Delays in Esterel

T. Bourke[1,2]* and A. Sowmya[1]

[1] NICTA, Locked Bag 6016, Sydney NSW 1466, Australia**
[2] School of Computer Science and Engineering, The University of New South Wales, Sydney, NSW 2052, Australia
timothy.bourke@irisa.fr, sowmya@cse.unsw.edu.au

Abstract. The timing details in many embedded applications are inseparable from other behavioural aspects. Time is also a resource; a physical constraint on system design that introduces limitations and costs. Design and implementation choices are often explored and decided simultaneously, complicating both tasks and encouraging platform specific programs where the meaning of a specification is mixed with the mechanisms of implementation.

The Esterel programming language is ideal for describing complex reactive behaviours. But, perhaps surprisingly, timing details cannot be expressed without making significant implementation choices at early stages of design. We illustrate this point with an example application where reactive behaviour and physical time are intertwined.

A simple solution is proposed: add a statement for expressing delays in physical time. While there are similar statements or library calls in many programming languages, the novelty of our proposal is that the delay statements are later replaced with standard Esterel statements when platform details become available. Delays are thus expressed directly in terms of physical time, but later implemented as a discrete controller using existing techniques. This approach is familiar in control system design where analytical models are constructed in continuous time and then later discretized to produce implementations.

We present some ideas for performing the translation and outline some of the remaining challenges and uncertainties.

1 Introduction

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.

* Now affiliated with INRIA / IRISA, Rennes and funded by the Synchronics large-scale initiative action of INRIA.

** NICTA is funded by the Department of Broadband, Communications and the Digital Economy, and the Australian Research Council, in part through the Australian Government's *Backing Australia's Ability* initiative.

Time is also a resource; a physical constraint that introduces limitations and costs. Balancing timing requirements and timing limitations is central to the design of many embedded systems. 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 language was designed for real-time programming [1, 2]. 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. They encourage unnecessarily platform-restricted programs.

These perceived limitations of Esterel are specific to certain applications and quite subtle. 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. This example exhibits two especial characteristics: it requires complex reactive behaviour in physical time, and its eventual implementations are on resource-constrained microcontrollers.

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 allows 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 [3], the macro statement implies the opposite approach: the program is stated in continuous time and the implementation 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 paper 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 paper comprises four sections. In §2, the microprinter example is presented. It is both a motivating, realistic application and a con-

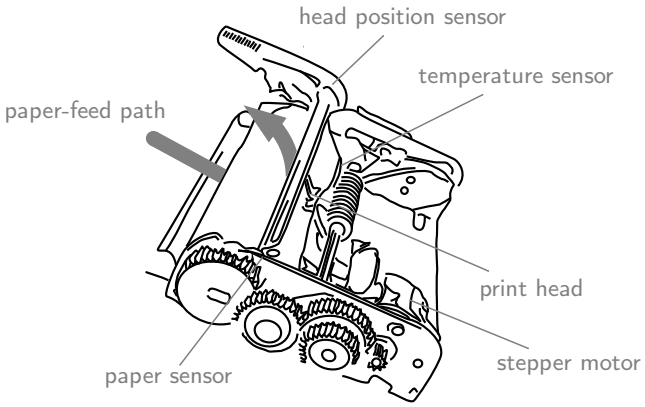


Fig. 1: Physical structure of the microprinter example

crete illustration of the issues under discussion. Extracts from the example are used throughout §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 §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 §5.

2 Motivating example: a microprinter controller

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 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 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. 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 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 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 the timing behaviours to be expressed in terms of its characteristics and features. Porting such programs to different platforms may require considerable efforts. Detailed verifications must consider combinations of program and platform.

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 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 2. There are three outputs: Enable, Coil1, and Coil2. The Enable signal is asserted to allow current into the stepper motor coils. The Coil1 and Coil2 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 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

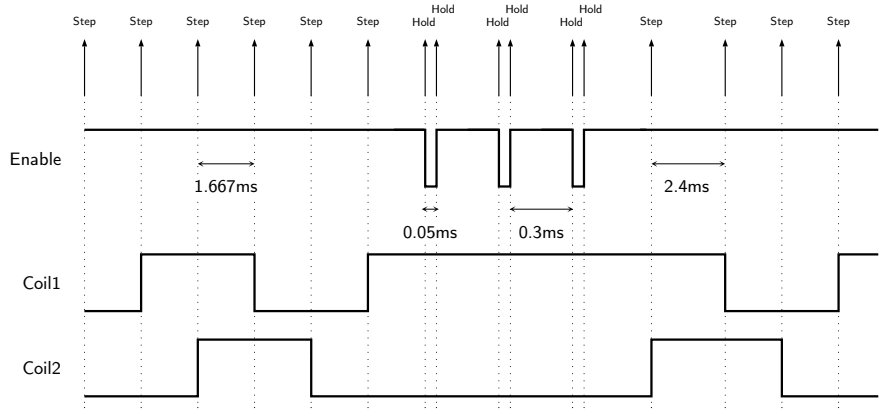


Fig. 2: Typical microprinter motor control signals

```
1  module PrintSteps:
2  input Hold;
3  output Step, Enable : boolean;
4
5  signal LongStep in
6    loop
7      emit Step;
8      present LongStep
9        then % delay 2.4ms
10       else % delay 1.667ms
11     end present;
12
13     present Hold then
14       trap Stalling in
15         loop
16           emit Enable(false);
17           % delay 0.05ms;
18           present Hold else
19             exit Stalling
20           end;
21
22           emit Enable(true);
23           % delay 0.3ms;
24           present Hold
25             else exit Stalling
26           end
27         end loop
28       ||
29         % delay 0.733ms;
30         sustain LongStep
31       end trap
32     end present
33   end loop
34 end signal
35 end module
```

(a) PrintSteps module

```
1  module Stepper:
2  input Step;
3  output
4    Coil1 := false : boolean;
5    Coil2 := false : boolean;
6
7  loop
8    await Step;
9    emit Coil1(true);
10
11   await Step;
12   emit Coil2(true);
13
14   await Step;
15   emit Coil1(false);
16
17   await Step
18   emit Coil2(false)
19 end loop
20
21 end module
```

(b) Stepper module

Fig. 3: Stepper motor controller in Esterel

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 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 standard techniques for expressing the delay are evaluated in the next section, but it turns out that 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.

3 Expressing delays in Esterel

Timing delays can be expressed variously in Esterel. Several standard 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.

3.1 Pause statements

In the modern semantics of Esterel [4, 5], **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** statement could be replaced with: **await** $[i_1$ **or** $\ldots$ **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.

In the sample-driven execution mode, 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. 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. An implementation choice must be made 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 microprinter 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.

3.2 Timing inputs

Counting specific signals instead of reactions is a partial remedy for some of the 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

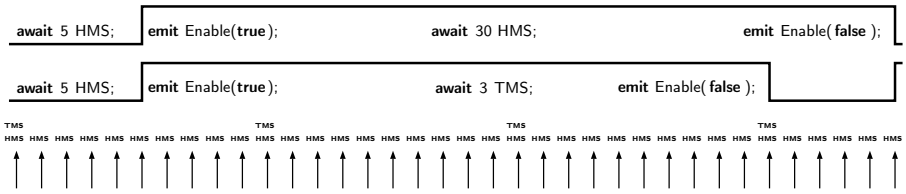


Fig. 4: Granularity of timing inputs

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. 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.

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; .
```

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 Enable(false);
  await 5 HMS;
  emit Enable(true);
  await 30 HMS
end loop.
```

Timing inputs are employed in several examples [6,7]. 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 Enable(false);
  await 5 HMS;
  emit Enable(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 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. Moreover, the timing inputs in early stages of a design may be in multiples of seconds, but those in later versions may differ. Discretization is ultimately an issue of implementation.

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, unlike in tools like Uppaal [8] where timed traces can be explored symbolically.

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 [9–11].

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 [12, §1.3.2] 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 Enable(false);
  emit START_HMSTIMER01(5); await HMSTIMER01;
  emit Enable(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 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 other Esterel statements. Consider this program fragment for example:

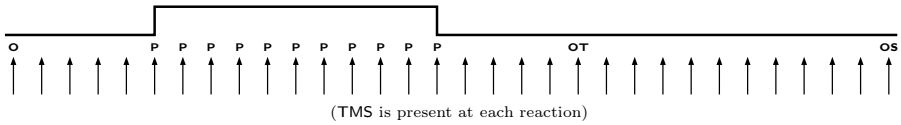


Fig. 5: Effect of suspend on delays

```
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.
```

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.

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 5. 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.

3.4 External intervals

Esterel is extended to CRP (Communicating Reactive Processes) [13] 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 Enable(false);
  exec HMSTIMER01(5);
  emit Enable(true);
  exec HMSTIMER01(30);
end loop.
```

The HMSTIMER01 process is assumed to sleep for the given number of hundredths of milliseconds and then terminate.

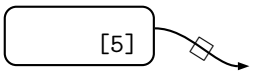
There are three main advantages over the timers described in the previous subsection. The delay is expressed as a single statement, which makes programs easier to read and simpler to analyze. The semantics of **exec** precisely defines its interaction with **abort**. The semantics also accounts for issues 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 [14], and there are similar issues with simulation and analysis tools.

3.5 Quantitative watchdogs

The Argos language defines a temporized state macro 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* [15] 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 [15]. 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 (Boolean Mealy Machine). In the continuous-time semantics, an Argos program is interpreted as a timed automaton[1] 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 [17], and there are fewer obstacles to creating simulation and debugging tools that take advantage of the timing parameters.

There are, however, at least three deficiencies. 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 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. This paper suggests an inverse approach: to specify delays in continuous time and then to implement them using standard synchronous language techniques.

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*.

[1] Technically, a *timed graph* [16].

The desired characteristics of a language for expressing the timing behaviour of applications like the microprinter controller are summarised in §4.1. The extension to Esterel that attempts to realise them is presented in §4.2. Some related approaches are discussed in §4.3.

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 with 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 §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.

4.2 Esterel+delay

The program of Figure 3 is already an excellent specification. It expresses the desired behaviours in the same terms as the physical model, as 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 §§3.1–3.4, 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.

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:

x h $= x * 3600$	x ms $= x * 10\mathsf{E}{-3}$
x m $= x * 60$	x us $= x * 10\mathsf{E}{-6}$
x s $= x * 1$	x ns $= x * 10\mathsf{E}{-9}$

Uncommenting the delay statements in the program of Figure 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 may 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 9 and 10 of Figure 3a.

At first glance the distinguished role of physical time in **delay** statements may seem to violate the doctrine of *multiform time* [18, §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 types of platform statement will be considered: one for sample-driven executions and two for event-driven executions.

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 1.4 ms

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

⟨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⟩ denote expressions that evaluate, respectively, to rational numbers and integers.

Fig. 6: Concrete syntax of platform statements

parameters: the physical time period of each tick, the minimum number of ticks possible, the maximum 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 is a list of input signal names together with the periods of their occurrence in physical time, for example:

```
event [SEC=1, OSC=90.0422ns]
```

Such statements clarify assumptions that are 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. 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 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 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 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.

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 4. *The* carrier function $\mathcal{C}(p)$ *is defined for every Esterel statement p:*

$$\begin{aligned}
\mathcal{C}(\textbf{nothing}) &= \textbf{nothing} \\
\mathcal{C}(\textbf{emit}\ \mathrm{s}) &= \textbf{emit}\ s \\
\mathcal{C}(\textbf{pause}) &= \textbf{pause} \\
\mathcal{C}(\textbf{present}\ \mathrm{s}\ \textbf{then}\ \mathrm{p}\ \textbf{else}\ \mathrm{q}\ \textbf{end}) &= \textbf{present}\ s\ \textbf{then}\ \mathcal{C}(p)\ \textbf{else}\ \mathcal{C}(q)\ \textbf{end} \\
\mathcal{C}(\textbf{suspend}\ \mathrm{p}\ \textbf{when}\ \mathrm{s}\ \textbf{end}) &= \textbf{suspend}\ \mathcal{C}(p)\ \textbf{when}\ s\ \textbf{end} \\
\mathcal{C}(\mathrm{p}\ ;\ \mathrm{q}) &= \mathcal{C}(p)\ ;\ \mathcal{C}(q) \\
\mathcal{C}(\textbf{loop}\ \mathrm{p}\ \textbf{end}) &= \textbf{loop}\ \mathcal{C}(p)\ \textbf{end} \\
\mathcal{C}(\mathrm{p}\ ||\ \mathrm{q}) &= \mathcal{C}(p)\ ||\ \mathcal{C}(q) \\
\mathcal{C}(\textbf{trap}\ \mathrm{T}\ \textbf{in}\ \mathrm{p}\ \textbf{end}) &= \textbf{trap}\ T\ \textbf{in}\ \mathcal{C}(p)\ \textbf{end} \\
\mathcal{C}(\textbf{exit}\ \mathrm{T}) &= \textbf{exit}\ T \\
\mathcal{C}(\textbf{signal}\ \mathrm{s}\ \textbf{in}\ \mathrm{p}\ \textbf{end}) &= \textbf{signal}\ s\ \textbf{in}\ \mathcal{C}(p)\ \textbf{end}
\end{aligned}$$

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

$$l = \max\left(\left\lfloor \frac{d}{\tau} \right\rfloor, 1\right) \tag{1}$$

$$u = \left\lceil \frac{d}{\tau} \right\rceil \tag{2}$$

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 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:

$$\frac{l_n}{l_d} = d - l \cdot \tau, \tag{3}$$

where the subscripts n and d stand for 'numerator' and 'denominator' respectively. And similarly for the overapproximation:

$$\frac{u_n}{u_d} = u \cdot \tau - d. \tag{4}$$

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

$$c = m \cdot \frac{l_n}{l_d} - n \cdot \frac{u_n}{u_d}, \tag{5}$$

which can be scaled to an integer by multiplying by $l_d \cdot u_d$, giving

$$c \cdot l_d \cdot u_d = m \cdot l_n \cdot u_d - n \cdot u_n \cdot l_d. \tag{6}$$

It can be tracked by an integer variable which is increased whenever the underapproximation is applied by

$$d_l = l_n \cdot u_d, \tag{7}$$

and decreased whenever the overapproximation is applied by

$$d_u = u_n \cdot l_d. \tag{8}$$

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 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,

$$\mathcal{T}_\tau(\textbf{delay}_{\mathsf{under}}\ \mathrm{d}) = \textbf{await}\ l\ \mathsf{tick}\ , \text{ and}$$
$$\mathcal{T}_\tau(\textbf{delay}_{\mathsf{over}}\ \mathrm{d}) = \textbf{await}\ u\ \mathsf{tick}\ ,$$

and, when $d \cdot \tau \geq 1$,

$$\begin{aligned}\mathcal{T}_\tau(\textbf{delay}_{\mathsf{avg}}\ \mathrm{d}) = {} & \textbf{if}\ \ \mathsf{abs}(\,\mathsf{diff} + d_l\,) <= \mathsf{abs}(\,\mathsf{diff} - d_u\,)\\ & \quad \textbf{then}\ \ \mathsf{diff} = \mathsf{diff} + d_l\,;\\ & \qquad \textbf{await}\ l\ \mathsf{tick}\\ & \quad \textbf{else}\ \ \mathsf{diff} = \mathsf{diff} - d_u\,;\\ & \qquad \textbf{await}\ u\ \mathsf{tick}\\ & \textbf{end}\ \ \textbf{if}\ ,\end{aligned}$$

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.

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 §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 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. *A* delay term *is formed from constants in* $\mathbb{Q}^{\geq 0}$*, and the two binary operators* ; *and* $\|$.

Definition 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 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 8. *The* timer count *function T_n gives the number of timers required for a delay term:*

$$\begin{aligned} T_n(0) &= 0 \\ T_n(d) &= 1 \\ T_n(d_1 \,;\, d_2) &= \max(C_n(d_1), C_n(d_2)) \\ T_n(d_1 \parallel d_2) &= C_n(d_1) + C_n(d_2) \end{aligned}$$

Two functions are introduced to evaluate the suitability of a particular timer for a particular delay.

Definition 9. *The* timer match *function T_m maps a delay d and 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 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 measures 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 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:*

$$\mathcal{T}_\tau(\textbf{delay}_i\ \mathrm{d}) = \textbf{emit}\ \mathit{start}(\mathit{timer}_a(i))\big(\mathit{value}_a(i)\big)\ ;\ \textbf{await}\ \mathit{finish}(\mathit{timer}_a(i))\,,$$

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.

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.[2] 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 sequencing of microsteps.

Issues of abortion and timer management are addressed better by the technique of §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 §4.2. Compilers should emit a warning for programs where **delay** statements are subject to suspension.

[2] It is unimportant if it is also aborted instantaneously because then there would be no statement awaiting the timeout, which would either be later reallocated or ignored.

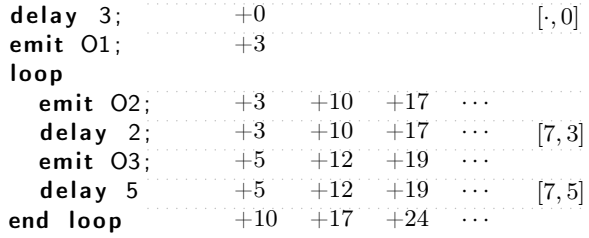


Fig. 7: Phase relationships in an Esterel+delay program

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 §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 $(n-1) \cdot \tau < t \leq n \cdot \tau$.[3] 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 1 and 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, however, sometimes be more suitable than the upper approximation, depending on the start time of a particular **delay** statement relative to the period of a given timing input. It is sometimes possible to statically determine the 'phase relationships' between **delay** statements, relative to system startup. An example is presented in Figure 7. 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 zero offset. The second and third both have a period of 7, the total delay of 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**

[3] The reason for the open lower bound of $(n-1) \cdot \tau$ is explained in §3.2.

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 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.

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 for implementing or otherwise discretizing timed automata, like, for instance, the AASAP (Almost As Soon As Possible) semantics [19]. The focus of this section is, however, on the incorporation of continuous time elements into synchronous languages. Five approaches are especially relevant: the TAXYS methodology, temporized Argos (as described in §3.5), two extensions to the Quartz language [20, 21], and a proposal for validating the real-time constraints of Esterel programs [22].

The proposal for Esterel+delay is influenced by the TAXYS [3, 23] 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 transformed to discrete controllers in 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 [15] 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 [21] and hybrid systems extensions [20] have been proposed.

Quartz programs can be translated into timed Kripke structures to verify quantitative properties [21]. Delays are expressed by **pause** statements. An abstraction statement is added to ignore intermediate polling states; for instance, **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 abstract designs prior to the consideration of implementation details. The translation is based on logical time since *physical time... depends on the hardware chosen for the realization* [21, §1]. The proposal for Esterel+delay suggests a different possibility.

The hyperQuartz language [20] 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 evolve according to 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 [22] for validating the real-time behaviour of Esterel programs,[4] locations and blocks of statements are annotated with markers to which timing constraints, that are stated separately, may then refer. For example [22, §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.
```

[4] The paper allows the **exec** statement but not the **suspend** statement.

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 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.

5 Unfinished work

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 §4, although the analysis of phase relationships does require 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 an-

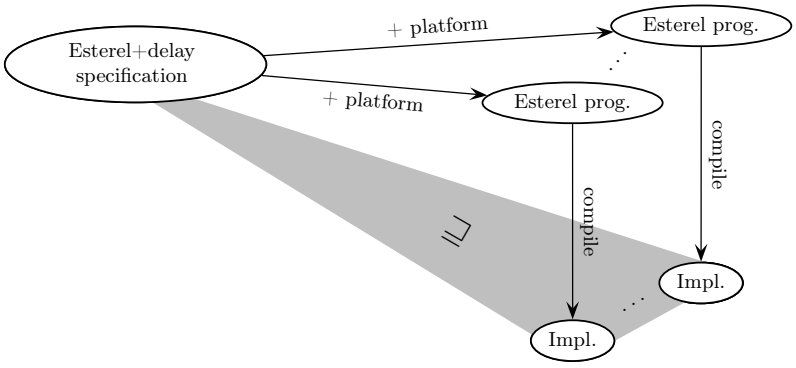


Fig. 8: Esterel+delay: artifacts and relations

other. But, in fact, there is no certainty that this is even possible. At least not without sacrificing some of the essential character and balance of Esterel, or without resorting to intricate formalisations.

A semantic treatment should address the comparison of Esterel+delay programs with the implementations generated from them. The basic idea is depicted in Figure 8. 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 [24] 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 Esterel programs would normally be 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 inputs.

6 Summary

It is argued in this paper 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.

7 Acknowledgements

S. Ramesh of GM Research Labs in Bangalore, India gave useful feedback on an early draft of this paper. 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.

References

1. Berry, G., Moisan, S., Rigault, J.P.: 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), Arlington, Virginia, USA, IEEE Computer Society (December 1983) 30–37
2. Berry, G.: Real time programming: Special purpose or general purpose languages. In Ritter, G., ed.: Proceedings of 11th International Federation for Information Processing (IFIP) World Computer Congress, San Francisco, USA (August–September 1989) 11–17
3. Sifakis, J., Tripakis, S., Yovine, S.: Building models of real-time systems from application software. Proceedings of IEEE **91**(1) (January 2003) 100–111
4. Berry, G.: The Constructive Semantics of Pure Esterel. Draft book, 3 edn. `ftp://ftp-sop.inria.fr/meije/esterel/papers/constructiveness3.ps` (July 1999)
5. Potop-Butucaru, D., Edwards, S.A., Berry, G.: Compiling Esterel. Springer-Verlag (2007)
6. Berry, G.: Programming a digital watch in Esterel v3. Rapport de recherche 1032, Institut National de Recherche en Informatique en Automatique (INRIA), Sophia Antipolis (May 1989)

7. Berry, G., Gonthier, G.: Incremental development of an HDLC protocol in Esterel. Rapport de recherche 1031, Institut National de Recherche en Informatique en Automatique (INRIA), Sophia Antipolis (May 1989)
8. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0. In: 3rd International Conference on Quantitative Evaluation of Systems, Riverside, California, USA, IEEE Computer Society (September 2006) 125–126
9. Castelluccia, C., Dabbous, W., O'Malley, S.: Generating efficient protocol code from an abstract specification. ACM SIGCOMM Computer Communication Review **26**(4) (October 1996) 60–72
10. Jagadeesan, L.J., Puchol, C., Olnhausen, J.E.V.: A formal approach to reactive systems software: A telecommunications application in Esterel. In: Proceedings of Workshop on Industrial-Strength Formal Specification Techniques, Florida, USA, IEEE (April 1995) 132–145
11. Murakami, G.J., Sethi, R.: Parallelism as a structuring technique: Call processing using the Esterel language. In van Leeuwen, J., ed.: Proceedings of 12th International Federation for Information Processing (IFIP) World Computer Congress. Number 92 in Information Processing, Madrid, Spain (1992) 10–16
12. Balarin, F., Chiodo, M., Giusto, P., Hsieh, H., Jurecska, A., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A., Sentovich, E., Suzuki, K., Tabbara, B.: Hardware-Software Co-design of Embedded Systems: The POLIS Approach. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers (1997)
13. Berry, G., Ramesh, S., Shyamasundar, R.K.: Communicating reactive processes. In: Proceedings of 20th ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages (POPL 1993), ACM Press (1993) 85–98
14. Berry, G.: Preemption in concurrent systems. In Shyamasundar, R.K., ed.: Foundations of Software Technology and Theoretical Computer Science. Volume 761 of Lecture Notes in Computer Science., Bombay, India, Springer-Verlag (December 1993)
15. Jourdan, M., Maraninchi, F., Olivero, A.: Verifying quantitative real-time properties of synchronous programs. In Courcoubetis, C., ed.: 5th International Conference on Computer Aided Verification. Volume 697 of Lecture Notes in Computer Science., Elounda, Greece, Springer-Verlag (June/July 1993) 347–358
16. Alur, R., Courcoubetis, C., Dill, D.: Model-checking for real-time systems. In: Proceedings of 5th Annual IEEE Symposium on on Logic in Computer Science (LICS '90), IEEE Computer Society (June 1990) 414–425
17. Yovine, S.: Kronos: A verification tool for real-time systems. International Journal of Software Tools for Technology Transfer **1**(1–2) (December 1997) 123–133
18. Berry, G.: The Esterel v5 Language Primer. Ecole des Mines and INRIA. 5.92 edn. (June 2000)
19. De Wulf, M., Doyen, L., Raskin, J.F.: Almost ASAP semantics: from timed models to timed implementations. In: HSCC 04: Hybrid Systems—Computation and Control. Number 2993 in Lecture Notes in Computer Science, Springer-Verlag (2004) 296–310
20. Baldamus, M., Stauner, T.: Modifying Esterel concepts to model hybrid systems. Electronic Notes in Theoretical Computer Science **65**(5) (July 2002) 819–833
21. Logothetis, G., Schneider, K.: Extending synchronous languages for generating abstract real-time models. In: Proceedings of Design, Automation and Test in Europe (DATE'02), Paris, IEEE Computer Society (March 2002) 795–803

22. Shyamasundar, R., Aghav, J.: Validating real-time constraints in embedded systems. In: 8th Pacific Rim International Symposium on Dependable Computing (PRDC 2001), Seoul, Korea, IEEE Computer Society (December 2001) 347–355
23. Bertin, V., Closse, E., Poize, M., Pulou, J., Sifakis, J., Venier, P., Weil, D., Yovine, S.: Taxys = Esterel + Kronos: A tool for verifying real-time properties of embedded systems. In: Proceedings of 40th IEEE Conference on Decision and Control, Orlando, Florida, USA, IEEE (December 2001) 2875–2880
24. Bohnenkamp, H., Stoelinga, M.: Quantitative testing. In: Proceedings of 8th ACM International Conference on Embedded Software (EMSOFT'08), Atlanta, Georgia USA, ACM, ACM Press (October 2008) 227–236