X := 0 }%
end loop.
```

These delays are not relative to an input signal nor to an eventual execution period. Rather they are expressed in terms of continuous-time clock variables. The syntax is a bit clumsy but there are no issues of timer naming and allocation.

There are three other issues to consider: the precision of constants, suspension, and compilation.

Clocks may only, by default, be compared to integer constants, which means both that a precision must be selected before any delays can be expressed, and that additional information, like the comment in the fragment above, is needed to interpret their meaning. This is a only a minor issue, though, and one that could be mitigated by compilers and other tools.

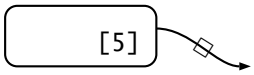
It is not clear how **npause** statements interact with suspension. Presumably, only their termination would be suspended. Clocks can be suspended in *stopwatch automata* [CL00], which are more expressive than timed automata and many analysis problems are not decidable in general.[1]

Non-deterministic choice in TAXYS is intended solely for modelling physical time delays, not for expressing them in programs. Models that contain **npause** statements are transformed into timed automata for reachability analysis, but they are not compiled into executables, which would require discretizing the delays and making other compromises for the constraints of particular implementation platforms. Compilation techniques and tools would have to be developed or adapted.

[1] This observation was made by an anonymous reviewer for EMSOFT 2007.

6.3.6 Quantitative watchdogs

The Argos language defines a temporized state macro, described in §2.4.2.8, for expressing timeouts; delays are stated by pairing an integer value with a signal name. Physical time delays can be expressed by counting timing inputs as previously described. There is an earlier proposal for *temporized Argos programs* [JMO93] where delays are written without an explicit signal name; timeout states are labelled with an integer constant between square brackets and they have a single timeout transition that is identified by a square box:



Two interpretations are defined for temporized Argos programs [JMO93]. In the discrete-time semantics, the timeout notation is just a macro that counts a special input event and an Argos program is interpreted as a BMM. In the continuous-time semantics, an Argos program is interpreted as a timed automaton[2] where the timeout notation is mapped to clocks, location invariants, and transition guards in a natural way. The separation of discrete and delay transitions in timed automata is also adopted for the discrete semantics: the special time input cannot occur synchronously with other inputs. There are implicit conflict relations.

Temporized Argos overcomes some of the limitations of counting timing inputs. Namely, quantitative timing properties can be verified by special-purpose tools, in this case Kronos [Yov97], and there are fewer obstacles to creating simulation and debugging tools that take advantage of the timing parameters.

But the proposal also suffers from the three deficiencies noted for TAXYS in §6.3.5. First, there are relatively minor issues surrounding the precision of timeout constants and the unit of measurement that applies in a given program. Second, the interaction of timeout states and suspension, or, in the case of Argos, with inhibition, is problematic. Third, there is no support for analyzing or making compromises for particular implementation platforms. There is only one discrete transformation and it does not allow for changes to the timing input granularity. The separation of timing inputs from other inputs, however, does allow timers to be treated separately. They could, at least in principle, run at a higher resolution than the rest of the program.

Essentially, in temporized Argos, statements that count timing inputs are treated as continuous-time delays. An alternative would be to do the inverse: to specify delays in continuous time and then to implement them using standard synchronous language techniques.

6.3.7 Other techniques

In other programming languages delays are often expressed by exploiting execution delays or using functions from libraries. These techniques are not normally used in synchronous programs, but they are interesting in comparison with the others and as another source of ideas.

The non-zero execution times of instructions can be exploited to effect delays in physical time. For instance, `nop` instructions may be inserted solely for the delays they cause; an approach demonstrated in the program of §4.5. The idea is less applicable in synchronous programs where instantaneous execution is assumed, although inserting **`pause`** statements has a similar effect. Such delays can be made less dependent on implementation details by adding feedback from clock variables or functions. Figure 6.7, for example, shows a reimplementation of a typical example [Bur95] in Esterel.

While the feedback reduces dependence on the precise length of the **`pause`** delay,[3] which is essential in the Esterel version, other deficiencies remain. The implementation must provide a clock input, and the desired effect—a delay of 10 units—is lost in a mass of mechanism. The relation between the units of the clock and those of physical time is unknown, or, at best, implicit. Furthermore, a precise delay of 10 units has already

[2] Technically, a *timed graph* [ACD90].

[3] The system must either be sample-driven, or, if event-driven, inputs must occur regularly, otherwise the **`pause`** statement may give an indefinite delay.

```
sensor Clock : integer;
⋮
trap Waiting in
  var start := Clock : integer in
    loop
      if CLOCK − START > 10
        then exit Waiting
      end if;
      pause
    end loop
  end var
end trap.
```

Figure 6.7: Delaying with feedback from a clock variable (adapted from [Bur95])

been diminished to one of at least 10 units. Such compromises are best left to later stages of design.

In most programming languages, timing behaviours are expressed using library functions, or keywords that may as well be library functions due to non-existent, loose, or varying timing semantics. Two distinct types can be identified: delays and timeouts.

The Portable Operating System Interface (POSIX) `sleep` function is an example of a library-based delay function. When called, it suspends a thread for a given number of seconds, although *the suspension time may be longer than requested due to the scheduling of other activity by the system* [IEE04, pp. 1407 and 1408], and a signal may truncate a delay. The **delay** keyword of Ada can be used similarly to delay program execution for a specified period, though not an exact one [Bur95]:

> It is important to appreciate that 'delay' is an approximate time construct, the inclusion of which in a task indicates that the task will be delayed by at least the amount specified. There is no upper bound given on the actual delay.

Library functions and keywords sometimes also provide timeouts. The Ada **delay** keyword can also be used, subject to the caveat above, to express timeouts on inter-process communications, for example:

```
select
  --rendezvous commands
or delay 10.0;
  --timeout handler
end select.
```

Similar functionality is provided by operating system APIs, as in the POSIX `select` function or the Windows `WaitForMultipleObjects` function.

These library functions have several drawbacks. While imprecision because of implementation details is inevitable, it would be better if the relationship between the delays requested and those delivered could be better determined or influenced as implementation details are clarified. This is particularly important for applications like the microprinter controller. The two libraries mentioned in the examples above, POSIX and Windows, are large and complex. The interaction of their many features and the non-determinism in their descriptions makes them difficult to use and to reason about, particularly when it comes to timing details. Real-time versions of these libraries contain more features for expressing behaviour in physical time, but at the expense of abstraction; there is more determinism but also more complexity and less portability.

6.4 An alternative

It has been argued that none of the existing techniques for expressing timing behaviour are ideal for programming systems like the microprinter controller. In this section,

several characteristics of an ideal programming language are identified, before an extension to Esterel that aims to meet them is proposed.

The extension has three parts: a macro statement that allows exact delays to be specified in the program text, a language for describing abstract details of implementation platforms, and a syntactic transformation that expands macros into standard Esterel statements suited to a particular platform. The extension is called *Esterel+delay*.

The desired characteristics of a language for expressing the timing behaviour of applications like the microprinter controller are summarised in §6.4.1. The extension to Esterel that attempts to realise them is presented in §6.4.2, and compared to related approaches in §6.4.3.

6.4.1 Desired characteristics

Esterel is ideal for specifying the discrete behaviour of applications like the microprinter controller, but, arguably, the specification of behaviour in physical time could be improved. Specifically, three characteristics are desired. First, it should be possible, at least in the early stages of design, to program in terms of physical time. Second, expressions of delay should not unduly bias the mechanisms by which they are eventually realised. Third, it should be possible to program initially in ideal terms and then later to make the inevitable compromises for implementations on specific platforms.

While digital implementations are inevitably discrete, early designs usually involve continuous models of the controller and plant; even if such models are incomplete or implicit. Engineers think about potential solutions as physical delays and movements orchestrated by discrete modes and steps. Delays are presented in specification sheets and described in design documents in physical time units. The details of discretization and realisation are worked out later when or after choosing an implementation platform.

All of the techniques described in §6.3 immediately require or assume information about the timing behaviour of eventual implementation platforms. It would be better if controllers could be specified, simulated, and analyzed well before making such implementation choices. In fact, the controller specifications themselves should guide choices: hardware or software, minimum processor speed, the number and resolution of timers, and similar.

An ideal language for applications like the microprinter controller would not only allow abstract descriptions of discrete behaviour in physical time, but would also facilitate the inevitable choices and compromises required to implement such programs on constrained platforms. A program should act as a reference against which possible implementations may be evaluated. Especially since perfect precision is not possible: quantitative specifications are given with explicit or implicit error tolerances, and accuracy may be compromised to better meet other constraints and requirements.

6.4.2 Esterel+delay

The program of Figure 6.3 is already an excellent specification. It expresses the desired behaviours in the same terms as the physical model described by the datasheet and without making too many assumptions about their implementation. Rather than immediately replace the timing comments with any of the constructions in §§6.3.1–6.3.7, it may be better to maintain those details for as long as possible, and only later, when platform details are known, to replace them with more concrete mechanisms, as automatically as possible. There are three parts to Esterel+delay: a statement for expressing exact delays, a language for describing relevant platform details, and syntactic transformations that instantiate delay statements. The first two parts are described below. The syntactic transformations are described in § 6.4.2.1, § 6.4.2.2, and § 6.4.2.3.

The statement for expressing exact delays is written:

delay *e*

where *e* is an expression that evaluates statically to a rational number which is interpreted as a duration in seconds. The expression may contain units, which are macros for multiplication by a suitable constant:

$$
\begin{array}{lcl}
x\ \mathsf{h} & = & x * 3600 \\
x\ \mathsf{m} & = & x * 60 \\
x\ \mathsf{s} & = & x * 1 \\
x\ \mathsf{ms} & = & x * 10\mathsf{E}{-3} \\
x\ \mathsf{us} & = & x * 10\mathsf{E}{-6} \\
x\ \mathsf{ns} & = & x * 10\mathsf{E}{-9}
\end{array}
$$

Uncommenting the delay statements in the program of Figure 6.3 gives a valid Esterel+delay program.

Insisting on the evaluation of delay expressions at compile time simplifies transformation and analysis but excludes some potential programs. Similar statements in Esterel, namely **repeat** *e* and hence also **await** *e s*, are less restrictive; they contain integer expressions that are evaluated at run time. The **delay** statement is different because the accompanying expression gives a rational value that is used in the calculation of execution parameters, which, in turn, determine how closely the value will actually be approximated. The restriction to static delay expressions does not preclude conditional or variable delays, but it becomes mandatory to state all possibilities explicitly. For example, step length in the microprinter controller is determined dynamically, but there are only two possible values, those at lines 10 and 11 of Figure 6.3a.

The **delay** statement is essentially a version of **pause** whose interpretation in physical time does not depend on implicit execution parameters. Another possible approach is suggested by the earlier semantics of Esterel [BG92]: an abortion statement for exact delays of the form **abort** *P* **after** *e*. Given this abortion statement as a primitive, the **delay** *e* statement can be derived as **abort halt after** *e*. Conversely, it can itself be derived from the **delay** primitive:

```
signal TO in
  abort P when TO
||
  delay e; emit TO
end signal.
```

Although it is not yet clear whether such equivalences are truly justified.

At first glance the distinguished role of physical time in **delay** statements may seem to violate the doctrine of *multiform time* [Ber00a, §3.10]. But, on the contrary, there is no dispute that a discrete controller perceives nothing but sequences of events and that it may as well count metres or heartbeats as seconds. Rather the approach proposed by Esterel+delay is to program at a slightly more abstract level that acknowledges the dual aspects of time as a behavioural dimension and as a computation resource. Whether it is of any utility to regard other dimensions similarly is a question left open.

The second part of Esterel+delay is a language for describing implementation platforms. Given extra platform details, an Esterel+delay program can be transformed into an Esterel program without **delay** statements, which can then be compiled using standard tools and techniques.

An implementation will be described by a *platform statement* that provides the abstract parameters necessary to approximate ideal delays. Three parameters were used to describe implementation details in the transformation of Chapter 3: execution mode, (minimum) period, and output lag. The output lag does not seem to be especially valuable and the Esterel+delay transformations instead rely on the usual assumption of perfect synchrony. The execution mode is, however, an essential parameter. The period parameter will be retained for the sample-driven execution mode, but the event-driven execution mode requires a different treatment.

Platform statements for sample-driven implementations simply state the execution period in seconds, but the concrete syntax also allows multiplying units, identically to those of **delay** statements, for example:

```
sample 1.4ms
```

Relating event-driven implementations to physical time is more complicated. Two types of platform statement are proposed. The first provides the list of the types of timers available on a platform. Each type of timer is described by four parameters: the physical time period of each tick, the minimum number of ticks possible, the maximum

⟨implstmt⟩→ **sample** ⟨ratexpr⟩ | **event [** ⟨events⟩ **]**
⟨events⟩→ ⟨signals⟩ | ⟨timertys⟩
⟨signals⟩→ ⟨signal⟩ | ⟨signal⟩ **,** ⟨signals⟩
⟨signal⟩→ ⟨name⟩ = ⟨ratexpr⟩
⟨timertys⟩→ ⟨timerty⟩ | ⟨timerty⟩ **,** ⟨timertys⟩
⟨timerty⟩→ **(** ⟨ratexpr⟩ **,** ⟨intexpr⟩ **,** ⟨intexpr⟩ **,** ⟨intexpr⟩ **)**

(where 'ratexpr' and 'intexpr' are expressions that evaluate to, respectively, rational and integer values.)

Figure 6.8: Concrete syntax of platform statements

number of ticks possible, and the number of such timers available. For example:

```
event [(1ms, 10, 65535, 2), (0.1s, 1, 255, 1)]
```

This platform statement describes a system with three timers. Two of them have a tick resolution of 0.001 seconds for countdowns from between 10 and 65535 ticks inclusive. The other has a tick resolution of 0.1 seconds for countdowns from between 1 and 255 ticks inclusive. The second type of platform statement lists input signal names together with their periods of occurrence in physical time, for example:

```
event [SEC=1, OSC=90.0422ns]
```

Such statements clarify assumptions that are otherwise at best implicit in the signal names.

Other platform statements for event-driven implementations can be imagined, for instance, platforms that provide both regularly occurring inputs and timers. Or regularly occurring inputs with offsets relative to system startup as well as periods; like the discrete sample time pairs of Simulink. These possibilities are not pursued because their practical utility is unclear and the two proposed platform statements provide challenge enough.

The concrete syntax for platform statements is summarised in Figure 6.8. The abstract definitions are similar in form. (In the following, $\mathbb{Q}^{\geq 0}$ is the set of non-negative rationals, $\mathbb{Q}^{>0}$ is the set of strictly positive rationals, $\mathbb{N}$ is the set of natural numbers, and $\mathbb{N}^{>0}$ is the set of natural numbers excluding zero.)

Definition 6.4.1
A *timer type* is a tuple $(\tau_t, l, u, n) \in \mathbb{Q}^{>0} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}^{>0}$, where $0 < l \leq u$. ■

In a timer type (τ_t, l, u, n), τ_t is the tick resolution in seconds, l and u are, respectively, the inclusive minimum and maximum values that the timer can provide, and n is the number of such timers that are available.

Definition 6.4.2
Given a set of signal names S, a *timing input* is a pair $(s, \tau_s) \in S \times \mathbb{Q}^{>0}$. ■

In a timing input (s, τ_s), s is the name of a signal and τ_s is its period of occurrence in seconds, relative to system startup.

Definition 6.4.3
Given a set of signal names S, a *platform statement* is an element of the set

$$\mathcal{P} = \mathbb{Q}^{>0} + \mathcal{T} + \mathcal{A},$$

where $\mathcal{T}$ is the set of finite sets of timer inputs, and $\mathcal{A}$ is the set of finite sets of timing inputs where $(s, \tau_{s_1}), (s, \tau_{s_2}) \in A \implies \tau_{s_1} = \tau_{s_2}$. ■

A timer statement is either a single, non-zero rational number that represents the sample period of a sample-driven implementation, or a finite set of timer inputs, or a finite set of timing inputs without duplicates.

The following three subsections describe the transformation of Esterel+delay programs into Esterel programs for each type of platform statement.

6.4.2.1 Sample-driven implementations

A platform statement of the form $\tau \in \mathbb{Q}^{>0}$ specifies a sample-driven implementation with an execution period of τ seconds. In this case, each **delay** e statement is essentially replaced by an **await** n tick statement, where n is chosen to effect the delay specified by the expression e for the given execution period τ. Three variations are proposed for approximating delays that are not multiples of the given execution period.

The transformations described in this section and the following two only replace the **delay** statements in Esterel+delay programs. The common part of their individual definitions is formalised in an obvious way.

Definition 6.4.4
The *carrier function* $C(p)$ is defined for every Esterel statement p:

$$\begin{aligned}
C(\textbf{nothing}) &= \textbf{nothing} \\
C(\textbf{emit}\ \mathrm{s}) &= \textbf{emit}\ s \\
C(\textbf{pause}) &= \textbf{pause} \\
C(\textbf{present}\ \mathrm{s}\ \textbf{then}\ \mathrm{p}\ \textbf{else}\ \mathrm{q}\ \textbf{end}) &= \textbf{present}\ s\ \textbf{then}\ C(p)\ \textbf{else}\ C(q)\ \textbf{end} \\
C(\textbf{suspend}\ \mathrm{p}\ \textbf{when}\ \mathrm{s}\ \textbf{end}) &= \textbf{suspend}\ C(p)\ \textbf{when}\ s\ \textbf{end} \\
C(\mathrm{p}\ ;\ \mathrm{q}) &= C(p)\ ;\ C(q) \\
C(\textbf{loop}\ \mathrm{p}\ \textbf{end}) &= \textbf{loop}\ C(p)\ \textbf{end} \\
C(\mathrm{p}\ \|\ \mathrm{q}) &= C(p)\ \|\ C(q) \\
C(\textbf{trap}\ \mathrm{T}\ \textbf{in}\ \mathrm{p}\ \textbf{end}) &= \textbf{trap}\ T\ \textbf{in}\ C(p)\ \textbf{end} \\
C(\textbf{exit}\ \mathrm{T}) &= \textbf{exit}\ T \\
C(\textbf{signal}\ \mathrm{s}\ \textbf{in}\ \mathrm{p}\ \textbf{end}) &= \textbf{signal}\ s\ \textbf{in}\ C(p)\ \textbf{end}
\end{aligned}$$

■

The carrier function and the identity function coincide for the subset of Esterel+delay without **delay** statements.

A program may contain a delay d that is not an exact multiple of the given execution period τ. An implementation can either underapproximate by waiting for l ticks, or overapproximate by waiting for u ticks, where

$$l = \max\left(\left\lfloor \frac{d}{\tau} \right\rfloor, 1\right) \tag{6.1}$$

$$u = \left\lceil \frac{d}{\tau} \right\rceil \tag{6.2}$$

The underapproximation l is not allowed to be zero because the replacement statement, **await** l tick, would then be instantaneous, which would drastically alter the meaning of the program and could introduce causality problems. Esterel+delay programs must always stop at **delay** statements.

When a **delay** is repeated, for instance if it occurs within a loop, choosing only one of the approximations gives a program whose actual timing behaviour drifts steadily from the ideal timing behaviour. Such cumulative errors are problematic in certain applications, for example in programs like the sensor driver of Chapter 4 that sample bits asynchronously. One possibility is to track the cumulative drift and, at each iteration, to choose whichever of the two approximations minimises it. This approach is only applicable when $l \cdot \tau < d$.

Some simple calculations show that approximations can be chosen at runtime using only operations on integers and a small amount of constant memory. The difference between a specified delay d and its underapproximation is equal to $d - l \cdot \tau$. Since both d and τ are rationals, this difference can be written as a ratio of two positive integers:

$$\frac{l_n}{l_d} = d - l \cdot \tau, \tag{6.3}$$

where the subscripts n and d stand for 'numerator' and 'denominator' respectively. And similarly for the overapproximation:

$$\frac{u_n}{u_d} = u \cdot \tau - d. \tag{6.4}$$

When a single, iterated **delay** d statement is approximated by m executions of an **await** l tick statement and n executions of an **await** u tick statement, the cumulative drift will be

$$c = m \cdot \frac{l_n}{l_d} - n \cdot \frac{u_n}{u_d}, \tag{6.5}$$

which can be scaled to an integer by multiplying by $l_d \cdot u_d$, giving

$$c \cdot l_d \cdot u_d = m \cdot l_n \cdot u_d - n \cdot u_n \cdot l_d. \tag{6.6}$$

It can be tracked by an integer variable which is incremented, whenever the underapproximation is applied, by

$$d_l = l_n \cdot u_d, \tag{6.7}$$

and decremented, whenever the overapproximation is applied, by

$$d_u = u_n \cdot l_d. \tag{6.8}$$

Any drift due to the approximations is mitigated by making local choices that minimise a tracking variable. This technique is most suitable for delay statements within loops whose values are midway between the lower and upper approximations at a given execution period, that is for d near $l + \frac{\tau}{2}$.

The three variations are combined in the translation function for sample-driven platform statements. It is assumed that each **delay** statement is annotated with one of {under, over, avg} that specify one of the approximations to apply; the annotation will be written as a subscript of the delay statement. The means of making these annotations is immaterial. This extra information can be provided by any convenient means. Annotations could, for instance, be given as per-delay pragmas, or they could be specified globally for an entire program.

Definition 6.4.5
The *sample-driven transformation* $\mathcal{T}_\tau(p)$ maps an Esterel+delay statement p to an Esterel statement. It extends the carrier function to the **delay** statement.

if $d = n \cdot \tau$,

$$\mathcal{T}_\tau(\textbf{delay}_{approx}\ \mathrm{d}) = \textbf{await}\ n\ \mathsf{tick},$$

otherwise,

$$\begin{aligned}\mathcal{T}_\tau(\textbf{delay}_{\mathsf{under}}\ \mathrm{d}) &= \textbf{await}\ l\ \mathsf{tick}, \text{ and}\\ \mathcal{T}_\tau(\textbf{delay}_{\mathsf{over}}\ \mathrm{d}) &= \textbf{await}\ u\ \mathsf{tick},\end{aligned}$$

and, when $d \cdot \tau \geq 1$,

$$\begin{aligned}\mathcal{T}_\tau(\textbf{delay}_{\mathsf{avg}}\ \mathrm{d}) = \ &\textbf{if}\ \mathsf{abs}(\mathsf{diff} + d_l) <= \mathsf{abs}(\mathsf{diff} - d_u)\\ &\quad\textbf{then}\ \mathsf{diff} = \mathsf{diff} + d_l;\\ &\qquad\textbf{await}\ l\ \mathsf{tick}\\ &\quad\textbf{else}\ \mathsf{diff} = \mathsf{diff} - d_u;\\ &\qquad\textbf{await}\ u\ \mathsf{tick}\\ &\textbf{end if},\end{aligned}$$

where the values of l and u for a given d and τ are as previously defined, and the variable name diff is unique within the module and declared as an integer variable. ■

As the avg translations introduce new variables they should be performed before any other source-code transformations, such as loop unrolling, which might otherwise affect the timing behaviour of the resulting system. This brittleness is an unfortunate side-effect of distinguishing the multiple dynamic occurrences of delays that are identified statically.

6.4.2.2 Event-driven with timers

A platform statement of the form $T \in \mathcal{T}$ specifies an event-driven implementation where T is a set of tuples (τ_t, l, u, n) describing available timers. The timers may be provided by hardware or an interface layer. The technique of §6.3.3 is applied: each **delay** e statement is replaced by an **emit** statement that starts an assigned timer and an **await** statement that waits for it to expire.

The transformation must allocate timers, from the multiset given by the platform statement, to **delay** statements in the program while minimising differences between required and actual delays. No single timer may be assigned to two simultaneous delays and all delays must be supported if possible. Issues of signal naming and aborted delays require care but do not present any fundamental problems.

The allocation of timers to delays can be simplified by forming a static over approximation of the original Esterel+delay program.

Definition 6.4.6
A *delay term* is formed from constants in $\mathbb{Q}^{\geq 0}$, and the two binary operators ; and $\|$. ■

Definition 6.4.7
The *delay abstraction* function D maps an Esterel+delay program, where delay expressions have been evaluated, to a delay term:

$$\begin{aligned}
D(\textbf{nothing}) &= 0 \\
D(\textbf{emit}\ s) &= 0 \\
D(\textbf{pause}) &= 0 \\
D(\textbf{delay}\ d) &= d \\
D(\textbf{present}\ s\ \textbf{then}\ p\ \textbf{else}\ q\ \textbf{end}) &= D'(p, q, ;) \\
D(\textbf{suspend}\ p\ \textbf{when}\ s\ \textbf{end}) &= D(p) \\
D(p\ ;\ q) &= D'(p, q, ;) \\
D(\textbf{loop}\ p\ \textbf{end}) &= D(p) \\
D(p\ \|\ q) &= D'(p, q, \|) \\
D(\textbf{trap}\ t\ \textbf{in}\ p\ \textbf{end}) &= D(p) \\
D(\textbf{exit}\ t) &= 0 \\
D(\textbf{signal}\ s\ \textbf{in}\ p\ \textbf{end}) &= D(p)
\end{aligned}$$

where:

$$D'(p, q, \otimes) = \begin{cases} D(q) & \text{if } D(p) = 0 \\ D(p) & \text{if } D(q) = 0 \\ D(p) \otimes D(q) & \text{otherwise.} \end{cases}$$

■

When a program p does not contain any **delay** statements, the delay abstraction function $D(p)$ gives the result 0. Otherwise, a delay term represents a binary tree with two types of internal nodes and leaves in $\mathbb{Q}^{>0}$. The constraints expressed by a delay term are conservative, they do not consider the reachable state-space of the program. A more accurate, but inevitably more expensive, analysis would permit a finer expression of constraints.

As an example, the delay term for the microprinter controller program of Figure 6.3a is: $(0.0024; 0.001667); ((0.00005; 0.0003) \parallel 0.000733)$. Note that the delays in

the branches of the **present** statement are combined with ';' in the delay term; all that matters is that they do not occur simultaneously.

The platform statement $T \in \mathcal{T}$ is a set of timer types. For the purposes of timer allocation, it may be considered a multiset of timer triples (τ_t, l, u). A certain number of timers are necessary to implement a given delay term, even before the closeness of their approximations is considered.

Definition 6.4.8
The *timer count* function T_n gives the number of timers required for a delay term:

$$\begin{aligned} T_n(0) &= 0 \\ T_n(d) &= 1 \\ T_n(d_1\,;d_2) &= \max(T_n(d_1), T_n(d_2)) \\ T_n(d_1 \parallel d_2) &= T_n(d_1) + T_n(d_2) \end{aligned}$$ ■

Two functions are introduced to evaluate the suitability of a particular timer for a particular delay.

Definition 6.4.9
The *timer match* function T_m maps a delay d and a timer (τ_t, l, u) to a rational number:

$$T_m(d, (\tau_t, l, u)) = \min(\max(l, \left\lfloor \frac{d}{\tau_t} \right\rfloor), u)$$ ■

The timer match function gives the closest delay to the ideal delay that is achievable by the timer. The possibility of implementing a delay with multiple successive timer invocations is not considered here, but it could be effected by a 'splitting transformation' on delay terms that breaks delays bigger than a given constant up into sequences of smaller delays.

Definition 6.4.10
The *timer delta* function T_δ maps a delay d and a timer (τ_t, l, u) to a positive rational number:

$$T_\delta(d, (\tau_t, l, u)) = \mid d - T_m(d, (\tau_t, l, u)) \mid$$ ■

The timer delta function is a measure of the suitability of a timer for meeting a delay.

Given a delay term d and a multiset of timers T such that $|T| \geq T_n(d)$, the *clock assignment problem* is to pair each delay in d with a timer from T such that no single timer is assigned to both subterms of any $\parallel$ operator. An optimal assignment is one that minimizes T_{delta} for each pairing.

The clock assignment problem may be solved automatically with standard constraint solving techniques. But since it is likely that engineers would prefer to make some or all of the allocations manually, compilers should provide pragmas for naming delays, and the platform statement should be extended so that timers can be associated with the names. These pragmas would further constrain the set of possible solutions.

In the definition of the transformation with allocated timers, it is assumed that each **delay** d statement is identified by a distinct index $i \in \mathcal{I}$, with which it is annotated, $\textbf{delay}_i\ d$.

Definition 6.4.11
Given an Esterel+delay program p where each **delay** statement is indexed from a set $\mathcal{I}$, and an allocation of timers represented by two functions, $timer_a$ from $\mathcal{I}$ to the name of a timer and $value_a$ from $\mathcal{I}$ to an integer, the *timer-allocated transformation* $\mathcal{T}_a(p)$ extends the carrier function to the **delay** statement:

$$\mathcal{T}_\tau(\textbf{delay}_i\ \mathrm{d}) = \textbf{emit}\ start(timer_a(i))(value_a(i))\ ;\ \textbf{await}\ finish(timer_a(i))\,,$$

where *start* gives the name of the integer-valued output signal that triggers a timer, and *finish* gives the name of the pure input signal emitted by a timer upon expiry. ■

```
delay 3;          +0                                  [·,0]
emit O1;          +3
loop
  emit O2;        +3    +10   +17   ···
  delay 2;        +3    +10   +17   ···   [7,3]
  emit O3;        +5    +12   +19   ···
  delay 5         +5    +12   +19   ···   [7,5]
end loop          +10   +17   +24   ···
```

Figure 6.9: Phase relationships in an Esterel+delay program

An implementation must manage timers properly when corresponding **await** statements are aborted. Two possibilities must be considered. First, a running timer could be aborted and then, in the same reaction, a new countdown could be requested. The interface layer should clear any latches for a timer after it has been restarted. Second, multiple timer requests could be made and aborted within the same reaction. Consider, for example, this fragment where two consecutive delay statements have been transformed to **emit**/**await** pairs that share a timer:

```
weak abort
  emit T1(100); await T1
when S;
emit T1(80); await T1.
```

When the signal S is present, T1 is emitted twice in a single reaction. Special **combine** functions are required to ensure that only the last request is honoured.[4] Such functions must normally be associative and commutative. An exception can be made for allocations against a delay term because timers are only reused for delays in sequence, provided that the compiler respects the sequence of microsteps.

Issues of abortion and timer management are better addressed by the technique of §6.3.4, where each **delay** e statement would be replaced by an **exec** statement that starts an assigned timer and awaits its completion. Unfortunately, the **exec** statement is not always supported by compilers.

The transformation with timers gives Esterel programs that suffer the inadequate interaction of suspension and delay described in §6.4.2.2. Compilers should emit a warning for programs where **delay** statements are subject to suspension.

6.4.2.3 Event-driven with timing inputs

A platform statement of the form $A \in \mathcal{A}$ where A is a set of *timing input pairs* (s, τ_s) specifies an event-driven implementation where each signal s occurs regularly with a period of τ_s relative to system startup. Delays are implemented by counting these timing inputs using the technique described in §6.3.2.

For a delay d, and a signal s with period τ_s, the statement **await** n s gives a physical-time delay t that satisfies $(n-1)\cdot\tau < t \leq n\cdot\tau$.[5] While there is again a choice between lower and upper approximations of the delay, that is between the values l and u given in equations 6.1 and 6.2, the $n-1$ multiplier in the lower bound for t means that the upper approximation is the safer choice; since $u = l + 1$.

The lower approximation may sometimes be more suitable than the upper approximation. It depends on the start time of a particular **delay** statement relative to the period of a given timing input signal. It is possible to statically determine the 'phase relationships' between **delay** statements, relative to system startup, in some Esterel+delay programs. An example is presented in Figure 6.9. Each statement has been labelled with its offset, in ideal time, from system startup. Multiple offsets are given for statements within the loop. In this example, the **delay** statements can be assigned a fixed period and offset. The first **delay** has no period, since it is only executed once, and a

[4] It does not matter if it is also aborted instantaneously because then there would be no statement awaiting the timeout input; it would either be reallocated later or ignored.

[5] The reason for the open lower bound of $(n-1)\cdot\tau$ is explained in §6.3.2.

zero offset. The second and third both have a period of 7, the total delay within the loop body. Their offsets are determined by delays before the loop is entered and also by those within the loop itself.

Phase relationships cannot be determined following **pause** statements or within **suspend** statements when they depend on the presence or absence of inputs whose timing characteristics are not known or not predictable. It would be possible to provide extra information about inputs, like timing offsets for instance, and to include timing inputs that do not occur regularly but may nevertheless only occur at certain times. It would also be possible to propagate known information about emitted signals to other parts of a program; for instance, that a certain signal is always emitted with a certain period and offset. It is not clear, however, how useful all of this would be in practice.

Determining phase relationships for the **present**, **trap**, and parallel constructs is difficult in general. An analysis could insist that both branches in a **present** or parallel construct have the same final offset and period, and similarly for each **exit** within a **trap** as well as for the **trap** body itself, but, again, it is not clear whether this would be especially useful.

An optimal choice of timing input also depends on phase relationships. Without this information, the timing input with the smallest granularity is the best choice because it provides the smallest range for the delay in physical time and the most accurate accounting in the presence of suspension. The selection of a timing input for a given delay may, moreover, affect the phase relationships and hence influence the selection of timing inputs for other delays. It is not clear how best to address this complication.

The most basic transformation always uses the finest timing input and takes the upper approximation.

Definition 6.4.12
Given a platform statement A, the *timing-input transformation* $\mathcal{T}_A(p)$ extends the carrier function to the delay statement:

$$\mathcal{T}_\tau(\textbf{delay}\ \mathrm{d}) = \textbf{await}\ n\ \mathsf{s}\,,$$

where (s, t_s) is chosen from A to minimise τ_s, and $n = \left\lceil \frac{d}{\tau_s} \right\rceil$. ■

More work is required to determine the usefulness and practicability of more sophisticated approaches.

6.4.3 Comparison to related work

The literature contains an abundance of proposals for modelling and implementing real-time systems. In particular, there are several techniques and methodologies for implementing or otherwise discretizing timed automata, like, for instance, the AASAP semantics discussed in §3.5. The focus of this section is, however, on the incorporation of continuous time elements into synchronous languages. Five approaches are especially relevant. Two of them, TAXYS and temporized Argos, are already described in §§6.3.6 and 6.3.5. Some further comparisons are nevertheless made in this section. Two extensions to the Quartz language [BS02, LS02] are outlined here, as is a proposal for validating the real-time constraints of Esterel programs [SA01].

The proposal for Esterel+delay is influenced by the TAXYS [BCP+01, STY03] methodology for building real-time systems with Esterel, but there are important differences. In TAXYS, application logic is specified in logical time and implementations are modelled in continuous time. A satisfaction relation is defined to judge the correctness of the latter against that of the former. It has been argued in this chapter, however, that applications like the microprinter controller are specified most naturally in terms of continuous time and only later transferred to discrete controllers with logical time. The timing annotations of TAXYS express the execution characteristics of a program on a specific platform, and also aspects of its environment, whereas the **delay** statements of Esterel+delay express desired application behaviours; platform limitations are stated separately. The platform models of Esterel+delay are more abstract and

less ambitious than those of TAXYS, where an asynchronous platform with dynamic scheduling is adopted. The relationship between ideal and executable models is more rigorously defined in TAXYS than it is in Esterel+delay.

The temporized version of Argos [JMO93] has both discrete-time and continuous-time semantics. The latter is derived from the former by treating discrete delays, expressed in terms of a distinguished timing input, as delays in terms of a continuous clock. The continuous-time semantics is motivated by and exploited for the automatic verification of quantitative properties. The direction of translation is reversed in Esterel+delay: continuous-time programs are translated into discrete-time programs. The motivation is different too: Esterel+delay aims to support both natural descriptions of certain types of programs and the adjustments required for implementation platform limitations. This latter issue is not addressed by temporized Argos.

Quartz is an Esterel-like language for which real-time verification [LS02] and hybrid systems extensions [BS02] have been proposed.

Quartz programs can be translated into timed Kripke structures to verify quantitative properties [LS02]. Delays are expressed by **pause** statements, essentially as described in §6.3.1. An abstraction statement is introduced to ignore intermediate polling states; for instance, the statement **await** n is not expanded into a sequence of n **pause** statements, but rather treated as a timed transition labelled with n. Quartz is intended for high-level designs before any implementation details have been considered. The translation is based on logical time since *physical time... depends on the hardware chosen for the realization* [LS02, §1]. The proposal for Esterel+delay suggests a different possibility.

The hyperQuartz language [BS02] is an extension of Quartz for modelling hybrid systems. Continuous execution intervals are expressed as lower and upper timing bounds on **pause** statements. The length of an interval may depend on an expression over a global time parameter and other continuous signals. Pure signals are piece-wise continuous over an interval, but hybrid variables may evolve according to given differential equations. It is not clear how multiple constraints are resolved to produce practical implementations. The timing limitations and characteristics of implementations are not discussed. The focus is modelling not programming.

In another proposal [SA01] for validating the real-time behaviour of Esterel programs,[6] locations and blocks of statements are annotated with markers to which timing constraints, that are stated separately, may then refer. For example [SA01, §4.1], this program fragment contains one pair of annotations:

```
%# block_1_begin
Y := 100;
emit S1(Y);
Y := Y + 100;
X := 7;
emit S2(Y)
%# block_1_end.
```

Timing constraints can then be stated relative to an external clock, for example:

```
time(block_1_end) − time(block_1_begin) ≤ 4 units.
```

A program is analyzed by replacing the marker annotations with 'ghost signals', which are observable after compilation to an automaton. The proposed design flow involves two steps. Logical correctness is first established under an assumption of perfect synchrony, then the timing analysis establishes that the constraints are met. There are several differences between this approach and that of Esterel+delay. In Esterel+delay, application timing details are stated within a program in physical time, that is as rational multiples of seconds, rather than as separate annotations in uncertain, discrete units. Platform timing constraints are given separately in Esterel+delay in terms of abstract execution models, whereas in the approach with annotations the form of eventual implementations is unclear, besides that they may be asynchronous and that their signal emissions may take time; no mention is even made of the standard event-driven and sample-driven execution schemes. The timing details of Esterel+delay programs

[6] The paper allows the **exec** statement but not the **suspend** statement.

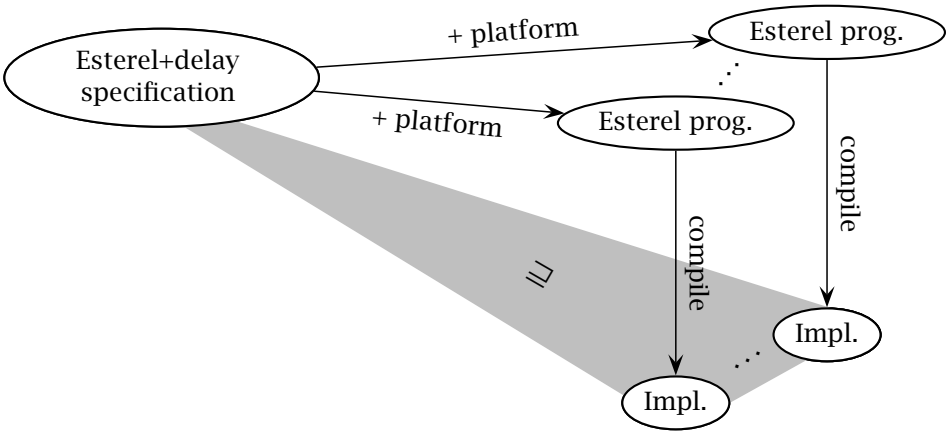


Figure 6.10: Esterel+delay: artifacts and relations

are stated in terms of physical time and later translated into discrete time for implementation. In the approach with annotations, as with other approaches, programs are designed in discrete-time and then validated in physical time.

Many other programming languages allow delays to be specified in terms of physical time – whether by special keywords, or by calls to library functions with either runtime or operating system support. It seems fair to state, however, that in most cases the menaing of these statements is approximate or subject to various special clauses and uncertainties. It is by no means certain how to derive discrete controller implementations with precise behaviour, nor how to describe or judge compromises between ideal behaviour and its approximations on specific platforms. The translation of Esterel+delay to Esterel is distinguished in this regard; it is possible in large part due to the synchronous and precise nature of the latter language.

6.5 Loose ends and further possibilities

The proposed **delay** statement and its interpretation relative to an abstract platform seem to be the right solution for designing and specifying applications like the microprinter controller. But, while the syntactic transformation addresses many issues well and takes advantage of existing tools and technology, it is not completely satisfactory. There are two issues: a lack of tool support and a certain semantic shallowness.

The lack of tool support may be the easier of the two to remedy. There seem to be no obstacles to implementing the transformations described in §§6.4.2.1–6.4.2.3, although the analysis of phase relationships sketched in §6.4.2.3 requires further investigation. Ideally, Esterel+delay would be supported by a simulation tool that combines features of Xes and Esterel Studio with those of Uppaal; rather than requiring repeated clicking through intervals, timing behaviours would be presented and manipulated symbolically. Esterel+delay programs might also be embedded into Simulink; **delay** statements would then be linked to the t parameter of a model.

The semantic issues are more difficult to address. Ideally, the discrete and continuous elements of Esterel+delay could be better integrated with one another. But, in fact, it is not certain that this is even possible without sacrificing some of the essential character and balance of Esterel, and without resorting to intricate formalisations.

6.5.1 Relating specifications and implementations

A semantic treatment should address the comparison of Esterel+delay programs with the implementations generated from them. The basic idea is depicted in Figure 6.10. An Esterel+delay program can be transformed, given different platform statements and parameters, into different Esterel programs, which can themselves be compiled

and executed. A refinement relation could be defined between an original program and its final implementations; much like the correctness property of TAXYS, although in this case, solely in continuous time.

Any such relation would have to allow some 'fuzziness' in the timing behaviour of implementations. The relative closeness of implementations to the original specification could be used to evaluate alternative implementation platforms. A maximum allowable divergence could be factored into verifications of properties against the specification, the results of which would then also apply to a range of implementations. The quantitative relations defined in some recent approaches [BS08a] may offer insights. An alternative approach would be to use a precise relation, but to 'blur' the Esterel+delay specification before applying it.

The equivalence of synchronous program traces, and hence of the programs themselves, is usually based on comparisons of discrete sequences. For Esterel+delay programs, the physical time between inputs and outputs, or between one output and another, may be more significant than the number of reactions between them—especially when nothing happens in the intervening reactions or when they simply count down reactions or discrete timing inputs. These ideas, though, only seem to hold for certain subclasses of Esterel+delay or Esterel programs, as is discussed in the next subsection.

Of course, a semantic model of Esterel+delay programs is requisite to defining relations between them and their eventual implementations. Several extended semantic domains have already been proposed for Esterel, including timed automata [LS02], hybrid automata [BS02], and formulas of the Duration Calculus [PRS95]. One of these may be suitable for defining Esterel+delay and related refinement relations.

6.5.2 Strictly event-driven programs

The **pause** statement is somewhat awkward in event-driven systems. Unlike in sample-driven systems, a precise interpretation in physical time is not possible. The discussion in §6.3.1 suggested, furthermore, that some uses of the **pause** statement sacrifice a form of stutter invariance for Esterel programs.

It may be possible to define a subclass of *strictly event-driven programs*[7] whose observable behaviour relative to inputs and physical time is relatively unaffected by their eventual execution rates; that is, programs that still function as expected after the insertion of extra reactions where all relevant inputs are absent. This would not be a new semantics, but rather a restriction on the existing one. There are several reasons for studying such a subclass. Conceptual issues of triggering would be clarified; such programs would operate most naturally in the event-driven execution scheme. Some aspects of program behaviour should be easier to reason about; for instance, the sort of equivalence and refinement relation described in the previous subsection. The extra assumption could be exploited to give more efficient systems; runtime layers could be more discerning about which parts of a program to execute. Moreover, the difficulties that arise when trying to define the subclass offer insights into the problems of giving a stronger definition of Esterel+delay.

Issues of triggering will be considered for standard Esterel first, and then, afterward, extended to encompass the **delay** statement.

The set of triggering inputs is important to determining whether or not a program statement is strictly event-driven. The subclass would thus be defined at the module level, where both a set of inputs and a program statement are given. A strictly event-driven module is one where observable behaviour, namely outputs, and state changes occur solely in response to explicit input events from the module interface. Consider four example statements given the declaration **input** A, B; **output** O:

```
await A; emit O,
loop pause end,
pause; emit O,
await A; pause; emit O.
```

[7] Some of the ideas sketched in this section were presented at the SYNCHRON workshop in 2006.

The first statement is strictly event-driven, the emission of O is simultaneous with the input A. The second statement, which is equivalent to **halt**, is also strictly event-driven: it does not emit anything and nor does it change state. The third and fourth statements are not strictly event-driven; their output emissions are related, respectively, to system startup and the occurrence of A in terms of logical reactions.

A strictly event-driven module should only respond when at least one input signal is present. The absence of a signal should be judged relative to other input signals and not solely to the occurrence of a reaction. Consider, for example, an event-driven implementation of:

```
module main:
input A;
output O;
await [not A]; emit O.
```

Will O ever be emitted? Or is the program equivalent to **halt**? It depends on whether the module is sometimes executed when A is absent. This module is not strictly event-driven. Given the same interface however, these two statements are strictly event-driven:

```
await [A and not B]; emit O,
await A; present B then emit O end.
```

The absence of B is only tested relative to the presence of A.

Suspension fits nicely with these ideas because it can only further constrain encapsulated statements. The negation implicit in **suspend** is not especially concerning. For instance, the statement

```
suspend
  await A;
  emit O;
when B
```

is equivalent to **await** [A **and not** B]; **emit** O, which is strictly event-driven. Though the statement

```
suspend
  await [not A];
  emit O;
when B,
```

which is equivalent to **await** [**not** A **and not** B], is not strictly event-driven, neither are the statements encapsulated by the **suspend**. In general, **suspend** p **when** e is strictly event-driven if p is strictly event-driven, since the expression e can only further restrict the instants of reaction. The converse is not always true. Consider, for example, enclosing one of the earlier examples of non strictly event-driven statements:

```
suspend
  pause;
  emit O
when [not A].
```

This statement is equivalent to **await** A; **emit** O, which is strictly event-driven. In fact, any statement p of a module with inputs i_1 through i_n can be made strictly event-driven by adding a **suspend** statement:

suspend p **when** [**not** (i_1 **or** $\cdots$ **or** i_n)].

Similar ideas are used to encode multiclock Esterel in standard Esterel [BS01, §4.2], and also to encode asynchronous processes in Esterel [HB02].

Surprisingly, the **sustain** statement does introduce conceptual problems. The expansion of **sustain** O into primitive Esterel gives **loop emit** O; **pause end loop**, which, according to the foregoing discussion is not strictly event-driven—but there is some justification for insisting that it should be. The **sustain** statement connotes an idea of duration, of a signal being held for an interval of time. While the expansion into a looping emit gives this effect within the standard semantics, there is no fundamental reason why a module should be repeatedly polled if it is only sustaining signals. The sampler components of multiclock Esterel [BS01, §2.3] embody this idea somewhat, al-

though they do not distinguish whether a signal was emitted or sustained. There may be justification for treating **sustain** as a primitive statement. Interestingly, a semantics of Esterel in the Duration Calculus [PRS95, §4.1] proposes two possible definitions for output signals, termed 'non-latching' and 'latching'. The same, or a similar, distinction could perhaps be made between the **emit** and **sustain** statements.

There are similarities between the idea of strictly event-driven programs and an earlier semantics for Esterel [BG92]. It seems that 'tick-free' programs in the earlier semantics would have similar properties. The earlier semantics does not treat the **suspend** and **sustain** constructs which, it turns out, complicate the definition of strictly-event driven modules.

The notion of strictly event-driven modules is extended to Esterel+delay by simply insisting that modules are triggered when a delay expires. For example, the statement

```
delay 5s; emit O.
```

would be strictly event-driven, since the output occurs at a specific time relative to initialization of the statement. The combination of strictly event-driven programs and **delay** statements on event-driven platforms may be a useful technique for minimising the number of reactions and maximising accuracy; for doing less computation while reducing lag and jitter.

6.5.3 Suspension

Since the **delay** statement is expanded into statements of standard Esterel, it shares their inadequate interactions with suspension. Consider, for instance, the statement:

```
  suspend
    delay 5s; emit O
  when P
||
  abort
    sustain P
  when Q
||
  delay 2s; emit Q.
```

Three components run in parallel. The first waits for five seconds and then emits O. But, it is suspended initially by the second component that sustains P until the third component emits Q, which it does after two seconds. The current description of Esterel+delay does not define when O would be emitted. There would seem to be two reasonable possibilities.

The first and simplest possibility is that **delay** statements are not affected by suspension. The example program would then emit O after precisely five seconds. Delays implemented by the timer-allocated transformation have this behaviour, while those implemented by the sample-driven or timing-input transformations do not. There are two justifications for this approach, though neither is completely satisfactory. One could argue that physical time cannot be suspended, but why should that limit the expressive possibilities of a programming or modelling language? Stopwatch automata [CL00] indicate that the suspension of continuous time is a powerful technique, albeit one whose analysis is undecidable. The second justification is that since signal emissions are conceptually instantaneous, their occurrence should have no impact on delays which are conceptually intervals of time. But, as mentioned in the previous subsection, the **sustain** statement seems rather to imply a continuous emission, rather than repeated instants of emission.

The second possibility is that **delay** statements should be suspended for intervals of time. The example program would then emit O after precisely seven seconds. The sample-driven and timing-input transformations approximate this behaviour, though neither is perfect. The closeness of the approximation depends on relationships between stated delays and their starting instants relative to sampling or timing input periods. Accuracy could be determined or measured by static analysis, and perhaps factored into the type of quantitative refinement relation suggested in §6.5.1. Alter-

natively either or both **suspend** and **sustain** statements could be encoded with two signals: one when an associated interval begins and another when it ends. This is a classic technique for representing intervals with discrete events. Ultimately, it is not clear whether 'Moore-like' elements can be incorporated into a 'Mealy-like' language like Esterel satisfactorily; that is, whether such a combination would benefit programmers without sacrificing too much of the simplicity of the underlying model and the efficacy of existing compilation and analysis techniques.

6.5.4 Indeterminate delays

In many timed formalisms, a model can specify that an action occurs or may occur within an interval of time, rather than at an explicit time. An informal attempt to model timed automata guards in Esterel+delay programs shows that the latter effectively operate under a form of maximal progress assumption.[8] This suggests the possibility of a statement for expressing indeterminate delays. Two other possible extensions are mentioned briefly.

Consider a transition from a state in a timed automaton labelled with an action I and a guard $2 \leq x \wedge x \leq 3$. Assuming that I is an input and that the clock x is reset when the state is entered, this transition expresses a response to an I action from the environment occurring within two and three units inclusive of the source state having been entered. The same idea is readily expressed in Esterel+delay:

```
delay 2;
weak abort
  await immediate I;
after 3,
```

using the **abort after** macro described in §6.4.2. Removing the **immediate** keyword gives a strict lower bound. Removing the **weak** keyword gives a strict upper bound. Thus the guard $2 < x \wedge x < 3$ is also expressible:

```
delay 2;
abort
  await I;
after 3.
```

The possibility in timed automata of resetting a clock at one state and expressing a guard against it from another can be modelled in Esterel+delay using parallelism and local signals.

Consider now a transition from a state in a timed automaton labelled with an action O and a guard $2 \leq x \wedge x \leq 3$. Assuming again that the clock x is reset when the state is entered, but this time that O is an output, this transition expresses the possibility, but not necessity, of performing an O action within the given interval. An attempt can be made in Esterel+delay:

```
  delay 2s;
  weak abort
    sustain S
  after 3s
||
  await immediate S;
  % delay ?
  present S then emit O end,
```

but the idea of an indeterminate delay cannot be expressed. Instead the signal O would be emitted as soon as possible, as if the program were subject to a form of maximal progress assumption. Inputs may occur non-deterministically in Esterel+delay, but not outputs.

Incidentally, the necessity of action, which is expressed with location invariants in timed automata, does not introduce any special difficulty. For example, the second

[8] Timed automata are described in §2.2.3. Uppaal and TIOA, described respectively in §§2.3 and B.5, are variants of timed automata that make a distinction between inputs and outputs. Guards in timed automata and the maximal progress assumption are discussed in §2.2.3.3.

branch in the above parallel statement could simply be replaced with:

```
abort
  % delay ?
when [not S];
emit O.
```

Another primitive could be introduced to express the indeterminate delays represented by the **delay** ? comments; an unparameterized version of the TAXYS **npause** statement. The precise delay could be chosen statically or dynamically. Adding a new primitive gives a separation between exact timing specification and non-deterministic timing specification,[9] without excluding macro expressions that combine the two; like, for example, **delay** [2 <= x <= 3]. Such a primitive would allow more flexible specifications; more design decisions could be postponed until a platform is chosen. But, it would also add further complications, and it would run contrary to the usual insistence on determinism in synchronous languages. The notion of a 'fuzzy' refinement relation may be better suited to the class of applications for which Esterel+delay is intended.

Two other possible extensions to Esterel+delay that may be worth considering are features for measuring elapsed time and estimating timing parameters. The former makes most sense if the measured values could then be used in calculations and possibly other delay statements. It would potentially make Esterel+delay more expressive than standard finite timed automata, and, in particular, introduce the possibility of Finite Internal Nondeterminism (FIN). Parameter estimation might be a useful technique for solving design problems using a combination of Esterel+delay programs and separately-stated timing constraints. Further investigation is required to determine the usefulness and challenges of these ideas.

6.6 Summary

It has been argued, in this chapter, that while Esterel is ideal for applications with complex sequential behaviour, there is no completely adequate way to express behaviours in physical time. The strengths and weaknesses of Esterel are well demonstrated by the microprinter controller example. The solution proposed is to allow the direct expression of delays in terms of physical time, and then to transform the stated delays according to the limitations of particular implementation platforms. The proposal differs from several others by recommending the expression of abstract designs in physical time with a later transformation to a discrete-time program; similarly to the usual approach for designing and implementing feedback controllers.

The proposal is simple and, it seems, practical, but further work is required to develop a rigorous semantic model for Esterel+delay, and also to define relations between specifications and implementations that account for inaccuracies introduced during translation to specific implementation platforms. Ideally, such a semantic model would assist in the definition of static analysis techniques for transforming Esterel+delay programs, and also provide a satisfactory explanation for Esterel constructs that embody an element of duration, like **suspend** and **sustain**. Ultimately, however, it is not clear whether it is possible to adapt a discrete, synchronous language in this way without sacrificing simplicity, clarity, and practicability.

[9] S. Ramesh recommended such a separation after reading an earlier draft.

Chapter 7

Conclusion

This thesis began with the broad goal of studying and improving the design and implementation of embedded systems, especially their timing aspects, using timed automata and synchronous languages. The specific contributions made towards this goal are summarised and evaluated in this chapter. Some ideas for extensions and improvements are included.

Four specific developments have been presented, their various strengths and weaknesses are discussed in §7.1. From these can be distilled some specific ideas for future research which are listed in §7.2. The chapter, and the thesis itself, ends with some final remarks in §7.3.

7.1 Summary, contributions, and significance

Each of the technical chapters contains a specific and largely self-contained development. Yet the individual results can be summarised and evaluated along four common axes. Each is discussed in a separate subsection. The relationship between the axes and the technical chapters is summarised in Table 7.1.

	Chapter 3 (Simulink)	Chapter 4 (Sensor)	Chapter 5 (Timed traces)	Chapter 6 (Esterel+delay)
Programs in physical time §7.1.1	∗	∗		∗
Real embedded controllers §7.1.2		∗		∗
Verification of timed models §7.1.3		∗	∗	
Implementations §7.1.4	∗		∗	

Table 7.1: Relations between the four common axes and the technical chapters

7.1.1 Programs in physical time

Embedded controllers interact in physical time with their environments. Two aspects of their design must be balanced: the timing requirements of the application and the timing characteristics of implementation platforms. There are good reasons for separating these two aspects: to reduce complexity, to simplify analysis, to support various platforms, and to make informed compromises between requirements and constraints. But eventually, and usually sooner rather than later in many embedded controller designs, they must be interweaved.

A general approach of this thesis is to transform abstract programs into implementation models while accounting for timing constraints. In the synchronous execution model of §3.2, abstract programs expressed as BMMs in discrete time are transformed into timed automata whose behaviour in continuous time accounts for implementation inevitabilities. In the assembly driver of §4.5, application requirements expressed in a timed automaton model are met by exploiting concrete timing characteristics of the implementation language. The transformation in this case is from a semantics in terms

of cycles per instruction to a timed automaton model incorporating the physical period of a cycle. In the proposal for Esterel+delay of §6.4, application requirements are expressed in terms of physical time and programs are transformed into a form suitable for discrete execution. The transformation requires the incorporation of timing aspects of eventual implementation platforms. Each of the three approaches is now discussed in turn.

Implementation characteristics in the synchronous execution model of §3.2 are expressed through three parameters: the execution mode, that is whether sample-driven or event-driven, the execution period τ, and the output lag δ_{out}. The mode and period parameters are necessary to relate logical instants to real time, which is essential to the Simulink embedding. The inclusion of both parameters in the Esterel+delay transformation confirms their usefulness. The output lag parameter is, however, less compelling. The simulation results do not seem to indicate any advantage and such fine details seem too brittle to be exploited in designs. From a control theory perspective, other implementation imperfections seem to be more important; for example, the lag between event occurrence and detection, which is modelled in other approaches [DWDR04].

The output lag parameter attempts to quantify the disparity between an implementation and the ideal of perfect synchrony. While no implementation is perfectly synchronous, it seems that the details of the imperfection are less important than whether or not the assumption of synchrony is warranted in the first place. This depends on the behaviour of the environment and on aspects of the implementation of the program, for example like those addressed in TAXYS [BCP+01, STY03] and in recent work on the WCET analysis of Esterel [BTH07, JHRC08]. Despite these facts, the δ_{out} parameter does capture some aspects of non-zero execution times, most notably the separation of input and output events in physical time. An alternative approach is suggested by Item 6 of §7.2.

The three parameter transformation makes details of the interface layer of a synchronous system explicit. Although these are not usually studied directly, there is some acceptance of their importance and intricacy [AMP91, AP93, BHHS93]. The model presented in this thesis formalises some of the details, notably, the necessity of latching new inputs between reactions and the concomitant complications for event-driven execution. The necessity of double buffering inputs is not modelled, although it is addressed indirectly in the implementation of the synchronous block. While this feature is not important in Chapter 3, other research indicates its significance in certain systems [STC06].

The broader theme of programs in physical time is also manifest in the assembly driver of §4.5. The semantics of the assembly language used are defined in terms of an abstract model of time, namely the number of cycles per instruction. This model is transformed into one in continuous time by accounting for details of the implementation, namely the cycle period. The mapping could have been made more detailed by explicitly modelling the cycle clock and its inaccuracies. The chosen processor effectively has a tractable semantics in terms of physical time, and engineers exploit these semantic properties, which arise necessarily from the nature of digital computing, to implement application timing requirements: either carefully tuning programs to meet upper timing bounds, or introducing delays to meet lower timing bounds. The model addresses a simple microprocessor, the MCS51 and it is not clear what can be inferred about the possibility of extending the results to more sophisticated devices. This is left to future work, see Item 9 of §7.2.

The proposal for Esterel+delay of §6.4 is a response to problems identified by the microprinter controller example and a survey of techniques for expressing delays in Esterel. It builds on ideas explored in the synchronous execution and assembler models, as well as on others from the research literature. The issue of what an Esterel+delay program means is addressed by a series of implementation-driven transformations into Esterel. The presented solution is pragmatic but incomplete.

Several problems are identified in the survey of delays, but the most critical, from the perspective of applications like the microprinter controller, is that expressing a timing delay requires early resolution of implementation issues. Furthermore, implemen-

tation choices that affect timing behaviour are not made explicit in the program text; at best they are stated as comments or incorporated into signal names. Esterel+delay extends Esterel with a statement for expressing delays in physical time and independently of implementation details. This deviation from the principle of multiform time is justified by the need to flexibly treat both aspects of time: application requirements and implementation constraints.

The proposal draws inspiration from other similar approaches, most notably those of TAXYS [BCP+01, STY03] and temporized Argos [JMO93]. A TAXYS program is specified in discrete time and refined to an implementation in continuous time. A temporized Argos program is given an interpretation in discrete time and another in continuous time. In Esterel+delay, by contrast, application timing behaviours are expressed in physical time and then refined to discrete delays for specific platforms; much as digital feedback controllers are designed and implemented.

The transformation to discrete programs exploits the assumption of synchrony and the various techniques for realising delays. It has four novel features: an idealised representation of implementation platforms, a simple feedback mechanism for approximating iterated delays, the statement of the timer allocation problem, and a sketch of a static analysis problem for determining phase relationships between timing inputs and program execution. Work remains, particularly regarding phase relationships in Esterel+delay programs; see Item 4 of §7.2. The transformations are a practical approach for applying Esterel+delay to systems like the microcontroller printer, but a principled semantic treatment is lacking, see Item 2 of §7.2, as is a formal relation between a program in Esterel+delay and corresponding implementations in Esterel, see Item 3 of §7.2.

Throughout this thesis, transformations between models are applied to reconcile application timing specifications with implementation platform characteristics. The possibility of generalising this approach is raised in Item 1 of §7.2.

7.1.2 Real embedded controllers

It is difficult to artificially concoct the problems faced when designing embedded controllers and futile to ignore them. The domain is characterised by intricate requirements arising from the physical necessities and limitations of the hardware to be controlled and the hardware that hosts controllers. While this thesis includes streamlined examples, namely the fuel, temperature, and railway controllers, special effort is made to describe and address more realistic and idiosyncratic examples, namely the infrared sensor of Chapter 4 and the microprinter of Chapter 6.

The infrared sensor case study presents a model of a realistic, if simple, embedded component at the level of the detail required by, and from the perspective of, an embedded systems engineer. Few compromises are made and all modelling choices are discussed. The model is an accurate and useful reference for this particular sensor. More importantly, however, the approach demonstrates the strengths and weaknesses of the notation, viz. timed automata, and the techniques, viz. model checking and timed trace inclusion, for studying a particular man-made artifact. The sequencing and timing constraints are readily expressed using timed automata, but less so the partial orderings of concurrent events, which are anyway limited in this particular example.

It turns out that there is a surprising amount of detail and ambiguity in the seemingly uncomplicated timing diagram. Chapter 4 shows that there is more to faithfully modelling the timing diagram than formalising its notation or re-expressing it in a formal notation.[1] Rather the meaning of the timing diagram must be interpreted against a background of engineering convention and practice. Furthermore, the modelling notation must be carefully applied to express these interpretations.

The idiosyncrasies of the sensor are simultaneously important and limiting. Important because this thesis aims to treat such engineering challenges as they are, rather than as they might be. But, also limiting because the same features may not occur in other components, and thus addressing them solves a particular, rather than a general problem. And, more importantly, features that are important in other components

[1] Appendix F demonstrates the possibility of re-expression in a formal timing diagram notation.

may not be represented. The problem of omitted features might be addressed in two ways: deductively, by proposing a modelling language that can express all possible components and features thought to be of interest and then studying it in detail, or, inductively, by studying more and more components. The deductive approach may be the more interesting from a theoretical perspective, but it risks oversimplification and irrelevancy. The inductive approach is surely more pragmatic. In any case, detailed instances are a necessary precursor for more general proposals.

The models presented in the sensor case study are complete in themselves, but there are many other styles of driver implementation that could be modelled and verified, see Item 9 of §7.2.

The microprinter controller is a novel example of a realistic embedded application, even though it is not treated as thoroughly as the infrared sensor (further possibilities are considered in Item 8 of §7.2). The extract that is presented demonstrates the timing characteristics of most importance in this thesis. With reference to §2.2.1, time is not solely exploited:

1. for error detection, as are time-outs in distributed systems [Lyn96, Chapters 23–25][Tel00, Chapters 12 and 15];
2. to constrain the reachable state-space, as in Fischer's algorithm [AL94];
3. for multiplexing, as in operating systems or on communication channels;
4. to increase simplicity, reliability, and predictability, as in timed-triggered systems [KB03, Kop97, Pon01]; or,
5. to state non-functional requirements, for example, acceptable response times.

Rather time is integral to the purpose of the microprinter controller. The program of Figure 6.3 means little without its precise timing details. Furthermore, the timing constants in the microprinter controller are most naturally expressed in terms of physical time; both in the original specification and in controller programs.

Broadly speaking, the mass of interrelated timing details that arise from devices like the microprinter are handled in three stages. First, relevant details must be extracted from specifications (which may be incomplete), application notes, and background knowledge. Second, these must be coerced into coherent, task-directed, and maintainable forms, which may take the forms of specifications, models, sketches, or programs. Third, the design must be implemented on resource-constrained hardware. The first stage was addressed prior to presenting the microprinter controller example, but the last two stages are discussed in some detail. It is argued that high-level designs and low-level implementation issues are too often intermingled, even in a more abstract language like Esterel. The microprinter controller example well demonstrates this essential challenge.

7.1.3 Verification of timed models

Verifying reactive systems is particularly important because they often involve complex sequential logic that is difficult to program correctly, and they also often fulfill functions whose failure can be expensive or dangerous. Adding intricate timing requirements only increases the difficulty of writing correct programs. Furthermore, such requirements are common in systems that act in the physical world where failures can cause damage to objects and harm to individuals. Verification by model checking in Uppaal is applied throughout Chapter 4 to the various sensor models, and a technique for verifying timed trace inclusion is extended in Chapter 5.

The sensor timing diagram model itself is not verified, but it is taken as a specification for the split models, which offer a different perspective on the relationship between sensor and driver. The distinction between inputs and outputs and the requirement of input enabled components is an important feature of the split models, and natural for the scenario being modelled. Two techniques for input enabling are presented: self-loops and broadcast channels. The broadcast technique is easier to

apply and the resulting models are less cluttered but, due to limitations in Uppaal, the self-loop technique is sometimes necessary.

Each of the split models is verified against the timing diagram model using a construction for testing timed trace inclusion in Uppaal. The approach is slightly unusual because it concerns the set of traces that occur between two components rather than the behaviour of a single component. Auxiliary channel constructions are required to verify both the self-loop split model and the broadcast split model. The latter would be simpler to verify if Uppaal allowed clock guards on broadcast receivers.

Data transmission in the split models is verified through reachability analysis of a property expressed as an observer automaton. The ability to express functions in Uppaal is invaluable for translating back and forth between data values and their representation as strings of bits.

The driver component in the split model is treated as a specification for the assembler implementation model. The timed trace inclusion testing technique is again applicable, but the restriction on clock guards in broadcast receivers means that only the self-loop model is readily useful. The verification would be more difficult if the driver model were less deterministic. The implication of the two timed trace inclusion verifications is that the assembly language implementation correctly implements the sensor timing diagram. While this approach is not proposed as a general methodology for verifying assembly language programs against timing diagrams, it has a certain intrinsic interest and it demonstrates well some of the associated pros and cons.

The interdependencies between the sensor and driver components in the split models, and the possibilities and limitations of using each in isolation is an aspect that could be explored in more depth.

The second verification tract considers the timed trace inclusion testing construction that was applied to the sensor case study. The basic construction is well known. It has been applied to the IEEE 1394 Root Contention Protocol [Sto02, §7.5], adapted to handle some non-deterministic specifications [Sto02, Appendix A], and extended to handle urgent channels and shared variables [JLS00].

It was discovered, while attempting to implement the basic construction, that extra manipulations are needed for some of the more recent Uppaal features, namely selection bindings, quantifiers, and channel arrays. These features are not usually represented in semantic definitions [BW04, BV08], presumably because they serve only to make the modelling language more convenient and not fundamentally more expressive. A semantic model in terms of processes, which are expanded to standard timed automata, is proposed to describe the extra features and their manipulation.

The basic timed trace inclusion construction is extended to include the additional features. Broadly speaking, two particular problems are encountered: meeting syntactic restrictions on transition guards, and clustering transitions that synchronize through channel arrays.

Transition guards are problematic because they are not closed under negation. In the absence of selection bindings and quantifiers, the negation of a conjunction of transition guards can be manipulated into Disjunctive Normal Form (DNF) and then split across multiple valid transitions. While selection bindings can be represented as existential quantifiers, their negation introduces universal quantifiers, which can inhibit this disjunct splitting. A looping construction is proposed to handle difficult expressions. Universal quantifiers alone are readily negated into existential quantifiers and represented by selection bindings—they seek out counter-examples to the original expression—which, due to their disjunctive character, can be split across multiple transitions. But, in combination with selection bindings, negated expressions will, in general, have the form $\forall \cdots \forall \exists \cdots \exists . e$, which cannot be represented directly in Uppaal. Two complementary solutions are proposed: a predicate that determines when the quantifiers can be swapped, and an enhanced looping construction.

Channels are grouped in the testing construction by name and direction. Grouping transitions on channel arrays is more challenging because the potential values of array subscript expressions must be considered. An initial solution that characterises possible groupings by predicates over those expressions works for some cases and offers insights into the problem, but it is difficult to generalise. A different solution based on

the introduction of 'sweep bindings' turns out to be more widely applicable.

The extended technique is demonstrated on a modified version of the railway controller example [WPD94, LS85]. Reachability analysis in Uppaal demonstrates that the timed traces of the original controller are a subset of those of the new controller.

7.1.4 Implementations

Two tools were implemented as part of the work described in this thesis: software for compiling (textual) Argos programs and incorporating them in Simulink models, and a tool for performing the extended validation construction on Uppaal models. Besides resulting in useful tools, these implementations were invaluable in coming to understand, develop, and evaluate the various ideas. Regrettably, there is not yet any tool support for Esterel+delay, some future possibilities are outlined in Item 4 of §7.2.

The tool-chain developed to support the Argos block is adequate for prototyping, but not for realistic applications. In hindsight, it would have been better to integrate the Argos compiler with the Lustre tool chain and to interface the latter's executable format into the synchronous block s-function. Furthermore, the simple graphical syntax of Argos is one of its strengths, but the prototype tools do not include a graphical editor or animator; it seems likely that the time it would have taken to develop such tools would not have been rewarded commensurately with relevant research insights.

The two case studies demonstrate that while Argos is a suitable language for expressing some discrete controllers in Simulink, more sophisticated control logic would probably be better expressed in Esterel or its graphical counterpart, SSM.

The Simulink s-function would have been more difficult to implement without the precision and clarity provided by the synchronous execution model. A variety of mechanisms—zero-crossing functions, inherited port-based sample times, and block-based sample-times—are employed to realise the required timing behaviour. This suggests that timed automata may be useful more generally for designing and implementing behaviours in Simulink. This possibility is further discussed in Item 7 of §7.2.

There do not seem to be any other tools for producing the timed trace inclusion validation construction—with or without extensions. The implementation described in §5.4 makes the technique much more practicable, particularly since, as was found while developing the sensor case study, timed trace inclusion testing reveals flaws in models, through counter-example traces, which must then be fixed before renewed testing. Such iterations are much faster and more accurate when partially automated.

The implementation directly reads and writes Uppaal XML models. It includes a simple expression language for describing manipulations, and newly generated models are formatted using Graphviz. This last step is not just for show; legible output allows the accuracy of the results to be examined, and makes animated counter-example traces in Uppaal easier to follow.

The implementation includes a parser and algebraic data type for Uppaal models, as well as several auxiliary routines. They are all publicly available and suitable for adaptation and reuse. The basic infrastructure has already been adapted to generate draft models from MCS51 assembler. While not perfect, this feature was nevertheless useful in creating the assembly language driver model.

The internal complexity of the tool is troubling. Subtle faults are hard to avoid in such intricate symbol manipulation programs. Some ideas for addressing this issue are discussed in Item 5 of §7.2.

7.2 Limitations and future work

1. Semantic transformations with platform timing details.

Three of the four technical chapters in this thesis—Chapters 3, 4, and 6—apply transformations that introduce implementation timing details. It seems possible to study this idea more generally: to determine which features are important, and how they differ between models for verification and those for implementation; to identify the various trade-offs; and, to determine the role of approximation and the kind of

'fuzziness' that engineers expect and exploit. More examples may be required before general principles become clear.

2. Semantics for Esterel+delay.

A semantic model is not defined for Esterel+delay. Any proposal should seek to provide a tractable and fundamental model, rather than just an impenetrable list of rules. Furthermore, the motivation should not be to produce another way of expressing timed or hybrid automata, or some other denotation, but rather to solve the design and analysis problems that are of importance to engineers.

One of the central challenges in giving a semantics to Esterel+delay is to address the interaction of instantaneous features with those that express or imply intervals, while not sacrificing the simplicity and other strengths of synchronous languages. Specifically, to find solutions to the interaction of **suspend** and **delay**—which may mean treating **sustain** differently to **emit**—and to make better sense of **pause** statements in event-driven programs. Some initial observations are made in §§6.5.2 and 6.5.3.

A semantic definition is an essential first step to simulating and analysing Esterel+delay programs. Ideally, direct verifications of these programs could guarantee something about their discrete implementations, or at least those within a certain range of accuracy. A definition is a prerequisite for defining the relation between ideal programs and actual implementations that is discussed in Item 3 of §7.2.

There exist proposals where programs in Esterel or a derived language are interpreted as timed automata [SA01], as hybrid automata [LS02], or as formulas in the duration calculus [BS02]. Related perspectives are found in other language proposals, like, for instance, Hybrid Statecharts [MMP91]. It is possible that Esterel+delay could be treated similarly, but more investigation is required. It is not ultimately certain, however, that a satisfying semantics can be devised.

3. Relations between programs and implementations.

The sort of relation that should exist between programs in Esterel+delay and their realisations for specific platforms in Esterel is outlined in §6.5.1. Definitions of such implementation relations and of the semantic model suggested in Item 2 of §7.2 would ideally be developed together.[2]

A rigorously defined relation would provide a basis for justifying the transformations to Esterel. It would have to account for the fact that the transformations give different results for different platforms, and that delays in the original program may only be approximated in implementations. This might be done by first 'loosening' parameters in Esterel+delay programs, or by allowing some 'fuzziness' in the relation itself; the two approaches are likely to be equivalent. Engineers routinely make such compromises, but the best way of supporting them formally is not clear; although recent work [BS08a] may help. Further investigation into both practical and theoretical aspects is required.

4. Static analysis for phase relationships in Esterel.

There are advantages to implementing delays by counting timing inputs. But one of the problems is that timing inputs are relative to system startup whereas delays are relative to the reaction of their commencement. As discussed in §6.4.2.3, this limits the effectiveness of transformations from Esterel+delay. It also makes it difficult to estimate inaccuracies. Static analysis could be applied to detect phase relations amongst timing inputs and statements, and the results applied to improve program program program transformation and estimation. There would certainly be limits, but certain common cases could surely be detected.

Static analysis could also enable new compilation techniques for standard Esterel programs. For instance, certain loops or subprograms could be assigned to interrupt handlers, others could be executed at different frequencies.

5. Uppaal transformations in Isabelle.

Both the extended construction for testing timed trace inclusion and its implementation in SML are intricate. Confidence in their correctness is especially important

[2] The intimate relationship between preorders and semantics is emphasized in Appendix C.

because they are proposed for use in verification. It could be increased by applying theorem proving techniques. The Isabelle theorem prover [NPW02] is a suitable candidate because it is also written in SML, its architecture supports programmatic extension [WW07], and the essential soundness of its proof kernel cannot be undermined, regardless of the complexity of an extension.

Ideally, the parser and data types of the implementation could be integrated with Isabelle. At a minimum, Isabelle's sophisticated and extensible term rewriter could be exploited to improve the manipulation of expressions, and the tool could prompt for interactive assistance if needed. A more sophisticated approach would be to develop one of the existing semantic models of Uppaal [BW04, BV08] and the extensions of Chapter 5 in Isabelle, and to use them as a basis for validating the output produced by the tool. For instance, each step of the transformation could generate a justification of its correctness such that chaining a sequence of justifications together would confirm the correctness of the whole.

6. Evaluating the assumption of synchrony in Simulink models.

The Argos block for Simulink enables deviations from perfect synchrony to be accounted for in simulations. In principle, this aids evaluations of whether the assumption of synchrony is valid for a specific implementation in a specific environment. An alternative would be to develop techniques and tools that determine whether the assumption is valid by examining the program, its implementation parameters, and the model of the environment. Such an approach could begin by adapting or extending existing semantic models [TSR03, ASK04] for continuous Simulink blocks.

7. A Simulink s-function for timed automata.

The synchronous execution model acts as a specification for the Argos block, which is implemented as a Simulink s-function. The s-function effectively implements a particular type of timed automaton using various Simulink features. This suggests the possibility of tools for embedding general timed automata models in Simulink. They could either allow the expression and simulation of timed automata directly, or they could facilitate the development of higher-level tools, like the Argos block. It is not clear which of the three options listed in §3.3.1 for embedding semantic models into Simulink would best support such an approach.

8. Complete the microprinter controller example.

The microprinter controller example of Chapter 6 is incomplete. The presentation and solution could be extended to include including serial communications of characters to print, the lookup of bitmaps in memory, and the transmission of pixels to the print head. It would be interesting to formalise the requirements from the specification relating to timing behaviour, temperature ranges, and exceptional conditions, and then to verify that they are met by implementations. This may be one way to combine the proposal for Esterel+delay with the techniques applied to the sensor case study.

9. More drivers for the infrared sensor case study.

Only a single assembly language driver was validated against the sensor timing diagram. It would be interesting to model and validate drivers that apply different techniques, for instance timers or OS features, and that are written in different languages, for instance in Esterel, C, or the assembly language of a more sophisticated platform. While the techniques developed in this thesis would be applicable to an Esterel driver, it seems likely that more research would be needed to treat timing in C programs, and also for more advanced assembler programs. In particular, it is not clear how to adapt the type of architecture-informed transformation that was used for the MCS51 assembler.

7.3 Final remarks

Programming embedded controllers is difficult. Intricate timing details, which arise from both the physical characteristics of the applications themselves and the resource

constrained platforms on which controllers are implemented, exacerbate the challenge of designing and expressing complex reactive behaviours. Synchronous languages like Esterel and Argos alleviate some of the problems, but more support could be provided for the compromises required to realise abstract designs efficiently on diverse platforms. This support could take the form of parameterised transformations between models that account for the timing characteristics of implementation platforms. Furthermore, while the synchronous languages benefit greatly from their discrete time foundation, the physical timing properties in hardware specifications and in models of implementations are often better expressed in continuous time. Timed automata are well-suited for expressing both of these kinds of timing details, with the distinct advantage that many properties, including implementation relations, can be analyzed automatically.

Appendix A

Process Algebra

Transition systems are sufficient for modelling and analyzing the fundamentals of discrete systems, but doing so for large systems is impracticable. The structural patterns important for understanding and reasoning cannot be seen for the masses of detail. Large systems are built, rather, from smaller components, and behaviours of the whole emerge from those of the interacting parts. Process algebra focuses on the composition and interaction of components.

In process algebra, as for the class of reactive systems, components operate concurrently with one another and their environment, and the actions performed over time are more important than any final result. In fact, the systems of interest must usually operate indefinitely and interact continually with their environments, which is in contrast to the study of algorithmic programming languages where termination is desired, and there are only two points of external interaction: initial acceptance of inputs and final delivery of results. Many of the semantic techniques, particularly notions of implementation and equivalence, developed for process algebras find application in the design of embedded systems.

The three main process algebra 'schools' are Communicating Sequential Processes (CSP) [Hoa85], Calculus of Communicating Systems (CCS) [Mil89], and the Algebra of Communicating Processes (ACP) [BW90].

CSP was originally intended as a concurrent programming language [Hoa78]. Theoretical issues later became more central, but models of real systems remain important. Both the Occam programming language, and the Language Of Temporal Ordering Specification (LOTOS) [BB87] language are derived from CSP.

CCS was developed to study concurrent and communicating agents. It draws inspiration from the Lambda calculus of sequential processes [Mil93]. CCS has a less operators, and is perhaps more mathematically ideal, than CSP. The pi-calculus extends CCS which features for naming processes and tracking evolving connections.

ACP is a series of process algebras. Each is described an equational system that axiomatizes the intuitions about a class of processes. The primacy of equations characterizes the mathematical approach of ACP. μCRL and mCRL2 are versions of ACP with abstract data types and tool support for verification and efficient execution.

The common features of most relevance are a mathematical syntax, a focus on semantic models, interleaved concurrency, synchronized communications, unobservable actions, choice operators, and treatments of refinement and equivalence. Each is described in turn over the next few sections, which do not aim for a rigorous and complete comparison, as may be found elsewhere [Gla97], but rather to summarise and exemplify the principles most relevant to later sections.

A.1 Mathematical syntax

The processes of a process algebra are expressed primarily as mathematical expressions, tree structures written linearly and formed of bound variables, constants, and function symbols. While perhaps forbidding at first, the notation has several advantages: it is convenient for pen and paper derivations; there is a one-to-one relation

CSP: $X = a \rightarrow (b \rightarrow STOP) \,\Box\, c \rightarrow Y$
$Y = d \rightarrow e \rightarrow Y$

CCS: $a.b.\mathbf{0} + c.\mathbf{fix}(Y = d.e.Y)$

ACP: $\mathsf{X} = \mathsf{a}\cdot\mathsf{b}\cdot\delta + \mathsf{c}\cdot\mathsf{Y}$
$\mathsf{Y} = \mathsf{d}\cdot\mathsf{e}\cdot\mathsf{Y}$

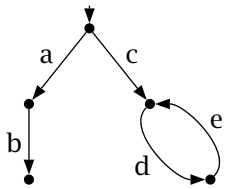


Figure A.1: Three process expressions and a corresponding process graph

between the syntax and operators on models; laws or propositions are expressed naturally as equations; semantic rules can be written compactly, usually entirely on a single page; and, there are fewer obstacles to writing rules and derivations precisely.

Three examples and a graphical representation of an associated process graph are shown in Figure A.1. The examples demonstrate four primary function symbols, those for *dynamic* operators [Mil89], and their interpretation on process graphs:

- action prefixing, $\rightarrow$, $a.$, $\mathsf{a}\cdot$, for transitions,
- deadlock constant, *STOP*, $\mathbf{0}$, δ, for a state[1] with no outgoing
- choice, $\Box$, $+$, for branching transitions from a state,
- recursion for transitions that loop back into other states.

These four concepts suffice to express all finite process graphs, that is those with a finite number of states and finite branching from each. The first three concepts alone can express the subset of graphs that are also trees. These facts can be exploited for reasoning about process graphs; the *head normal form* [BW90, Definition 2.4.6] and related lemmas of ACP are good examples.

An indexed choice operator is required for process graphs with infinitary branching. Infinitary branching encompasses *image finite* processes, where an infinite number of actions are possible but where there are only a finite number of destination states for each, and *infinitely branching* processes where there may be an infinite number of destination states for an individual action. Mutual recursion with infinitely many variables is required to express infinite state process graphs.

Variables, that stand for processes, are used in expressions for three main reasons: to break complicated definitions into smaller parts, to describe recursive processes, and to stand for arbitrary or unknown processes. The variables in the first two cases are bound, explicitly or otherwise, by the process specification. In the last case they are free variables. Only concrete terms,[2] those where no variables are free, define processes. But, in informal use, unspecified behaviours are often represented by variables. Similarly, (infinitely) repeated behaviours may be written informally with ellipses.

A.2 Unobservable actions

There are various reasons for localizing some of the actions of a subsystem: to prevent them from influencing other subsystems; to hide the internal operations of a system from its environment; and, to abstract from certain actions when analyzing or comparing processes. Two distinct solutions for this task have arisen. In CCS and ACP, and sometimes CSP [Ros97], a special action τ is distinguished to represent an internal step, which can neither be seen nor controlled by the environment (control is the ability to force or block). In CSP, and sometimes CCS [DNH87], an internal choice operator represents possibilities within a process that cannot be influenced by an environment.

In the process graph model, τ is just another action relation $\xrightarrow{\tau}$, but it is treated specially in semantic rules, where it occurs without synchronization, and in traces,

[1] The term 'process' is more usual, but 'state', used synonymously, emphasizes the link with transition systems.

[2] Also called 'closed process expressions'

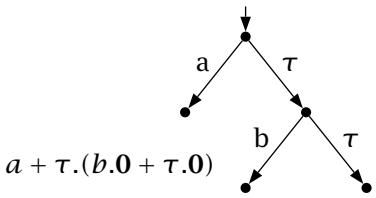


Figure A.2: Example for discussing the effect of τ transitions

where it is normally omitted, and also in those equivalence and refinement relations that respect abstraction. The apparent simplicity of τ-actions belies their expressive power and subtle semantic effect. Besides explicit prefixing, τ-actions are introduced by completed communications in CCS, and the *hiding* operator of ACP. Any operation that introduces τ-actions can introduce non-determinism.

Internal steps have a preemptive power [Mil89, §2.3] to change a process's possibilities for action. For example, the process of Figure A.2 is initially willing to perform an a action, but it may take an internal step that spontaneously removes that possibility. There is no guarantee that an a will occur, even in an environment where another process wishes to synchronize with it. If there are no other processes willing to synchronize on a, the internal step will inevitably occur, under the usual assumption of progress, giving a new process $b.\mathbf{0} + \tau.\mathbf{0}$, where a similar choice exists between b and an internal step.

It is somewhat awkward that interaction possibilities may change spontaneously in this way. It complicates thinking about process behaviours, for example, thought experiments based on testing [Gla93]. For example, in Figure A.2, the ability to perform a b may appear at some point and disappear at another. In asynchronous composition, both the possibility and non-possibility of synchronization with b must be considered; that is, neither internal nor external actions may take priority over the other.[3]

A change in action possibilities could be interpreted as an observable event, though this is contrary to the idea that τ actions are unobservable. Standard process operators are not sensitive to the removal of action possibilities, but operators able to detect it, for example a 'polling' operator [Blo94], can be defined through SOS rules with negative premises.

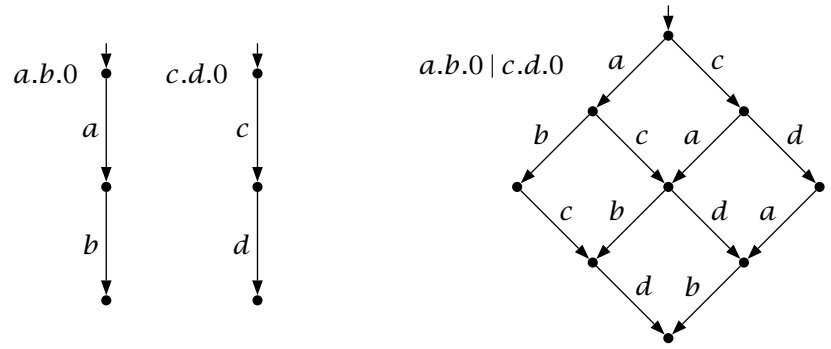
An alternative to τ-steps is to distinguish process branchings where the environment decides which path is taken from those where the process alone decides; operators for expressing the distinction are discussed in §A.5. CSP is usually defined in this way, and some versions of CCS are too CCS [DNH87, Hen88].

Although, CSP was originally presented [Hoa85] without τ-steps, they can be introduced in models of CSP expressions. The key idea is that an internal choice between behaving like P or behaving like Q, written $P \sqcap Q$ is essentially the same as a non-deterministic choice between two τ-steps, one leading to P and the other to Q, that is: $\tau.P + \tau.Q$. This idea and the mutual distributivity of the $\sqcap$ operator with the operator for external choice $\Box$ [Hoa85, §3.3.1] limits the range of process graphs that are considered,[4] which, in turn, gives insight into the treatment of τ-steps in the CSP-approach: they occur immediately whenever possible before the environment has a chance to act, and loops or cycles of τ-steps are considered as pathological.

In versions of CCS without τ-steps, the external and internal choice operators are usually written as $+$ and $\oplus$, respectively [Hen88]. The operational semantics is presented using 'invisible moves' [DNH87] which, unlike τ-steps, have no preemptive power; that is, invisible moves can remove possibilities offered by internal choice, but they must preserve those offered through external choice. All internal choices must be resolved before any external possibilities are presented; synchronisations cannot suddenly be offered to or retracted from the environment. All CCS terms can be translated into the τ-free calculus; testing equivalence, but not weak bisimulation equivalence, is preserved [DNH87].

[3] In CCS, for instance, $((a + \tau.(b.\mathbf{0} + \tau.\mathbf{0})) \mid (\bar{b}.d.\mathbf{0}))\backslash\{a, b\}$ and $\tau.(\tau.d.\mathbf{0} + \tau.\mathbf{0})$ are equivalent.

[4] Graphs are rewritten so internal and external outgoing transitions are not mixed at any node [Hoa85, §3.5.4].



(a) individual processes

(b) Parallel composition

Figure A.3: CCS expressions and corresponding LTSs

A.3 Interleaved concurrency

The four dynamic operators presented in §A.1 are adequate for reasoning about individual sequential processes. But many systems comprise multiple components acting individually and concurrently, especially reactive systems where, at the very least, the environment is considered as a concurrent entity. Thus process algebras invariably have one or more operators for *parallel composition*. The parallel composition of processes P and Q, is written variously:

CSP	$P \mid\mid\mid Q$	*interleaving*
	$P \parallel Q$	*lock-step*
CCS	$P \mid Q$	*composition*
ACP	$\mathsf{P} \parallel \mathsf{Q}$	*merge*

The operators differ in the details of synchronization, which is discussed in the next section, but all are interpreted as specifying interleaved concurrency. Parallel composition is classed as a *static* operator [Mil89].

In an interleaved model causally-independent events are ordered sequentially in any single trace. For example, considered from the point of view of another concurrent process or the environment, in a process of two enabled and independent actions a and c, either a will occur first and then c, or c will occur first and then a. Neither the possibility of the two occurring independently but simultaneously, nor of the order of occurrence being relative to the observer, are admitted.

The CCS expression in Figure A.3 shows two independent processes in parallel, and a corresponding process graph model. A trace of the process graph will run from top to bottom. All orderings of the four actions that respect the sequentiality of the individual components are possible. In the interleaved model, concurrency is transmuted to choice. An equivalent[5] expression to the one in the figure is

$$a.(b.c.d.\mathbf{0} + c.(b.d.\mathbf{0} + d.b.\mathbf{0})) + c.(a.(b.d.\mathbf{0} + d.b.\mathbf{0}) + d.a.b.\mathbf{0}).$$

The example increases in size from an expression of two three-state processes to a process graph of nine states. This is typical, the parallel composition of independent processes in the interleaved model yields a Cartesian product of their states: each pairing must be considered.

The interleaving model simplifies semantic rules, reasoning, and formal proofs. It is a natural description of multitasking where the actions of individual sequential processes are interleaved by an operating system. Interleaving seems also to be appropriate for modelling distributed systems provided the realities of networked communication are accounted for in the models themselves.

Interleaving is not the only possible representation of concurrency. There are partial order, or 'true concurrency' models, including Petri nets [Rei85, Pet81] and Event

[5] For any equivalence relation on interleaved processes.

structures [NPW81]. Synchronous models also exist, including the SCCS [Mil83], a variant of CCS where all concurrent proceses must participate in each step of the system if only to take an explicit idling step; actions then form an algebra of their own being combined to give composite actions.

A.4 Synchronized/handshake communication

Communication and concurrency are intertwined in the process algebraic approach. Concurrent components communicate with one another through synchronized actions, which, for two participants, are called a *handshake*.[6] The various approaches differ in their details.

The parallel composition operator of CSP is parameterized over two sets of synchronizing actions, one for each component [Ros97, §2.2]. In the process represented by $P \,{}_X\|_Y\, Q$, actions in $X \cap Y$ may only occur if performed simultaneously by both P and Q, actions not in $X \cap Y$ only occur when performed independently by either of P or Q. The interleaving operator $|||$ is equivalent to ${}_\varnothing\|_\varnothing$; all actions are performed independently. The lock-step operator $\|$ is equivalent to ${}_A\|_B$ where A and B are called the *alphabets* of P and Q respectively.[7] The alphabet of a (CSP) process must include all actions that the process might perform, but it may include actions that are not otherwise mentioned, and, which may thereby affect the behaviour of concurrent processes.

In CCS, every action a, except τ, has a complement $\bar{a}$. Only complementary actions may synchronise, but they need not. The semantics of parallel composition $|$ contain a rule [Mil89, §2.5] for each possibility:

$$\frac{E \xrightarrow{\alpha} E'}{E|F \xrightarrow{\alpha} E'|F}\ \text{Com}_1 \qquad \frac{F \xrightarrow{\alpha} F'}{E|F \xrightarrow{\alpha} E|F'}\ \text{Com}_2 \qquad \frac{E \xrightarrow{a} E' \quad F \xrightarrow{\bar{a}} F'}{E|F \xrightarrow{\tau} E'|F'}\ \text{Com}_3$$

where $a \in A \setminus \{\tau\}$ and $\alpha \in A$. Typically for CCS, the rules inductively describe part of the transition relations of a process graph in terms of the syntax of an expression. In rules Com_1 and Com_2 the processes act independently of one another. In Com_3 complementary actions synchronise to become a τ-transition. Note that τ-steps cannot synchronise with one another.[8] An example expression and corresponding process graph are shown in Figure A.4a. The structure is essentially identical with Figure A.3, where there is no synchronization, but for the two τ-transitions. In CCS, instead of forcing synchronization as done in CSP, unwanted transitions are pruned with a *restriction* operator, as shown in Figure A.4b.[9]

Composition in ACP also preserves the possibility of three actions, either interleaving or shared, but does not insist that synchronization results in a τ-transition. Rather a communication partial function $\gamma : A \times A \to A$ is defined. It maps each pair of actions that may synchronize to a resultant action, pairs not in the domain may not synchronize. The function must be commutative and associative so that the order of arguments is unimportant, neither of the arguments nor the result may be τ. The semantics of composition in ACP are given axiomatically as a set of (conditional) equations [BW90, Table 51]:

$$\mathsf{x} \parallel \mathsf{y} = \mathsf{x} \between \mathsf{y} + \mathsf{y} \between \mathsf{x} + \mathsf{x}|\mathsf{y} \tag{CM1}$$

$$\mathsf{a}\cdot\mathsf{x} \between \mathsf{y} = \mathsf{a}\cdot(\mathsf{x} \parallel \mathsf{y}) \tag{CM3}$$

$$\mathsf{a}|\mathsf{b} = \gamma(a,b) \quad \text{if } \gamma \text{ defined} \tag{CF1}$$

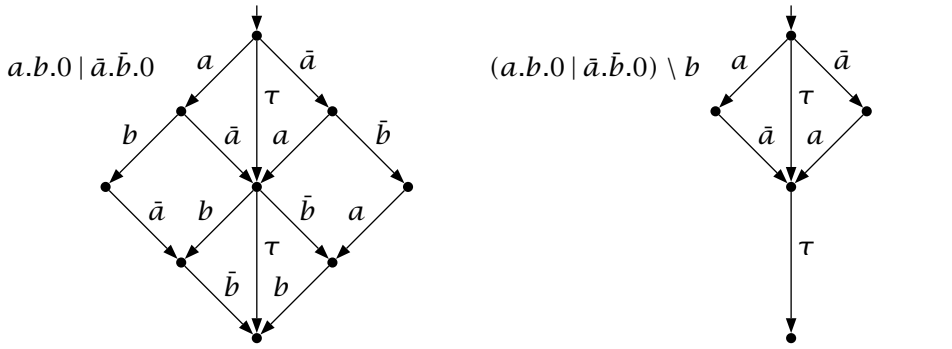
$$\mathsf{a}|\mathsf{b} = \delta \quad \text{otherwise} \tag{CF2}$$

[6] They are also sometimes called a 'rendezvous', but this term is more appropriate for the type of remote procedure call in Ada, or 'synchronous communication', but this term is easily confused with the synchronous broadcast communication discussed in §2.4.

[7] Alphabets are presented differently in [Hoa85].

[8] Properly, a complete set of rules should be presented otherwise it is not possible to conclude which transitions are not included.

[9] The CSP concealment and CCS restriction operators are both written as the symbol \. Both remove transitions, but the former 'reconnects' the processes after concealed actions or alternatively renames actions to τ, whereas the latter prunes them altogether.



(a) Two synchronizing processes **(b)** Restriction of two synchronizing processes

Figure A.4: CCS expressions with process graphs for synchronization and restriction

Rules for several other cases have been omitted. The symbols $\between$ and $\mid$ are auxiliary operators for defining parallel merge in ACP, the latter should not be confused with the similar symbol in CCS. The use of a communication function increases both flexibility and bookkeeping. The *combine* function of Esterel, §2.4.3, is a similar idea for combining the effect of simultaneous actions

All three process algebras define ways of communicating values between concurrent processes. The approach and syntax of CSP have been particularly influential, for example in an extension to Esterel [BRS93], and in Uppaal as described in §2.3. The roles of sender and receiver are distinguished when values are communicated. Both participants remain equal in terms of synchronisation, neither can occur without the other and both actions occur simultaneously, but the sender specifies a value, usually as an expression over local variables and constants, whereas the receiver specifies a name to bind the communicated value. In CSP, sending and receiving are expressed by the syntax [Hoa85, Chapter 4]:

$$c\,!\,v \to P \qquad\qquad c\,?\,x \to Q(x),$$

where both processes refer to a global *channel* name c rather than to one another, output is marked with an exclamation mark between channel name and value $c\,!\,v$, and input with a question mark between channel and variable names $c\,?\,x$, the latter is bound over the residual process expression $Q(x)$. Formally, the set of actions includes an element $c.v$ for each combination of channel name and value, and the input notation denotes a branch for each possible v. Communication of values in ACP [BW90] is similar, but rather than name a common channel, each process specifies a local *port*. Port interconnections are defined via the communication function, for instance $\gamma(\mathsf{send}(5), \mathsf{receive}(5)) = \mathsf{comm}(5)$.

While handshake communication is conceptually appealing, its merits for modelling and programming are also important. Several applications may be modelled directly: subroutines calls [Hoa85, §7.3.4], where the instants of call and return are shared by two functions; process synchronisation in multitasking operating systems, since it does not matter which action occurs next after synchronisation; and, conceptual interactions like pushing a button or picking up a chopstick [Hoa85, §2.5]. Handshake communication can also benefit programmers: reasoning is simplified because there is only a single instant of communication, not separate instants of sending and receiving. Also something of the state of both participating processes is known at that instant: they are blocked at communication instructions. Handshake communication can be efficient because buffering is avoidable when not required. The absence of buffering, or at least its restriction, can simplify verification [Hoa85, §7.3.4].

Some complain that handshake communication is unrealistic or not practicable to implement, particularly for distributed systems and those with multiple processors or multiple clocks. But this is an issue of modelling not mechanism. Buffering or communications delay, whether programmed, through a synchronizer, or in a network, should be modelled explicitly, for instance by adding buffer processes, or by mandating

that asynchronous actions may only prefix the deadlock process thereby syntactically excluding dependencies on message receipt in the sender.[10] Consider, for example, an ACP process C, triggered by a local action s that sends an asynchronous query q and awaits a reply of either ack or nack before reporting the result locally, in parallel with another process P that acknowledges every second query:

$$\begin{aligned} \mathsf{C} &= \mathsf{s}\cdot(\mathsf{q}\cdot\delta \parallel (\mathsf{ack}\cdot\mathsf{yes}\cdot\mathsf{C}) + (\mathsf{nack}\cdot\mathsf{no}\cdot\mathsf{C})) \\ \mathsf{P} &= \mathsf{q}\cdot(\mathsf{P'} \parallel \mathsf{ack}\cdot\delta) \\ \mathsf{P'} &= \mathsf{q}\cdot(\mathsf{P} \parallel \mathsf{nack}\cdot\delta) \end{aligned}$$

Modellers must decide whether direct handshake communication is accurate for a given circumstance, depending on the level of abstraction, relative atomicity of actions, and either the properties to be validated or the eventual implementation architecture.

Although buffer processes are easily included in models, it is less reasonable to program distributed systems in the same way. The constructs of a language influence programmed solutions, compilation, and optimization. Systems with handshake communications are most naturally implemented with message passing through a local arbiter, rather than through shared memory or over asynchronous links. Attempts at implementing handshake communication across networks require either a central arbiter or constraints on when computations may occur [RS98].

The parallel composition operator of CSP allows each step of a system to be influenced by multiple processes. An action may only occur if all controlling processes agree. Parallel composition is equivalent to conjunction of trace specifications [Hoa85, §2.3.3] [Ros97, §2.5], and leads to a style of *constraint-oriented specification* [BB87, Ros97] where the set of required behaviours is gradually refined through the addition of new restrictions in parallel. Implementations, however, need not involve multiway handshaking, it is sufficient if their behaviours can be shown to refine those of the specification.

A.5 Determinism

In the process algebraic approach the environment of a system is treated like any other concurrent process. In particular, system actions must synchronize with those of the environment. Only τ-actions can be performed autonomously, all other actions are presented to the environment which selects the next to occur.

Decisions that a system can make autonomously, that is non-deterministically, from its environment are expressed in CSP by an *internal choice*[11] $\sqcap$ operator. For example, the process $a \to P \sqcap b \to Q$ may offer to synchronize on a but not b, on b but not a, or on either a or b. The latter is termed *angelic non-determinism*, which can be inefficient to implement. Internal choice is intended for use in specifications, either explicitly or resulting from combinations of operators like concealment, or general choice when the component processes are prefixed with the same action,

$$(c \to P) \,\Box\, (c \to Q) = (c \to P) \sqcap (c \to Q).$$

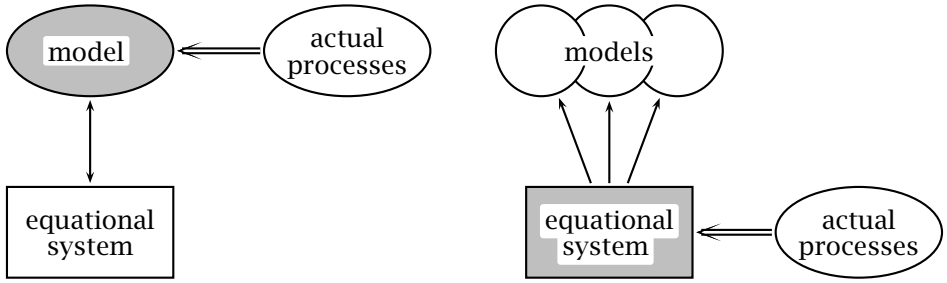
Neither ACP nor CCS has an explicit operator to represent non-deterministic choice, since the autonomy of τ-actions can be exploited to the same effect. The CSP process $P \sqcap Q$ could be written in CCS as $\tau.P + \tau.Q$.

The philosophy that implementations are, ideally, deterministic is reflected in the refinement ordering of CSP and in programming languages like Esterel. The distinction between specification and implementation, though, is not always clear. A program may give the same result regardless of the way particular choices are resolved. The compiler or runtime environment would then be free to resolve the choices to satisfy other requirements, possibly improving some factor of efficiency.

[10] Matthew Hennessy pointed this out during a tutorial in 2005.

[11] Also called *nondeterministic or* [Hoa85, §3.2]

A.6 Semantic models



(a) CCS and CSP approach **(b)** ACP approach

Figure A.5: Sketch of relations between process algebras, models, and actual processes

In CSP and CCS, processes are modelled as mathematical structures, refer Figure A.5a. The models formalise an abstraction of actual processes: like vending machines [Hoa85, §1.1] or job shops [Mil89, §1.3]. Constants, operators, and equational identities are conceived to represent and reason about process models, but the focus of study is the models themselves.

In contrast, ACP is primarily concerned with equational systems as formalizations of actual processes, refer Figure A.5b. An equational system is a set of function symbols and their arities, and a set of equational axioms. Technically, a model is just a witness to the consistency of a set of axioms. But, models are also interesting as mathematical objects in their own right, and they aid the development of intuitions on the implications of axioms.

There are two standard process models in CSP: a trace model for deterministic processes [Hoa85, §2.8], and a failures/divergences model for non-deterministic processes [Hoa85, §3.9].

In the trace model a process is a pair (A, S), where A is a set of symbols and S is a prefix-closed set of finite traces over A. Constants in the expression language correspond to specific pairs and function symbols to operators on pairs. The solutions of recursive equations are defined through fixed point theory.

In the failures/divergences model, each trace is augmented with additional information about the actions possible thereafter.

Definition A.6.1
Given an LTS $(S, S_0, \longrightarrow)$ over A, a *refusal* of a state $s \in S$ is a set $X \subseteq A$ such that $\forall a \in X.\, (s \stackrel{a}{\not\longrightarrow} \wedge\, s \stackrel{\tau}{\not\longrightarrow})$. The set of all refusals of s is written $\mathit{refusals}(s)$. ■

Definition A.6.2
A *failure* of an LTS $\mathcal{T}$ over A is a pair (t, R), where t is a finite trace of $\mathcal{T}$ and $R = \bigcup\{\mathit{refusals}(s_n) \mid \alpha = s_0, a_0, \ldots, a_{n-1}, s_n \in \mathit{execs}(\mathcal{T}) \text{ and } \mathit{trace}(\alpha) = t\}$. ■

The set of all failures of an LTS $\mathcal{T}$ is written $\mathit{failures}(\mathcal{T})$; the traces are prefix-closed, and the refusals of each trace are subset-closed. An element is a failure of a process, if, after performing the sequence of actions T and all possible internal actions, the process *may* deadlock if only offered actions in X for synchronisation; two examples, that cannot be distinguished by their traces alone, are given in Figure A.6. The union after grouping executions by trace in Definition A.6.2 induces a subtle and important structure. A failure set encodes the effects of all possible executions for the associated trace, and thus expresses the effect of non-determinism.

In the failures/divergence model a process is a triple (A, F, D), where A is a set of symbols, F is a set of failures, and D is a set of *divergences*. A divergence is a finite trace where the process *may* afterward perform an infinite number of internal steps. The set of divergences is closed under trace extension.

CCS adopts a single transition system model. The closed process expressions of the language form the set of states of an LTS where $S_0 = \varnothing$ [Mil89, §2.5], usually modulo

$a \rightarrow ((b \rightarrow STOP) \sqcap (c \rightarrow STOP))$	$\{(\epsilon, X) \mid X \subseteq \{b, c\}\}$ $\cup \{(a, X) \mid X \subseteq \{a, c\} \vee X \subseteq \{a, b\}\}^*$ $\cup \{(ab, X) \mid X \subseteq \{a, b, c\}\}$ $\cup \{(ac, X) \mid X \subseteq \{a, b, c\}\}$
$a \rightarrow ((b \rightarrow STOP) \Box (c \rightarrow STOP))$	$\{(\epsilon, X) \mid X \subseteq \{b, c\}\}$ $\cup \{(a, \emptyset), (a, \{a\})\}$ $\cup \{(ab, X) \mid X \subseteq \{a, b, c\}\}$ $\cup \{(ac, X) \mid X \subseteq \{a, b, c\}\}$

* Note that $\{a, b, c\}$ is not in this refusal set.

Figure A.6: Failure sets where $A = \{a, b, c\}$

a bisimulation equivalence. The transition relation $\xrightarrow{a}$ is described using SOS. The definition is given inductively by rule schemas over the structure of expressions. A rule schema has two parts: a sequence of *hypotheses* and a *conclusion*, the former is usually written over the latter with a line between like Com_1, Com_2, and Com_3 of §A.4. If all the hypotheses are true of the relation then the conclusion must be too.

No distinction is made in CCS between processes and states. The model denoted by a process expression is a connected subgraph of the transition system defined by the whole language; it is a process graph where s_0 is the state corresponding to the expression and the predicate $\surd$ is unused.

In ACP, reasoning proceeds, ideally, from the axioms alone, independently of any model. Rather than distinguish one process model, various models are developed and compared. The most usual are [BW90] a term model, similar to the LTS of CCS, a projective limit model, and a process graph model, which is similar to, but more abstract than the term model. The relations between a set of axioms and a model are established rigorously: soundness means that all statements derivable from the axioms are true of the model, and completeness that all statements true of the model are derivable from the axioms.

Appendix B

Input/Output Automata and related approaches

Input/Output Automata (IOA) [Lyn96, Chapter 8][LT89] are another formalism for the rigorous modelling and verification of reactive and concurrent systems. They were influenced primarily by the CSP approach to process algebra [Lyn96, §8.9] but they have developed a distinct character and body of research of their own. The extension of IOA to model time [KLSV06] is particularly relevant for modelling embedded systems. Hybrid and probabilistic extensions are also important but they are outside the present scope.

The three distinguishing characteristics of IOA are the partitioning of external actions into sets of inputs and outputs with related restrictions on transition structures, §B.1, the description of states via variable valuations §B.2, and the attention given to fairness, §B.3. IOA incorporate external actions as in process algebra, with particular inspiration from CSP, with description and reasoning over states and executions as do formalisms like the Temporal Logic of Reactive and Concurrent Systems (TLRCS) and Temporal Logic of Actions (TLA$^+$), which are described in §B.4. A version of IOA in continuous time is outlined in §B.5.

B.1 Input-enabling and composition

Definition B.1.1
An *input/output automaton* over I, O, and L is an LTS $(S, S_0, \longrightarrow)$ over $A = I \dot\cup O \dot\cup L$ and $P = \emptyset$ where $\forall i \in I, s \in S. \exists s' \in S. s \xrightarrow{i} s'$, and a *task partition* T that partitions $O \dot\cup L$ into equivalence classes. The pair will be written $(S, S_0, \longrightarrow, T)$.[1] ■

The three types of action are respectively *input*, *output*, and *internal*. The first two are together termed *external* actions $I \dot\cup O$, and the latter two together *locally-controlled* actions $O \dot\cup L$. Internal actions are like the τ actions of process algebra, but distinguishing different actions allows for specific reasoning about the behaviour and properties of individual components.

The clause that requires transitions on every input action to be defined for every state defines a class of *input-enabled* LTSs. It marks a distinct shift in perspective from the consensus-requiring handshakes of the process algebras to a setting where processes must always be ready to react to stimuli from the environment which is, arguably, more natural for a large class of embedded systems where there is no arbitration between processes, rather just detection and response. While in process algebras a larger class of processes can be considered, the restriction has the advantage of simplifying notions of refinement and equivalence [Vaa91, LV95a]. In terms of specifying

[1] I/O automata are usually presented differently [Lyn96, Chapter 8][LT89]. For an automaton A, the sets of states, initial states, transitions, and the task partition are written, respectively, $states(A)$, $start(A)$, $trans(A)$, and $tasks(A)$. A separate 'signature' $sig(A) = S$ partitions the set of actions, written $acts(S)$ into inputs $in(S)$, outputs $out(S)$, and internal $in(S)$ actions. The set $trans(A)$ is a subset of $states(A) \times acts(sig(A)) \times states(A)$, rather than a family of transition relations.

algorithms, it can be argued [Lyn96, §8.1] that input-enabling forces designers to specify what should happen when an unexpected input occurs. Furthermore, it simplifies the theory [Lyn96, §8.1].

An IOA may model implementations comprising multiple distinct threads of control. While there is a tendency to abstract from such details in process algebra, they are integral to the treatment of fairness in IOA. Rather than identify tasks, locally-controlled actions are considered equivalent, for the purpose of defining execution fairness, when they are controlled by the same component.

The properties of locally-controlled actions are ensured, while preserving the synchronising character of communications, by restricting when IOA may be composed. A (countable) set of IOA are *compatible* if their local actions are truly local, each action is controlled by at most one of them, and only finitely many of them share any one action [Lyn96, §8.2.1].

Definition B.1.2
A set of IOA $\{\mathcal{I}_j\}_{j\in J}$ is *compatible* iff

1. $\forall j,k \in J.\, j \neq k \Longrightarrow L_j \cap A_k = \emptyset$
2. $\forall j,k \in J.\, j \neq k \Longrightarrow O_j \cap O_k = \emptyset$
3. $\forall a \in \bigcup_{j\in J} A_j.\, \mathsf{finite}\,(\{A_k \mid k \in J \text{ and } a \in A_k\})$ ■

Compatibility is a prerequisite for forming the composition of IOA [Lyn96, §8.2.1].

Definition B.1.3
Given a set of compatible IOA $\left\{\mathcal{I}_j = \left(S_j, S_{0_j}, \longrightarrow_j, T_j\right)\right\}_{j\in J}$ the *composition* $\|_{j\in J}\,\mathcal{I}_j$ is an IOA $(S, S_0, \longrightarrow, T)$ over $I = \bigcup_{j\in J} I_j \setminus O$, $O = \bigcup_{j\in J} O_j$, and $L = \bigcup_{j\in J} L_j$ where

- $S = \prod_{j\in J} S_j$
- $S_0 = \prod_{j\in J} S_{0_j}$
- $\vec{s} \xrightarrow{a} \vec{s}'$ iff $\begin{cases} s_j \xrightarrow{a} s'_j & \text{if } a \in A_j \\ s_j = s'_j & \text{otherwise} \end{cases}$
- $T = \bigcup_{j\in J} T_j$

The symbol $\prod_{j\in J}$ denotes a Cartesian product indexed by J, each element is a vector $\vec{s}$ whose components are written s_j for $j \in J$. ■

The definition of composition over a set of IOA, rather than as a binary operator as in the process algebras, emphasizes the structure of the composition, and allows compositions of an unbounded number of components without a recursion operator. The restriction to compatible IOA ensures that exactly one component controls each internal and output action, and also that only a finite number of components participate in any transition. The structure of task partitions is preserved by composition.

Communication between composed IOA is similar to that of CSP: the potential for further synchronization with an action is preserved and all components who can synchronize on an action must. But the input-enabled and compatibility clauses restrict the structure of interactions. The occurrence of an output action is determined by at most one component, and any other component that shares the action must synchronise with it. Communication is thus by synchronized broadcast from a single source. Importantly, and as in CSP [Hoa85, §2.8.1, D6], the traces of an IOA composition have the property [Lyn96, Theorems 8.1–8.3] that when restricted to the alphabet of an individual component they are also a trace of that component; similarly for executions.

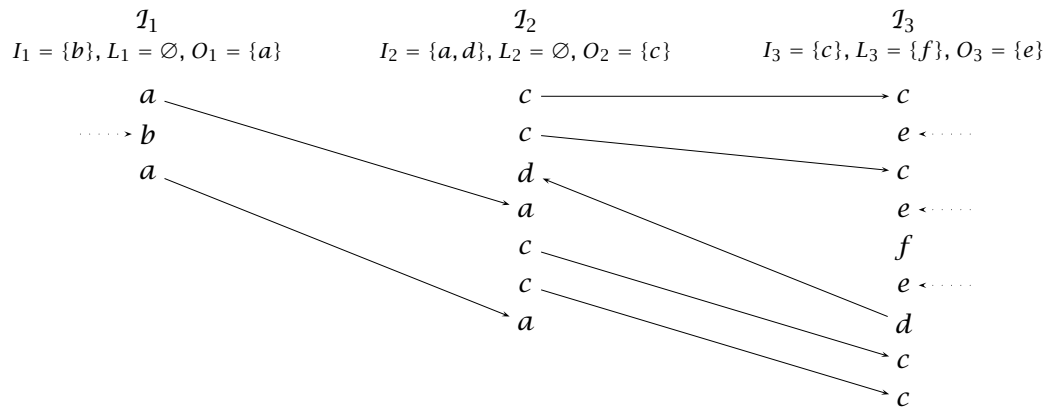


Figure B.1: 'Pasting' together Input/Output Automata (IOA) traces

This property makes compositional reasoning possible; properties of the whole follow directly from properties of the components.

Composition constrains the executions and traces of components. The executions or traces of individual components can only be 'pasted together' if there is a mutually consistent ordering of the actions of each. That is, all components with an action in common must correspond on its occurrences, whether it is generated inside or out. The correspondences must respect the sequential ordering within each execution or trace. See Figure B.1 for an example. Even given a specific execution for each component, different executions of the composition may still be possible if the components do not synchronise together on all actions. Each would represent one projection of a partial ordering amongst the events. This characteristic is accounted for in the usual statements of the properties of compositionality [Lyn96, Theorems 8.2/8.5, 8.3/8.6] by postulating a trace in the composition that is consistent with individual component behaviours, from which a combined effect can then be concluded.

Hiding is the only other operator commonly defined on IOA [Lyn96, §8.2.2]. It has a similar effect to the ACP encapsulation operator, or the CSP hiding operator defined with τ-actions.

Definition B.1.4
Given an IOA $\mathcal{I}$ over I, O, and L, a subset of output actions $H \subseteq O$ can be *hidden* to produce an IOA $\mathcal{I} \setminus H$ with identical structure but defined over I, $O \setminus H$, and $L \cup H$. ■

B.2 Modelling and specification

The states of an IOA in Definition B.1.1 are elements of an abstract set, a description which, although sufficient for describing many operations and properties, does not adequately represent the way that IOA models are created or analysed in practice. Rather the state space of a model is defined implicitly by declaring a set of variables and their possible values; computing an exact set of reachable states may be non-trivial and possibly the main reason for constructing a model. The subset of initial states is defined by stating the values that variables may take initially. Input, output, and internal actions are stated directly, and often parameterized for convenience, as are the task sets. The transition relation is described by associating a *precondition* and an *effect* with each action. The precondition is a predicate over state-space variables defining a set of states whence a transition on an action departs. By describing how a subset of those variables may change—those not mentioned are unchanged—the effect describes transition destinations relative to an action and source state.

IOA are typically presented in a stylized form similar to Figure B.2. The example shows an abstraction of software [Hol03, Chapter 5] from the NASA Pathfinder demonstrating a priority-inversion fault. The pseudo-code notation [KLSV06] is more stylized than the mix of informal natural language and mathematics in other examples [Lyn96]. The description comprises the declaration of two type domains and an IOA described

```
type PcValue = enumeration of idle, wait, run
type MutexValue = enumeration of free, busy

automaton Pathfinder
  signature
    input pause(v: Bool)
    output enterh, exith, enterl, exitl
    internal waith, waitl
  states
    pch: PcValue := idle,
    paused: Bool,
    pcl: PcValue := idle,
    mutex: MutexValue := free
  transitions
    input pause(v: Bool)
      eff
        paused := v

    internal waith
      pre
        pch = idle
      eff
        pch := wait

    output enterh
      pre
        pch = wait ∧ mutex = free
      eff
        pch := run;
        mutex := busy

    output exith
      pre
        pch = run
      eff
        pch := idle;
        mutex := free

    internal waitl
      pre
        pcl = idle ∧ pch = idle
      eff
        pcl := wait

    output enterl
      pre
        pcl = wait ∧ mutex = free
          ∧ pch = idle
      eff
        pcl := run;
        mutex := busy

    output exitl
      pre
        pcl = run ∧ pch = idle
      eff
        pcl := idle;
        mutex := free
  tasks
    high = {enterh, exith, waith}
    low  = {enterl, exitl, waitl}
```

Figure B.2: Pathfinder abstraction [Hol03, Chapter 5] as an IOA in TIOA style [KLSV06]

over four sections. The `signature` describes the I, O and L sets of actions, note the parameterization giving $\mathsf{pause}_{\mathsf{true}}$ and $\mathsf{pause}_{\mathsf{false}}$ actions. The reachable `states` are a subset of a cross-product of the domains of the four variables, and there are two initial states because `paused` is not assigned an explicit initial value. Two of the state variables encode program counters, `pch` and `pcl` for the high- and low-priority processes respectively, which is an effective technique for small examples and abstract models but manual reasoning about complex control structures may be unnecessarily difficult. The `transitions` section contains a precondition/effect pair for each action or action schema. There is no explicit precondition for the `pause` actions, and they are thus enabled in each state as required for input-enabledness. The two task partitions are given last.

The low-priority process is only allowed to run when the high-priority process is idle, which, although it would likely be effected by an operating system scheduler, is here modelled in the preconditions of `waitl`, `enterl`, and `exitl`. With handshake communication, particularly the CSP-multiway version, it would be natural to model each of the two processes, the mutex, and the scheduler as a distinct IOA. However, as processes must be input-enabled, the occurrence of an action does not give the controlling process any information about the state of other processes: separate acknowledgement actions, which complicate modelling, would be required.

In the example two separate processes are encoded in a single IOA. They interact via a shared variable, the mutex, rather than through input and output events. This style of *asynchronous shared memory model* [Lyn96, Chapter 9], where interaction with the system environment is through actions but local tasks communicate through shared variables, differs from that of the *asynchronous network model* [Lyn96, Chapter 14] where processes and channels are modelled as separate IOA that communicate in composition by sending and receiving messages. In the shared memory model, locality restrictions on local variables and variable type restrictions on shared variables limit transition preconditions and effects. The pathfinder example does not meet the locality restrictions because the actions of one task depend on the state of another.

Dynamic behaviour in IOA models is expressed directly by individual automata, in contrast to the process algebras where individual processes are built from constants, action prefixing, choice, and recursion. The static operators for composition and hiding are used, however, to describe how automata are connected to form larger structures. This is sufficient for modelling a variety of algorithms and accords with the importance in IOA of variable-structured states for description and reasoning. An alternative is to build and compose IOA with action transducers [LV95a], a formalism for describing operators on the domain of IOA. Even using action transducers, little distinction is made between syntax and model, between description and denotation.

Mandating input-enabledness restricts the type of transition systems that can be described. It is possible to define syntactic conditions on transition rules to ensure input-enabling, and then, with a further condition ensuring *output non-refusal*, to define an input/output form of process algebra [Vaa91]. Interestingly, the choice operator must be parameterised by the triggering actions of each component since they are input-enabled:

$$\frac{E \xrightarrow{e} E'}{E \,{}_I{+}_J\, F \xrightarrow{e} E'} \text{if } e \in I \cup \overline{A} \qquad \frac{F \xrightarrow{e} F'}{E \,{}_I{+}_J\, F \xrightarrow{e} F'} \text{if } e \in J \cup \overline{A}$$

where A is the set of input actions, $\overline{A}$ is the set of output actions, and $I, J \subseteq A$.

B.3 Fair executions and analysis

In an asynchronous model nothing can be said about the execution times, relative or absolute, of parallel processes. Systems modelled or designed under this simplifying assumption are less dependent on brittle details of implementation [Dij68, §2]; the length of a wire, temperature of a chip, or number of instructions in a scheduler for instance.

But there may be executions of such models where the enabled locally-controlled actions of some components are never performed. Such executions may be unrealistic, for instance physically distinct components operating at any non-zero speed would be expected to act eventually, and infinitely often if possible, or it may be possible to assume or mandate that they are, for instance by requiring that a scheduling mechanism eventually gives every process that is able to act a chance to do so. Such *fairness assumptions* either restrict the applicability of a model, or form part of a specification that must be validated of or ensured by implementations. Stating them separately adds an additional complication, but obviates the need to explicitly model scheduling mechanisms, which may be complicated, tedious, or irrelevant. Fairness assumptions are stated in terms of infinite sequences, which makes them abstract, and thus cannot be expressed, in full generality, by a (finitely branching) transition system which can only act on the finite prefixes of sequences [MP92, §2.10].

For an IOA $(S, S_0, \longrightarrow, T)$, each task partition $C \in T$ is a set of locally-controlled actions, $C \subseteq O \cup L$, that belong to a subcomponent which must make progress when able regardless of other subcomponents. The integrity of a task partition is maintained through composition, Definition B.1.3, some structure of the parts is thereby retained in the whole [LT89, p. 12]. The approach taken by the process algebras is more abstract because the details of how a composite is formed are lost afterward, but this has an effect on the expression of fairness. The effect of task partitioning is considered by defining a subset of *fair executions* [Lyn96, §8.3]. Fairness is a type of progress that is not lost in composition.

Definition B.3.1
Given an IOA $\mathcal{I} = (S, S_0, \longrightarrow, T)$ over I, O, and L, let $\mathsf{fair}_{\mathcal{I}}$ be a predicate that is true of an execution α of $\mathcal{I}$ iff

- if $\alpha = s_0, a_0, s_1, a_1, \ldots, a_{n-1}, a_n$ then $\forall a \in O \cup L. s_n \stackrel{a}{\not\longrightarrow}$,
- otherwise $\forall C \in T. (\neg\mathsf{finite}(\alpha \lceil C) \text{ or } \neg\mathsf{finite}(\alpha \lceil \{s \mid \forall a \in C. s \stackrel{a}{\not\longrightarrow}\}))$

A finite execution is considered fair if no locally-controlled actions can occur in the final state. An infinite execution is fair if for each task either infinitely many of its locally-controlled actions occur, or there are infinitely many states where such actions cannot occur.

Definition B.3.2
Given an IOA $\mathcal{I}$ let $\mathit{fairexecs}(\mathcal{I})$ denote $\{\alpha \mid \alpha \in \mathit{execs}(\mathcal{I}) \text{ and } \mathsf{fair}_{\mathcal{I}}(\alpha)\}$. ■

Definition B.3.3
Given an IOA $\mathcal{I}$ let $\mathit{fairtraces}(\mathcal{I})$ denote the set of *fair (weak) traces*[2] in $\mathit{fairexecs}(\mathcal{I})$. ■

The set of fair traces is a subset of the set of traces.

Trace properties [Lyn96, §8.5.2][LT89] are one way of specifying requirements on, or expectations of IOA models. A trace property $\mathcal{P}$ is a set of traces $\mathit{traces}(\mathcal{P})$ defined over sets of inputs $I_{\mathcal{P}}$ and outputs $O_{\mathcal{P}}$. No specific formalism is proposed for defining traces sets; it could be done informally or with a linear temporal logic [Lyn96, p. 220], for instance. An IOA $\mathcal{I}$ over I, O and L can only satisfy a trace property if $I = I_{\mathcal{P}}$ and $O = O_{\mathcal{P}}$. There are two notions of satisfaction either $\mathit{traces}(\mathcal{I}) \subseteq \mathit{traces}(\mathcal{P})$, or $\mathit{fairtraces}(\mathcal{I}) \subseteq \mathit{traces}(\mathcal{P})$. In the latter case, traces resulting from executions that are not fair need not be considered by proofs.

The IOA approach is characterised by the attention paid to rigorous and practical proof techniques. Three main techniques [Lyn96, §8.5] are invariant assertions, compositional reasoning, and hierarchical proofs.

Despite its name, an invariant assertion, in the IOA approach, is a property that is *always-true* of reachable states of an automaton [Lyn96, §8.5.1]. Thus it need not be *invariant* in the sense of being preserved by all transitions regardless of source state reachability [GT90]. The distinction makes little difference for IOA because components in parallel can only influence one another through actions, not by changing

[2]The original [LT89] IOA terms for strong and weak traces were, respectively, 'schedules' and 'behaviours'.

non-local state components. Invariant assertions are usually proved by induction over the steps in an execution, with case-analysis over possible actions.

Compositional reasoning allows the properties of the whole to be inferred from those of the parts. It is made possible because of properties of the definitions of IOA and their composition. The executions and traces of a composition, when suitably restricted, are also executions and traces of individual components [Lyn96, Theorem 8.1]. The merged traces of individual components that correspond on shared actions form a trace of the composition [Lyn96, Theorem 8.3], and similarly for executions [Lyn96, Theorem 8.2].

Hierarchical decomposition [Lyn96, §8.5.5] is a technique for developing a proof, implementation, or both simultaneously, by beginning with a relatively simple IOA and developing a series of *successive refinements* of increasingly detailed IOA until the desired levels of detail and efficiency are reached. The essentials of the technique are also applicable for forming successive abstractions from an implementation model through to one where analysis becomes feasible or where it is obvious that desired properties hold. One IOA $\mathcal{I}_c$ is a valid implementation of another $\mathcal{I}_a$ if a one-way *simulation relation* can be found between initial states of $\mathcal{I}_c$ and $\mathcal{I}_a$. A relation exists if for every step from a reachable state of $\mathcal{I}_c$ there exists a sequence of steps from the corresponding state in $\mathcal{I}_a$ to a corresponding destination state. The existence of a relation implies trace inclusion, $traces(\mathcal{I}_c) \subseteq trace(\mathcal{I}_a)$. Simulation techniques are used in many formalisms; notably in CCS.

B.4 Related techniques

Many systems for specifying concurrent and distributed systems using assertional reasoning have been proposed. They are extensions of classical techniques for analysing the properties of sequential programs. Two approaches, both based on temporal logic, are particularly close in style and technique to IOA: the Temporal Logic of Reactive and Concurrent Systems (TLRCS) [MP92] and the Temporal Logic of Actions (TLA$^+$) [Lam02].

Executions,[3] Definition 2.1.6, are primary in all three approaches. States assign values to variables. In all three frameworks, actions represent discrete transitions between states, but they play a more central role in IOA where they are given a distinct identity, act as inputs, outputs, or internal steps, and affect operators on automata. In both the TLRCS and the TLA$^+$ an action is an assertion on the values of variables in two consecutive states—those in the second state being distinguished by primes—actions are not observed directly, rather only by their effect on variables. Thus, in the TLRCS and the TLA$^+$, traces are usually not defined or studied.

Temporal logic is an optional extra for IOA [Lyn96, p. 220], but it is central to the approaches of the TLRCS and the TLA$^+$ although the two differ slightly in style.

The basic model of TLRCS is a Fair Transition System (FTS), which is essentially—like an IOA—an LTS over A, where states are described by assignments to a finite set of variables, augmented with sets $J \subseteq A$ and $C \subseteq A$ that play a role, like the partitions of IOA, in defining fair executions. Four modelling languages [MP92, Chapter 1] are proposed: graphical transition diagrams communicating through shared variables, a shared-variable concurrent programming language, a message-passing concurrent programming language, and Petri nets. Each is given a semantics in terms of FTSs by an encoding in state variables; for instance, rather than associate states with program expressions, like the transition systems of CCS, §A.6, control variables and a system of nested labels are used. An FTS, in turn, implicitly defines a set of executions. Properties are expressed in temporal logic, which is a topic of interest in its own right [MP92, Chapter 3]. Interpretations are given for executions and thereby for FTSs.

Everything in the TLA$^+$ is a formula of temporal logic; the focus is on concepts rather than language []. In fact, predicate logic is preferred and temporal operators are used in a limited and structured way. A system is specified by a single formula of the form

$$Init \wedge \Box [Next]_v \wedge Liveness,$$

[3] Called variously 'computations' [MP92] and 'behaviours' [Lam02].

where *Init* characterises the initial states of the system, *Next* characterises the actions—pairings of states—that may occur, and *Liveness* limits the set of infinite behaviours; usually in terms of fairness constraints.

The subscript v represents the set of variables whose values may change as a result of *next*. The notation $[next]_v$ means $Next \vee (v = v')$, the variables in v either change due to the action or they are left unchanged by *stuttering steps*: other transitions that may occur but that do not concern the specification. The square brackets of the action are suggestive of the box-shaped *always* operator because the two always occur together (so that stuttering steps may always occur). Another action notation is used with the diamond-shaped *eventually* operator in liveness properties $\langle Next \rangle_v$ to mean $Next \wedge (v \neq v')$, an action that does change variables in v (since stuttering steps may occur infinitely often anyway).

TLA$^+$ formulas are interpreted over infinite executions. A specification σ represents a set $\{\sigma \mid \sigma \vDash Spec\}$. Properties are also written in temporal logic and imply a set of executions. A property P holds of a specification *Spec* if the executions of the latter are a subset of those of the former, or equivalently if $Spec \implies P$.

A state in an execution is said to be *stuttering* when it is identical to an adjacent state. A formula of temporal logic is said to be *invariant under stuttering* when its truth value is unchanged by the addition or removal of stuttering states [Lam02, Chapter 8]; such a formula cannot distinguish between two formulas that only differ in this way. Invariance under stuttering is the basis for hierarchical specification [Lam83, §2.3]; a single step in an abstract specification might be refined by multiple steps in an implementation model. Additionally, a single specification describes behaviours only as they are relevant to a set of variables, which may model only one aspect of a system. A behaviour of the broader system, when reduced to the set of variables, may have stuttering states where actions were occurring elsewhere.

All 'sensible' TLA$^+$ formulas should be invariant under stuttering [Lam02, Chapter 8]. The FTSs of TLRCS are required to have 'idling transitions' on every state that leave the state unchanged; they are self-loops. An execution may only contain an infinite number of idling transitions after having reached a state where such transitions are the only possibility.[4] So stuttering is always possible in TLRCS but it need not be ignored, in particular the temporal logic defines a *next operator* $\bigcirc$, which is sensitive to stuttering. A next operator is considered too expressive for the TLA$^+$ [Lam83]. The local state components of IOA stutter when an action occurs in an unrelated component, Definition B.1.3, but because actions are more essential to a description of system behaviours stuttering is irrelevant.

The type of fairness described previously, §B.3, is known as *weak fairness*: executions where a transition is continuously enabled, that is without interruption, but does not occur are excluded from consideration. Weak fairness can be interpreted in terms of individual transitions or in terms of processes [MP92]. The latter interpretation is expressed in IOA by grouping transitions into task partitions, although the former can also be expressed by defining a separate partition for each transition. In TLRCS, weak (transition) fairness is assumed of all actions in the J-set of an FTS. In TLA$^+$, weak fairness is asserted for an action by conjoining a predicate, defined ultimately in temporal logic, onto the *Liveness* subformula.

Weak fairness is inadequate for ensuring individual progress when an action depends on access to a shared resource or the participation of another process [MP92]. Then *strong fairness* is more appropriate: executions where a transition is repeatedly enabled, but not necessarily continuously, but does not occur are excluded from consideration. It is not relevant for IOA because the enabledness of output and internal actions depends on the controlling process alone, and any assumptions required of input actions are better modelled explicitly. Strong fairness in TLRCS is assumed of all actions in the C-set of an FTS, and in TLA$^+$ by asserting predicates on actions.

While more general liveness assertions can be made in TLA$^+$, nearly all specifications are adequately expressed in terms of conjunctions of fairness conditions on action clauses [Lam02, §8.9.2].

[4] Which removes the need to consider finite executions, and also mandates a kind of global progress property.

B.5 Timed I/O Automata

Timed Input/Output Automata (TIOA) [KLSV06] are an extension of IOA for reasoning about algorithms and systems in quantitative time. They are a good example of how modelling formalisms can be adapted by adding explicit clock variables.

Definition B.5.1
A Timed Input/Output Automata (TIOA) over pairwise disjoint I, O, and L is a tuple $(V, \mathcal{V}_0, \longrightarrow, \mathcal{T})$, where V is a finite set of variables, $\mathcal{V}_0 \subseteq 2^{\mathsf{Vals}_V}$ is a set of initial variable valuations, $\longrightarrow \subseteq \mathsf{Vals}_V \times A \times \mathsf{Vals}_V$, where $A = I \dot\cup O \dot\cup L$, and $\forall i \in I, \mathsf{val}_V \in \mathsf{Vals}_V . \mathsf{val}_V \xrightarrow{i}$, and $\mathcal{T}$ is a set of V-trajectories that satisfy:

1. If $\mathsf{val}_V \in \mathsf{Vals}_V$ there is a $\tau \in \mathcal{T}$ from $[0,0]$ to val_V *(existence of point trajectories)*,
2. If $\tau \in \mathcal{T}$ and $\tau' \preceq \tau$ then $\tau' \in \mathcal{T}$ *(prefix closure)*,
3. If $\tau_0 \frown \tau_1 \in \mathcal{T}$ and $\tau_1^0 = \tau_0^{ltime}$ then $\tau_1 \in \mathcal{T}$ *(suffix closure)*,
4. If $\tau_0, \tau_1, \tau_2, \ldots$ is a sequence of trajectories, each in $\mathcal{T}$, such that each τ_i is closed and $\tau_i^{ltime} = \tau_{i+1}^0$ then $\tau_0 \frown \tau_1 \frown \tau_2 \frown \cdots \in \mathcal{T}$ *(concatenation closure)*, and,
5. If $\mathsf{val}_V \in \mathsf{Vals}_V$ there is a $\tau \in \mathcal{T}$ which is either open with $ltime(\tau) = \infty$, or closed and $\exists l \in O \dot\cup L . \tau^{ltime} \xrightarrow{l}$ *(time-passage enabled)*. ■

Definition B.5.1, and particularly the five axioms, is more or less directly from the theory of TIOA [KLSV06, §§4.1 and 6.1], where the development is more detailed and the input/output distinction is introduced separately. Like basic IOA, Definition B.1.1, the states and structure of TIOA are described in terms of a finite set of variables, and there is an input-enabled discrete transition relation. Unlike basic IOA, there is an additional element, the set of trajectories, describing when time may pass and how variables evolve as it does.

The clauses in Definition B.5.1 for prefix and suffix closure are analogues of the time interpolation axiom in Definition 2.2.1, and similarly for concatenation closure and time additivity. The requirement that there be a point trajectory for each state is a technicality that simplifies the definitions of executions and traces of TIOA; given below. The input-enabling requirement on discrete transitions is characteristic of the input/output approach. Time passage enabling excludes certain unrealistic models as discussed in §2.2.3.4.

Definition B.5.2
An (A,V)-sequence $\tau_0, a_0, \tau_1, a_1, \tau_2, \ldots$ is an *execution* of a TIOA $(V, \mathcal{V}_0, \longrightarrow, \mathcal{T})$ over I, O, and L if $\tau_0^0 \in \mathcal{V}_0$, every $\tau_i \in \mathcal{T}$, every $a_i \in I \dot\cup O \dot\cup L$, and if τ_i is not the last trajectory then $\tau_i^{ltime} \xrightarrow{a} \tau_{i+1}^0$. ■

Definition B.5.3
A *trace* of an execution α of a TIOA over I, O, and L is the sequence $\alpha \lceil (I \dot\cup O, \varnothing)$. ■

Appendix C

Preorders

There are both theoretical and practical reasons for considering relations between process models; and there is a large amount of literature on the subject, particularly on the various distinctions made by different notions of equivalence [Gla90, Gla93]. A broad overview, rather than a comprehensive survey, is provided in this appendix. Essential intuitions are presented rather than detailed technicalities.

Reasons for studying relations between models are outlined in §C.1, followed by a summary of some basic concepts in §C.2. In §C.3, trace-based process relations that expose increasing amounts of the internal structure of models are discussed, which lead to simulation techniques that are defined directly in terms of internal structure §C.4. Some remarks on preorders for process algebras and IOA are made in §C.5.

The TTSs of §2.2.2 are not explicitly addressed. It has been noted [LV96] that the standard concepts are directly applicable when delay transitions are treated as discrete transitions but that the proofs are more difficult.

C.1 Motivation

Techniques for relating models to one another are important to semantic theory, where arbitrary models are considered; to system development, where models of increasing detail may be created between a specification and an implementation; and to verification, where increasingly abstract models of an implementation are often a prerequisite for analysis, and where relationships with models derived from properties or stated as ideal are of prime importance.

C.1.1 Semantic theory

Central to the semantic theory of any formalism is a definition of how arbitrary models are to be compared. Such definitions express which features of a model are considered important enough to distinguish, and thereby also which are to be ignored. They guide the definition of operators, for instance to ensure compositionality, and determine what can be expressed and how precisely. They also provide a means for judging and comparing multiple semantic constructs, for instance, ensuring a correspondence between operational and denotational descriptions.

C.1.2 Design and development

Development can, at least ideally, proceed from a high-level model of a design through successive refinements to a detailed implementation model. This allows development to begin with an abstract model that is easily understood or obviously correct and to gradually add the details required of an implementation, such as distribution over a network, specific protocols, error handling, more efficient data structures, or security. The different models may be expressed in the same formalism, for instance as terms of a process algebra (see Appendix A), or as IOA (see Appendix B), or as formulas of

the TLA$^+$ (see §B.4). Showing or ensuring relations between successive pairs of models guarantees that certain properties of the most abstract model also hold in the final concrete one.

C.1.3 Verification

Verification often amounts to showing a relationship between models. One model may be designated as a *specification* and another as an *implementation*. The latter is typically an abstraction of the step-by-step behaviour of a concrete implementation, while the former may be less operational. A specification need not be comprehensive, and may permit many different implementations. It may, indeed, state only a single, specific *property* that is desired of an implementation model. Properties such as deadlock-freedom and mutual exclusion are typical examples.

In many approaches, specifications and properties are stated in a different form to implementation models. For instance, in CSP specifications are stated as predicates over free variables representing system traces and refusal sets [Hoa85], and similarly in IOA where a *trace property* [Lyn96, §8.5.2] groups explicit sets of input and output actions with a set of traces, possibly expressed in linear temporal logic, over those actions. Specification and implementation models may be translated into the same form for comparison. For instance, the semantics of a logic used for specification and the semantics of a modelling language may both be given in the same domain. Or, a proof system or algorithm may allow relationships between the two to be shown without recourse to explicit translation and comparison. In other approaches models are unnecessary: for instance, in TLA$^+$ specifications are stated in the same language used to express system details and logical implication is the relationship of choice [Lam02]; as another example, the ACP insists on model-independent reasoning, see §A.6.

An alternative to stating properties in a separate specification language is to construct a system model whose correctness is obvious or readily verifiable, and then to compare it to a more detailed model. For example, a CCS model of a network protocol could be compared to a simpler First-in-First-Out (FIFO) buffer which obviously has the desired properties of not losing messages, nor duplicating them, and of preserving their order [Mil89, Chapter 1]. In CCS (and ACP) such comparisons involve the introduction of τ-steps, see §A.2, to abstract from low-level details. Comparing the models requires ignoring τ-steps while ensuring that the relevant properties of each model also hold of the other; the subtleties of this task require careful attention.

A complex verification proof can be decomposed, just like a complex programming task, into smaller subproblems whose individual solutions can be combined to give a complete solution [Lyn96, §8.5.5]. Ensuring that the extracted models are related to the original model is essential for sound analysis. Such proof techniques rest on properties of operators in the modelling language, for instance that the synchronous product of CSP processes is effectively restriction and conjunction in the underlying model [Hoa85, §3.9] and similarly for IOA composition [Lyn96, Theorems 8.1–8.3], and the existence of relations between models [LV95b].

C.2 Technical Background

The different ways of comparing models of processes are built on the basic mathematics of relations for ordering and equivalence.

A relation R on a set A is a subset of the cross-product: $R \subseteq A \times A$. Two elements $a, a' \in A$ are related by R if $(a, a') \in R$, written $a \mathrel{R} a'$. The relations used for comparing processes satisfy certain structural restrictions.

Definition C.2.1
A *preorder* $\preceq$ is a relation satisfying:

- *(reflexivity)* $\forall a \in A.\, a \preceq a$
- *(transitivity)* $\forall a, b, c \in A.\, a \preceq b \wedge b \preceq c \implies a \preceq c$ ■

There are two common specialisations of a preorder relation. In a partial order, two elements are equal if the relation holds between them in both directions. In an equivalence relation, the relation either holds in both directions or not at all.

Definition C.2.2
A *partial order* $\preceq$ is a preorder satisfying:

- *(antisymmetry)* $\forall a, b \in A.\, a \preceq b \wedge b \preceq a \implies a = b$ ■

Definition C.2.3
An *equivalence* $\preceq$ is a preorder satisfying:

- *(symmetry)* $\forall a, b \in A.\, a \preceq b \implies b \preceq a$ ■

Thus a relation is a preorder if it is reflexive and transitive. Additionally, it may be termed a partial order if it is antisymmetric, or an equivalence if it is symmetric. Any preorder is readily extended to a specific equivalence relation.

Definition C.2.4
The *kernel* $\equiv$ of a preorder $\preceq$ is the relation that holds between two elements a and b iff $a \preceq b \wedge b \preceq a$. It is an equivalence relation. ■

Elements are not distinguished by the kernel relation if the underlying preorder holds between them in both directions; the resulting equivalence classes are partially ordered by the underlying preorder since it is antisymmetric with respect to them.

The relations defined between models are usually partial orders or equivalences because their respective constraints express the minimum expectations of such relations. Partial orders are used to define when one model is better than another for some purpose, such as for implementation or analysis. Equivalences are used to define the differences between models that are important and those which may be ignored. In most cases it is sufficient to define a preorder on a class of models; its kernel then divides the models into a partial ordering of classes of equivalent models.

In most modelling frameworks, models are built by combining smaller models using operators from a fixed set. As described in §C.1, a large model may be developed or analyzed in terms of its components. For instance, a component may be replaced with a more detailed version as a design is gradually refined into an implementation, or a complex component may be replaced with a more abstract version to make analysis feasible. The ability to predict the effect of replacing an arbitrary component with an equivalent or better one, that is to reason *compositionally*, rests on the interaction between the chosen relation on models and the set of operators from which the models are built. One way to make this idea precise is by defining, abstractly, those contexts in which a single component could find itself operating.

Definition C.2.5
For a fixed set of operators Σ, a *context* $C_\Sigma[\_]$ is a term built from those operators where the only variable $\_$ occurs once. The substitution of another term t for the variable in context $C_\Sigma[\_]$ is written $C_\Sigma[t]$. ■

Not only are contexts useful in the technicalities of definitions and proofs, they also give an abstract representation of the 'environment', which is of central importance to models of reactive systems.

Definition C.2.6
A relation R is a *congruence* for operators in Σ iff for all processes p and q, $p\,R\,q$ implies $\forall C_\Sigma[\_].\, C_\Sigma[p]\,R\,C_\Sigma[q]$. ■

Having a preorder that is a congruence for the operators of a modelling framework means that replacing one component of a model with a better component, according to the ordering, gives a better model. Replacing one component with an equivalent component gives a model that cannot be distinguished by those features discernible through the equivalence from the original. This requirement is so fundamental that relations are chosen so as to be congruent for desired operators, and operators are

defined so that a desired relation is a congruence for them; it allows properties of the whole to be inferred from properties of the parts.

Many preorders have been proposed. There are broad philosophical differences about which features should be distinguished and which ignored. There are also smaller technical differences on how precisely this should be done. And there are adjustments to ensure congruence with proposed operators. Relations can be compared with one another by considering their relative granularities [Gla90, Gla93].

Definition C.2.7
Given relations R_f and R_c on a set A, R_f is *finer* than R_c, or equivalently R_c is *coarser* than R_f, iff $\forall a, b \in A.\, a\, R_f\, b \implies a\, R_c\, b$. ■

The concept of relative granularity for two notional equivalence classes $=_f$ and $=_c$ is illustrated by Figure C.1. The solid lines mark divisions between $=_c$ classes. Divisions between $=_f$ classes are marked by both the solid and dotted lines. Two elements related by the finer equivalence $=_f$ will also be related by the coarser one $=_c$. Two elements considered equal in the coarser equivalence may or may not be related in the finer equivalence.

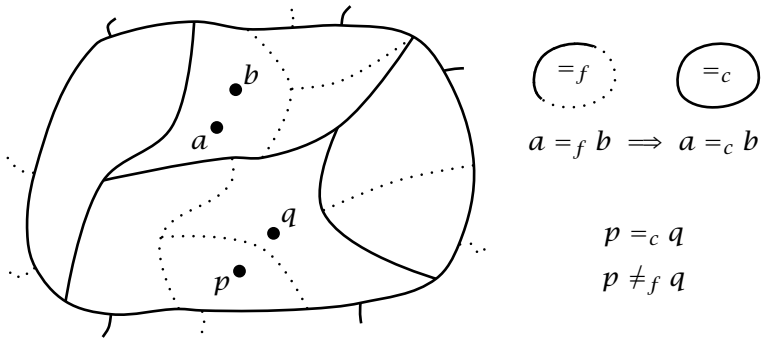


Figure C.1: Relative granularity of equivalence classes

Every modelling framework designates a particular preorder as ideal for comparing the models that it can express. Although the various preorders are defined against different semantic models, they can be characterised and thereby compared by their power to distinguish labelled transition systems, Definition 2.1.1, specifically process graphs, Definition 2.1.3 [Gla90, Gla93]. A chosen preorder can be considered the semantics for a framework because from it suitable models can be derived. A semantics N can be characterised as a function $\mathcal{O}_N$ that maps a process graph to the set of observations that may be made by experimenting or testing it externally [Moo56, Gla90, Hen88]. A semantic preorder on process graphs is then defined $p \sqsubseteq_N q$ iff $\mathcal{O}_N(p) \subseteq \mathcal{O}_N(q)$, the kernel gives $p =_N q$ iff $\mathcal{O}_N(p) = \mathcal{O}_N(q)$. To properly characterise process graphs with τ-steps, more sophisticated notions of possible observability [Gla93] and preorders based on must-testing [Hen88] are necessary, but they will not be further discussed.

The observational characterisations can be ordered by relative granularity which, since not all pairs of preorders can be said to be finer or coarser than one another, gives a lattice called the *linear time–branching time spectrum* [Gla90, Gla93]. It is so named because the coarsest relation compares process graphs based only on the linear runs they may make, whilst the finest relation compares their computation trees—that is, all potential future behaviours at each step of an execution. A simplified version of the lattice containing the eight semantics to be discussed over the following subsections, from 155 more precise definitions [Gla93], is shown in Figure C.2. A line between two names indicates that the upper is strictly finer than the lower; lines are transitive. A practical advantage of such a lattice is that once it is known which equivalence distinguishes models sufficiently, it is immediately evident which finer equivalences would also do so. Some equivalences may be more desirable than others due to their simplicity of definition, their congruence for a set of operators, or because there exist complete proof techniques or practical decision procedures for them. There are

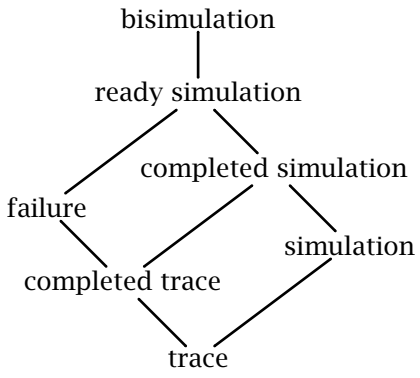


Figure C.2: Simplification of the linear time–branching time spectrum [Gla90, Gla93]

advantages for the theory too; the characterisation of the various semantics in terms of observations based on testing isolates the central concepts and insights from the details of particular approaches and unifies the underlying ideas.

It is generally agreed that models of practical systems should only be distinguished by the observations that can be made externally, ignoring irrelevant details of internal structure, and compromising only to achieve congruence with operators. But there are different ideas about what can be observed externally and which details are irrelevant.

Thought experiments on machines with various combinations of buttons, lights, and displays are one way to conceptualize observations [Gla90]. They can be adapted to understand the practical consequences for developing or trouble-shooting a real system that has been developed from or as a model in a particular semantics.

The remaining subsections consider each of the semantics of Figure C.2 in turn, working from the coarsest upward.

C.3 Traces

The trace preorder is the coarsest in the linear time–branching time spectrum. Traces are defined in Definition 2.1.9.

Definition C.3.1
The two process graphs p and q are ordered by *(partial) trace inclusion* $p \sqsubseteq_t q$ iff $\mathit{traces}(p) \subseteq \mathit{traces}(q)$. ■

The notion of observation $\mathcal{O}_t$ is the *traces* function itself. The kernel of the preorder is called *trace equivalence*. Two processes are *trace equivalent* exactly when their trace sets are identical; coarser notions of equivalence are not considered because there is little sense in identifying reactive processes that have different trace sets.

There are appealing reasons for using trace semantics. It is truly an external view of model behaviour; all internal structure, including τ-steps, is ignored leaving only the set of all linear runs that can be observed. The definition is relatively simple.

Trace semantics is a congruence for certain sets of operators; broadly, those that can be defined using SOS rules with premises that are positive and where each variable occurs only once [Blo94].[1] The former constraint precludes predicates on actions that cannot happen, the latter prevents discernment of the branching structure of a process. The basic operators of CCS, CSP, and ACP meet these restrictions.

A set of traces expresses all action sequences that may be performed, but nothing about which actions or subsequences are inevitable after observing an initial trace prefix. One classic example is the set of traces $\{\epsilon, a, ab\}$ which is a trace model for the process graphs, written as CCS expressions, $(a.b.\mathbf{0})$ and $(a.b.\mathbf{0} + a.\mathbf{0})$. The two processes are thus trace equivalent, but, under an assumption of progress, the former must eventually perform a b action, whereas the latter may stop after performing an a.

[1] More precisely: 'tyft' rules [GV92] that are 'straight' and 'patient' [Blo94].

The semantics of transformational programs distinguish partial correctness, where a program that terminates gives correct results, from total correctness, with the additional requirement that a program does eventually terminate. Termination is also desired for certain reactive algorithms, for instance that a series of network messages result in the election of a leader node. In many other cases, however, a system that stops prematurely, or deadlocks, is considered incorrect. The trace model does not distinguish a process that deadlocks from another with the same potential behaviours that does not. A simple remedy is to enhance traces with additional information about when deadlock can occur.

Definition C.3.2
Given an LTS $\mathcal{T}$ over A, where $0 \notin A$, the *completed traces*, written $ctraces(\mathcal{T})$, comprise $traces(\mathcal{T})$ and additional elements $a_0, a_1, \ldots, a_n, 0$ for each trace $a_0, a_1, \ldots, a_n$ that corresponds to an execution ending in a state with no outgoing transitions. ■

Definition C.3.3
The two process graphs p and q are ordered by *completed trace inclusion* $p \sqsubseteq_{ct} q$ iff $ctraces(p) \subseteq ctraces(q)$. ■

Completed traces are a simple form of *decorated trace* where externally visible sequences of actions are augmented with additional information about the internal structure of the generating model, namely whether it is in a deadlock state. They allow models to be distinguished based on differences in deadlock and termination, for example $ctraces(a.b.\mathbf{0}) = \{\epsilon, a, ab, ab0\} \neq \{\epsilon, a, ab, a0, ab0\} = ctraces(a.b.\mathbf{0} + a.\mathbf{0})$, but they mandate an increased capacity for observation. One could ask whether an observer really could always distinguish a deadlocked system from one that is simply slow, particularly in the asynchronous setting where (almost) nothing is known of relative execution rates. The issue of deadlock in concurrent systems is important enough to trump such philosophical objections.

Operators defined on a completed trace model, or any finer model, can utilise the additional structural information. An example is true sequencing, $P;Q$, where process Q begins execution as soon as P terminates or deadlocks. Such operators may be expensive or impractical to implement since they require a deadlock detection algorithm [Blo94]. While they simplify the expression of certain designs, which is an advantage for specification, the ramifications for eventual implementations, particularly of embedded systems, cannot be ignored.

It is also possible to distinguish successful termination from deadlock [Gla90, §19].

If a particular system were shown to be free of deadlock states and a component were replaced with an equivalent component, one would expect the resulting system to also be free of deadlock. But this is only so when the notion of equivalence is a congruence for the operators used to describe the system. Unfortunately, completed traces are not congruent for common operators, like the CSP synchronous product or the CCS restriction operator, which means they are not a suitable model.

Given a relation that is considered ideal for a fixed modelling language but for a lack of congruence, the general solution is to consider all finer relations that give congruence for the language and from them to choose the coarsest. That is, find a new relation that distinguishes per the original except when necessary to ensure that operator mappings respect the equivalence classes of the new kernel. The new relation makes just enough additional distinctions to avoid 'being tricked' by the operators of the calculus. By definition, any relation finer than completed trace inclusion also distinguishes processes that differ in possible traces or deadlock states.

The coarsest relation that is a congruence for the operators of CSP is the failure preorder,[2] refer Figure C.2.

Definition C.3.4
Given two process graphs p and q, they are related by *failure inclusion* $p \sqsubseteq_f q$ iff $failures(p) \subseteq failures(q)$. ■

Failures are another form of decorated trace. They are more discriminating than completed traces. Rather than indicate when no further actions are possible, they

[2] Only divergence-free processes are considered.

record exactly which actions are not possible, see §A.6 for a precise definition. They expose even more information about the internal structure of a model.

Failure sets are less relevant for input-enabled formalisms, like IOA, where the intersection of failures and inputs will always be empty, and, besides, such formalisms can only be placed in contexts that never restrict local or output actions [Vaa91, Definition 5.6]; that is, parallel composition is only defined for compatible automata (Definition B.1.2). Alternately, the completed and partial trace sets of an IOA are always identical because deadlock states are impossible[3] because there must be a transition for every input from every state. So, rather than deadlock states, the states of interest in an input-enabled setting are those without outgoing local or output transitions, or equivalently, those where only input transitions are enabled. Such states, where a process waits indefinitely for stimulus from its environment, are termed *quiescent*. Quiescent trace sets can be defined similarly to completed trace sets—including partial traces, but with quiescent states marked rather than deadlock states—and the resulting preorder is substitutive for any input-enabled calculus, even when fairness is considered [Vaa91].

Definitions C.3.1 and C.3.3 describe sets where each element is finite. The trace set of a process able to continually and indefinitely perform a-actions will contain every finite sequence: a, aa, aaa, ...; but no single infinitely long sequence of as. That is, it will include every sequence in a^* but not the sequence a^ω. In an *infinitary trace semantics* observation sets comprise both partial and infinite traces. For *infinitary completed trace semantics*, completed and infinite traces are included; they suffice to recover partial traces [Gla90, §3].

Thinking about infinitary traces requires care because everyday intuitions do not necessarily apply. Infinitary traces are also difficult to justify observationally since an observation or test may never actually finish. Despite these drawbacks, they are sometimes a convenient and necessary abstraction, allowing what would be an evolving behaviour to be treated as a static object. Sets of infinite traces can be described as formulas of linear temporal logic (for which they are a natural model) or ω-automata.

Although not evident from Figure C.2, infinitary trace semantics is strictly finer than (finitary[4]) trace semantics [Gla93, Figure 2].

For certain classes of LTSs, finitary trace sets are valid approximations of infinitary behaviour and concluding that two LTSs are ordered or equal in terms of their finitary traces implies the same of their infinitary traces, which means that only the finitary traces need ever be considered. This property is termed variously *finite distinguishability* [MP92, §2.1] or *limit closure* of infinitary trace sets. There is a simple restriction on LTSs that ensures this property.

Definition C.3.5
An LTS $(S, S_0, \longrightarrow)$ over A has Finite Internal Nondeterminism (FIN) if S_0 is finite and for every $\sigma \in A^*$ and $s \in S$ the set $\{s' \mid s \stackrel{\sigma}{\Longrightarrow} s'\}$ is finite. ■

An LTS with FIN is *image finite*, meaning that, after any finite trace, it can only be in one of a finite number of states. The finitary traces of an LTS with FIN are sufficient to determine its infinitary traces.[5]

Although finitary traces suffice for models of 'real' systems, infinitary trace sets are required for more abstract descriptions, like those that express fairness [MP92, §2.10].

C.4 Simulations

When models are compared in terms of traces, no direct reference is made to internal states or interconnecting transitions, even as increasingly decorated traces discern

[3] Provided the set of inputs is not empty: $I \neq \emptyset$.

[4] The adjective 'finite' is preserved to distinguish different versions of failure semantics [Gla93].

[5] This result is shown with König's Lemma, see for example [KLSV06, Lemma 4.18] and [Gla90, Proposition 2.4].

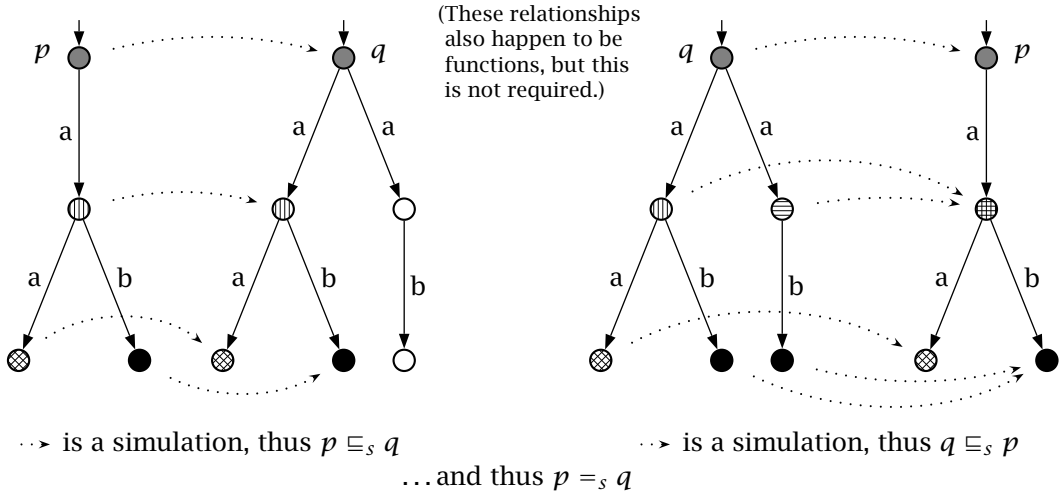


Figure C.3: Simulations between two processes p and q.

more internal differences. A simulation relation, in contrast, is defined on the states of models based on the actions possible and the states resulting from each.[6]

The basic simulation relation and two extensions, completed simulation and ready simulation, are presented in this subsection, though without treating τ-transitions in detail. The existence of simulation relations between two models implies a close correspondence between their behaviours, which both leads to distinct new preorders as well as giving a partial proof method for decorated trace preorders. Although simulation preorders arise from the comparison of internal structure, they can also be characterised in terms of external observations and experiments.

A relation between the states of two models is termed a simulation relation if it satisfies an asymmetric constraint on possible actions and future states.

Definition C.4.1
Given two LTSs $(S_1, S_{0_1}, \longrightarrow_1)$ and $(S_2, S_{0_2}, \longrightarrow_2)$ over a single set of actions A, a relation between their states $R \subseteq S_1 \times S_2$ is a *simulation relation* provided that whenever $s_1 \, R \, s_2$ and $s_1 \xrightarrow{a}_1 s_1'$ then $\exists s_2' . \, s_2 \xrightarrow{a}_2 s_2'$ and $s_1' \, R \, s_2'$. ■

Between two models there may be many such relations; the existence of at least one that relates initial states is enough to guarantee a close correspondence between the model's behaviours.

Definition C.4.2
Two process graphs p and q, with respective initial states p_0 and q_0, are ordered by *simulation* $p \sqsubseteq_s q$ iff there exists a simulation relation R between their states such that $p_0 \, R \, q_0$. ■

Simulation relations and the associated preorder are fundamental and occur frequently in the literature [Gla90, §8][Par81].

Given process graphs p and q where $p \sqsubseteq_s q$, q could be considered as 'better' because it can simulate any execution of p such that both executions exhibit the same external behaviour. That is, a requirement for a machine behaving per p would also be satisfied given one behaving per q. Alternatively, q could be seen as a specification of allowed behaviours and p an implementation. To meet the specification, every action of p must be checked against q to see both that it is allowed and that all behaviours from the subsequent state are also permitted. Note that, due to the asymmetry of simulation relations, p is not required to match all possible behaviours of q.

The example in Figure C.3 shows two processes p and q. At left, the dotted lines show a simulation relation from the states of p to the states of q. The relation, in

[6] The term 'simulation' refers to a relation between states; 'simulates' to the associated preorder; 'simulation equivalence' to the full equivalence relation; and, 'similar' to two processes that are equal according to that relation.

this case a function, shows how every execution of p can be simulated by one in q to produce the same trace. Other simulation relations are possible and any one is sufficient to show that p is related to q by the simulation preorder $p \sqsubseteq_s q$. At right, a simulation relation is shown in the other direction: from q to p. Thus, these two processes are simulation equivalent, or similar, $p =_s q$.

Although the simulation preorder is defined through a relation on internal states it can be characterised in terms of observations, provided experimenters can somehow save and restore the state of machines; perhaps by allowing machines to be replicated during an experiment or by equipping them with undo buttons [Gla90, §8]. Different preorders arise if machines can only be replicated a finite number of times from a given state, or if replication is only allowed in states that have no outgoing τ-transitions [Gla93].

A replication or undo mechanism allows experimenters to reconstruct the branching structure of a machine by performing multiple experiments from each reachable state (or at least from those that cannot be internally preempted). Such mechanisms may not be realistic for all systems, they serve rather as thought experiments on models and as a way to relate simulation with other preorders. The observations made during such experiments can be characterised as *branching traces* where, when expressed as formulas of modal logic, multiple experiments from a replicated state are put together with nested conjunction [HM80]. The notion of observation for simulation $\mathcal{O}_s$ is then sets of branching traces, with inclusion and equality of sets for its preorder and equivalence relations [Gla90].

As replication (conjunction) can be added to traces to give basic simulation, so can it be added to decorated trace preorders to give corresponding notions of simulation.

Completed simulation, refer Figure C.2, distinguishes processes more finely than do either of the simulation or completed trace preorders. It is defined by constraining the definition of basic simulation to relations that only pair deadlock states in one model with deadlock states in the other.

Definition C.4.3
Given two LTSs $(S_1, S_{0_1}, \longrightarrow_1)$ and $(S_2, S_{0_2}, \longrightarrow_2)$ over a single set of actions A, a relation between their states $R \subseteq S_1 \times S_2$ is a *completed simulation relation* if it is a simulation and whenever $s_1 \, R \, s_2$ then $(\forall a \in A.\, s_1 \stackrel{a}{\not\longrightarrow}_1) \iff (\forall a \in A.\, s_2 \stackrel{a}{\not\longrightarrow}_2)$. ■

Definition C.4.4
Two process graphs p and q, with respective initial states p_0 and q_0, are ordered by *completed simulation* $p \sqsubseteq_{cs} q$ iff there exists a completed simulation relation R between their states such that $p_0 \, R \, q_0$. ■

Processes are observed in terms of completed simulation by enhancing completed trace observation with the ability to replicate processes; giving sets of branching traces where a leaf may be marked as complete if the process could deadlock after performing the sequence of actions in the branch leading to it.

The ready simulation [Gla90, §9][LS91, BIM95] preorder is finer than either of the completed simulation or failure preorders. A ready simulation is constrained to pairs of states from which the same actions are possible.

Definition C.4.5
Given two LTSs $(S_1, S_{0_1}, \longrightarrow_1)$ and $(S_2, S_{0_2}, \longrightarrow_2)$ over a single set of actions A, a relation between their states $R \subseteq S_1 \times S_2$ is a *ready simulation relation* if it is a simulation and whenever $s_1 \, R \, s_2$ then $\forall a \in A.\, (s_1 \stackrel{a}{\not\longrightarrow}_1 \iff s_2 \stackrel{a}{\not\longrightarrow}_2)$. ■

There are at least two other ways to state the key constraint, each offering additional insight. First [LS91], by checking transition possibilities back in the other direction, that is requiring that states are only related $s_1 \, R \, s_2$ if $s_2 \stackrel{a}{\longrightarrow}_2$ implies $s_1 \stackrel{a}{\longrightarrow}_1$ for all actions a. Second [Gla90], by defining the set of transitions enabled from a state $I(s) = \{a \mid s \stackrel{a}{\longrightarrow}\}$ and requiring that if $s_1 \, R \, s_2$ then $I(s_1) = I(s_2)$.

Definition C.4.6
Two process graphs p and q, with respective initial states p_0 and q_0, are ordered by *ready simulation* $p \sqsubseteq_{rs} q$ iff there exists a ready simulation relation R between their states such that $p_0 \, R \, q_0$. ■

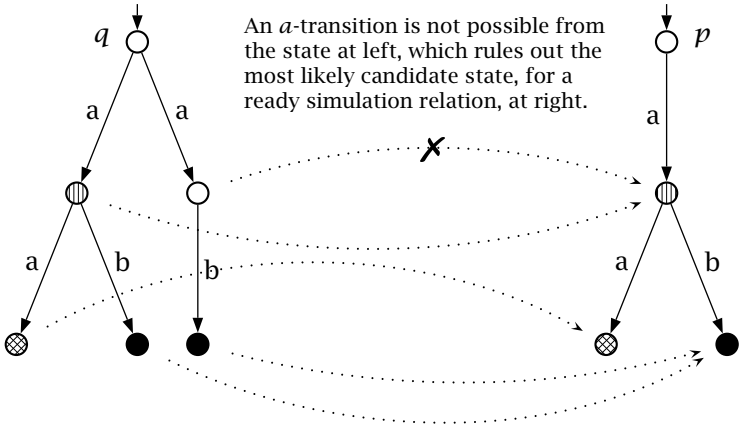


Figure C.4: Failure to find a ready simulation between the processes of Figure C.3

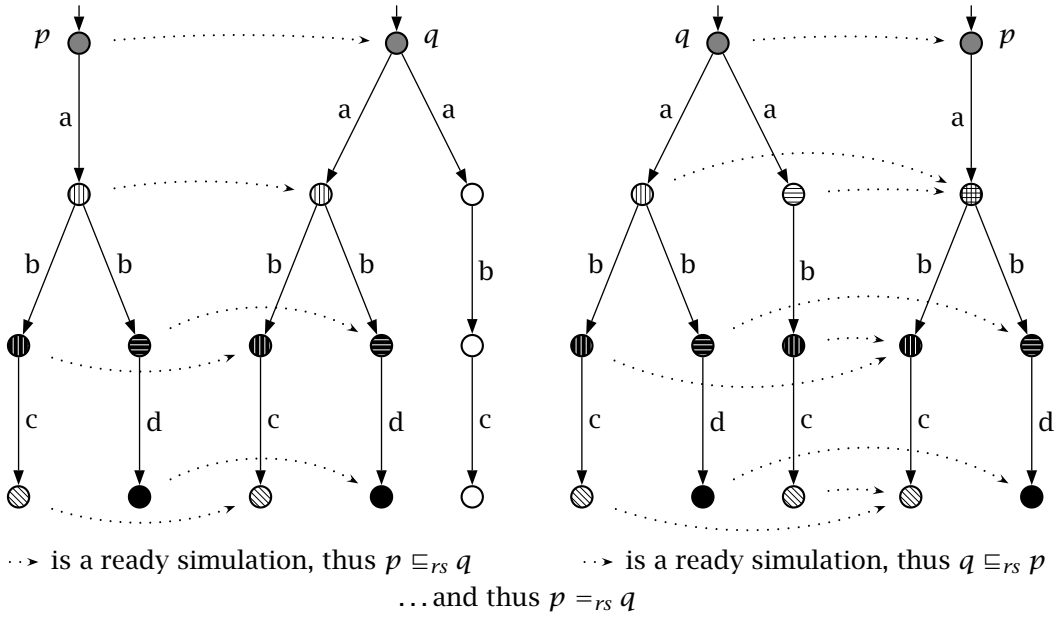


Figure C.5: Ready simulations between two processes p and q [LS91, from Figure 1].

There are several different but equivalent observational characterisations of ready simulation [Gla90, §9]. For the simplification presented in Figure C.2, it suffices to consider the combination of failures (Definition A.6.2), with replication, giving branching traces where each state is labelled with a refusal set (Definition A.6.1).

Ready simulation is more discriminating than both basic and completed simulation. The example in Figure C.4 depicts the two processes that were earlier shown to be simulation equivalent, but for which only one of the simulation relations is also a ready simulation. The figure shows that the other relation does not qualify because there is a state in q (at left) from which less actions are possible than from any otherwise potentially related state in p (at right).

An example where two processes are ready simulation equivalent is given in Figure C.5. The first direction $p \sqsubseteq_{rs} q$ is obvious because p is a subtree of q and no additional actions are possible from the state of q where the additional branch is attached. The other direction $q \sqsubseteq_{rs} p$ is also readily established. Note that the right-hand a-transition of q leads to a state from which only a b-transition followed by a c-transition is possible, whereas in the corresponding state of p in the ready simulation, there are two b-transitions with different destination states. Ready simulation

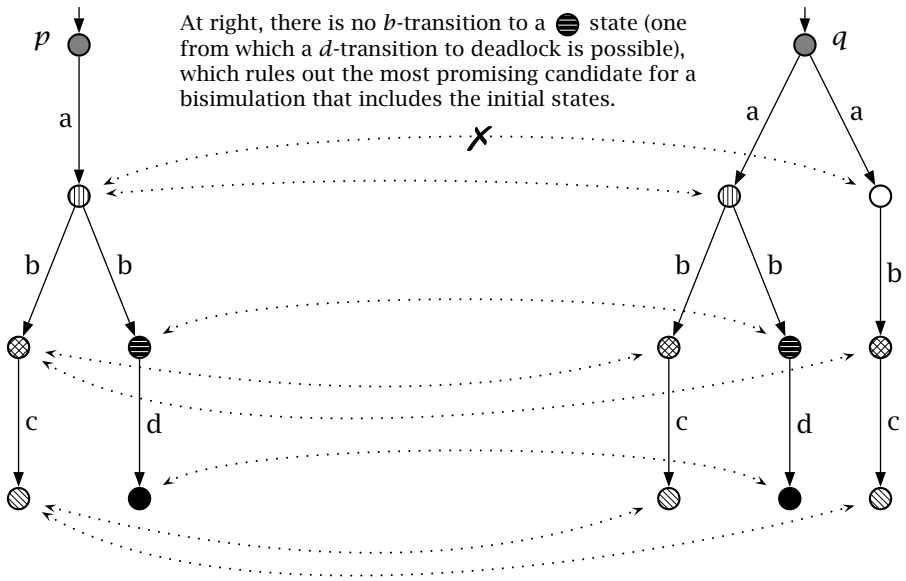


Figure C.6: Failure to find a bisimulation between the processes of Figure C.5.

only requires that all possible actions are matched, not all possible derivatives.

Requiring that related states match all possible derivatives of one another gives strong bisimulation [Mil89].

Definition C.4.7
Given two LTSs $(S_1, S_{0_1}, \longrightarrow_1)$ and $(S_2, S_{0_2}, \longrightarrow_2)$ over a single set of actions A, a relation between their states $R \subseteq S_1 \times S_2$ is a *strong bisimulation* provided, for all $a \in A$, that $s_1 \, R \, s_2$ implies:

1. whenever $s_1 \xrightarrow{a}_1 s_1'$ then $\exists s_2' . \, s_2 \xrightarrow{a}_2 s_2'$ and $s_1' \, R \, s_2'$.
2. whenever $s_2 \xrightarrow{a}_2 s_2'$ then $\exists s_1' . \, s_1 \xrightarrow{a}_1 s_1'$ and $s_1' \, R \, s_2'$. ■

In contrast to the other simulations, bisimulations are symmetric; reversing the order of the two clauses does not change anything. A bisimulation found between two models in one direction is also a bisimulation between them in the reverse direction. Thus there is no preorder relation, only an equivalence.

Definition C.4.8
Two process graphs p and q, with respective initial states p_0 and q_0, are *bisimilar*, $p =_b q$, iff there exists a bisimulation relation R between their initial states $p_0 \, R \, q_0$. ■

The two models that were earlier shown to be ready similar are presented again in Figure C.6, where it is shown that they are not bisimilar. After an a-transition, the process at right q cannot always match all future possibilities of the process at left p, even though, q can, from both a-descendants, match all immediate possibilities for action. Bisimulation is considered to be a branching time equivalence because of this requirement on all future possibilities, even though any run of a system must resolve each branching point to a single choice.

Powerful experiments are required to characterise bisimulation [Gla90]. In addition to being able to replicate processes, an observer must have the power to ensure that all possible transitions for an action at a state have been tested. This grants the power to observe not just which actions cannot occur at a given point, as per ready simulation, but also which future branchings are not possible, or equivalently which future branchings are inevitable. For example, in Figure C.6, one could observe of p that initially it is not possible to do an a such that all b actions then lead to states where c is inevitable (equivalently actions in $A \setminus \{c\}$ are impossible), while from q it is possible to do an a

to a state where all b actions must be followed by a c. Such observations can be stated more clearly as formulas in the Hennessy-Milner Logic [HM80], sets of which give an observational characterisation of bisimulation $\mathcal{O}_b$.

A strong bisimulation does not treat τ-transitions differently to other actions, contrary to their distinguished meaning. But a weak bisimulation [Mil89, §5] does, allowing a τ-transition in one process to be matched by none, or any number of τ-transitions in the other; and for any other type of transition the matching transition may be preceeded and trailed by any number of τ-transitions provided the final states, but not necessarily any intermediate ones, are also in the relation.[7] Weak bisimulation equivalence is not a congruence for the choice operator due to the preemptive power of τ-transitions; thus a refinement, observational congruence [Mil89, §7], that requires explicit matching of τ-transitions from the initial state, is often preferred.

By ignoring τ-transitions, weak bisimulation also ignores differences in divergent behaviour (the potential for infinite sequences of internal steps) between models; it is suggested that such issues can be addressed 'by a separate argument' [Mil89, p. 149].

The remainder of this section outlines the use of simulation preorders and bisimulation for proving relationships between models. Simulation proofs are often also an effective way to prove trace inclusion.

Proving simulation ordering or bisimulation equivalence between two models requires first proposing a relation between the states of both and then showing that it is indeed a (bi)simulation relation; that is, initial states are interrelated, transitions occur between related states, and other relevant requirements on possible actions and derivatives are met. For simulation equivalence, the proposal and proof must also be repeated in the converse direction. The approach has two important advantages. First, the existence of any relation that meets the requirements is sufficient to show ordering or equivalence; the most convenient can be chosen and state reachability can be ignored or accounted for as required, possibly by incorporating invariants [LV95b, §6]. Second, reasoning is mostly local, though τ-transitions complicate matters, to pairs of states, which simplifies analysis.

While simulation relations are defined as, and usually also presented as, relations on abstract states, like the lines between the dots in Figures C.3–C.6 reasoning proceeds in practice from more structured descriptions; like, for instance, parameterised definitions in a process algebra [Mil89], or variable valuations in IOA. Relations need not involve specific models, they can also be proved over more abstract descriptions to show desired properties of transformations within a modelling framework, or between modelling frameworks. They can justify an equational proof system for a model [Mil89, §7], see Figure A.5a, or a model for a set of equational axioms, see Figure A.5b.

Simulation proof techniques are also important for verifying trace inclusion and equivalence relations because they 'reduce reasoning about executions to reasoning about individual states and steps' [LV95b]. The simplified linear time–branching time spectrum of Figure C.2 shows that for each of the (decorated) trace preorders, there is a corresponding simulation preorder. The lines between each of the simulation and trace preorders indicate that the former are finer: simulation is a sound proof technique for trace inclusion, but it is not complete.

Simulation techniques are usually an effective way to show trace inclusion [LV95b]. The basic idea is to posit a relation between the states of two systems, and then to show that it is indeed a simulation relation. The advantage is the ability to reason in terms of states, rather than in terms of sequences of actions. Ideally, the relation between the states will be a function, in which case it is termed a *refinement*. Simulation is sufficient to show finite trace inclusion within a deterministic specification [LV95b, Theorem 3.11].

Simulation fails to show trace inclusion between models that differ in the timing of (non-deterministic) choices. The classic example is shown in Figure C.7 where a simulation exists from p to q, but not from q to p due to the latter's initial branching that

[7] Branching bisimulation [GW96] is a stricter variant where the intermediate states traversed by preceding τ-transitions are related to one another and, similarly, those traversed by trailing τ-transitions are also interrelated; branching potentials are always respected along corresponding runs of branching bisimilar models.

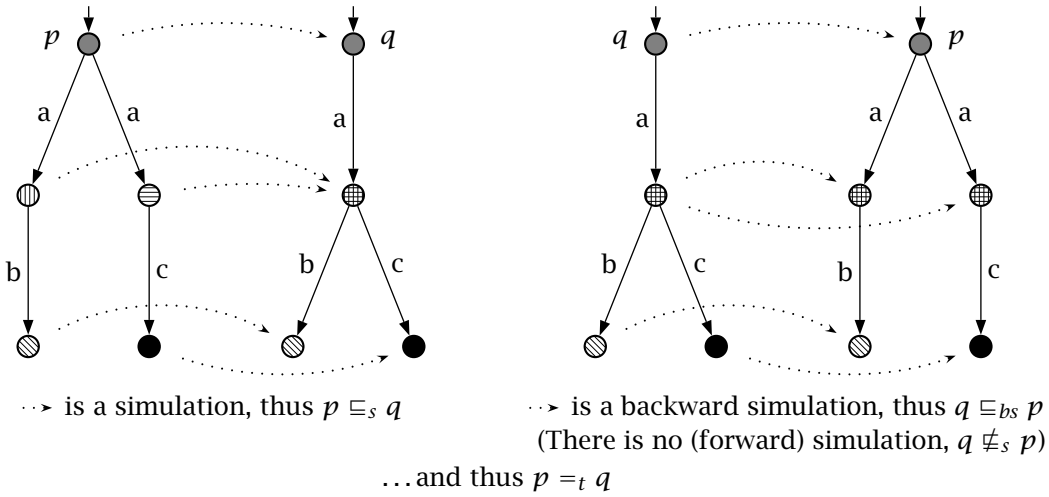


Figure C.7: Trace equivalence via forward and backward simulation.

limits later possibilities. A backward simulation overlooks such differences in branching. In the figure, the a-descendant of q is related to two states in p. The whole relation is a backward simulation because both b- and c-transitions in q are matched in p even though there each originates from a different related state. Combining forward and backward simulations gives a complete proof technique for trace inclusion, and hence equivalence [LV95b, Theorem 3.22].

It is sometimes easier to propose and prove simulation and backward simulation relations between models if history and prophecy variables are introduced [AL91, LV95b]. Such variables track properties of complete executions while maintaining the ability to reason about states and steps individually.

A simulation between models $\mathcal{I}$ and S implies that the traces of $\mathcal{I}$ are a subset of those of S, and thereby that any safety property shown of S will not be violated by $\mathcal{I}$. This property partially justifies the use of simulation in successive refinements from high-level models, be they design specifications or verified abstractions, to more detailed low-level models. As simulations imply a closer relationship between processes than trace inclusion alone, they can also help in showing that liveness properties are preserved [Lyn96, §8.5.5].

The lattice of Figure C.2 collapses for deterministic processes—no preorder distinguishes any more processes than another. This is advantageous for deterministic formalisms [MR01] where simulation is complete for trace inclusion and bisimulation for trace equivalence.

C.5 Remarks on Process Algebra and IOA

Which of the various preorders is most appropriate for comparing process models depends on several factors [Gla90, §18][Gla93, §5]. No single preorder is intrinsically better than the others and the presented modelling languages differ in which they distinguish as primary.

CSP adopts the coarsest preorder that is both a congruence for its operators and able to distinguish differences in deadlock behaviour. Coarse preorders require less imaginative testing scenarios and, thus, any distinctions made between processes are easier to justify in practical terms. For algebraic approaches, the coarsest semantics gives the most valid equations [Gla90]. For deterministic CSP trace inclusion is sufficient [Hoa85, §2.8]. But, although it is a congruence for CSP more generally, it fails to distinguish non-deterministic processes with different deadlock behaviours. The coarsest equivalence that does and that is also a congruence is failures equivalence. Distinguishing divergent behaviour is important in the CSP approach, so the preorder

of choice compares both failures and divergence sets.

As demonstrated by CSP: the coarser a preorder is, the less likely it will remain suitable if new operators or manipulations are introduced. This is one reason for verifying equivalence with the finest possible relation: the result is more likely to hold if the process is later used in an environment that makes more observations [Gla90].

For CCS, and also ACP, the equivalence of choice is bisimulation, which, because it distinguishes processes finely on details of internal structure, is congruent for all 'reasonable' process operators and contexts. This is an advantage for theoretical investigations: new operators can be introduced and trialled without risking existing results, and processes can be discriminated with accuracy. The proof techniques associated with bisimulation are convenient and they can often also be applied to show coarser equivalences. There are effective algorithms for verifying bisimulation between finite processes.

In the study of IOA more emphasis is typically placed on modelling and practical proof techniques than on concerns about algebraic properties. As for CSP, processes are related via the simplest observations possible. The restriction to input-enabled processes and output-accepting contexts means that trace sets are compositional for IOA [Vaa91]. Finitary trace inclusion is enough to show satisfaction of safety properties, which are represented as non-empty, prefix-closed, and limit-closed sets of infinitary traces [Lyn96, §8.5.3]. Satisfaction of liveness properties, which must include an infinitary extension for every finitary trace, is shown by fair trace inclusion, which means quiescent (finitary) traces and the subset of infinitary traces that meet the requirements of being fair. Divergence is less of an issue for input-enabled models: they must always respond to external stimulus.

Both TLRCS and TLA$^+$ eschew distinctions based on branching alone, and, like IOA, are less concerned about interactions of relations and operators. States, rather than actions, are central to both frameworks and thus the preorders discussed in this section have less relevance.

In the TLRCS approach, transition diagrams, shared-variable programs, message-passing programs, and petri nets are valid system descriptions [MP92, TODO], but the focus of study is on fair transition systems. Properties or interrelations are based on executions that either contain an infinite number of non-stuttering transitions, or that, after a finite suffix, consist of an infinite repetition of a (deadlock) state where the only possible transition is a stuttering self-loop [MP92, TODO]. Often only fair, strongly or weakly, executions are considered. The TLA$^+$ is similar in perspective but with less emphasis on fairness and liveness [Lam02, §8.9.5]. Stuttering is important to both approaches, and particularly the latter [Lam83], because it simplifies reasoning, separate cases for finite executions are unnecessary [Lam02, §8.9.5], and composition, where the conjunction of specifications is essentially synchronous and asynchronous transitions are matched by stuttering transitions in unrelated components.

C.6 Omissions

Perhaps the biggest omissions of this section are the may and must preorders of process testing [Hen88] and detail on abstraction (τ-transisions) and divergence [Gla93]. The importance of these topics is not in doubt, but they are not needed directly in this thesis.

Appendix D

Textual Argos

D.1 Explanation

This grammar was constructed by examining online Argos examples [Ver05]. It contains two enhancements:

1. An `after` keyword (see ⟨transition⟩) for specifying the timeout transition and parameters for temporized states.

2. Process encapsulation (`INTERNAL`) is not limited to the top level process description, but braces must be used to delimit fine-grained scopes.

D.2 Lexical details

Comments begin with a percentage symbol and extend to the next new-line character. White space is otherwise ignored. `IDENT` represents the regular expression: `[A-Za-z0-9_]+`. Matching is case-sensitive. The following keywords are reserved:

```
AUTOMATON  init  INTERNAL  from  PAR     ENDPAR
PROCESS    RAFF  STATES    to    TARGOS  ENDTARGOS
TRANS      with  after
```

D.3 Grammar

⟨program⟩ →
    `TARGOS` ⟨automata⟩ ⟨process⟩ `ENDTARGOS`
⟨automata⟩ →
    ⟨automaton⟩
  | ⟨automata⟩ ⟨automaton⟩
⟨process⟩ →
    `PROCESS` ⟨symbol⟩ `INTERNAL (` ⟨symbol list⟩ `)` ⟨process expr.⟩
  | `PROCESS` ⟨symbol⟩ ⟨process expr.⟩
⟨process expr.⟩ →
    ⟨raff⟩
  | ⟨raff⟩ `{` ⟨refinement list⟩ `}`
  | `PAR` ⟨parallel list⟩ `ENDPAR`
  | `INTERNAL (` ⟨symbol list⟩ `)` `{` ⟨process expr.⟩ `}`
⟨raff⟩ →
    `RAFF` ⟨symbol⟩
⟨parallel list⟩ →
    ⟨process expr.⟩
  | ⟨parallel list⟩ ⟨process expr.⟩
⟨refinement list⟩ →
    ε

| ⟨refinement list⟩ ⟨refinement expr.⟩
⟨refinement expr.⟩ →
⟨symbol⟩ ⟨process expr.⟩
⟨automaton⟩ →
`AUTOMATON` ⟨symbol⟩ `STATES` ⟨state list⟩ `TRANS` ⟨transition list⟩
⟨state list⟩ →
ε
| ⟨state list⟩ ⟨symbol⟩
| ⟨state list⟩ ⟨symbol⟩ `init`
⟨transition list⟩ →
ε
| ⟨transition list⟩ ⟨transition⟩
⟨transition⟩ →
⟨fromto⟩
| ⟨fromto⟩ `with` ⟨transition expr. list⟩
| ⟨fromto⟩ `after` ⟨integer⟩ ⟨symbol⟩
| ⟨fromto⟩ `after` ⟨integer⟩ ⟨symbol⟩ `/` ⟨symbol list⟩
⟨fromto⟩ →
`from` ⟨symbol⟩ `to` ⟨symbol⟩
⟨transition expr. list⟩ →
⟨transition expr.⟩
| ⟨transition expr. list⟩ `+` ⟨transition expr.⟩
⟨transition expr.⟩ →
⟨guard expr.⟩
| ⟨guard expr.⟩ `/` ⟨symbol list⟩
| `/` ⟨symbol list⟩
⟨guard expr.⟩ →
⟨symbol⟩
| `-` ⟨symbol⟩
| ⟨guard expr.⟩ `.` ⟨guard expr.⟩
⟨symbol list⟩ →
⟨symbol⟩
| ⟨symbol list⟩ `,` ⟨symbol⟩
⟨symbol⟩ →
IDENT

Appendix E

Argos Case Studies

This appendix contains textual Argos, see Appendix D, source code for the examples in Chapter 3.

E.1 Sensor failure detection

The Sensor failure detection controller is described in §3.4.1. A graphical version appears in Figure 3.14.

```
1  TARGOS
2
3  AUTOMATON Oxygen_Sensor_Mode
4    STATES
5      O2_warmup init
6      O2_normal
7      O2_fail
8    TRANS
9      from O2_warmup to O2_normal after 48 TSEC
10     from O2_normal to O2_fail with −Ego_inrange/FAIL_O2
11     from O2_fail to O2_fail with −Ego_inrange/FAIL_O2
12     from O2_fail to O2_normal with Ego_inrange
13
14 AUTOMATON O2ColdLoop
15   STATES
16     stillcold init
17   TRANS
18     from stillcold to stillcold with / O2_cold, FAIL_O2
19
20 AUTOMATON Pressure_Sensor_Mode
21   STATES
22     press_norm init
23     press_fail
24   TRANS
25     from press_norm to press_fail with −Press_inrange / FAIL_PRESS
26     from press_fail to press_fail with −Press_inrange / FAIL_PRESS
27     from press_fail to press_norm with Press_inrange
28
29 AUTOMATON Throttle_Sensor_Mode
30   STATES
31     throt_norm init
32     throt_fail
33   TRANS
34     from throt_norm to throt_fail with −Throt_inrange / FAIL_THROT
35     from throt_fail to throt_fail with −Throt_inrange / FAIL_THROT
36     from throt_fail to throt_norm with Throt_inrange
37
38 AUTOMATON Speed_Sensor_Mode
```

```
39    STATES
40      speed_norm init
41      speed_fail
42    TRANS
43      from speed_norm to speed_fail
44        with −Moving.Press_underthresh / FAIL_SPEED
45      from speed_fail to speed_fail with −Moving / FAIL_SPEED
46      from speed_fail to speed_norm with Moving
47
48  AUTOMATON SingleFail
49    STATES test init
50    TRANS from test to test with
51         FAIL_O2.−FAIL_PRESS.−FAIL_SPEED.−FAIL_THROT / SINGLE +
52        −FAIL_O2.FAIL_PRESS.−FAIL_SPEED.−FAIL_THROT / SINGLE +
53        −FAIL_O2.−FAIL_PRESS.FAIL_SPEED.−FAIL_THROT / SINGLE +
54        −FAIL_O2.−FAIL_PRESS.−FAIL_SPEED.FAIL_THROT / SINGLE
55
56  AUTOMATON ZeroFail
57    STATES test init
58    TRANS from test to test with
59        −FAIL_O2.−FAIL_PRESS.−FAIL_SPEED.−FAIL_THROT / ZERO
60
61  AUTOMATON MultiFail
62    STATES test init
63    TRANS from test to test with −ZERO.−SINGLE / MULTI
64
65  AUTOMATON Fuelling_Mode_Running
66  % −fuel_mode_disabled added to each transition to simulate whennot().
67    STATES
68      Warmup init
69      Normal
70      Single_Failure
71    TRANS
72      from Warmup to Warmup
73        with O2_cold.−gorich.−fuel_mode_disabled / fuel_mode_low
74      from Warmup to Normal
75        with −O2_cold.−SINGLE.−fuel_mode_disabled.−gorich / fuel_mode_low
76      from Warmup to Single_Failure
77        with −O2_cold.SINGLE.−fuel_mode_disabled / fuel_mode_low
78          + gorich.−fuel_mode_disabled / fuel_mode_rich
79      from Normal to Single_Failure
80        with SINGLE.−fuel_mode_disabled / fuel_mode_rich
81          + gorich.−fuel_mode_disabled / fuel_mode_rich
82      from Normal to Normal
83        with −SINGLE.−gorich.−fuel_mode_disabled / fuel_mode_low
84      from Single_Failure to Normal
85        with −SINGLE.−gorich.−fuel_mode_disabled / fuel_mode_low
86      from Single_Failure to Single_Failure
87        with SINGLE.−fuel_mode_disabled / fuel_mode_rich +
88          gorich.−fuel_mode_disabled / fuel_mode_rich
89
90  AUTOMATON Fuelling_Mode_Disruption
91    STATES
92      Running init
93      Overspeed
94      Shutdown
95    TRANS
96      from Running to Shutdown with −toofast.MULTI / fuel_mode_disabled
97      from Running to Overspeed with toofast / fuel_mode_disabled
98      from Overspeed to Running with −FAIL_SPEED.speed_under_hys.−MULTI
99      from Overspeed to Overspeed with −speed_under_hys / fuel_mode_disabled
100       + FAIL_SPEED / fuel_mode_disabled
101     from Overspeed to Shutdown
```

```
102         with −FAIL_SPEED.speed_under_hys.MULTI / fuel_mode_disabled
103       from Shutdown to Shutdown with MULTI / fuel_mode_disabled
104       from Shutdown to Running with −MULTI / gorich
105
106   PROCESS fuelcontrol
107   INTERNAL (O2_cold, SINGLE, MULTI, ZERO)
108   PAR
109     RAFF Oxygen_Sensor_Mode { O2_warmup RAFF O2ColdLoop }
110     RAFF Pressure_Sensor_Mode
111     RAFF Throttle_Sensor_Mode
112     RAFF Speed_Sensor_Mode
113     RAFF ZeroFail
114     RAFF SingleFail
115     RAFF MultiFail
116     INTERNAL (gorich) {
117       PAR
118         RAFF Fuelling_Mode_Running
119         RAFF Fuelling_Mode_Disruption
120       ENDPAR
121     }
122   ENDPAR
123
124   ENDTARGOS
```

E.2 Bang-bang temperature controller

The Bang-bang temperature controller is described in §3.4.2. A graphical version appears in Figure 3.16.

```
1   TARGOS
2
3   AUTOMATON Heater
4   STATES
5     Off init
6     On
7   TRANS
8     from Off to On with COLD.onOk / BOILER
9     from On to Off after 20 SEC
10    from On to Off with −COLD
11
12  AUTOMATON Wait
13  STATES
14    Hold init
15    Free
16  TRANS
17    from Hold to Free after 40 SEC
18    from Free to Free with / onOk
19
20  AUTOMATON Flash_led_red
21  STATES
22    Off init
23    On
24  TRANS
25    from Off to On after 5 SEC
26    from On to Off after 5 SEC
27
28  AUTOMATON Sustain_red
29  STATES
30    Single init
31  TRANS
32    from Single to Single with / RED
33
```

```
34 AUTOMATON Flash_led_green
35 STATES
36   Off init
37   On
38 TRANS
39   from On to Off with SEC
40   from Off to On with SEC / GREEN
41   from On to On with −SEC / GREEN
42
43 AUTOMATON Sustain_boiler
44 STATES
45   Single init
46 TRANS
47   from Single to Single with / BOILER
48
49 PROCESS bangbang
50 INTERNAL (onOk)
51   RAFF Heater { Off PAR
52       RAFF Flash_led_red { On RAFF Sustain_red }
53       RAFF Wait
54     ENDPAR
55     On PAR
56       RAFF Flash_led_green
57       RAFF Sustain_boiler
58     ENDPAR }
59 ENDTARGOS
```

Appendix F

Formal Timing Diagram Model

The sensor timing diagram of §4.2.2 has, at best, a semi-formal meaning. The formal model that is constructed manually in §4.3 expresses an interpretation of the diagram as a timed automaton. A model could also be created by transcribing the diagram into the syntax of a formal timing diagram language with a well-defined semantics. The semantic domain may also be timed automata, or it may be readily transformed into timed automata. The two approaches are sketched in Figure F.1. Whereas the earlier chapter presents an example of interpretation, this appendix presents an example of transcription.

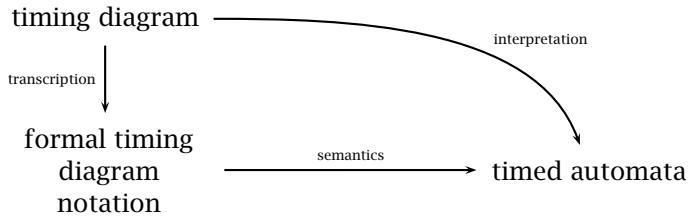


Figure F.1: Different ways of formalizing a timing diagram

Timing diagrams are so useful in engineering practice, particularly in hardware systems design, that they have been widely studied as: a design notation [Ron80], a form of specification permitting automatic synthesis [Tie91, Bor92, SD93, MAP93, FF89, FJ97], an application of model checking [AEKN00], and a formal language of intrinsic interest [KC98, Fis99].

Timing diagrams have much in common with Message Sequence Charts [RGG96, Pel02], both are tools with a practical focus, for digital signals and network protocols respectively, where each chart generally focuses on a single scenario, patterns of signal or message exchanges respectively, within a system that may comprise many such scenarios. As Message Sequence Charts express a causal (partial) ordering of atomic events, timing diagrams indicate causal relations between signal transitions.

The intent, here, is not to survey the field of 'scenario-oriented specifications', the references given here are not even exhaustive, but rather to gain some insights into the relative advantages and disadvantages of expression in formal timing diagrams compared to manual modelling in timed automata for the sensor case study. This section begins with a brief introduction §F.0.0.1 to one formalization of timing diagrams called Real Time Symbolic Timing Diagrams (RTSTD) [SD93, FJ97]. The original timing diagram is then modelled in that notation §F.0.0.2, before some conclusions are stated §F.0.0.3.

F.0.0.1 (Real Time) Symbolic Timing Diagrams

RTSTDs are an extension of Symbolic Timing Diagrams (STDs), which are a declarative specification language for qualitative constraints on system state changes [SD93]. They

have been employed as a visual means for stating formulas of Propositional Temporal Logic in the VHDL/S system that combines VHDL and Statecharts with support for model checking and theorem proving [DJS95]. Techniques for synthesising controllers from STDs have been investigated [KS94]. Case studies are available [KS95].

An STD specification may comprise multiple Timing Diagrams (TDs), each of which is itself comprised of *bundles*. A bundle is a set of waveforms. They are normally presented graphically. Each waveform is a sequence of alternating *regions* and events that are totally ordered from left to right. Regions are labelled with propositional expressions over one or more state variables, like, for example, equality between a variable and value Vin = '1'.

Events mark discrete changes of state variables, where an *activation condition* is true beforehand, and either a *continuation condition*, meaning that a bundle remains applicable, or an *exit condition*, meaning that a bundle is no longer applicable, is true afterward. Arrowed arcs drawn from one event to another, usually between different waveforms, further constrain the possible sequences of state variable values, thus restricting the specified set of *behaviours*. STDs differentiate between environment actions and system actions. The former are associated with *weak* constraints and the latter with *strong* constraints. The weak constraints act as assumptions for the guarantees provided by the strong constraints.

An STD defines traces of system variable values. A single TD will apply at different times over a system trace, depending on whether it is *initial* or *dynamic*, and the details of its activation, continuation, and exit conditions. When a bundle is applicable, it is termed *activated*, and at other times *deactivated*. If initial, a TD is activated from the initial state. If dynamic, a TD is activated when system variables satisfy the conjunction of its left-most activation conditions. Multiple, overlapping invocations of TDs are permitted, they act in parallel to simultaneously constrain system traces.

Allowable behaviours are defined by the *unwindings* of TDs [DJS95].[1] A *front* consists of one event for each waveform. These are the changes to state variables that are waiting for one or more continuation conditions to become true before occurring. Waiting is only permitted while all activation conditions are satisfied. At any state change each event may:

1. Remain *stable*: The activation condition remains true.

2. *Unwind*: The activation condition no longer holds, but the continuation condition is true in the next state.

3. *Exit*: The activation condition no longer holds and in the next state the continuation condition is false but the exit condition is true.

4. *Fail*: The activation condition no longer holds, and both continuation and exit conditions are false in the next state.

The constraints and conditions of a TD are distilled to these four possibilities that apply to each front configuration. Violations of weak constraints result in exit deactivations, meaning that a given TD is not applicable to the subsequence of behaviour in question. Failure deactivations, caused when strong constraints are violated, distinguish behaviours that are outside a specification. Certain fronts are distinguished to mandate eventual progress.

STDs are given semantics in terms of Temporal Logic formulas.

An RTSTDs [FJ97] is an STD where constraint arcs may be annotated with upper and lower bounds. The four main types of constraint are listed in Table F.1. Simultaneity and precedence impose an ordering on events, conflict and leads-to mark timing restrictions. Pairs of events may be constrained in multiple ways.

An RTSTD is translated to a timed Büchi automaton by adding a clock for each event, and then, as a diagram is unwound, resetting clocks when events occur, and later using them in guards to fulfil relative timing constraints. Care is required when determining which of the system or environment breaks a leads-to constraint.

[1] The original paper contains detailed definitions.

constraint	symbol	bounds*	description†
simultaneity	$RT^{=}$	$[0,0]$	e_1 and e_2 must occur in the same *unwinding step*.
precedence	$RT^{\leq}$	$[0,\infty]$	e_2 cannot occur unless e_1 has already occurred.
conflict	$RT_t^{\neq}$	$[t,\infty]$	e_2 must not occur in the same *unwinding step* as, nor before t time units after, e_1.
leads-to	$RT_t^{\rightsquigarrow}$	$[-\infty,t]$	e_2 must occur with t time units of e_1 (bounded liveness).

*$[l,u]$, lower and upper bounds, respectively, on the interval between two events.
†for $(e_1,e_2) \in RT$

Table F.1: RTSTD timing constraints

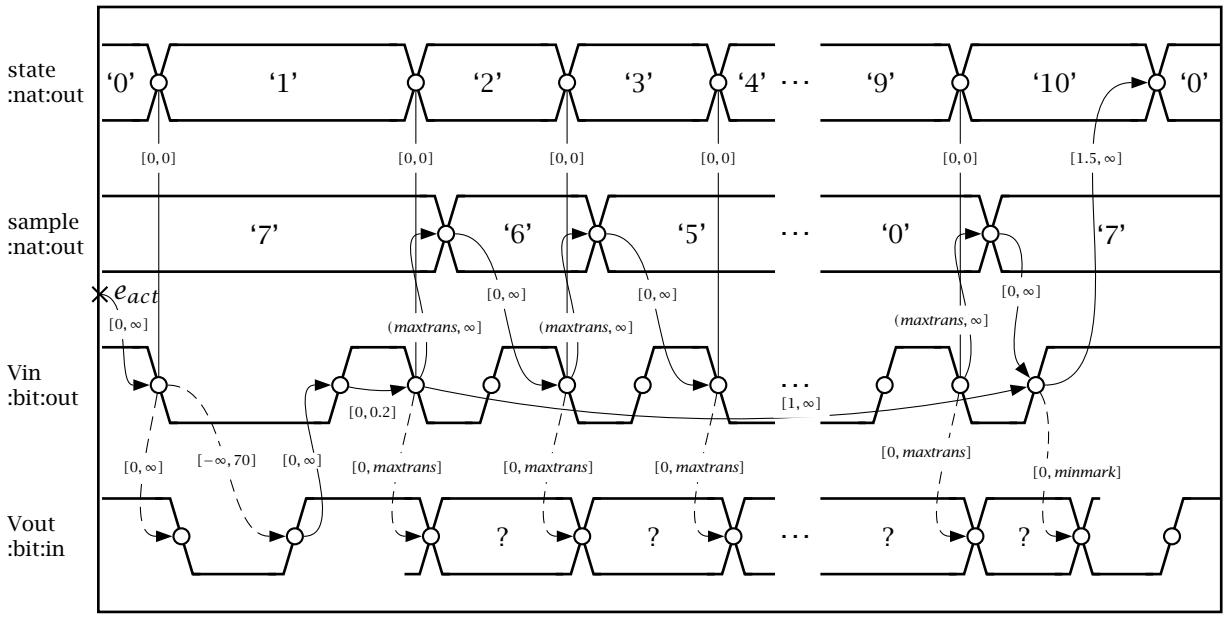


Figure F.2: RTSTD version of the timing diagram of Figure 4.3

F.0.0.2 RTSTD model of the timing diagram

An RTSTD version of the sensor timing diagram is shown in Figure F.2. The centre of the diagram is omitted for reasons of clarity, arcs to and from elided events are also omitted. The RTSTD diagram embodies the same decisions that lead to the timed automaton model. Whereas the original timing diagram had only two waveforms, the RTSTD version has four. The waveforms labelled *Vin* and *Vout* mimic those with the same names in the original, though the formatting differs. The introduction of the two others, *sample* and *state*, is unfortunate because they are not present in the original and they make the diagram more complicated. The justification for *sample* was given earlier §4.3.2. The necessity of *state* is described below. A third extra waveform *power* is omitted.

The RTSTD is *dynamic* rather than *initial*, which would have been indicated by a double bar at the left-hand side, so as to constrain repeated readings. An additional *initial* RTSTD is required but not shown. It simply restricts the values of *state*, *sample*, *Vin* and *Vout* to 0, 7, 1, and 1 respectively.

RTSTDs distinguish between system and environment, in contrast to the approach taken for the timed automaton model. Here the diagram is treated as the specification of a driver, thus, perhaps confusingly, *Vin* is labelled at left as an output and *Vout* as an input. Constraints relating waveforms to *Vout* are weak, indicated by dashed lines,

while those from *Vout* are strong, indicated by solid lines.

The notation used for *Vin* varies from the published [FJ97] examples, which appear, like *state* and *sample*, with '0' and '1' written in alternate regions. Regions are bordered above or below by a single line intended to imply one of the propositions $Vin = 0$ or $Vin = 1$ in the obvious way. The waveform *Vout* is depicted similarly, but the events of indeterminate change are left as crossed lines, and the regions between contain question marks. It is not clear how to model such a data signal; published accounts [DJS95, FJ97] avoid the issue because they focus on control events. A predicate $Vin = 0 \vee Vin = 1$ is unsuitable because it is always satisfied and events are defined by waveform changes. One solution would be to exclude the *Vout* waveform altogether, since range readings can be made without any feedback. This would, however, forbid implementations that are triggered by the first rising *Vout* transition, or those that look for transitions to detect data changes, and it departs from the ideal of modelling the timing diagram itself. Instead control events and valid data intervals are mixed on the same waveform, but arcs are only drawn to and never from the changes on *Vout* that signify data transmission.

The *sample* waveform in the RTSTD performs the same function as the sample event in the timing diagram model; it specifies the intervals where sampling may occur. Since events are characterised by changes in state variables, *sample* is declared as a natural number that decreases in value each time a sample is taken. The regions of this waveform are marked with values that stand for propositional expressions, for example '6' abbreviates sample = '6'.

Understanding the inclusion of the *state* waveform requires considering the meaning of the diagram, that is its semantics in terms of Timed Propositional Temporal Logic (TPTL). A dynamic diagram is associated with a formula of the form

$$\Box(actcond \Rightarrow z_{act}.TL(\ldots)),$$

where $actcond$ is a conjunction of the activation conditions from the heads of each waveform, z_{act} resets the clock associated with the activation event (e_{act}), and $TL(\ldots)$ is the rest of the translation. For Figure F.2, $actcond$ is

$$state = 0 \wedge sample = 7 \wedge Vin = 1 \wedge Vout = 1.$$

Without the *state* waveform the RTSTD would exclude some valid observations. Since the activation condition would also be true after the first rising edges of *Vin* and *Vout*, a second invocation of the formula would apply. The new invocation starts anew and thus mandates a 70ms delay before any subsequent transition on *Vin*. The addition of the state waveform precludes such multiple invocations.

The *state* waveform is synchronized with each falling edge of *Vin*: the vertical lines marked $[0, 0]$ represent simultaneity constraints. The two waveforms could also have been merged into a single waveform with a conjunction in each range, for example, $Vin = 0 \wedge state = 2$.

The addition of a *state* waveform violates the ideal of modelling observable behaviour at external interfaces. Unfortunately it cannot be made a local waveform because it would then have a distinct value for each instantiation of the RTSTD whereas, for the desired effect, a single value must be shared across all invocations.

Having discussed the RTSTD diagram in broad terms, the finer details are now considered.

Working through Figure F.2 from the left, the first constraint is a strong precedence constraint between e_{act}, which occurs implicitly when an RTSTD is activated, and the first falling *Vin* event. The first falling events of *Vin* and *Vout* are linked by a weak precedence constraint. A weak leads-to constraint $[-\infty, \epsilon]$ would give an upper bound. A simultaneity constraint $[0, 0]$ would give a composite event.

The *70ms or more* restriction is expressed as a weak leads-to constraint, requiring a rising edge on *Vout* to occur within 70 units of the first falling edge on *Vin*. The next rising edge on *Vin* must occur after the rising *Vout* edge. For strict compliance with the timing diagram, a conflict constraint $[70, \infty]$ should be added between the first falling edge on *Vin* and the subsequent rising edge.

Timing constraints are added between events on the *Vin* waveform, in addition to its implicit total order: $[0, 0.2]$ expresses the *0.2ms or less* restriction and $[1, \infty]$ expresses the *1ms or more* restriction across the sampling pulses. Not shown are the conflict constraints $[minspace, \infty]$ between adjacent falling and rising edges on *Vout* sampling pulses, nor $[minmark, \infty]$ between adjacent rising and falling edges, as these render the diagram less readable. They could be expressed in a simultaneous RTSTD.

Each falling edge on *Vin* during the sampling phase anchors two out-going arcs. The first, a strong combination of conflict, $RT^{\neq}_{maxtrans}$, and precedence, $RT^{\leq}$, restricts the subsequent sampling event. The second, a weak leads-to, $RT^{\rightsquigarrow}$, constrains changes on *Vout*. It is not clear that open intervals can be specified in the RTSTD language. Events on *sample* must precede subsequent falling events on *Vin*. The independence of sampling relative to rising transitions, except the last one, on *Vin*, is much more naturally expressed in the this model than in the timed automaton model.

The *1.5ms or more* constraint on beginning another reading or switching off is expressed between the last rising transition on *Vin* and the last change in *state*. After *state* has returned to zero, and the final rising transition of *Vout* has occurred, the RTSTD may repeat since the initial activation conditions are then true again.

The powerOff event could be modelled by adding a *power* waveform. In Figure F.2 the waveform would have a predicate of $power = 1$ with an exit condition of $power = 0$ strongly constrained by $[1.5, \infty]$ on the last rising edge of *Vin*. An additional initial RTSTD, containing only *power*, would have the same predicates, though the second would be a continuation condition.

F.0.0.3 Analysis and conclusions

This brief evaluation of formalised timing diagrams suffers from two main limitations. First, the RTSTD model was not transformed into a timed automaton or formula of temporal logic for formal comparison with the manually constructed model. Second, RTSTDs are but one formalisation of timing diagrams and no attempt has been made to isolate their particular peculiarities. Yet even with these limitations, it still seems reasonable to draw some general conclusions on the use of precedence arcs and overlapping constraints, and on whether redrawing alone is sufficient for formalisation, and to compare the modelling approaches and argue that the timed automaton model is most suitable for the sensor case study.

Ideally, the RTSTD diagram would have been transformed into a semantic model, but the synthesis software seems no longer to be available. Moreover, it is not clear how the data changes on *vout* could be represented. It would be interesting to determine whether the roles of system and environment in the RTSTD correspond directly with the presence and absence, respectively, of location invariants in the timing diagram model, and the necessity of response that they imply.

The precedence arcs of the RTSTD model are a distinctive and useful feature. They make explicit causal relations that must otherwise be inferred from the relative placements and dashed vertical lines of the original diagram. Some informal timing diagrams [Ron80] also contain such arcs, so perhaps the sensor diagram is not representative in this regard. While the arcs provide a necessary precision, it comes, arguably, at the cost of readability. Interpreting a diagram requires, potentially, understanding the simultaneous effect of multiple interwoven arcs, while a timed safety automaton[2] may be understood one state and transition at a time, with relevant clock constraints stated at each, although it may ultimately be only a question of taste and familiarity.

Another expressive, but potentially complicating feature of RTSTDs is the possibility of multiple overlapping constraints, which are similar to the parallel operator of CSP and the constraint-oriented specifications of LOTOS (see Appendix A). This feature was not necessary to express the sensor timing diagram so no practical conclusions can be made. It would seem, however, to be quite complicated to define [DJS95].

In principle, a timing diagram is formalized using RTSTDs by simply redrawing it, and then generating a semantic model. Yet, in practice, it is still necessary to carefully interpret the original diagram to ensure a correct translation. Similarly, the meaning

[2] Timed Büchi automata require more complicated accounting.

of a formal timing timing diagram may seem immediately clear, but precision depends on semantics, not intuition; having something that looks like the specification sheet gives no absolute guarantees about the underlying model.

At a high-level, comparing formalised timing diagrams with timed automata involves weighing the appeal of a problem-specific notation against the benefits of a more general formalism. RTSTDs, for instance, are good at mimicking the appearance of the sensor timing diagram, but the additional details needed to remove ambiguity and arrive at a complete description are awkward to include and the results may well lack the great advantages of simplicity and abstraction possessed by informal timing diagrams. While some behaviours are more naturally expressed in the RTSTD notation, namely mutually unconstrained events, others, namely choice and state, are more easily expressed with timed automata. The gap between a timing diagram and a timed automata is advantageous in a discussion of modelling details because each provides a different perspective on the behaviours being studied.

Further, timed automata are both simpler and better understood than RTSTDs and surely other similar formalisms. While STDs and RTSTDs are defined over several pages [DJS95], the essence of timed automata can be expressed with a tuple and two types of transition—though, admittedly adding other features, like variables, complicates matters. Also, the translation of a timing diagram into an automaton or formula is less direct than the relation between a timed automaton graph and the corresponding tuple. Although realistic programming languages require more complicated definitions, SML for example [MTHM97], the trade-off for RTSTDs does not seem as worthwhile.

Bibliography

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proceedings of 5th Annual IEEE Symposium on on Logic in Computer Science (LICS '90)*, pp. 414–425. IEEE Computer Society, Jun. 1990.

[ACM08] *Proceedings of 8th ACM International Conference on Embedded Software (EMSOFT'08)*. ACM Press, Atlanta, Georgia USA, Oct. 2008.

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):pp. 183–235, Apr. 1994.

[AEKN00] N. Amla, E. A. Emerson, R. P. Kurshan, and K. S. Namjoshi. Model checking synchronous timing diagrams. In W. A. J. Hunt and S. D. Johnson, eds., *Proceedings of 3rd International Conference on Formal Methods in Computer-Aided Design*, vol. 1954 of *Lecture Notes in Computer Science*, pp. 283–298. Springer-Verlag, Austin, Texas, Nov. 2000.

[AFH99] R. Alur, L. Fix, and T. A. Henzinger. Event-clock automata: A determinizable class of timed automata. *Theoretical Computer Science*, 211(1–2):pp. 253–273, Jan. 1999.

[AH97] R. Alur and T. A. Henzinger. Real-time system = discrete system + clock variables. *International Journal of Software Tools for Technology Transfer*, 1(1–2):pp. 86–109, Dec. 1997.

[AKRS08] R. Alur, A. Kanade, S. Ramesh, and K. Shashidhar. Symbolic analysis for improving simulation coverage of Simulink/Stateflow models. In ACM [ACM08], pp. 89–98.

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):pp. 253–284, 1991.

[AL94] —. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):pp. 1543–1571, Sep. 1994.

[AMP91] C. André, J. Marmorat, and J. Paris. Execution machines for Esterel. In *Proceedings of European Control Conference*, vol. 2, pp. 1672–1677. Hermes Editions, Grenoble, France, Jul. 1991.

[AMT94] A. W. Appel, J. S. Mattson, and D. R. Tarditi. *A lexical analyzer generator for Standard ML*. Department of Computer Science, Princeton University, 1994.

[And95] C. André. SyncCharts: A visual representation of reactive behaviors. Technical report, I3S, Sophia-Antipolis, France, Oct. 1995. RR 95-52.

[And96] —. Representation and analysis of reactive behaviors: A synchronous approach. In *Computational Engineering in Systems Applications*, pp. 19–29. IEEE, Lille, France, Jul. 1996.

[AP93] C. André and M.-A. Péraldi. Effective implementation of ESTEREL programs. In *5th Euromicro Workshop on Real-Time Systems*, pp. 262–267. Oulu, Finland, Jun. 1993.

[ASK04] A. Agrawal, G. Simon, and G. Karsai. Semantic translation of Simulink/Stateflow models to hybrid automata using graph transformations. In *4th Workshop on Language Descriptions, Tools and Applications (LDTA 2004), satellite event of ETAPS 2004*. Barcelona, Spain, Apr. 2004. Preliminary Version.

[AT05] K. Altisen and S. Tripakis. Implementation of timed automata: An issue of semantics or modeling? Tech. Rep. TR-2005-12, Verimag, Gières, France, Jun. 2005.

[BASRB04] M. Biglari-Abhari, Z. Salcic, P. Roop, and A. Bigdeli. REFLIX: A processor core with native support for control dominated embedded applications. *Journal of Microprocessors and Microsystems*, 28(1):pp. 13–25, Feb. 2004.

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):pp. 25–59, 1987.

[BB91] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of IEEE*, 79(9):pp. 1270–1282, Sep. 1991.

[BC84] G. Berry and L. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In S. D. Brookes, A. W. Roscoe, and G. Winskel, eds., *Seminar on Concurrency*, vol. 197 of *Lecture Notes in Computer Science*, pp. 389–448. Springer-Verlag, Pittsburg, USA, Jul. 1984.

[BCD+07] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime. UPPAAL-Tiga: Time for playing games! In W. Damm and H. Hermanns, eds., *19th International Conference on Computer Aided Verification*, vol. 4590 of *Lecture Notes in Computer Science*, pp. 121–125. Springer-Verlag, Berlin, Germany, Jul. 2007.

[BCE+03] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of IEEE*, 91(1):pp. 64–83, Jan. 2003.

[BCG+97] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-design of Embedded Systems: The POLIS Approach*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1997.

[BCLG99] A. Benveniste, B. Caillaud, and P. Le Guernic. From synchrony to asynchrony. In J. C. M. Baeten and S. Mauw, eds., *Proceedings of 10th International Conference on Concurrency Theory (CONCUR '99)*, vol. 1664 of *Lecture Notes in Computer Science*, pp. 162–177. Springer-Verlag, Eindhoven, The Netherlands, Aug. 1999. ISBN 3-540-66425-4.

[BCLG+02] A. Benveniste, P. Caspi, P. Le Guernic, H. Marchand, J.-P. Talpin, and S. Tripakis. A protocol for loosely time-triggered architectures. In Sangiovanni-Vincentelli and Sifakis [SVS02], pp. 252–265.

[BCP+01] V. Bertin, E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. Taxys = Esterel + Kronos: A tool for verifying real-time properties of embedded systems. In *Proceedings of 40th IEEE Conference on Decision and Control*, pp. 2875–2880. IEEE, Orlando, Florida, USA, Dec. 2001.

[BDL04] G. Behrmann, A. David, and K. Larsen. A tutorial on Uppaal. In M. Bernardo and F. Corradini, eds., *Formal Methods for the Design of Real-Time Systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*,

vol. 3185 of *Lecture Notes in Computer Science*, pp. 200–236. Springer-Verlag, Bertinora, Italy, Sep. 2004.

[BDL+06] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks. Uppaal 4.0. In *3rd International Conference on Quantitative Evaluation of Systems*, pp. 125–126. IEEE Computer Society, Riverside, California, USA, Sep. 2006.

[Bee94] M. von der Beeck. A comparison of Statecharts variants. In H. Langmaack, W. de Roever, and J. Vytopil, eds., *Formal Techniques in Real-Time and Fault-Tolerant Systems*, vol. 863 of *Lecture Notes in Computer Science*, pp. 128–148. Springer-Verlag, Sep. 1994. ISBN 3-540-58468-4.

[Ber89a] G. Berry. Programming a digital watch in Esterel v3. Rapport de recherche 1032, INRIA, Sophia Antipolis, May 1989.

[Ber89b] —. Real time programming: Special purpose or general purpose languages. In G. Ritter, ed., *Proceedings of 11th International Federation for Information Processing (IFIP) World Computer Congress*, pp. 11–17. San Francisco, USA, Aug.–Sep. 1989.

[Ber92] —. Esterel on hardware. *Philosophical Transactions of the Royal Society: Physical and Engineering Sciences*, 339(1652):pp. 87–103, Apr. 1992.

[Ber93] —. Preemption in concurrent systems. In R. K. Shyamasundar, ed., *Foundations of Software Technology and Theoretical Computer Science*, vol. 761 of *Lecture Notes in Computer Science*. Springer-Verlag, Bombay, India, Dec. 1993. ISBN 3-540-57529-4.

[Ber99] —. *The Constructive Semantics of Pure Esterel*. `ftp://ftp-sop.inria.fr/meije/esterel/papers/constructiveness3.ps`, draft book, current version 3.0 ed., Jul. 1999.

[Ber00a] —. *The Esterel v5 Language Primer*. Ecole des Mines and INRIA, version 5.92 ed., Jun. 2000.

[Ber00b] —. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, eds., *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Foundations of Computing Series. MIT Press, 2000.

[BG89] G. Berry and G. Gonthier. Incremental development of an HDLC protocol in Esterel. Rapport de recherche 1031, INRIA, Sophia Antipolis, May 1989.

[BG92] —. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):pp. 87–152, Nov. 1992.

[BHHS93] R. Bernhard, L. Hazard, F. Horn, and J.-B. Stefani. Implementation of a synchronous execution machine on Chorus micro-kernel. In *Proceedings of 14th IEEE Real-Time Systems Symposium (RTSS 1993)*, pp. 189–193. IEEE, Raleigh-Durham NC, USA, Dec. 1993.

[BIM95] B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced. *Journal of the ACM*, 42(1):pp. 232–268, Jan. 1995.

[BKB99] C. Banphawatthanarak, B. Krogh, and K. Butts. Symbolic verification of executable control specifications. In *Proceedings of 10th International Symposium on Computer Aided Control System Design*, pp. 581–586. IEEE, Hawaii, USA, Aug. 1999.

[Blo94] B. Bloom. When is partial trace equivalence adequate? *Formal Aspects of Computing*, 6(3):pp. 317–338, May 1994.

[BM02] J. Baeten and C. Middelburg. *Process Algebra with Timing*. Monographs in Theoretical Computer Science. Springer-Verlag, 2002. ISBN 3-540-43447-X.

[BMR83] G. Berry, S. Moisan, and J.-P. Rigault. Esterel: Towards a synchronous and semantically sound high level language for real time applications. In *Proceedings of 4th IEEE Real-Time Systems Symposium (RTSS 1983)*, pp. 30–37. IEEE Computer Society, Arlington, Virginia, USA, Dec. 1983.

[Bor92] G. Borriello. Formalized timing diagrams. In *Proceedings of 3rd European Conference on Design Automation*, pp. 372–377. IEEE Computer Society, Brussels, Mar. 1992.

[Bou91] F. Boussinot. Reactive C: An extension of C to program reactive systems. *Software: Practice and Experience*, 21(4):pp. 401–428, Apr. 1991.

[Bou97] A. Bouali. XEVE: an ESTEREL verification environment. Rapport de recherche 0214, INRIA, Sophia Antipolis, Dec. 1997.

[BR01] D. Buck and A. Rau. On modelling guidelines: Flowchart patterns for STATEFLOW. *Gesellschaft für Informatik, FG 2.1.1: Softwaretechnik Trends*, 21(2), 2001.

[BRS93] G. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating reactive processes. In *Proceedings of 20th ACM SIGPLAN-SIGACTSymposium on Principles Of Programming Languages (POPL 1993)*, pp. 85–98. ACM Press, 1993. ISBN 0-89791-560-7.

[BS91] F. Boussinot and R. de Simone. The Esterel language. *Proceedings of IEEE*, 79(9):pp. 1293–1304, Sep. 1991.

[BS96] —. The SL synchronous language. *IEEE Transactions on Software Engineering*, 22(4):pp. 256–266, Apr. 1996.

[BS00] G. Berry and E. M. Sentovich. An implementation of constructive synchronous programs in POLIS. *Formal Methods in Systems Design*, 17(2):pp. 135–161, 2000.

[BS01] —. Multiclock Esterel. In T. Margaria and T. F. Melham, eds., *Proceedings of 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, vol. 2144 of *Lecture Notes in Computer Science*, pp. 110–125. Springer-Verlag, Livingston, Scotland, Sep. 2001.

[BS02] M. Baldamus and T. Stauner. Modifying Esterel concepts to model hybrid systems. *Electronic Notes in Theoretical Computer Science*, 65(5):pp. 819–833, Jul. 2002.

[BS05] T. Bourke and A. Sowmya. Formal models in industry standard tools: An Argos block within Simulink. In F. E. Tay, ed., *International Journal of Software Engineering and Knowledge Engineering: Selected Papers from the 2005 International Conference on on Embedded and Hybrid Systems*, vol. 15, pp. 389–395. World Scientific, Singapore, Apr. 2005.

[BS06] —. A timing model for synchronous language implementations in Simulink. In Min and Wang [MW06], pp. 93–101.

[BS08a] H. Bohnenkamp and M. Stoelinga. Quantitative testing. In ACM [ACM08], pp. 227–236.

[BS08b] T. Bourke and A. Sowmya. Automatically transforming and relating Uppaal models of embedded systems. In ACM [ACM08], pp. 59–68.

[BST97] S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In W. P. de Roever, H. Langmaack, and A. Pnueli, eds., *International Symposium on Compositionality: The Significant Difference (COMPOS '97)*, vol. 1536 of *Lecture Notes in Computer Science*, pp. 103–129. Springer-Verlag, Bad Malente, Germany, Sep. 1997.

[BTH07] M. Boldt, C. Traulsen, and R. von Hanxleden. Worst case reaction time analysis of concurrent reactive programs. In *International Workshop on Model-driven High-level Programming of Embedded Systems (SLA++P 2007)*, pp. 65–79. Braga, Portugal, Mar. 2007. ETAPS 2007 Satellite Event.

[Bur95] A. Burns. *Concurrent programming in Ada*. The Ada Companion Series. Cambridge University Press, 1995.

[BV08] J. Berendsen and F. Vaandrager. Compositional abstraction in real-time model checking. In F. Cassez and C. Jard, eds., *Proceedings of 6th International Conference on Formal Modeling and Analysis of Timed Systems*, no. 5215 in Lecture Notes in Computer Science, pp. 233–249. Springer-Verlag, Saint Malo, France, Sep. 2008.

[BW90] J. Baeten and W. Weijland. *Process Algebra*. No. 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[BW04] J. Bengtsson and Y. Wang. Timed automata: Semantics, algorithms and tools. In J. Desel, W. Reisig, and G. Rozenberg, eds., *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, vol. 3098 of *Lecture Notes in Computer Science*, pp. 87–124. Springer-Verlag, 2004.

[Cas92] P. Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94(1):pp. 125–140, Mar. 1992.

[Cas01] —. Embedded control: From asynchrony to synchrony and back. In T. A. Henzinger and C. M. Kirsch, eds., *Proceedings of 1st International Conference on Embedded Software (EMSOFT'01)*, vol. 2211 of *Lecture Notes in Computer Science*, pp. 80–99. Springer-Verlag, Tahoe City, USA, Oct. 2001. ISBN 3-540-42673-6.

[CCM+03] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *Proceedings of 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '03)*, pp. 153–162. ACM Press, 2003. ISBN 1-58113-647-1.

[CDO96] C. Castelluccia, W. Dabbous, and S. O'Malley. Generating efficient protocol code from an abstract specification. *ACM SIGCOMM Computer Communication Review*, 26(4):pp. 60–72, Oct. 1996.

[CGP94] P. Caspi, A. Girault, and D. Pilaud. Distributing reactive systems. In *Seventh International Conference on Parallel and Distributed Computing Systems (PDCS '94)*. International Society for Computers and their Application, Las Vegas, USA, Oct. 1994.

[CJGP00] E. M. Clarke Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Jan. 2000. ISBN 0-262-03270-8.

[CL00] F. Cassez and K. Larsen. The impressive power of stopwatches. In C. Palamidessi, ed., *Proceedings of 11th International Conference on Concurrency Theory (CONCUR '00)*, vol. 1877 of *Lecture Notes in Computer Science*, pp. 138–152. Springer-Verlag, Pennsylvania, USA, Aug. 2000. ISBN 978-3-540-67897-7.

[Com99] F. M. Company. Structured analysis and design using Matlab/Simulink/Stateflow: Modelling style guidelines, version 2.4.2. `http://vehicle.berkeley.edu/mobies/papers/stylev242.pdf`, 1999.

[CP95] P. Caspi and M. Pouzet. A functional extension to Lustre. In M. Orgun and E. Ashcroft, eds., *International Symposium on Languages for Intentional Programming*. World Scientific, Sydney, Australia, May 1995.

[Dij68] E. Dijkstra. Cooperating sequential processes. In F. Genuys, ed., *Programming Languages*, pp. 43–112. Academic Press, 1968.

[DJS95] W. Damm, B. Josko, and R. Schlör. Specification and verification of VHDL-based system-level hardware designs. In E. Börger, ed., *Specification and Validation Methods*, International Schools for Computer Scientists, pp. 331–409. Oxford Science Publications, 1995.

[DNH87] R. De Nicola and M. Hennessy. CCS without τ's. In H. Ehrig, ed., *Proceedings of International Joint Conference on Theory and Practice of Software Development (TAPSOFT'87); Volume 1: Advanced Seminar on Foundations of Innovative Software Development I and Colloquium on Trees in Algebra and Programming (CAAP'87)*, vol. 249 of *Lecture Notes in Computer Science*, pp. 138–152. Springer-Verlag, Pisa, Italy, Mar. 1987.

[DWDR04] M. De Wulf, L. Doyen, and J.-F. Raskin. Almost ASAP semantics: from timed models to timed implementations. In *HSCC 04: Hybrid Systems — Computation and Control*, no. 2993 in Lecture Notes in Computer Science, pp. 296–310. Springer-Verlag, 2004.

[Edw00] S. A. Edwards. Compiling Esterel into sequential code. In *37th Design Automation Conference*, pp. 322–327. ACM Press, Los Angeles, USA, Jun. 2000.

[Edw02] —. An Esterel compiler for large control-dominated systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(2):pp. 169–183, Feb. 2002.

[EG02] D. Evans and P. Gruba. *How to Write a Better Thesis*. Melbourne University Press, 2nd ed., 2002.

[EL03] S. A. Edwards and E. A. Lee. The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, 48:pp. 21–42, 2003.

[EL07] —. The case for the precision timed (PRET) machine. In *44th Design Automation Conference*, pp. 264–265. ACM Press, San Diego, USA, Jun. 2007.

[ET05] S. A. Edwards and O. Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. In Wolf [Wol05], pp. 264–272.

[FF89] M. Fujita and H. Fujisawa. Specification, verification and synthesis of control circuits with propositional temporal logic. In J. A. Darringer and F. J. Rammig, eds., *Computer Hardware Description Languages and their Applications: Proceedings of IFIP WG 10.2 ninth International Symposium on*, pp. 265–279. Elsevier Science, Washington DC, USA, Jun. 1989.

[Fis99] K. Fisler. Timing diagrams: Formalization and algorithmic verification. *Journal of Logic, Language and Information*, 8(3):pp. 323–361, 1999.

[FJ97] K. Feyerabend and B. Josko. A visual formalism for real time requirement specification. In M. Bertran and T. Rus, eds., *Proceedings of 4th International Algebraic Methodology and Software Technology (AMAST) Workshop on Real-Time Systems (ARTS'97)*, vol. 1231 of *Lecture Notes in Computer Science*, pp. 156–168. Springer-Verlag, Palma, Mallorca, Spain, May 1997.

[Gla90] R. van Glabbeek. The linear time-branching time spectrum. In J. Baeten and J. Klop, eds., *Proceedings of 2nd International Conference on Concurrency Theory (CONCUR '90)*, vol. 458 of *Lecture Notes in Computer Science*, pp. 278–297. Springer-Verlag, Amsterdam, Aug. 1990.

[Gla93] —. The linear time-branching time spectrum II: The semantics of sequential processes with silent moves. In E. Best, ed., *Proceedings of 4th International Conference on Concurrency Theory (CONCUR '93)*, vol. 715 of *Lecture Notes in Computer Science*, pp. 66–81. Springer-Verlag, Hildesheim, Germany, Aug. 1993. Unpublished full version: `http://theory.stanford.edu/~rvg/abstracts.html#26`.

[Gla97] —. Notes on the methodology of CCS and CSP. *Theoretical Computer Science*, 177(2):pp. 329–349, May 1997.

[GN00] E. R. Gansner and S. C. North. An open graph visualisation system and its applications to software engineering. *Software: Practice and Experience*, 30(11):pp. 1203–1233, 2000.

[GR04] E. R. Gansner and J. H. Reppy. *The Standard ML Basis Library*. Cambridge University Press, 2004.

[Gri99] E. T. Griebling. GP2D02 assembly language driver for 68HC12B32 microcontroller. `http://home.earthlink.net/~tdickens/68hc11/code/sharpirhc12.asm`, Feb. 1999.

[GT90] A. van Gasteren and G. Tel. Comments on "On the proof of a distributed algorithm": Always-true is not invariant. *Information Processing Letters*, 35:pp. 277–279, Sep. 1990.

[GTL03] P. L. Guernic, J.-P. Talpin, and J.-C. L. Lann. Polychrony for system design. *Journal of Circuits, Systems and Computers*, 12(3):pp. 261–303, 2003.

[GV92] J. Groote and F. Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100(2):pp. 202–260, Oct. 1992.

[GW96] R. J. van Glabbeek and P. W. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):pp. 555–600, May 1996.

[Haa81] V. H. Haase. Real-time behavior of programs. *IEEE Transactions on Software Engineering*, SE-7(5):pp. 494–501, Sep. 1981.

[Hal93] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[Ham05] G. Hamon. A denotational semantics for Stateflow. In Wolf [Wol05], pp. 164–172.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):pp. 231–274, Jun. 1987.

[HB02] N. Halbwachs and S. Baghdadi. Synchronous modeling of asynchronous systems. In Sangiovanni-Vincentelli and Sifakis [SVS02], pp. 240–251.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of IEEE*, 79(9):pp. 1305–1320, Sep. 1991.

[Hen88] M. Hennessy. *Algebraic theory of processes*. Foundations of Computing. MIT Press, 1988.

[HL94] C. Heitmeyer and N. Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *Proceedings of 15th IEEE Real-Time Systems Symposium (RTSS 1994)*, pp. 120–131. IEEE Computer Society, San Juan, Puerto Rico, Dec. 1994.

[HLJ93] C. Heitmeyer, B. Labaw, and R. Jeffords. A benchmark for comparing different approaches for specifying and verifying real-time systems. In H. F. Wedde, ed., *Proceedings of 10th IEEE Workshop on Real-time operating systems and software*, pp. 122–129. IEEE Computer Society, New York, May 1993.

[HLN+88] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring. STATEMATE: A working environment for the development of complex reactive systems. In *Proceedings of the 10th international conference on Software engineering*, pp. 396–406. ACM Press, Singapore, Apr. 1988.

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, eds., *Proceedings of 3rd International Conference on Algebraic Methodology and Software Technology (AMAST'93)*. Workshops in Computing, Springer Verlag, Twente, Jun. 1993.

[HM80] M. Hennessy and R. Milner. On observing nondeterminism and concurrency. In J. de Bakker and J. van Leeuwen, eds., *Proceedings of 7th Colloquium on Automata, Languages and Programming*, vol. 85 of *Lecture Notes in Computer Science*, pp. 299–309. Springer-Verlag, Noordwijkerhout, The Netherlands, Jul. 1980.

[HMP92] T. A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In W. Kuich, ed., *Proceedings of 19th International Colloquium on Automata, Languages and Programming*, vol. 623 of *Lecture Notes in Computer Science*, pp. 545–558. Springer-Verlag, Vienna, Austria, Jul. 1992.

[HMU01] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2nd ed., 2001.

[HN96] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):pp. 293–333, Oct. 1996.

[HNSY94] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):pp. 192–244, Jun. 1994.

[Hoa78] C. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):pp. 666–677, Aug. 1978.

[Hoa85] —. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[Hol03] G. J. Holzmann. *The Spin Model Checker*. Addison Wesley, 2003.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. *Logics and models of concurrent systems*, pp. 477–498, 1985.

[HPSS87] D. Harel, A. Pnuelli, J. Schmidt, and R. Sherman. On the formal semantics of Statecharts. In *2nd IEEE Symposium on Logic in Computer Science*. 1987.

[HR95] M. Hennessy and T. Regan. A process algebra for timed systems. *Information and Computation*, 117(2):pp. 221–239, Mar. 1995.

[HR99] N. Halbwachs and P. Raymond. Validation of synchronous reactive systems: from formal verification to automatic testing. In P. Thiagarajan and R. Yap, eds., *5th Asian Computing Science Conference (ASIAN'99)*, vol. 1742 of *Lecture Notes in Computer Science*, pp. 1–12. Springer-Verlag, Phuket, Thailand, Dec. 1999.

[HR04] G. Hamon and J. Rushby. An operational semantics for Stateflow. In M. Wermelinger and T. Margaria-Steffen, eds., *Proceedings of 7th International Conference on on Fundamental Approaches to Software Engineering*, vol. 2984 of *Lecture Notes in Computer Science*, pp. 229–243. Springer-Verlag, Barcelona, Spain, Apr. 2004.

[IEE04] IEEE Computer Society and The Open Group. *Standard for Information Technology—Portable Operating System Interface (POSIX®): System Interfaces*, IEEE std 1003.1 ed., 2004.

[Int94] Intel Corporation. MCS® 51 microcontroller family user's manual, Feb. 1994.

[Jef93] K. Jeffay. The real-time producer/consumer paradigm: A paradigm for the construction of efficient, predictable real-time systems. In *Proceedings of 1993 ACM/SIGAPP Symposium on Applied Computing: states of the art and practice*, pp. 796–804. ACM Press, Indianapolis, USA, Feb. 1993.

[JHRC08] L. Ju, B. K. Huynh, A. Roychoudhury, and S. Chakraborty. Performance debugging of Esterel specifications. In *6th ACM/IEEE International Conference on Hardware/Software Co-Design and System Synthesis (CODES+ISSS'08)*, pp. 173–178. ACM Press, Atlanta, Georgia USA, Oct. 2008.

[JLS00] H. E. Jensen, K. G. Larsen, and A. Skou. Scaling up Uppaal: Automatic verification of real-time systems using compositionality and abstraction. In Joseph [Jos00], pp. 19–30.

[JMO93] M. Jourdan, F. Maraninchi, and A. Olivero. Verifying quantitative real-time properties of synchronous programs. In C. Courcoubetis, ed., *5th International Conference on Computer Aided Verification*, vol. 697 of *Lecture Notes in Computer Science*, pp. 347–358. Springer-Verlag, Elounda, Greece, Jun./Jul. 1993.

[Jos00] M. Joseph, ed. *Proceedings of 6th International Symposium on Formal Techniques for Real-Time and Fault-Tolerance (FTRTFT '00)*, vol. 1926 of *Lecture Notes in Computer Science*. Springer-Verlag, Pune, India, Sep. 2000. ISBN 3-540-41055-4.

[JPO95] L. J. Jagadeesan, C. Puchol, and J. E. V. Olnhausen. A formal approach to reactive systems software: A telecommunications application in Esterel. In *Proceedings of Workshop on Industrial-Strength Formal Specification Techniques*, pp. 132–145. IEEE, Florida, USA, Apr. 1995.

[JS88] F. Jahanian and D. A. Stuart. A method for verifying properties of modechart specifications. In *Proceedings of 9th IEEE Real-Time Systems Symposium (RTSS 1988)*, pp. 12–21. IEEE Computer Society, Huntsville, USA, Dec. 1988.

[KB03] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of IEEE*, 91(1):pp. 112–126, Jan. 2003.

[KC98] K. Khordoc and E. Cerny. Semantics and verification of action diagrams with linear timing constraints. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 3(1):pp. 21–50, Jan. 1998.

[KLSV06] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. *The Theory of Timed I/O Automata*. Synthesis Lectures on Computer Science. Morgan & Claypool Publishers, 2006.

[KMSL83] J. Kramer, J. Magee, M. Sloman, and A. Lister. CONIC: an integrated approach to distributed computer control systems. *IEE Proceedings (Part E)*, 130(1):pp. 1–10, Jan. 1983.

[Kop97] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[KRSW98] T. Kropf, J. Ruf, K. Schneider, and M. Wild. A synchronous language for modeling and verifying real time and embedded systems. In *GI/ITG/GME Workshop: Methoden des Entwurfs und der Verifikation digitaler Schaltungen und Systeme und Beschreibungssprachen und Modellierung von Schaltungen und Systemen*, pp. 11–20. HNI-Verlagsschriften, Paderborn, Germany, Mar. 1998.

[KS94] F. Korf and R. Schlör. Interface controller synthesis from requirement specifications. In *Proceedings of European Conference on Design Automation (with ETC and EUROASIC)*, pp. 385–394. IEEE Computer Society, Paris, Mar. 1994.

[KS95] —. *Synthesis of a Production Cell Controller using Symbolic Timing Diagrams*, chap. 18, pp. 311–331. Vol. 891 of Lewerentz and Lindner [LL95], 1995.

[Lam77] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):pp. 558–565, Jul. 1977.

[Lam83] —. What good is temporal logic? In R. Mason, ed., *Proceedings of 9th International Federation for Information Processing (IFIP) World Computer Congress*, pp. 657–668. International Federation for Information Processing (IFIP), Paris, France, Sep. 1983.

[Lam02] —. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison Wesley, 2002.

[LGGLBLM91] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of IEEE*, 79(9):pp. 1321–1336, Sep. 1991.

[LHHR94] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):pp. 684–707, Sep. 1994.

[LL95] C. Lewerentz and T. Lindner, eds. *Formal Development of Reactive Systems—Case Study Production Cell*, vol. 891 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1995. ISBN 3-540-58867-1.

[LL97] L. Léonard and G. Leduc. An introduction to ET-LOTOS for the description of time-sensitive systems. *Computer Networks and ISDN Systems*, 29(3):pp. 271–292, 1997.

[LLB+05] X. Li, J. Lukoschus, M. Boldt, M. Harder, and R. von Hanxleden. An Esterel processor with full preemption support and its worst case reaction time analysis. In *Proceedings of International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'05)*, pp. 225–236. ACM Press, San Francisco, USA, Sep. 2005.

[LPW97] K. G. Larsen, P. Pettersson, and Y. Wang. Uppaal in a nutshell. *International Journal of Software Tools for Technology Transfer*, 1(1–2):pp. 134–152, Oct. 1997.

[LS85] N. Leveson and J. Stolzy. Analyzing safety and fault tolerance using time petri nets. In G. Goos and J. Hartmanis, eds., *Formal Methods and Software Development, Proceedings of International Joint Conference on-Theory and Practice of Software Development (TAPSOFT) Volume 2: Colloquium on Software Engineering (CSE)*, vol. 186 of *Lecture Notes in Computer Science*, pp. 339–355. Springer-Verlag, Berlin, Mar. 1985.

[LS91] K. G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1):pp. 1–28, Sep. 1991.

[LS02] G. Logothetis and K. Schneider. Extending synchronous languages for generating abstract real-time models. In *Proceedings of Design, Automation and Test in Europe (DATE'02)*, pp. 795–803. IEEE Computer Society, Paris, Mar. 2002.

[LT87] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of 6th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 137–151. Vancouver, Canada, Aug. 1987.

[LT89] —. An introduction to input/output automata. *CWI Quarterly*, 2(3):pp. 219–246, Sep. 1989.

[LV95a] N. Lynch and F. Vaandrager. Action transducers and timed automata. *Formal Aspects of Computing*, 8(5):pp. 499–538, 1995.

[LV95b] —. Forward and backward simulations. Part I: Untimed systems. *Information and Computation*, 121(2):pp. 214–233, Sep. 1995.

[LV96] —. Forward and backward simulations. Part II: Timing-based systems. *Information and Computation*, 128(1):pp. 1–25, Jul. 1996.

[Lyn96] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[MAP93] P. Moeschler, H. P. Amann, and F. Pellandini. High-level modelling using extended timing diagrams: A formalism for the behavioral specification of digital hardware. In *Proceedings of European Design Automation Conference (EuroDAC93 and EURO-VHDL 93)*, pp. 494–499. Hamburg, Sep. 1993.

[Mar91] F. Maraninchi. The Argos language: Graphical representation of automata and description of reactive systems. In *Proceedings of IEEE Workshop on Visual Languages*, pp. 254–259. Oct. 1991.

[Mar92] —. Operational and compositional semantics of synchronous automaton compositions. In R. Cleaveland, ed., *Proceedings of 3rd International Conference on Concurrency Theory (CONCUR '92)*, vol. 630 of *Lecture Notes in Computer Science*, pp. 550–564. Springer-Verlag, Stony Brook, NY, USA, Aug. 1992. ISBN 3-540-55822-5.

[Mat01] MathWorks Automotive Advisory Board (MAAB). Controller style guidelines for production intent using MATLAB, Simulink and Stateflow. `http://www.mathworks.com/applications/controldesign/MAAB_Style_Guide_html_v1_00/MAAB_v1p00.htm`, Apr. 2001.

[Mat03a] The Mathworks, Natick, MA, U.S.A. *Simulink—Using Simulink*, 5th ed., Sep. 2003. Release 13SP1.

[Mat03b] The Mathworks, —. *Simulink—Writing S-Functions*, 5th ed., Sep. 2003. Release 13SP1.

[Mat03c] The Mathworks, —. *Stateflow® and Stateflow Coder® User's Guide*, 5th ed., Sep. 2003. Stateflow 5.1 (Release 13SP1).

[Mat04] The Mathworks, —. *Stateflow® and Stateflow Coder® User's Guide*, 6th ed., Jun. 2004. Stateflow 6.0 (Release 14).

[Mea55] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):pp. 1045–1079, 1955.

[MH96] F. Maraninchi and N. Halbwachs. Compiling Argos into Boolean equations. In B. Jonsson and J. Parrow, eds., *Proceedings of 4th International Symposium on Formal Techniques for Real-Time and Fault-Tolerance (FTRTFT '96)*, vol. 1135 of *Lecture Notes in Computer Science*, pp. 72–89. Springer-Verlag, Uppsala, Sweden, Sep. 1996. ISBN 3-540-61648-9.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):pp. 267–310, 1983.

[Mil89] —. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, 1989.

[Mil93] —. Elements of interaction: Turing award lecture. *Communications of the ACM*, 36(1):pp. 78–89, Jan. 1993.

[MM97] B. R. Montague and C. E. McDowell. Synchronous/reactive programming of concurrent system software. *Software: Practice and Experience*, 27(3):pp. 207–243, Mar. 1997.

[MMP91] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In J. de Bakker, C. Huizing, W. de Roever, and G. Rozenberg, eds., *Real-Time: Theory in Practice*, vol. 600 of *Lecture Notes in Computer Science*, pp. 447–484. Springer-Verlag, Mook, The Netherlands, Jun. 1991.

[Moo56] E. F. Moore. Gedanken-experiments on sequential machines. In C. Shannon and J. McCarthy, eds., *Automata Studies*, no. 34 in Annals of Mathematics Studies, pp. 129–153. Princeton University Press, 1956.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[MP93] —. Verifying hybrid systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, eds., *Hybrid Systems*, vol. 736 of *Lecture Notes in Computer Science*, pp. 4–35. Springer-Verlag, 1993.

[MR01] F. Maraninchi and Y. Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, 27(1–3):pp. 61–92, 2001.

[MS92] G. J. Murakami and R. Sethi. Parallelism as a structuring technique: Call processing using the Esterel language. In J. van Leeuwen, ed., *Proceedings of 12th International Federation for Information Processing (IFIP) World Computer Congress*, no. 92 in Information Processing, pp. 10–16. Madrid, Spain, 1992.

[MT90] R. Milner and M. Tofte. *Commentary on Stndard ML*. MIT Press, Nov. 1990.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, May 1997.

[MW06] S. L. Min and Y. Wang, eds. *Proceedings of 6th ACM International Conference on Embedded Software (EMSOFT'06)*. ACM Press, Seoul, South Korea, Oct. 2006. ISBN 1-59593-542-8.

[Neu99] A. Neumann. *Parsing and Querying XML Documents in SML*. Ph.D. thesis, Universität Trier, Germany, Dec. 1999.

[NPW81] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13:pp. 85–108, 1981.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, vol. 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[NS94] X. Nicollin and J. Sifakis. The algebra of timed processes ATP: Theory and application. *Information and Computation*, 114(1):pp. 131–178, 1994.

[NSY93] X. Nicollin, J. Sifakis, and S. Yovine. From ATP to timed graphs and hybrid systems. *Acta Informatica*, 30(2):pp. 181–202, 1993.

[Par81] D. Park. Concurrency and automata on infinite sequences. In G. Goos and J. Hartmanis, eds., *5th GI-Conference*, vol. 104 of *Lecture Notes in Computer Science*, pp. 167–183. Springer-Verlag, Karlsruhe, Mar. 1981.

[PBEB07] D. Potop-Butucaru, S. A. Edwards, and G. Berry. *Compiling Esterel*. Springer-Verlag, 2007.

[Pel02] D. Peled. Specification and verification using Message Sequence Charts. *Electronic Notes in Theoretical Computer Science*, 65(7), 2002.

[Pet81] J. L. Peterson. *Petri Net Theory and The Modeling of Systems*. Prentice-Hall, 1981.

[Pnu94] A. Pnueli. Development of hybrid systems. In H. Langmaack, W.-P. de Roever, and J. Vytopil, eds., *Proceedings of 3rd International Symposium on Formal Techniques for Real-Time and Fault-Tolerance (FTRTFT '94)*, vol. 863 of *Lecture Notes in Computer Science*, pp. 77–85. Springer-Verlag, Lübeck, Germany, Sep. 1994. ISBN 3-540-58468-4.

[Pon01] M. Pont. *Patterns for time-triggered embedded systems: Building reliable applications with the 8051 family of microcontrollers*. ACM Press/Addison Wesley, 2001. ISBN 0-201-331381.

[PRS95] P. Pandya, Y. Ramakrishna, and R. Shyamasundar. A compositional semantics of Esterel in duration calculus. In *Proceedings of 2nd International Algebraic Methodology and Software Technology (AMAST) Workshop on Real-Time Systems (ARTS'95)*. Bordeaux, France, Jun. 1995.

[Puc98] R. R. Pucella. Reactive programming in Standard ML. In *International Conference on Computer Languages, ICCL '98*, pp. 48–57. IEEE, Chicago, USA, May 1998.

[Ram01] A. Ramsey. Interfacing the GP2D02 to a Microchip PIC. Encoder: The Newsletter of the Seattle Robotics Society, Dec. 2001.

[Rei85] W. Reisig. *Petri Nets: An Introduction*, vol. 4 of *EATCS Monographs on Computer Science*. Springer-Verlag, 1985.

[RGG96] E. Rudolph, P. Graubmann, and J. Grabowski. Tutorial on Message Sequence Charts. *Computer Networks and ISDN Systems: Special issue on SDL and MSC*, 28(12):pp. 1629–1641, Jun. 1996.

[Ron80] P. R. Rony. Interfacing fundamentals: Timing diagram conventions. *Computer Design*, 19(1):pp. 152–153, Jan. 1980.

[Ros97] A. Roscoe. *The Theory and Practice of Concurrency*. Concurrent and Distributed Systems. Prentice-Hall, 1997.

[RS98] S. Ramesh and C. M. Shetty. Impossibility of synchronization in the presence of preemption. *Parallel Processing Letters*, 8(1):pp. 111–120, Mar./Apr. 1998.

[SA01] R. Shyamasundar and J. Aghav. Validating real-time constraints in embedded systems. In *8th Pacific Rim International Symposium on Dependable Computing (PRDC 2001)*, pp. 347–355. IEEE Computer Society, Seoul, Korea, Dec. 2001.

[Sch08] B. Schlich. *Model Checking of Software for Microcontrollers*. Ph.D. thesis, RWTH Aachen University, Aachen, Germany, Jun. 2008.

[SD93] R. Schlör and W. Damm. Specification and verification of system-level hardware designs using timing diagrams. In *Proceedings of 4th European Conference on Design Automation (with EUROASIC)*, pp. 518–524. IEEE, Paris, Feb. 1993.

[SGSAL94] R. Segala, R. Gawlick, J. Søgaard-Andersen, and N. Lynch. Liveness in timed and untimed systems. In S. Abiteboul and E. Shamir, eds., *Proceedings of 21st International Colloquium on Automata, Languages and Programming*, no. 820 in Lecture Notes in Computer Science, pp. 166–177. Springer-Verlag, Jul. 1994.

[Sha97] Sharp Corporation. GP2D02: Compact, high sensitive distance measuring sensor, 1997.

[Spe02] C. Spencer. *Formal Verification of Stateflow Diagrams Using SMV*. Master's thesis, Carnegie Mellon University, 2002. Not Sighted, Supervised by Bruce Krogh.

[SSC+04] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a "safe" subset of Simulink/Stateflow into Lustre. In G. Buttazzo, ed., *Proceedings of 4th ACM International Conference on Embedded Software (EMSOFT'04)*, pp. 259–268. ACM Press, Pisa, Italy, Sep. 2004. ISBN 1-58113-860-1.

[STC06] C. Sofronis, S. Tripakis, and P. Caspi. A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling. In Min and Wang [MW06], pp. 21–33.

[Sto02] M. I. Stoelinga. *Alea Jacta est: Verification of probabilistic, real-time and parametric systems*. Ph.D. thesis, Katholieke Universiteit Nijmegen, The Netherlands, Apr. 2002.

[STY03] J. Sifakis, S. Tripakis, and S. Yovine. Building models of real-time systems from application software. *Proceedings of IEEE*, 91(1):pp. 100–111, Jan. 2003.

[SVS02] A. L. Sangiovanni-Vincentelli and J. Sifakis, eds. *Proceedings of 2nd International Conference on Embedded Software (EMSOFT'02)*, vol. 2491 of *Lecture Notes in Computer Science*. Springer-Verlag, Grenoble, France, Oct. 2002. ISBN 3-540-44307-X.

[TA00] D. R. Tarditi and A. W. Appel. *ML-Yacc User's Manual*. Department of Computer Science, Princeton University, 2000.

[Tec05] E. Technologies. *The Esterel v7 Reference Manual*. Esterel Technologies, Villeneuve-Loubet, France, v7_30 ed., Nov. 2005.

[Tel00] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2nd ed., Sep. 2000. ISBN 0-521-79483-8. 1st ed.: Nov. 1994.

[The03a] The MathWorks. Bang-bang temperature controller example, 2003.

[The03b] —. Fault-tolerant fuel controller example, 2003.

[The06] S. Thesing. Modeling a system controller for timing analysis. In Min and Wang [MW06], pp. 292–300.

[Tie91] W.-D. Tiedemann. Bus protocol conversion: From timing diagrams to state machines. In F. Pichler and R. M. Díaz, eds., *Proceedings of 2nd International Workshop on Computer Aided Systems Theory (EUROCAST '91)*, vol. 585 of *Lecture Notes in Computer Science*, pp. 365–377. Springer-Verlag, Krems, Austria, Apr. 1991.

[Tiw02] A. Tiwari. Formal semantics and analysis methods for Simulink/Stateflow models. Unpublished report, SRI International, 2002.

[TNTBS00] S. Tudoret, S. Nadjm-Tehrani, A. Benveniste, and J.-E. Strömberg. Co-simulation of hybrid systems: Signal-Simulink. In Joseph [Jos00], pp. 134–151.

[TS05] O. Tardieu and R. de Simone. Loops in Esterel. *ACM Transactions on Embedded Computing Systems*, 4(4):pp. 708–750, Nov. 2005.

[TSCC05] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating discrete-time Simulink to Lustre. *ACM Transactions on Embedded Computing Systems*, 4(4):pp. 779–818, Nov. 2005.

[TSR03] A. Tiwari, N. Shankar, and J. Rushby. Invisible formal methods for embedded control systems. *Proceedings of IEEE*, 91(1):pp. 29–39, Jan. 2003.

[Vaa91] F. W. Vaandrager. On the relationship between process algebra and input/output automata (extended abstract). In *Proceedings of 6th Annual IEEE Symposium on on Logic in Computer Science (LICS '91)*, pp. 387–398. IEEE Computer Society, Amsterdam, Jul. 1991.

[Ver05] Verimag. Argos and Argonaute examples. `http://www-verimag.imag.fr/SYNCHRONE/ARGONAUTE/Examples.html`, 2005.

[VG04] F. Vaandrager and A. de Groot. Analysis of a biphase mark protocol with Uppaal and PVS. Technical Report NIII-R0445, Radboud University, Nijmegen, The Netherlands, 2004.

[WA85] W. W. Wadge and E. A. Ashcroft. *LUCID, the dataflow programming language*. Academic Press Professional, Inc., 1985.

[Wan90] Y. Wang. Real-time behaviour of asynchronous agents. In J. W. K. Jos C.M. Baeten, ed., *Proceedings of 2nd International Conference on Concurrency Theory (CONCUR '90)*, vol. 458 of *Lecture Notes in Computer Science*, pp. 502–520. Springer-Verlag, Amsterdam, Aug. 1990.

[Wan91] —. CCS + time = an interleaving model for real time systems. In J. L. Albert, M. Rodríguez-Artalejo, and B. Monien, eds., *Proceedings of 18th International Colloquium on Automata, Languages and Programming*, vol. 510 of *Lecture Notes in Computer Science*, pp. 217–228. Springer-Verlag, Madrid, Spain, Jul. 1991.

[WBC+00] D. Weil, V. Bertin, E. Closse, M. Poize, P. Venier, and J. Pulou. Efficient compilation of Esterel for real-time embedded systems. In *Proceedings of International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'00)*, pp. 2–8. ACM Press, San Jose, USA, Nov. 2000.

[WEE+08] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):pp. 36–88, Apr. 2008.

[Wol05] W. Wolf, ed. *Proceedings of 5th ACM International Conference on Embedded Software (EMSOFT'05)*. ACM Press, Jersey City, USA, Sep. 2005. ISBN 1-59593-091-4.

[WPD94] Y. Wang, P. Pettersson, and M. Daniels. Automatic verification of real-time communicating systems by constraint-solving. In D. Hogrefe and S. Leue, eds., *Formal Description Techniques VII, Proceedings of 7th IFIP WG6.1 International Conference on Formal Description Techniques*, pp. 223–238. International Federation for Information Processing (IFIP), Chapman & Hall, Berne, Switzerland, 1994.

[WW07] M. Wenzel and B. Wolff. Building formal methods tools in the Isabelle/Isar framework. In K. Schneider and J. Brandt, eds., *Proceedings of 20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2007)*, vol. 4732 of *Lecture Notes in Computer Science*, pp. 352–367. Springer-Verlag, Kaiserslautern, Germany, Sep. 2007.

[Yov97] S. Yovine. Kronos: A verification tool for real-time systems. *International Journal of Software Tools for Technology Transfer*, 1(1–2):pp. 123–133, Dec. 1997.