International Journal of Software Engineering and Knowledge Engineering

FORMAL MODELS IN INDUSTRY STANDARD TOOLS: AN ARGOS BLOCK WITHIN SIMULINK

T. Bourke

School of Computer Science and Engineering, University of NSW
& NICTA Australia
Sydney, NSW 2052, Australia
tbourke@cse.unsw.edu.au

A. Sowmya

School of Computer Science and Engineering, University of NSW
& NICTA Australia
Sydney, NSW 2052, Australia
sowmya@cse.unsw.edu.au

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

Simulink is widely used within industry for simulation and model-driven development, and reactive behaviors are often modeled using an add-on called Stateflow. Argos is one of the synchronous languages that have been proposed for the specification, validation and implementation of reactive systems. It is a rigorously defined graphical notation which, though not as powerful as Stateflow, is much less complicated. This paper describes the implementation of an Argos block for Simulink.

Keywords: reactive systems, synchronous languages, Simulink, Stateflow, Argos.

1. Introduction and Background

Simulink* is a popular simulation environment that includes tools for code generation. In this way it serves as a platform for model-driven development of embedded systems. The Stateflow package allows reactive behaviors to be modeled within Simulink using a notation that looks like Statecharts [5]. Stateflow is powerful, but its semantics is intricate.

The synchronous languages [1] also address the design of reactive systems, but they benefit from an underlying formal model that facilitates verification, synthesis, and manual reasoning. This minimizes differences between implementations and free designers to concentrate on application functionality rather than details of program interpretation.

This paper proposes embedding the restricted but rigorously defined Argos [7] synchronous language into the Simulink environment. Prototype tools and a Simulink block for Argos have been developed to evaluate our approach. Although it is not a complete

* Matlab, Simulink and Stateflow are products and registered trademarks of The MathWorks, Inc.

environment (there is yet no graphical editor), we think that the work to date could provide a practical solution where other tools are unnecessarily powerful and complex. Practitioners can describe executable models, which have precise and easily understood behaviors, for use within the familiar Simulink system.

The remainder of this section provides background and outlines related work. Section 2 describes the synchronous block for Argos that we have developed. In Section 3 we demonstrate our approach with an example. Conclusions follow in Section 4.

1.1. *Simulink and Stateflow*

Simulink models comprise interconnected blocks which have ports for input and output, e.g. Fig. 3(a). Blocks may be discrete or continuous. They are abstract entities which are connected to model system elements. Lines, or *signals*, connect outputs to inputs.

Stateflow provides a block whose reactive behavior is specified using Statecharts-like diagrams. Behaviors are either triggered implicitly as a simulation progresses, or in response to input changes. Internal functions and transitions between states may be given in a flowchart-like language that allows sequencing, branching and iteration.

Stateflow has been described as a graphical *sequential imperative language* [4]. Diagrams are executed by an interpreting algorithm, the operation of which, in combination with natural language descriptions and examples [8], defines the behavior of a program.

1.2. *The Synchronous Language Argos*

The synchronous languages [1] assume that computations, or *reactions*, occur in response to input events and produce state changes and output events in the same instant. In practice a system is synchronous if it processes input events quickly enough so that none are missed. Reactions are atomic and each corresponds to a discrete instant of time.

Argos [7] is uncompromising in adapting Statecharts to the synchronous model. Argos programs combine Mealy machines with operators for hierarchy, parallelism, encapsulation and inhibition. Timed states are available as macros. Argos does not have history junctions, incoming multi-level transitions, or conditional default states. The pure version has no variables and is limited to Boolean signals. Unlike Stateflow, Argos has no concept of an input event queue, rather all simultaneous inputs present as a single event. A reaction transforms this event, relative to the current state, into output values.

The austerity of Argos contrasts with the impressive features of Stateflow. The simpler Argos semantics allow tractable reasoning about designs, both on-paper and automatically. Programs may be compiled into code or silicon with minimum infrastructure.

1.3. *Related work*

Of the other formal approaches to Simulink and Stateflow, two are most directly relevant.

The first [3] translates the discrete controller part of a Simulink model, usually simulated in feedback with a hybrid model of the environment, into the Scade/Lustre language for validation and synthesis and a later extension [9] includes a subset of Stateflow.

Other work [4] clarifies the operation of a large subset of Stateflow with an operational semantics, thereby permitting formal reasoning about unmodified programs. Such an approach is compelling but does not alleviate the underlying complexity of Stateflow.

Our approach complements current research proposals and Stateflow itself. Instead of trying to match the power of Statecharts, or to convert models into an external tool, we propose using a simpler language, suitable for a subset of tasks, *within* a Simulink model.

2. A Synchronous Block

We have implemented a tool-chain to assess our approach. The key component is a synchronous block, or *syncblock*, that provides a bridge between a Simulink model and an Argos program. It encompasses elements from two of three implementation layers [2]: an *interface* for turning Simulink signals into logical signals, and a *reactive kernel* defined by an Argos program. It does not currently support *data-handling* functions.

2.1. *Implementation*

The prototype system is composed of several distinct programs (Fig. 1).

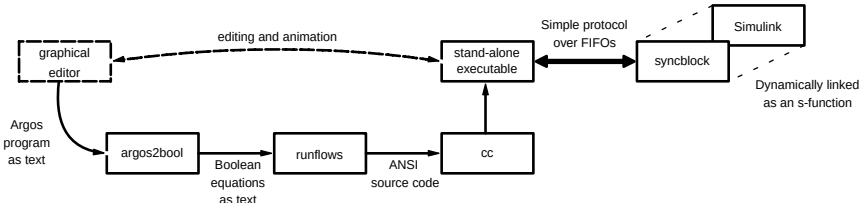

Fig. 1. Toolchain

An Argos compiler, `argos2bool`, converts a textual Argos program into Boolean equations using the technique of [6]. It is implemented in C using a yacc generated parser and a BDD package. Reachability analysis ensures that programs are deterministic and reactive. Cyclic dependencies are detected but not yet resolved. The equations are transformed into C source code by a second program, `runflows`. This code is linked with a library implementing a stream-based interface. It could easily be compiled for an embedded platform, but glue code would be required to interface with the logical signals. The syncblock is implemented as a Simulink s-function, it spawns the compiled Argos program and communicates with it over FIFO channels.

2.2. *Inputs and Outputs*

In order for a program within the syncblock to interact within a model, values from Simulink must be mapped to logical signals and vice versa. Input signals may represent any discrete event of interest. Output signals trigger actions. Any change in the value of a Simulink signal could be regarded as a triggering event, the syncblock only detects the

subset of these events shown in Fig. 2(a). A rising event occurs when an input changes from zero to one in consecutive simulation steps. Falling and transition events are defined similarly. This approach mimics that of Stateflow. A sampled input signal is read, and marked true if non-zero, when an event occurs on a clocking signal. Sensors inputs are sampled at instants of reaction.

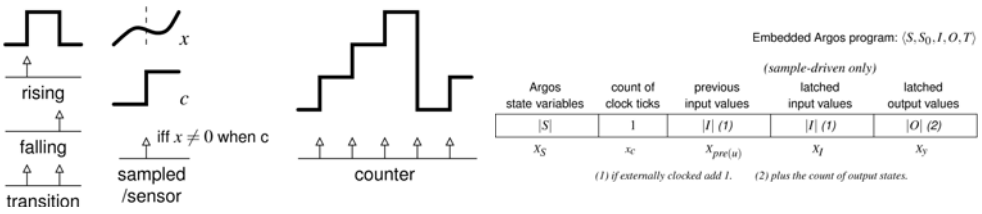

Fig. 2. syncblock implementation (a) Signal mapping (b) State vector

Events are registered for the counter mode whenever the input signal is incremented by at least one, or decremented by more than one (i.e. $\Delta \geq 1$ or $\Delta < -1$). This mode allows clocking from either a rising edge signal or a counter that resets on overflow.

Block outputs are set to one (zero) when a signal is emitted (not emitted), they are latched between reactions. The block allows program states to be signaled, but given that Argos state names have no semantic value [7] it is a debugging mechanism only.

2.3. *Clocking and general operation*

Synchronous programs are either *event-driven*, triggered by input events, or *sample-driven* [1], triggered periodically with input latching. The latter case may be clocked from either the simulation time or an explicit signal. Explicit signals are to be preferred as they clarify the modeling of time and any synchronization assumptions.

The reactive program state is a subset of the syncblock state vector, shown in Fig. 2(b). Elements for clocking, detecting inputs, and latching are required. It is possible, though not effective, to implement these elements with other Simulink blocks.

Four operations occur at each step of a simulation:

(i) Update latch (X_I) by comparing inputs (u) to those of the previous step ($X_{pre(u)}$).
(ii) Increment the clock count (x_c) if necessary.
(iii) In reaction instants – based on clock count, simulation time, or latched inputs – update the Argos state (X_S) and latched outputs (X_y) before clearing the input latch.
(iv) Transfer latched outputs to Simulink outputs.

3. Experience

Stateflow is supplied with a number of example diagrams. As part of evaluating the feasibility of our approach we have modeled some using our tool-set.

Fig. 3(a) shows a subsystem for fuel system sensor modeling. The comparison blocks implement predicates over valued Simulink signals. These are necessary when using pure

Argos. Some predicates were altered to simplify the presentation (for instance `Ego` < `max_ego` and `Ego` > `max_ego` become `Ego` ≤ `max_ego`). Other blocks either merge multiple signals into a vector or encode them in a single value.

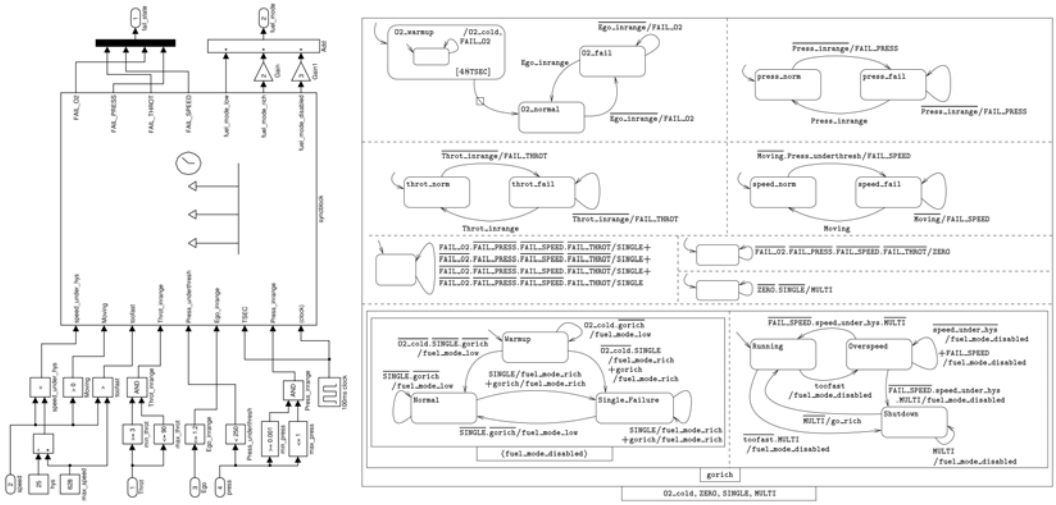

Fig. 3. Fuel control subsystem (a) Simulink interface (b) Argos controller

The syncblock in Fig. 3(a) is sample-driven by an external 100ms clock. The Argos program within is shown in Fig. 3(b) (a textual version was created manually). Moore-style outputs are fabricated with self-loops, the `O2_warmup` state has the self-loop on a refining state to avoid resetting the timer, which itself responds to an explicit input signal rather than the simulation time. This is a subtle difference between modeling reactive behavior for simulation and specifying it for implementation. The failure counter is replaced by three self-looping states – as multiple failures can occur simultaneously (Argos is without Esterel-style combinations) – whose formulae are evaluated at each reaction.

Argos uses labeled rectangles as unary operators, for instance encapsulation declares signals to be local, the box labeled `gorich` in Fig. 3(b) is an example. Within this box is another labeled `<fuel_mode_disabled>`, the angle braces indicate the inhibition operator; the program inside such a box will only participate in reactions where the given signal is false. Inhibition is used in the example to simulate entry-by-history for the Running mode. The `gorich` signal replaces a multi-level transition between the `Shutdown` state and the `Single_Failure` state.

Several transitions in the example are triggered by conjunctions of signals and negated signals (over-lined). Such triggers are irrelevant in an event-queue model, but are otherwise necessary for programs to behave deterministically. 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. Prioritizing by relative graphical position may be less cumbersome, but it is also less explicit and more susceptible to inadvertent change.

4. Concluding Remarks

We have implemented tools to embed a synchronous language within Simulink, in order to compare an approach with strong formal underpinnings to that of the feature-rich but complicated Stateflow product. The pure Argos language can cope with simple examples of reactive behavior but the paucity of features can make modeling awkward. Moore machine outputs, conditional default states, and outputs on default transitions would mitigate some difficulties, as would valued signals, variables, and predicate labels on transitions. It may be better, however, to maintain the simplicity of pure Argos for the systems to which it is suited and to examine other reactive programming languages for more complex instances. We are confident that the syncblock could be easily adapted to work with other synchronous languages that have base periods, such as Lustre and Esterel.

Acknowledgments

The authors would like to thank Ralf Huuck, of NICTA Sydney, for his interest and feedback while the work described in this paper was under development. National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

References

1. A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic and R. de Simone, The synchronous languages 12 years later, *Proceedings of the IEEE*, 91(1)(2003) 64–83.
2. G. Berry and G. Gonthier, The Esterel synchronous programming language: Design, semantics, implementation, *Science of Computer Programming*, 19(2)(1992) 87–152.
3. 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 *Proc. 2003 ACM SIGPLAN conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '03)*, (ACM Press, 2003), pp. 153–162.
4. G. Hamon and J. Rushby, An operational semantics for Stateflow, in *Proc. 7th International Conference on Fundamental Approaches to Software Engineering*, eds. M. Wermelinger and T. Margaria-Steffen, volume 2984 of Lecture Notes in Computer Science, (Springer-Verlag, 2004), pp. 229–243.
5. D. Harel, Statecharts: A visual formalism for complex systems, *Science of Computer Programming*, 8(3)(1987) 231–274.
6. F. Maraninchi and N. Halbwachs, Compiling Argos into Boolean equations, in Proc. 4th International Symposium on Formal Techniques for Real-Time and Fault-Tolerance (FTRTFT '96), eds. B. Jonsson and J. Parrow, volume 1135 of Lecture Notes in Computer Science, (Springer Verlag, 1996), pp. 72–89.
7. F. Maraninchi and Y. Rémond, Argos: an automaton-based synchronous language, *Computer Languages*, 27(1–3)(2001) 61–92.
8. The Mathworks, Natick, MA, U.S.A., *Stateflow® and Stateflow Coder® User's Guide*, 5.1 edition, 2003. Stateflow 5.1 (Release 13SP1).
9. N. Scaife, C.Sofronis, P. Caspi, S. Tripakis and F. Maraninchi, Defining and translating a "safe" subset of Simulink/Stateflow into Lustre, in *Proc. 4th ACM International Conference on Embedded Software (EMSOFT '04)*, eds. G. Buttazzo and S. Edwards (ACM Press, 2004), pp. 259–268.