Albert Benveniste, Stephen A. Edwards,
Edward Lee, Klaus Schneider, and
Reinhard von Hanxleden

SYNCHRON'09

Proceedings of Dagstuhl Seminar 09481

Schloß Dagstuhl, Germany, November 22-27, 2009

Dagstuhl Seminar Proceedings 09481
SYNCHRON 2009
http://drops.dagstuhl.de/opus/volltexte/2010/2434

Organization

The 16th SYNCHRON workshop has been organized as Dagstuhl seminar 09481 from November 22-27, 2009. Online material of the seminar is available at the following web page:

`http://www.dagstuhl.de/de/programm/kalender/semhp/?semnr=09481`

Previous SYNCHRON workshops that were organized as Dagstuhl seminars:

04491 2004 "Synchronous Programming"
01491 2001 "Synchronous Languages"
9650 1996 "Synchronous Languages"
9448 1994 "Synchronous Languages"

Organizers

Albert Benveniste	INRIA — Rennes, FR
Stephen A. Edwards	Columbia University, US
Edward A. Lee	University of California — Berkeley, US
Klaus Schneider	TU Kaiserslautern, DE
Reinhard von Hanxleden	Universität Kiel, DE

Participants

Joaquin Aguado, Universität Bamberg
Cedric Auger, INRIA — Orsay
Fernando Barros, University of Coimbra
Albert Benveniste, INRIA — Rennes
Egon Boerger, University of Pisa
Julien Boucaron, INRIA Sophia Antipolis
Timothy Bourke, INRIA — Rennes
Jens Brandt, TU Kaiserslautern
Benoit Caillaud, INRIA — Rennes
Paul Caspi, VERIMAG — Gières
Albert Cohen, INRIA — Orsay
Christian Colombo, University of Malta
Gwenaël Delaval, INRIA Rhône-Alpes
Patricia Derler, Universität Salzburg
Philippe Dumont, INRIA — Orsay
Stephen A. Edwards, Columbia University
Giovanni Funchal, VERIMAG — Gières
Abdoulaye Gamatie, INRIA — Lille

Mike Gemünde, TU Kaiserslautern
Leonard Gerard, INRIA — Orsay
Manuel Gesell, TU Kaiserslautern
Dan Ghica, University of Birmingham
Alain Girault, INRIA Rhône-Alpes
Ursula Goltz, TU Braunschweig
Nicolas Halbwachs, VERIMAG — Gières
David Harel, Weizmann Institute — Rehovot
Peter Hintenaus, Universität Salzburg
Edward A. Lee, University of California — Berkeley
Gerald Luettgen, Universität Bamberg
Louis Mandel, INRIA — Orsay
Florence Maraninchi, VERIMAG — Gières
Slobodan Matic, University of California — Berkeley
Mohamed Nabih Menaa, University of Birmingham
Michael Mendler, Universität Bamberg
Lionel Morel, INSA — Lyon
Matthieu Moy, VERIMAG — Gières
Gordon Pace, University of Malta
John Plaice, Univ. of New South Wales
Florence Plateau, INRIA — Orsay
Dumitru Potop-Butucaru, INRIA — Le Chesnay
Marc Pouzet, Université Paris-Sud and INRIA
Pascal Raymond, VERIMAG — Gières
Eric Rutten, INRIA Rhône-Alpes
Stephan Scheele, Universität Bamberg
Klaus Schneider, TU Kaiserslautern
Peter Schrammel, INRIA Rhône-Alpes
Sandeep K. Shukla, Virginia Polytechnic Institute — Blacksburg
Satnam Singh, Microsoft Research UK — Cambridge
Martin Strecker, Université Paul Sabatier — Toulouse
Jean-Pierre Talpin, INRIA — Rennes
Stavros Tripakis, University of California — Berkeley
Reinhard Wilhelm, Universität des Saarlandes
Willem-Paul de Roever, Universität Kiel
Robert de Simone, INRIA Sophia Antipolis
Reinhard von Hanxleden, Universität Kiel

Sponsoring Institutions

Artist — Network of Excellence on Embedded Systems Design

Table of Contents

Executive Summary on Dagstuhl Seminar 09481 about Synchronous Languages

A. Benveniste, S.A. Edwards, E. Lee, K. Schneider, and R. von Hanxleden

Synchronous languages have been designed to allow the unambiguous description of reactive, embedded real-time systems. The common foundation for these languages is the synchrony hypothesis, which treats computations as being logically instantaneous. This abstraction enables functionality and real-time characteristics to be treated separately, facilitating the design of complex embedded systems. Digital hardware has long been designed using the synchronous paradigm; our synchronous languages were devised largely independently and have placed the technique on a much firmer mathematical foundation.

Feedback from the user base and the continuously growing complexity of applications still pose new challenges, such as the sound integration of synchronous and asynchronous, event- and time-triggered, or discrete and continuous systems. This seminar aims to address these challenges, building on a strong and active community and expanding its scope into relevant related fields. This year's workshop includes researchers in model-based design, embedded real-time systems, mixed system modeling, models of computation, and distributed systems.

The seminar was successful in bringing together researchers and practitioners of synchronous programming, and furthermore in reaching out to relevant related areas. With a record participation in this year's SYNCHRON workshop of more than 50 participants and a broad range of topics discussed, the aims seem to have been well-met. The program of the seminar was composed of around 36 presentations, all of which included extensive technical discussions. The fields covered included synchronous semantics, modeling languages, verification, heterogeneous and distributed systems, hardware/software integration, reactive processing, timing analyses, application experience reports, and industrial requirements. The discussion identified and collected specific needs for future topics, in particular the integration of different models of computation.

The SYNCHRON workshop constitutes the only yearly meeting place for the researchers in this exciting field. The workshops on Synchronous Languages started in 1993 at Schloss Dagstuhl. Since then, the workshop has evolved significantly in its sixteen years of existence. One obvious change is the citizenship of its attendees, which has shifted from being largely French to being truly world-wide. But the biggest change is in its scope, which has grown to expand many languages and techniques that are not classically synchronous but have been substantially influenced by the synchronous languages' attention to timing, mathematical rigor, and parallelism. Also, while many of the most senior synchronous language researchers are still active, many younger researchers have also entered the fray and taken the field in new directions. We look forward to seeing where they take us next.

Synchronous and Asynchronous Abstract State Machines

Egon Börger

Dipartimento di Informatica, University of Pisa, Italy

Sequential ASMs (Abstract State Machines) are characterized by the synchronous parallelism of executing in one step each of their fireable rules. They are turned into asynchronous ASMs by changing the underlying notion of run from sequences to partial orders which satisfy three natural constraints. What is a step (and a sequence of steps) can furthermore be explicitly managed by using control-state ASMs, which typically are components of asynchronous ASMs. Concurrent control-state ASMs are equivalent to Lamport's +CAL programs.

Clock Refinement in Imperative Synchronous Languages

Mike Gemünde, Jens Brandt, and Klaus Schneider

Embedded Systems Group
Department of Computer Science, University of Kaiserslautern
P.O. Box 3049, 67653 Kaiserslautern, Germany
http://es.cs.uni-kl.de

Abstract. The fundamental principle of synchronous languages is the division of the program execution into a sequence of logical steps. On the one hand, this programmer's view simplifies many analyses and synthesis procedures, but on the other hand, it imposes inflexible restrictions on developers and compilers. To lower these restrictions, we propose the refinement of logical steps into substeps. We illustrate our approach by extending the imperative synchronous language Quartz by new statements, which allow developers to explicitly declare subclocks. Thereby, clocks can be refined by several independent subclocks so that a controlled amount of asynchrony between subsequent synchronization points can be exploited by compilers. The advantages obtained by the new modeling feature are finally illustrated by various examples.

1 Introduction

The synchronous model of computation [1] is the basis for many programming languages, including Esterel [3], Lustre [12] or Quartz [20]. Its core is the synchronous hypothesis, which postulates that computation and communication are executed as *micro steps* in zero time. Consumption of time is explicitly modeled by grouping micro steps to macro steps, which all consume the same amount of logical time. As a consequence, all parts of the program run in lockstep and automatically synchronize at the end of each macro step. Since all micro steps of a macro step are executed at the same point of time (at least from the semantic point of view), their ordering within the macro step is irrelevant. Therefore, values of variables are determined with respect to macro steps instead of micro steps, i. e. variables do not change within a macro step.

This abstraction guarantees many properties which are desirable for development of safety-critical embedded systems. First, *deterministic* single-threaded code can be generated from multi-threaded synchronous programs, which, for instance, can be directly executed on simple micro-controllers without using complex operating systems [9, 10, 13]. Second, synchronous programs can be straightforwardly translated to hardware circuits [2, 18, 20]. Finally, the concise formal semantics of synchronous languages makes them particularly attractive

for reasoning about program properties, correctness and worst-case execution time [4, 15, 16, 19, 23, 25].

However, the synchronous model of computation imposes tight restrictions that lead to an unfortunate inflexibility of already created systems: An apparent drawback is the single abstraction layer provided by micro and macro steps, which may lead to a possible *over-synchronization* in a synchronous program. For example, compilers (at the back-end of the language) are challenged when generating efficient code for programs consisting of sporadically communicating threads, since all parts of the program implicitly synchronize after each step, even if there are no data dependencies. While a static clock and data-flow analysis may be able to detect this effect and to desynchronize such programs [7, 8], adding an explicit notion of independence makes it possible for compilers to create desynchronized code without expensive analyses. Additionally, spurious synchronization can be prevented by construction. Similarly, developers (at the front-end of the language) are limited by the single temporal abstraction layer of the synchronous model of computation. It abstracts from the causality and scheduling of the operations within a single macro step but there is no support for a more coarse-grained structure of logical time. For example, this immediately causes problems if several existing *modules of different abstraction levels* should be combined.

Using a hierarchy of clocks in the system description is an apparent approach to tackle these problems, while still preserving all advantages of the synchronous model of computation. A crucial point in the design of a derived multi-clock model of computation is the construction of the clock hierarchy. In general, one can distinguish three different alternatives for this: (1) new clocks are created independently from each other and subsequently related by *explicit clock constraints* to form a clock hierarchy, (2) new clocks are created by *downsampling* already existing clocks (bottom-up hierarchy), and (3) new clocks are created by *upsampling* already existing clocks (top-down hierarchy). While all alternatives seem to be equivalent at first sight, a closer look reveals that there are significant differences. In particular, the last alternative bears an enormous potential, which has not been used so far due to the lack of full support by state-of-the-art synchronous languages.

In this paper, we present an extension of the imperative synchronous programming language Quartz which gives developers the possibility to declare so-called *subclocks*. These subclocks are the first full support of a top-down hierarchy of clocks. The allow developers to refine the temporal behavior of synchronous systems to avoid over-synchronization. Additionally, subclocks allow to exchange components with others having a different internal temporal behavior. Hence, clock refinements provide additional degrees of freedom for synthesis and design space exploration. At the same time, we preserve all desired advantages of the synchronous languages: fundamental properties such as the input-output determinism are maintained as well as the fully orthogonal structure of the programming language, which allows developers to arbitrarily nest all kinds of statements.

The rest of the paper is structured as follows: Section 2 first reviews basic concepts of synchronous systems in general and the synchronous language Quartz. Section 3 shows how they can be enriched by a top-down hierarchy of clocks. Thereby, we show how syntactical restrictions of the language extension can be used to enforce desired properties. Finally, Section 4 draws some conclusions and sketches future work.

2 Synchronous Quartz

2.1 Synchronous Model of Computation

The synchronous model of computation [1,11] divides the execution of a program into a sequence of reaction steps. In each of these steps (which are often called macro steps [14]), the system reads its inputs, does some computation and finally provides all outputs (that appear from the semantic point of view at the same instant). In practice, this means that the execution implicitly follows the data dependencies of the system.

```
a = 1;          b = a;          a = 1;
b = a;          a = 1;          if(b = 1)
pause;          pause;             b = a;
a = b;          a = b;          pause;
                                if(a ≠ 2)
                                   a = b;
```

(a) (b) (c)

Fig. 1: Three Quartz Programs Illustrating Synchronous Execution

In imperative synchronous languages, this model of computation is represented as follows: All actions are assumed to be executed in zero-time. The `pause` statement determines the end of the current macro step and is therefore *responsible* for consuming (logical) time. A simplified view on the programs is therefore as follows: In each macro step, a synchronous program resumes its execution at the `pause` statements where the control flow has been stopped at the end of the previous macro step, then it reads new inputs and executes the following statements until the next `pause` statements are reached (see Figure 1 (a)). Thereby, the control flow can rest at multiple `pause` statements. In concurrent programs, all threads follow this principle, i. e. they run in lockstep and automatically synchronize at each macro step.

However, the assumption that all statements are executed in zero-time has some consequences which might be confusing at a first glance: Since a statement does not take time for its execution, it is evaluated in the same variable environment as another statement following it in a sequence. In principle, both statements may therefore be interchanged without changing the behavior of the

program. So, the program in Figure 1 (b) has the same behavior as the program in Figure 1 (a). Thus, each statement knows and depends on the results of all operations in the current macro step. Obviously, this generally leads to dependencies that are not present in sequential programming languages. In particular, it is possible that a statement influences its own activation condition, which may lead to semantic problems (see the program in Figure 1 (c)). So-called causally incorrect programs are the result, which have no consistent behavior. This is a well-studied problem for synchronous systems and many analysis procedures have been developed to spot these problems [17,21,22,24,26]. While this model of computation seems to be unnatural at first, this represents exactly the way a synchronous hardware circuit works: all computation and communication within a clock cycle happens more or less simultaneously according to the data dependencies.

Another problem related to the synchronous model of computation are write conflicts. They occur if several actions try to assign different values to the same variable in the same step. Since a variable can only carry a single value in each step, programs that show this behavior are also considered to be incorrect, and have to be rejected by compilers.

2.2 Synchronous Quartz

In this paper, we extend the imperative synchronous language Quartz [20], which has been derived from Esterel [5,6]. In the following, we give a brief overview of the core of the language, which is sufficient to define most other statements as simple syntactic sugar.

For each statement we will describe its behavior; for the sake of simplicity, we do not give a formal definition; the interested reader is referred to [20] instead, which provides a complete structural operational semantics. The Quartz core (see Figure 2) consists of the following statements, provided that S, S_1, and S_2 are also core statements, ℓ is a location variable, x is a variable, σ is a Boolean expression, and α is a type.

`nothing`	(empty statement)
$x = \tau$ and `next`$(x) = \tau$	(assignments)
ℓ : `pause`	(start/end of macro step)
`if`(σ) S_1 `else` S_2	(conditional)
$S_1; S_2$	(sequence)
`do` S `while`(σ)	(iteration)
$S_1 \parallel S_2$	(synchronous concurrency)
`[weak] [immediate] abort` S `when`(σ)	(abortion)
`[weak] [immediate] suspend` S `when`(σ)	(suspension)
$\{\alpha\ x;\ S\}$	(local variable x of type α)
$inst$: $name(\tau_1, \ldots, \tau_n)$	(call of module *name*)

Fig. 2: Quartz Core Statements

There are two kinds of assignments: Both kinds of assignments immediately evaluate the right-hand side expression τ in the variable environment of the current macro step. Immediate assignments $x = \tau$ instantaneously transfer the obtained value of τ to the left-hand side x, whereas delayed ones $\texttt{next}(x) = \tau$ transfer this value in the following macro step. If a variable is not set by an action in the current macro step, its value is determined by the so-called *reaction to absence*, which generally depends on the data type of the variable. For usual memorized variables, the value of the previous step is kept.

A `pause` statement is used to mark the beginning of a new macro step and to define a *control-flow location*. Since all other statements are executed in zero time, the control flow can only rest at these positions in the program.

```
{                          ||   {
   b = true;               ||
   ℓ1 : pause;             ||      ℓ3 : pause;
   if(a)                   ||      if(¬b)
      b = false;           ||         c = true;
                           ||      a = true;
   ℓ2 : pause;             ||      ℓ4 : pause;
                           ||      b = true;
}                          ||   }
```

Fig. 3: Synchronous Concurrency in Quartz

In addition to the usual control-flow constructs, which are known from typical imperative languages (conditionals, sequences and iterations), Quartz offers synchronous concurrency. The *parallel statement* $S_1 \parallel S_2$ immediately starts the statements S_1 and S_2. Then, both S_1 and S_2 run in lockstep, i. e. they automatically synchronize when they reach the next `pause` statement. The whole parallel statement runs as long as one of the sub-statements is active.

Figure 3 shows a simple example consisting of two parallel threads. In the first step, the program, and thus both threads, are started. The first thread executes the assignment to b and stops at location ℓ_1, while the second thread directly moves to location ℓ_3. In the second macro step, the program resumes at the labels ℓ_1 and ℓ_3. Since the second thread contains an immediate assignment to a, the action resetting b in the first threads is activated, which in turn activates the actions setting c in the second thread. The last step then resumes from ℓ_2 and ℓ_4, where the second thread performs the final assignment to variable b.

Preemption can be conveniently implemented by the `abort` and `suspend` statements. Their meaning is as follows: A statement S which is enclosed by an `abort` block is immediately terminated when the given condition σ holds. Similarly, the control flow in a statement S enclosed by `suspend` is frozen when the given condition σ holds. Thereby, two kinds of preemption must be distinguished: strong (default) and weak (indicated by keyword `weak`) preemption.

While strong preemption deactivates both the control and data flow of the current step, weak preemption only deactivates the control flow, but retains the current data flow of this macro step. The immediate variants check for preemption already at starting time, while the default is to check preemption only after starting time.

(a)
```
abort {
  a = 1;
  ℓ1 : pause;
  b = 2;
  ℓ2 : pause;
} when(true);
c = 3;
```

(b)
```
immediate abort {
  a = 1;
  ℓ1 : pause;
  b = 2;
  ℓ2 : pause;
} when(true);
c = 3;
```

(c)
```
weak abort {
  a = 1;
  ℓ1 : pause;
  b = 2;
  ℓ2 : pause;
} when(true);
c = 3;
```

(d)
```
weak immediate abort {
  a = 1;
  ℓ1 : pause;
  b = 2;
  ℓ2 : pause;
} when(true);
c = 3;
```

Fig. 4: Variants of the `abort` Statement

Figure 4 shows examples of all four abort variants. The execution of the programs (a) and (c) take two macro steps. The first step is executed without checking the abort condition and a is set to 1. In the second macro step, the condition is checked and the abortion takes place. Hence, c is set in both cases, whereas b is because of the weak abortion just set for the program (c). The other two program examples (b) and (d) take just one step to execute. Since for the immediate variants the abort condition is checked already in the first step, the abortion takes place directly. For the strong variant shown in (b) just c is set, while the weak variant shown in (d) additionally assigns the variable a.

Modular design is supported by the declaration of modules in the source code and by calling these modules in statements. Any statement can be encapsulated in a module, which further declares a set of input and output signals for interaction with its context statement.

In contrast to many concurrent languages, all statements of Quartz are fully orthogonal to each other, i. e. they can be arbitrarily nested. In particular, it is no problem to mix sequential and parallel composition and to apply preemption statements to several threads running in parallel. Furthermore, there are no restrictions for module calls, which can be part of sequences or conditionals and which can be located in any abortion or suspension context, which possibly preempts their execution. In all cases, the synchronous model of computation

and its logical time scale will take care that the program behavior remains deterministic.

3 Subclocked Quartz

In this section, we introduce the subclock extension for the synchronous language Quartz. We start by sketching the basic idea of subclocks in Section 3.1, before the actual extension is presented in Section 3.2 by introducing new statements. Thereafter, in Sections 3.3 to 3.7, we highlight the effects that the new statement has on the existing statements.

3.1 Subclocks

Traditional synchronous systems are based on a single clock that provides a sequence of ticks, which determine logical instants of time. According to the synchronous paradigm, every variable has a uniquely determined value in every instant of time. For the implementations, this generally means that the execution must follow data dependencies, so that a value is not written after it has been read before within the same instant. Figure 5 (a) shows a sample trace of reaction instants for a single-clocked system. As already described in the introduction, this view imposes many restrictions, which are caused by the single abstraction layer between micro and macro steps: they define the interaction points with the environment as well as the rate of all internal computations.

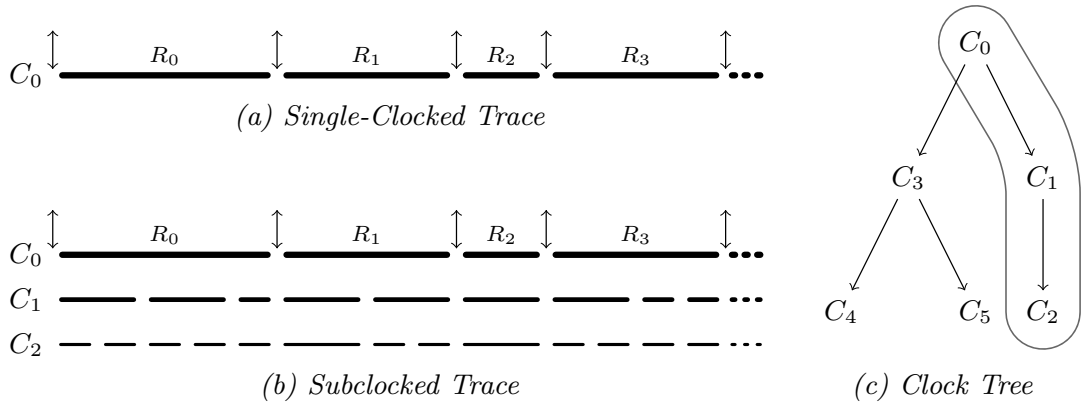


Fig. 5: Timing of Single Clocked and Subclocked Systems

To overcome these problems, we propose the introduction of so-called subclocks, which allow developers to divide macro steps on the system level into smaller steps — thus, macro steps are no longer atomic. However, *we do not make this abstraction visible to the environment of modules.* The system interface has the same timing behavior, while its internal implementation has more

freedom due to its internal subclocks. One instant on a clock level is divided into some smaller steps of the lower clock level as shown in Figure 5 (b). Thereby, variables of subclocks can have multiple values during a step on a higher level, but the variables are not visible to the higher level. Similar to the distinction of micro and macro steps, the computation which is done in one step, is hidden to the higher level and only the result is visible. The advantage compared to micro and macro steps is that the clock hierarchy provided hereby can be arbitrarily deep nested.

It is possible to refine a clock by multiple unrelated subclocks. This leads to a tree of clocks shown in Figure 5 (c), whereas only the marked branch of the tree provides the clocks shown in the trace in Figure 5 (b). In the same way as micro steps are executed within a macro step, the subclock steps are executed in a step of a higher clock. For unrelated subclocks, this is the same as micro steps, whereas in both cases they can be executed independently with respect to data dependencies on common variables. A synchronization is enforced by the end of a step of a common superclock.

Variables can be declared for every clock, but those variables are only visible for lower clocks and not for the higher ones. Hence, unrelated clock domains can just communicate through variables declared on a common shared superclock. Such variables remain constant during the execution of the subclocks until a synchronization.

In the following, we write $C' \prec C$, iff C' is a subclock of C, i. e. when C' is on a lower level in the tree and both are on the same branch. For *unrelated* clocks C, C', i. e. neither $C' \preccurlyeq C$ nor $C \preccurlyeq C'$ holds, we write $C \# C'$. E. g. $C_3 \# C_2$ and $C_4 \# C_1$ holds for the tree in Figure 5 (c). In this paper we denote the clock of the system with C_0, which is also the clock of the interface variables.

The whole model gives some internal flexibility for desynchronized implementations through unrelated subclocks while the synchronous interface to the outside environment can be kept at the same time. Since subclocks are not visible outside the module, a clock ticks for them cannot be provided by the environment, which emphasizes the view of a logical refinement instead of a multi-clock extension.

3.2 Language Extension

To make subclocks accessible to developers, we extend the synchronous programming language Quartz. Thereby, we aim at providing a comprehensive support without introducing unnecessary complexity. As it will be shown in the following, it is not simple to find a good compromise, but we try to make the design decisions transparent to show subtle problems avoided by our choice.

The core of our subclock extension is the new statement `clock`(C) $\{S\}$, which declares a subclock C and additionally determines the scope S where this subclock is visible and can be referenced. The only statement that directly addresses the new clock is a labeled pause statement `pause`(C), which enforces a synchronization with respect to the clock C. Thereby, parallel running threads

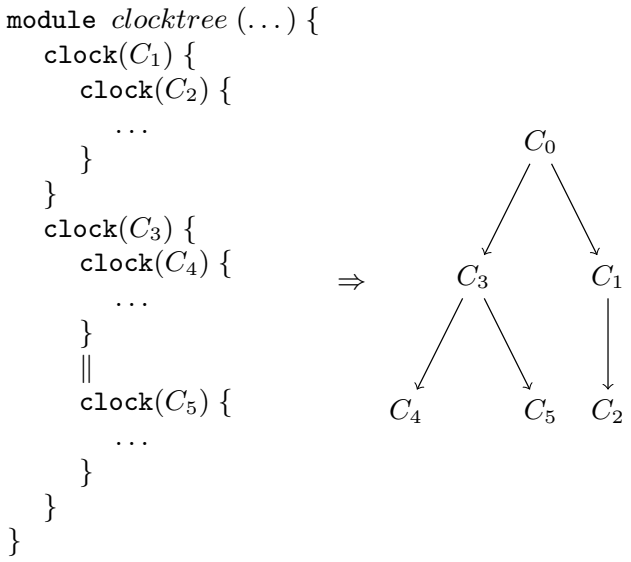


Fig. 6: Clock Tree defined by a Quartz Program

can synchronize at this clock as with any other clock. Details about this will be given later.

```
module deadlock (...) {
  clock C1, C2;
  {
    ℓ11 : pause(C1);
    ℓ12 : pause(C2);
  } || {
    ℓ21 : pause(C2);
    ℓ22 : pause(C1);
  }
}
```

Fig. 7: Deadlocks due to Interleaved Clocks

The visibility of a clock is the same as the one of a local variable declared at the beginning of a block. Nevertheless, there is a good reason for declaring clocks not in the same way as variables ($\alpha\ x$;). As shown in Figure 6, the clock tree can be directly derived from the source code and it ensures, that unrelated clocks are never visible in the same scope. Allowing clock declarations like variables would destroy this assumption, and parallel threads could request a synchronization for unrelated clocks, which ultimately leads to a deadlock. This is a well-known problem in many concurrent languages, which is due to parallel threads waiting for different events. The issue is illustrated in Figure 7 by a Quartz program, which *does not conform* to our proposed extension, since the

two declared subclocks are unrelated but visible in the same scope. It justifies why new clocks can be only declared for a fixed scope and why the shown case is prohibited. Deadlocks on clock levels cannot happen with the scope definition described above, since parallel threads can only synchronize on related clocks so that waiting for an unrelated but shared clock is impossible. The two threads would need to synchronize for the same clock, which is impossible because they both wait first for a tick of the other one.

3.3 Variable Declarations

In order to take advantage of subclocks, one needs to declare variables related to them. With the subclock statement introduced above, each variable declaration is now in at least one scope of a clock (the whole program is trivially in the scope of the system clock C_0). The clock of a variable x is directly inferred from *lowest* clock visible for the declaration, hence the next surrounding clock block (or system clock). We write $\mathsf{clk}(x) = C$ for a variable x related to clock C, whereas x can change its value each time the clock C ticks. Thus, only one assignment to x can take place between two consecutive ticks of clock C. By construction, the scope of a subclock variable is always smaller than the scope of its clock. This ensures that the variables bound to a subclock are not visible outside of the subclock domain.

3.4 Immediate and Delayed Assignments

Assignments are the core statements of the data flow and therefore, they are closely related to the clock of the contained variables. They are the only statements that change the value of variables. Each assignment consists of a target (left-hand side) and of an expression that determines its value (right-hand side). Both the target and the expression contain arbitrary variables visible at the current position in source code. Immediate assignments change the value of the variable in the current step, whereas delayed assignments commit the result in the following step. Obviously, for immediate assignments it is determined when they take place, whereas for delayed assignments it is not clear what the *following* step in the context of a subclocked time model is. Since the assignment affects the value of the target variable, the delayed assignment takes place in the next step of the clock of the assigned variable.

Together with the visibility of variable declarations, it is guaranteed that in each expression only variables of one *branch* of the clock tree occur, i. e. for each two variables x_1, x_2 which occur in the same expression, the clock relation $\mathsf{clk}(x_1) \ \# \ \mathsf{clk}(x_2)$ never holds. This is an essential property that ensures a deterministic evaluation of the expressions. If unrelated variables could occur, it would not be specified which value to take, since the other clock could make an independent step or not, and the variable could have a different value for the steps. The syntax of our extension guarantees that the only variables visible in unrelated clock domains are declared on a common higher level: these variables

```
module M1 (bool ?a, bool ?b, bool !c) {
  bool x;
  if(a) c = true;
  ℓ1 : pause;
  clock(C1) {
    bool x1, y1;
    ℓ11 : pause(C1);
    y1 = ¬c;
    if(a)
      c = true;
    else
      c = false;
    next(x1) = b;
    next(y1) = a;
    ℓ12 : pause(C1);
    next(x) = x1;
    if(x1 & y1) next(y1) = a;
    ℓ2 : pause;
    if(b) c = x & a;
    ℓ13 : pause(C1);
  }
}
```

	a	b	c	x	x_1	y_1
1.	true	true	true	false	–	–
2.	false	true	false	false	false false true	false true false
3.	false	true	false	true	false false	false false

Fig. 8: Example: Step Behavior

have a determined value, which does not change before the next synchronization of the unrelated domains.

The program shown in Figure 8 explains the basic semantics of Quartz modules with subclocks. In particular, it shows the declaration of variables related to different clocks and how the variables are changed according to the steps.

The modules M_1 has two inputs a, b and one output c. The variables of the interface and the local variable x follow to the system clock C_0. The module additionally contains a part driven by the subclock C_1, which declares the local variables x_1 and y_1. The right part of the figure shows a sample input trace together with the values of the output and local variables. The numbers in the first column of the table count the steps of the system clock. It can be seen that the subclock variables have different values for the C_1-steps — during a single C_0-step. In contrast, system clock variables only have a single value. Therefore, there may be only one assignment in all substeps (during a step of the system clock) for those variables. Otherwise, a *write conflict* occurs, which is exactly the same as the one in the synchronous case shown in Section 2.1. The example also illustrates that next assignments are naturally issued at the following step w. r. t. the clock of the assigned variable.

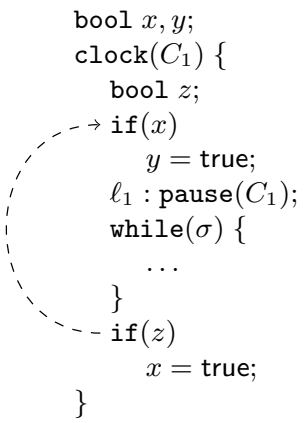


Fig. 9: Example: Information Flows Backward

Finally, a very important point remains to explain. From the semantic point of view, substeps are just micro steps for the higher clock level. In principle, they are executed simultaneously in zero time. For a subclocked program this has the consequence that information can actually flow backwards. Theoretically, a variable declared on a higher clock level can transport information backwards in time, since subclocks postulate that all read and write accesses to the super-variables occur simultaneously.

Figure 9 shows an example. The variable x is used to transport information from the end of the subclock block to the beginning. Since this variable is related to the module clock, it only has one value during the whole execution of the subclock block. According to the synchronous hypothesis, this value is also visible at the beginning of the subclock block. However, when we try to execute the subclock block, it is unclear whether the actions that set x will be executed at all. To avoid all these problems by construction, we do not rely on an extensive and complex analysis, but we add the constraint that variables of a higher clock level can only be read if they have been already assigned in the same or previous subclock step. That means, that the order of substeps must preserve the data dependencies between variables. Relaxing this condition generally makes hardware and software synthesis infeasible from programs with backward information flow[1].

3.5 Parallelism

Synchronous languages provide deterministic parallelism, which is due to the concept of micro and macro steps. All threads execute their micro steps simultaneously in zero time and synchronize at the end of each macro step. As already

[1] These programs can be only executed with a partial environment, which follows all possibilities and derives more and more information from the run until all variables are known. This behavior is similar to causality analysis [21,22,24], but it additionally requires to define transitions for partial environments.

pointed out in the introduction, this execution scheme leads to a potential over-synchronization for sporadically communicating threads. Figure 10 (a) illustrates this issue by a simple example.

The first thread of the program does some calculation and changes the value of x sporadically. The calculation is not explicitly shown in the example but it is assumed to be accomplished in the sequences of `pause` statements. The second thread runs in parallel and sums each changed value of x to y. It needs to get the correct values of x and thus it must synchronize to the first one. The second thread therefore *simulates* the same *timing* of the first one. Both threads synchronize for every `pause` and thus, more than needed, since the second thread only needs *some* values of x.

Figure 10 (b) shows the alternative subclocked implementation, where a subclock is used in the first thread. Thereby, both threads only synchronize in the steps where a synchronization is needed, i. e. only the exchange of x is controlled by the shared system clock. The second thread does not need to take care about the timing of the first thread any more, it just needs to add the value of x in each step.

Naturally, the synchronization of both threads in the example can be alternatively implemented by an explicit synchronization with the help of `await`(σ) statements, which pause as long as their condition σ is not fulfilled yet. However, even there both threads unnecessarily synchronize at each pause statement, which is problematic if the computations of both threads are unbalanced in the macro steps.

The previous example only illustrated a *dependency* in one direction, i. e. only the second thread has to adapt the timing of the first one. Apparently, it gets more complicated, when both threads must be adapted in a two-way dependency. The example just points out the issue, in the given form, it can obviously be implemented much simpler with the same behavior.

A second issue, which is pointed out by the example is the following one. For the single-clocked implementation the timing of both threads highly depend on each other. In the sense of modularity, one part of the program can only be substituted by another one, which provides the same timing. Completely changing the timing by using another algorithm destroys the global behavior. With subclocks, this can be avoided, since internal timing can be completely hidden to the outside.

Since the timing can be completely hidden to the outside, parallel threads can be executed fully asynchronous if there are no data dependencies. Figure 11 shows an example: both threads get the same input by the interface variables a, b. The first thread computes the *greatest common divisor (GCD)* of the input values. The second one multiplies them by a sequential algorithm. The result of both computations is obtained and provided as output in the same step the input is given. Thus, for every module step, the same calculation can be done for two input values.

(a)

```
{
   x = 3;
   pause;
   pause;
   pause;
   x = 4;
   pause;
   pause;
   x = 9;
   pause;
} || {
   y = x;
   pause;
   pause;
   pause;
   next(y) = y + x;
   pause;
   pause;
   next(y) = y + x;
   pause;
}
```

(b)

```
clock(C1) {
   x = 3;
   pause;
   pause(C1);
   pause(C1);
   x = 4;
   pause;
   pause(C1);
   x = 9;
   pause;
} || {
   y = x;
   pause;
   next(y) = y + x;
   pause;
   next(y) = y + x;
   pause;
}
```

Fig. 10: Example: Over-Synchronization and Temporal Abstraction

```
module M2 (nat ?a, nat ?b, nat !gcd, nat !prod) {

   clock(C1) {
      int x1, y1;
      loop {
         x1 = a;
         y1 = b;
         while(x1 ≥ 0) {
            if(x1 ≥ y1)
               next(x1) = x1 − y1;
            else
               next(y1) = y1 − x1;
            ℓ11 : pause(C1);
         }
         gcd = y1;
         ℓ1 : pause;
      }
   }

   ||

   clock(C2) {
      int x2, y2;
      loop {
         x2 = a;
         y2 = 0;
         while(x2 ≥ 0) {
            next(y2) = y2 + b;
            next(x2) = x2 − 1;
            ℓ21 : pause(C2);
         }
         prod = y2;
         ℓ2 : pause;
      }
   }

}
```

Fig. 11: Example: Parallel Computation

Since the execution time of the shown algorithm depends on the input value, an instantaneous implementation of the GCD is not possible, and therefore both algorithms have to synchronize at each step during their internal computation.

3.6 Preemption

```
clock(C1) {
   weak abort {
      clock(C2) {
         x = true;
         ℓ2 : pause(C2);
         y = true;
         ℓ1 : pause(C1);
         z = true;
         ℓ0 : pause(C0);
      }
   } when(σ);
}
```

Fig. 12: Example: Preemption

A designated feature of imperative synchronous languages are abortion and suspension statements which deterministically preempt the program execution (see Section 2.2). The subclock extension should preserve this property. The crucial point of the semantics is the definition at which points of time the preemption condition is checked (at the beginning of the current macro step) and at which point of time the preemption actually takes place (at the end of the current macro step in the case of weak abort). Again, the definition of *current macro step* has to be clarified in a subclocked context.

Consider the example in Figure 12. The `weak abort` statement is in the scope of subclock C_1, which is in the scope of C_0. In its body, another subclock C_2 is declared. By construction, the abortion condition σ cannot contain variables of clock C_2 but only variables of clocks C_1 and C_0. Hence, the condition σ can only become true if a tick of C_1 occurs. If the condition only contains variables of clock C_0, the condition will only change at ticks of C_0. Nevertheless, we check the condition at each tick of C_1 for the following reason: assume that the abortion condition σ is $x_0 \vee \mathsf{false} \wedge x_1$, where eliminating the second part would then change the set of variables and thereby the temporal behavior. In that case, compile-time optimizations would be almost impossible. The difference becomes visible for a weak abortion, since it first completes the macro step before jumping to the end of the aborted block. The execution of the example given by Figure 12 is the following if we assume that σ holds in the first step of C_1: The actions for x and y will be executed, while the one for z will be skipped, since the control

flow leaves the abort block when it hits the label ℓ_1. This label is declared on the same level as the whole abort block.

3.7 Schizophrenia

A well-known problem related to synchronous programs is schizophrenia, which is caused by local variables within loops. In the context of perfect synchrony, which groups a number of micro steps into an instantaneous macro step, a limited scope which does not match with the macro steps boundaries may cause problems. In particular, this is the case if the scope of a local declaration is left and reentered within the same macro step. This scenario is not as unusual as it may appear at first. It always occurs when local declarations are nested within loop statements. In such a problematic macro step, the micro steps must then refer to the right incarnation of the local variable, depending on whether they belong to the old or the new scope of the local declaration.

```
do {
   bool x;
   if(x)
      y = 1;
   ℓ1 : pause;
   x = true;
} while(true) ;
```

(a)

```
do {
   bool x;
   if(x)
      y = 1;
   if(a)
      ℓ1 : pause;
   x = true;
   if(¬a)
      ℓ2 : pause;
} while(true) ;
```

(b)

```
do {
   bool x;
   clock(C1) {
      bool y;
      ℓ11 : pause;
      x = true;
      ℓ12 : pause;
      ...
   }
   ℓ1 : pause;
   ...
} while(true) ;
```

(c)

Fig. 13: Schizophrenic Quartz Programs

Figure 13 (a) gives a simple example. The local variable x is referenced at the beginning and at the end of the loop. In the second step of the program, when it resumes from the label ℓ_1, all actions are executed, but they refer to two different incarnations of x.

While traditional software compilation can solve this problem simply by shadowing the variables of the old scope, this is not possible for the synchronous model of computation. Since each variable has exactly one value per macro step, we need one value for the scope which is left and one for the scope which is entered. Therefore, we have to generate a copy of the locally declared variable, which is called a *incarnation* of the variable. Additionally, the actions of the program must be mapped to the corresponding incarnation of the variable in the intermediate code. Furthermore, we have to create additional actions in the

compiled code that link the copies so that the value of the new incarnation at the beginning of the scope is eventually transported to the old one, which is used as the end of the scope.

However, the problem can be even worse in general: first, whereas in the previous example each statement always referred to the same incarnation (the old or the new one), the general case is more complicated as can be seen in Figure 13 (b). The statements between the two `pause`statements are sometimes in the context of the old and sometimes in the context of the new incarnation. Therefore, these statements are usually called *schizophrenic* in the synchronous languages community [4]. Second, there can be several reincarnations of a local variable, since the scope can be reentered more than once. In general, the number of loops that are nested around a local variable declaration determines an upper bound on the number of possible reincarnations.

Subclocks do not make the problem worse, but they widen its scope (see Figure 13 (c)). In synchronous languages, only instantaneous parts (data flow) must be copied in the course of the compilation. In the presence of subclocks the compiler needs to copy the first macro step of loops with respect to the current clock at the beginning of the loop. Thereby, it may be needed to copy variables (data flow) of subclocks as well as labels (control flow), which is not the case for single-clocked programs.

4 Conclusions

In this paper, we presented a subclock extension to imperative synchronous languages, which allows developers to use several abstractions layers in a program given by subclocks. Thereby, two significant drawbacks of the synchronous model of computation can be eliminated: First, the reuse and the maintenance of existing synchronous designs is improved, since timing can be modified locally without any global effect. Subclocks will make the changes invisible to the context. Second, implementations can make use of more internal freedom, since spurious synchronization can be prevented by construction. Whereas the synchronous model of computation and its strict determinism tends to *over-synchronize* system parts due to the single clock level, independent subclocks make it possible to omit a fixed schedule in the program, which can then be optimally adapted to the target platform by the synthesis tool.

The proposed Quartz extension is completely orthogonal to the other statements. Furthermore, we made some crucial design decision which make sure that semantic problems are prevented by the language syntax: First, declarations of clocks must be always given a scope so that a tree of clocks for the whole system sprouts automatically. Second, variables can be only declared for visible clocks. Third, assignments are not allowed to contain variables of unrelated clocks. By introducing these syntactical constraints, we could eliminate the most significant problems which usually occur in the context of multiple clocks.

However, there remains a lot of work to do in order to make use of the proposed extension. Since the synchronous language Quartz is compiled to an

intermediate format from which hardware and software can be synthesized, the same design flow must be supported for subclocks. Therefore, the formal semantics of Quartz must be extended by subclocks. To allow compilation, the existing procedures must be adjusted, and the intermediate format needs to be extended to preserve the freedom introduced by subclocks on this level. The causality analysis has to be revised and finally, the synthesis algorithms have to be extended in order to exploit the new features.

References

1. Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., de Simone, R.: The synchronous languages twelve years later. Proceedings of the IEEE 91(1), 64–83 (2003)
2. Berry, G.: A hardware implementation of pure Esterel. Sadhana 17(1), 95–130 (March 1992)
3. Berry, G.: The foundations of Esterel. In: Plotkin, G., Stirling, C., Tofte, M. (eds.) Proof, Language and Interaction: Essays in Honour of Robin Milner. MIT Press (1998)
4. Berry, G.: The constructive semantics of pure Esterel. `http://www-sop.inria.fr/esterel.org/` (July 1999)
5. Berry, G.: The Esterel v5 language primer. `http://www-sop.inria.fr/esterel.org/` (July 2000)
6. Berry, G., Gonthier, G.: The Esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming 19(2), 87–152 (1992)
7. Brandt, J., Gemünde, M., Schneider, K.: Desynchronising synchronous programs by modes. In: Conference on Application of Concurrency to System Design (ACSD). IEEE Computer Society, Augsburg, Germany (2009)
8. Brandt, J., Schneider, K.: Static data-flow analysis of synchronous programs. In: Bloem, R., Schaumont, P. (eds.) International Conference on Formal Methods and Models for Co-Design (MEMOCODE). pp. 161–170. IEEE Computer Society, Cambridge, Massachusetts, USA (2009)
9. Edwards, S.: An Esterel compiler for large control-dominated systems. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems 21(2), 169–183 (February 2002)
10. Edwards, S.: Compiling concurrent languages for sequential processors. ACM Transactions on Design Automation of Electronic Systems (TODAES) 8(2), 141–187 (2003)
11. Halbwachs, N.: Synchronous programming of reactive systems. Kluwer (1993)
12. Halbwachs, N.: A synchronous language at work: the story of Lustre. In: International Conference on Formal Methods and Models for Co-Design (MEMOCODE). pp. 3–11. IEEE Computer Society, Verona, Italy (2005)
13. Halbwachs, N., Raymond, P., Ratel, C.: Generating efficient code from data-flow programs. In: Maluszynski, J., Wirsing, M. (eds.) International Symposium on Programming Language Implementation and Logic Programming(PLILP). LNCS, vol. 528, pp. 207–218. Springer, Passau, Germany (1991)
14. Harel, D., Naamad, A.: The STATEMATE semantics of statecharts. ACM Transactions on Software Engenieering Methods 5(4), 293–333 (1996)

15. Li, Y.T., Malik, S.: Performance Analysis of Real-Time Embedded Software. Kluwer (1999)
16. Logothetis, G., Schneider, K.: Exact high level WCET analysis of synchronous programs by symbolic state space exploration. In: Design, Automation and Test in Europe (DATE). pp. 196–203. IEEE Computer Society, Munich, Germany (2003)
17. Malik, S.: Analysis of cycle combinational circuits. IEEE Transactions on Computer Aided Design 13(7), 950–956 (July 1994)
18. Rocheteau, F., Halbwachs, N.: Implementing reactive programs on circuits: A hardware implementation of LUSTRE. In: Proceedings of the Real-Time: Theory in Practice. LNCS, vol. 600, pp. 195–208. Springer, Mook, The Netherlands (1991)
19. Schneider, K.: Embedding imperative synchronous languages in interactive theorem provers. In: Conference on Application of Concurrency to System Design (ACSD). pp. 143–154. IEEE Computer Society, Newcastle upon Tyne, UK (2001)
20. Schneider, K.: The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany (2009)
21. Schneider, K., Brandt, J.: Performing causality analysis by bounded model checking. In: Conference on Application of Concurrency to System Design (ACSD). pp. 78–87. IEEE Computer Society, Xi'an, China (2008)
22. Schneider, K., Brandt, J., Schuele, T.: Causality analysis of synchronous programs with delayed actions. In: Compilers, Architecture, and Synthesis for Embedded Systems (CASES). pp. 179–189. ACM, Washington, DC, USA (2004)
23. Schneider, K., Brandt, J., Schuele, T.: A verified compiler for synchronous programs with local declarations. Electronic Notes in Theoretical Computer Science (ENTCS) 153(4), 71–97 (2006)
24. Schneider, K., Brandt, J., Schuele, T., Tuerk, T.: Maximal causality analysis. In: Desel, J., Watanabe, Y. (eds.) Application of Concurrency to System Design (ACSD). pp. 106–115. IEEE Computer Society, St. Malo, France (2005)
25. Schuele, T., Schneider, K.: Abstraction of assembler programs for symbolic worst case execution time analysis. In: Design Automation Conference (DAC). pp. 107–112. ACM, San Diego, CA, USA (2004)
26. Shiple, T.: Formal Analysis of Synchronous Circuits. Ph.D. thesis, University of California at Berkeley, Berkeley, CA, USA (1996)

The New Averest: Version 2.0

Jens Brandt

Embedded Systems Group, TU Kaiserslautern, Germany

Averest is a set of tools for the specification, verification, and implementation of reactive systems. It includes a compiler and a simulator for synchronous programs, a symbolic model checker and a tool for hardware-software synthesis. The new version will be released at SYNCHRON '09. This talk briefly introduces its completely revised design and highlights new features which the synchronous community can benefit from.

Cartesian Programming: The Power of the Index

John Plaice

Univ. of New South Wales, Sydney, Australia

The current development of increasingly diverse physical computing architectures, with large numbers of distinct computational nodes, forces the creation of a unique formalism in which programmers can simply write equations and for which compiler designers can write implementations for the different architectures. This formalism must be sufficiently powerful and simple to ensure that the semantics of an entire system can be written directly, even if it encompasses reactivity, context-awareness, mobility, ubiquity or pervasiveness.

Cartesian programming provides a multidimensional context, in the form of an index, to programming, using an infinite dimensional space. In the same way that René Descartes's coordinate geometry allowed for the algebraisation of geometry, Cartesian programming makes it possible to have a single formalism in which to describe the entire development of a software system, with multiple heterogeneous components, in a fully declarative manner.

In this talk, we present the TransLucid programming language, through which the concepts of Cartesian programming have been developed, and demonstrate that with a very restricted set of primitives, it is possible to write massively parallel systems, incorporating real-time streaming and system reconfiguration, with full tracking of provenance.

On Compositionality and Modular Code Generation for Synchronous and Other Models of Computation

Stavros Tripakis

University of California, Berkeley, USA

What is the parallel composition of two stateless functions? Perhaps surprisingly, the answer to this question is not "a stateless function". What is the parallel composition of two Mealy machines? Again, the answer is not "a Mealy machine". Indeed, consider function f_1 with input x_1 and output y_1, and function f_2 with input x_2 and output y_2. Suppose we represent the parallel composition of f_1 and f_2 as a function f with inputs x_1, x_2 and outputs y_1, y_2, such that $f(x_1, x_2) = (f_1(x_1), f_2(x_2))$. Now suppose we wish to connect output y_2 to input x_1: we cannot do this in f, because the feedback connection creates an a-priori cyclic dependency. This is a false dependency, however, because output y_2 does not depend on input x_1, it only depends on input x_2. By treating the parallel composition of f_1 and f_2 as a "monolithic" function f, this dependency information has been lost.

We study the above questions, in the context of modular code generation for synchronous block diagrams (SBDs). SBDs are a hierarchical signal-flow diagram notation with synchronous semantics. They are the fundamental model behind widespread tools in the embedded software domain, such as SCADE and the discrete time subset of Simulink. Automatic code generation from such models is key to the success of so-called model-based design. We are interested in *modular* code generation for SBDs: modular means that code is generated for a given composite block independently from context (i.e., without knowing in which diagrams this block is to be used). Existing methods fail to address this problem in a satisfactory manner. They generate "monolithic" code, e.g., a single "step-function" per block. As illustrated above, this introduces false dependencies between block inputs and outputs, and compromises reusability, by not allowing the block to be used in some contexts. As a result, state-of-the-art tools either impose restrictions on the diagrams they can handle or resort to flattening.

We propose a framework that fixes this by generating, for a given block, a variable number of interface functions, as many as needed to achieve maximal reusability, but no more. In the worst case, $N + 1$ functions may be needed, where N is the number of outputs of the block. It is crucial to minimize the number of interface functions, for reasons of scalability, but also because of IP concerns. We are thus led to define a quantified notion of modularity, in terms of the size of the interface of a block. The smaller the interface, the more modular the code is. Our framework exposes fundamental trade-offs between reusability, modularity and code size. We show how to explore these trade-offs by choosing appropriate graph clustering algorithms. We present a prototype implementation

and experimental results carried on Simulink models, and describe extensions of our framework to triggered and timed diagrams. This work is joint with Roberto Lublinerman and Christian Szegedy and is described in detail in [2–4].

The above observations extend to models other than SBDs and triggered SBDs. In particular, we have studied them in the context of *synchronous (or static) data flow* (SDF) [1]. Hierarchical SDF models are not compositional: a composite SDF actor cannot be represented as an atomic SDF actor without loss of information that can lead to deadlocks during feedback composition. We have studied extensions of the modular code generation method for SBDs described above to SDF. This is joint work with Dai Bui, Bert Rodiers and Edward A. Lee, described in detail in [5].

References

1. E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.
2. R. Lublinerman, C. Szegedy, and S. Tripakis. Modular Code Generation from Synchronous Block Diagrams – Modularity vs. Code Size. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*, pages 78–89. ACM, January 2009.
3. R. Lublinerman and S. Tripakis. Modular Code Generation from Triggered and Timed Block Diagrams. In *14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'08)*, pages 147–158. IEEE CS Press, April 2008.
4. R. Lublinerman and S. Tripakis. Modularity vs. Reusability: Code Generation from Synchronous Block Diagrams. In *Design, Automation, and Test in Europe (DATE'08)*, pages 1504–1509. ACM, March 2008.
5. S. Tripakis, D. Bui, B. Rodiers, and E.A. Lee. Compositionality in Synchronous Data Flow: Modular Code Generation from Hierarchical SDF Graphs. Technical Report UCB/EECS-2009-143, EECS Department, University of California, Berkeley, Oct 2009. http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-143.html.

The Modal Model Muddle

Edward A. Lee

University of California — Berkeley, USA

Modal models in Ptolemy II are models that have multiple distinct behaviors and a state machine that selects among these behaviors. This talk will chronicle an exploration of semantics for modal models for timed systems.

Formal and Executable Contracts (Using 42) for Transaction-Level Modeling in SystemC

Florence Maraninchi[1], Tayeb Bouhadiba[1], and Giovanni Funchal[2]

[1] Verimag, Grenoble INP, France
[2] Verimag/STMicroelectronics

Transaction-Level Modeling (TLM) for systems-on-a-chip (SoCs) has become a standard in the industry, using SystemC. With SystemC/TLM, it is possible to develop an executable virtual prototype of a hardware platform, so that software developers can start writing code long before the actual chip is available. A hardware model in System- C/TLM can be very abstract, compared to the detailed RTL model. It is clearly component-based, with guidelines defining how components should be designed for use in any TLM context. However, these guidelines are quite informal for the moment. In this paper, we establish a structural correspondence between functional SystemC/TLM models and a formal component-model for embedded systems called 42, for which we have defined a notion of control contract, and an execution mode for systems made of components' contracts. This is a way of formalizing the principles of functional SystemC/TLM. Moreover, it allows the combined use of SystemC/TLM components with 42 components. Demonstrating that such a combined use is possible is key to the adoption of formal components' definitions in the community of TLM users.

Deterministic, Time-Predictable, and Light-Weight Multithreading Using PRET-C

Alain Girault

INRIA Rhône-Alpes, Grenoble, France

Support for light-weight concurrency in C is gaining recent momentum. Another area of research focus has been the development of code that guarantees precise worst case timing. We present a new language called Precision Timed C, for predictable and lightweight multithreading in C. PRET-C supports synchronous concurrency, preemption, and a high-level construct for logical time. In contrast to existing synchronous languages, PRET-C offers C-based shared memory communications between concurrent threads that is guaranteed to be thread safe via the proposed semantics. Mapping of logical time to physical time is achieved by a Worst Case Reaction Time (WCRT) analyzer. To improve throughput while maintaining predictability, a hardware accelerator specifically designed for PRET-C is augmented to a softcore processor. We then demonstrate through extensive benchmarking that the proposed approach not only achieves complete predictable execution but also improves overall throughput when compared to the software execution of PRET-C. PRET-C software approach is also significantly more efficient in comparison to two other light-weight concurrent C variants called SC and Protothreads, as well as the well-known synchronous language Esterel.

SyncCharts in C – A Proposal for Light-Weight, Deterministic Concurrency

Reinhard von Hanxleden

Real-Time and Embedded Systems Group, Christian-Albrechts-Universität zu Kiel, Germany

Synchronous C (SC) and Synchronous Java extend C and Java with control flow operators for deterministic, light-weight concurrency and preemption. SC/SJ is based on SyncCharts, a synchronous variant of Statecharts with a sound formal basis. SC/SJ implements concurrency via a simulation of multi-threading, inspired by reactive processing. This approach permits very fast context switches and allows to express SC operators with regular, sequential C code. Thus a concurrent SC program requires neither a special compiler nor OS support for concurrency.

A reference implementation of SC, based on C macros, is available as open source code (http://www.informatik.uni-kiel.de/rtsys/sc/); SJ will be available by the end of the year. SC can be used in a number of scenarios: 1) as a regular programming language, requiring just a C compiler; 2) as an intermediate target language for synthesizing graphical SyncChart models into executable code, in a traceable manner; 3) as instruction set architecture for programming precision timed (PRET) or reactive architectures, abstracting functionality from physical timing; or 4) as a virtual machine instruction set, with a very dense encoding.

Modelica for Hybrid Systems Modeling: Problems and Difficulties, Mathematical Semantics and Execution Schemes (work in progress)

Albert Benveniste, Benoit Caillaud and Marc Pouzet

INRIA Rennes and INRIA Saclay, France

Modelica is a very interesting hybrid systems modeler. It fits the modeling discipline in use for physical systems where the system model results from non-directed interactions between physical entities. As an example, think of the Ohm and Kirchoff laws in electrical circuits: these lead to balance equations with no pre-specified inputs or outputs. Mathematically speaking, this leads to considering DAEs (Differential Algebraic Equations) of the form $f(\dot{x}, x) = 0$ instead of ODEs (Ordinary Differential Equations) of the form $\dot{x} = g(x)$. Unlike ODEs, DAEs compose with no restriction (loops with no integrators are allowed).

Modelica allows specifying such systems (unlike Simulink) and supports discrete mode changes based on zero-crossings. All together, Modelica is often viewed as a hybrid systems modeler suited to component based design. In this talk we will address two central issues raised by Modelica.

First, since Modelica is continuous time based, it faces the problem of deciding whether two triggering events should be considered simultaneous or not (a run-time comparison is not appropriate). Different Modelica compilers have adopted different disciplines for this. We will propose a type system reminiscent of clock calculi in synchronous languages to fix this.

Second, the style of Modelica is somehow schizophrenic: it supports DAE and is therefore relational in nature regarding its continuous time facet; it is, however, imperative in style regarding its discrete mode change facet. The two don't fit together at all and this is a source of problems when developing a mathematical semantics for Modelica on which compilation techniques could rely. One deep difficulty is that, on the one hand, an operational semantics should be step-based, whereas, on the other hand, discretization steps must be adjusted adaptively at run-time for good performance and accuracy. In our work, we try to reconcile the two requirements with the help of *non-standard analysis*, a heterodox mathematical framework in which infinitesimals can be effectively and rigorously handled. This approach allows us to derive mathematically sound executions schemes for Modelica.

Liberating Programming

David Harel

Weizmann Institute — Rehovot, Israel

This talk describes a dream about freeing ourselves from the straight-jackets of programming, making the process of getting computers to do what we want intuitive, natural, and also fun. It recommends harnessing the great power of computing and transforming a natural and almost playful means of programming so that it becomes fully operational and machine-doable.

Technically, the three issues are: (1) having to produce a tangible artifact in some language; (2) having actually to produce two separate artifacts (the program and the requirements) and having then to pit one against the other; (3) having to program each piece/part/object of the system separately. The talk then got a little more technical, providing some modest evidence of feasibility of the dream, via live sequence charts (LSCs) and the play-in/play-out approach to scenario-based programming.

An Alternative Compilation Scheme for Polychrony

Sandeep K. Shukla

Virginia Polytechnic Institute — Blacksburg

In this talk we will discuss how to the solve the clock calculus for polychronous programs in SIGNAL using prime implicates of Boolean clauses. This provides an alternative top down compilation scheme for SIGNAL for code synthesis. It also provides a different view of endochrony.

Modular Static Scheduling of Synchronous Data-flow Networks

Marc Pouzet[1] and Pascal Raymond[2]

[1] Université Paris-Sud and INRIA
[2] Verimag-CNRS — Grenoble

This work addresses the question of producing modular sequential imperative code from synchronous data-flow networks. Precisely, given a system with several input and output flows, how to decompose it into a minimal number of classes executed atomically and statically scheduled without restricting possible feedback loops between input and output? Though this question has been identified by Raymond in the early years of Lustre, it has almost been left aside until the recent work of Lublinerman, Szegedy and Tripakis. The problem is proven to be intractable, in the sense that it belongs to the family of optimization problems where the corresponding decision problem — there exists a solution with size c — is NP-complete. Then, the authors derive an iterative algorithm looking for solutions for c = 1,2,... where each step is encoded as a SAT problem.

Despite the apparent intractability of the problem, our experience is that real programs do not exhibit such a complexity. Based on earlier work by Raymond, this paper presents a new symbolic encoding of the problem in terms of input/output relations. This encoding simplifies the problem, in the sense that it rejects solutions, while keeping all the optimal ones. It allows, in polynomial time, (1) to identify nodes for which several schedules are feasible and thus are possible sources of combinatorial explosion; (2) to obtain solutions which in some cases are already optimal; (3) otherwise, to get a non trivial lower bound for c to start an iterative combinatorial search.

The solution applies to a large class of block-diagram formalisms based on atomic computations and a delay operator, ranging from synchronous languages such as Lustre or SCADE to modeling tools such as Simulink.

Dynamic Structure Multisampling Synchronous Programming

Fernando Barros

University of Coimbra, Portugal

Traditional discrete time machines operate at the same rate simplifying the semantics of their interconnection. More complex systems require machines to specify their sampling rate independently, make their coordination a challenging problem. We present the Continuous Flow System Specification (CFSS), a formalism aimed to the representation of dynamic structure, generalized sampling systems. In this formalism, sampling can change over time and from component to component, making CFSS a framework for representing multisampling synchronous systems. The ability to join machines with different sampling periods is enabled by a novel representation of continuous systems based on digital computers.

Clock-Driven Distributed Real-Time Implementation of Endochronous Systems

Dumitru Potop-Butucaru

INRIA — Le Chesnay

Round Abstraction, Compositionally

Mohamed Nabih Menaa

University of Birmingham

We revisit a form of temporal scaling called "round abstraction" as a solution to the problem of building locally-synchronous representations of asynchronous behavior. We will show how round abstraction can be defined on sets of traces in a compositional model akin to the category of games in game semantics (albeit with less restrictions). In particular, we establish a soundness result under certain conditions.

Three Ways of Compiling Programs into Circuits

Satnam Singh

Microsoft Research UK — Cambridge

We want to compile programs into circuits to take advantage of performance or power improvements that might be available from a hardware implementation. However, trying to pervert a piece of software written in an imperative language into a digital synchronous circuits has many challenges and this talk discusses how some of the may be overcome. In this presentation I shall describe how we can in certain cases synthesize programs that are written with explicit dynamic memory allocation into corresponding programs that only use statically allocated memory (a stepping stone towards a hardware implementation). A second case study illustrates how we can use a modern imperative object orientated language to model synchronous systems using a standard multi-threading library (the Kiwi project). The compiled .NET byte can then be compiled into circuits. A third case study shows how we can embed a data-parallel domain specific language into C# or C++ and target GPUs, SIMD multi-core code and FPGA circuits. By learning how to compile programs into circuits we gain insight into the general problem of how to write software for heterogeneous multi-core systems.

Dynamic Aspects in Synchronous Languages

Louis Mandel, Florence Plateau, and Marc Pouzet

Université Paris-Sud and INRIA

In this talk, we will discuss about old works that have shown that it is possible to have dynamic aspects in synchronous languages with strict causality analysis and static scheduling. Then we will go a little further with the presentation of an interaction loop to program in Lucid Synchrone.

Towards Formal Evaluation of QoS Properties in Data Acquisition Systems Using Synchronous Models

Lionel Morel[1], Jean-Philippe Babau[2], and Belgacem Ben-Hedia[1]

[1] Université de Lyon, INRIA
INSA-Lyon, CITI, F-69621, France
[2] Université de BrestUBO, LISyC, UEB F-29238 Brest Cedex 3, France

In the field of process control, data acquisition software (such as device drivers) require special care in their design, because they usually stand as bottlenecks between hardware devices and control applications. In particular timing constraints on occurrences of data are often given based on intuition and empirical experience.

The work presented here intends to provide a formal model to characterize timing properties such as input data delay. This model is based on an encoding in Lustre of some basic components organized in certain patterns to form data acquisition software subsystems. The scheduling of these components, intrinsically asynchronously is encoded as a synchronous controller. With this encoding, we can benefit from existing formal proof tools available for Lustre (especially abstract interpretation) to verify timing properties.

How Far Can We Go With Synchronous Programming?

Paul Caspi

VERIMAG — Gières

In this talk we argue that synchronous programming has been growing based on a start-small approach: at the beginning it aimed modestly at designing single-threaded control programs. But the scope has been extending in the course of time in several directions: on the modelling side, by mixing both dataflow and state machines, and by allowing the modelling of distributed control systems; on the implementation side, by allowing both time and event triggered, single and multi-threaded, centralized and distributed implementations. These impressive achievements lead us to the question raised in the title of the talk.

Synchronous and Asynchronous Interaction in Distributed Systems

Rob van Glabbeek[1], Ursula Goltz[2] and Jens-Wolfhard Schicke[2]

[1] NICTA, Sydney, Australia
[2] TU Braunschweig

When considering distributed systems, it is a central issue how to deal with interactions between components. We investigate the paradigms of synchronous and asynchronous interaction in the context of distributed systems. We choose Petri nets as our system model and formalize a general concept of distributed systems as sequential components interacting asynchronously by defining a corresponding class of Petri nets, called LSGA nets. We investigate to what extent or under which conditions synchronous interaction is a valid concept for specification and implementation of such systems by proving that certain system specifications can not be implemented as LSGA nets up to step readiness equivalence.

Compensations and Runtime Monitoring

Gordon Pace and Christian Colombo

Department of Computer Science, University of Malta

Reasoning about long-lived transactions with possibility of failure at various stages, and with complex error-recovery and implicit application of compensations, poses a number of challenges. One major challenge, which has been extensively explored and addressed in different ways, is that of compositional and concise specification of such systems, complete in that they include the description of exceptional behaviors. This enables the runtime monitoring of systems against such complete specifications. We propose the extension of such an approach to use complete specifications with compensations to aid the runtime verification system to enable automated rectification of failures and a combination of system- and monitor-controlled compensations. Furthermore, the approach we propose, enables us to build a framework for monitors weakly synchronized to the running system. The use of such a loosely connected verification system enables us to recover from errors discovered even if the underlying system has, in the meantime, proceeded further.

Mode automata and timed imperative programs in SSA form

Jean-Pierre Talpin

INRIA — Rennes

Analysis of Scheduled Latency Insensitive Systems with Periodic Clock Calculus

Sandeep Shukla

Reconfigurations and Adaptations: In Need of Control (with 'pataphysical in/out-troduction)

Eric Rutten and Gwenael Deleval

We outline ongoing work on the model-based control of adaptive and reconfigurable systems, especially their logical aspects. It is considered as a closed-loop control problem, modeled using the synchronous approach to reactive systems, with the application of discrete controller synthesis techniques, encapsulated in a mixed imperative-declarative programming language. We illustrate the use of our language by a simple case study. The presentation also has a 'pataphysical introduction and outroduction.

References

1. Soufyane Aboubekr, Gwenael Delaval, and Eric Rutten. A Programming Language for Adaptation Control: Case Study. In *Proc. of the 2nd Workshop on Adaptive and Reconfigurable Embedded Systems, APRES 2009 (in conjunction with ESWeek 2009)*, Grenoble, France, October 11, 2009. ACM SIGBED Review Volume 6, Number 3, October 2009 (see `http://sigbed.seas.upenn.edu/vol6_num3.html`).
2. Gwenael Delaval, Hervé Marchand, and Eric Rutten. BZR Contracts for Modular Discrete Controller Synthesis. Research Report/Rapport de Recherche INRIA, no. 7111, November 2009. (to appear in Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems, LCTES 2010, in conjunction with CPSWeek 2010, Stockholm, Sweden, April 12-16, 2010) (see `http://hal.inria.fr/inria-00436560`).

Programming in the n-Synchronous Model with Lucy-n

Florence Plateau, Louis Mandel, and Marc Pouzet

Université Paris-Sud and INRIA

The n-synchronous model aims at relaxing the synchronous programming model to allow the communication through bounded buffers. It is done by introducing a subtyping rule in the clock calculus. Last year, we presented at Synchron how to abstract clocks in order to check this subtyping relation. This year, we will present a n-synchronous programming language : Lucy-n. We will show through examples how to program in Lucy-n.

Regular Switching Schemes for Interconnect in Process Network Models: K-Periodically Routed Event Graphs

Robert de Simone

INRIA Sophia Antipolis

In the times of flourishing multicore embedded devices, concurrent applications and parallel architectures are becoming standard. Process networks provide formal models that capture in many cases the essence of such concurrency, and allow to reason mathematically on some compilation issues, while at very abstract level. As compilation consists mainly of mapping of applications onto execution platforms, it comprises in this case both spatial allocation and temporal scheduling of functions onto distributed resources. Because global efficiency depends largely on communication rates, interconnect components must also be carefully modeled for optimal traffic.

In this presentation, we first recall known and recent results from our group on static, ultimately k-periodic schedules for Marked Graphs and several control-free Process Network extensions. Then, we introduce a new simple model, named K-periodic Routing extended Event Graphs (KREG). It can be seen as an extension of Marked Graphs with only two switching schemes, named Merge and Select, for multiplexing and demultiplexing respectively. The important fact here, which is that the switching condition patterns have to be also k-periodic, is a way borrowed from previous works on schedules. The model can be seen as a specialization of boolean and cyclo-static data flow graphs altogether (with nevertheless the expressive power of the full cyclo-static case). The main importance of KREGs comes from the fact that they allow algebraic identities which compose a full axiomatic theory for finite network expressions regarding (asynchronous) behavior equivalence. At the same time, these algebraic identities can be seen as transformations on the interconnect topology, sharing links through appropriate interleaving, or splitting them as point-to-point links. Loop-free networks then admit canonical normal forms, with fully expansed point-to-point like topology. But in many senses, this is a worst-case solution, and link-sharing is highly desirable, under the condition that it does not constraint further the data traffic.

In the talk, we introduce the relevant formal models, their algebraic properties, and motivate the underlying methodology through examples.

KPASSA PN

Julien Boucaron

INRIA Sophia Antipolis

This presentation provides a short introduction to our KPASSA tool. Then, we survey quickly Latency-Insensitive Design (aka Synchronous Elastic) and show transformations implemented in the tool.

From Supercomputing to Volkscomputing a Data-Flow Synchronous Perspective to Performance Portability

Albert Cohen

INRIA — Orsay

Many researchers claim that the manycore era is a revolution, with various reasons. Many researchers claim that reinventing the wheel is not a revolution. Both extremes are obviously wrong. Behind the myths, what is it that the software industry needs and that parallel computing research has failed and continues to fail to deliver?

Portability of performance has been the underlying assumption for general-purpose software projects. After initial successes with the first FORTRAN compilers, modern compilers, run-time systems and architecture designs have progressively failed to hide the non-uniformity and the parallelism of the hardware. Every day, more programmers are forced to resort to platform-specific optimizations, committing early on specific parallel implementations. This is a dramatic regression.

Giving manual control to the programmer provides some short-term relief and has been somewhat successful for supercomputing applications or signal-processing embedded devices. But attempting to extend this approach to the global software industry demonstrates a misconception of the inertia and cost of software development. This talk argues that portability of performance has to be our ambition, and not only a long-term one.

We will discuss some research directions and results, combining implicit parallelism, data-flow with relaxed forms of synchrony, deterministic concurrency, adaptive/auto-tuning frameworks operating over novel concurrent intermediate languages.

Resource-Bounded Runtime Verification of Java Programs with Real-Time Properties

Christian Colombo[1], Gordon J. Pace[1], and Gerardo Schneider[2]

[1] Department of Computer Science, University of Malta
[2] Department of Informatics, University of Oslo

Given the intractability of exhaustively verifying software, the use of runtime verification, to verify single execution paths at runtime, is becoming increasingly popular. Undoubtedly, the overhead introduced by runtime verification is a concern for system developers planning to introduce this technique in their work. By using Lustre to write security-critical properties, we exploit the language's guarantees on bounded resources. By translating these properties into the existing monitoring framework Larva, we manage to monitor Java programs with guaranteed use of bounded-resources. Another recurrent issue is the identification of appropriate notations to represent properties. We use a subset of QDDC as an alternative specification notation for real-time properties because it is translatable into Lustre. Thus, QDDC also enjoys the same guarantees given when using Lustre.

Clock Type Soundness in Synchronous Languages

Martin Strecker

Université Paul Sabatier — Toulouse

This talk takes a closer look at clock types, which are meant to ensure that synchronous programs do not manipulate invalid data. We first establish an abstract type soundness result for synchronous languages: given a program, we can derive a system of set equations whose solution guarantees the absence of invalid data during execution. We then instantiate this result for synchronous languages with periodic clocks and show how to effectively solve the resulting set constraints. We will briefly comment on our formalization in the Isabelle proof assistant. More details can be found in the following paper:

`http://www.irit.fr/~Martin.Strecker/Publications/clock_type_soundness.html`

Contract and Interface Theories for Embedded System Design: Users' Requirements, Failure or Success To Meet Them, and a New Proposal

Benoit Caillaud

INRIA — Rennes

In this talk we will present the modal interface theory, a unification of interface automata, modal specifications, and some contract theories, in the context of requirements engineering for embedded system design. In this talk we will unveil the power of modal interfaces, as a means to capture system or component level requirements and support compositional reasoning, with low computational complexity algorithms.

The Timing Definition Language and High-Priority Interrupts

Peter Hintenaus

Universität Salzburg

For the design of deeply embedded systems we propose extensions to the Timing Definition Language (TDL) that allow integration of asynchronous activities running at highest priority levels.

Concurrency and Communication: Lessons from the SHIM Project

Stephen A. Edwards

Columbia University

Describing parallel hardware and software is difficult, especially in an embedded setting. Five years ago, we started the SHIM project to address this challenge by developing a programming language for hardware/software systems. The resulting language describes asynchronously running processes that has the useful property of scheduling-independence: the I/O of a SHIM program is not affected by any scheduling choices. I will present a history of the SHIM project with a focus on the key things we have learned along the way.

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-

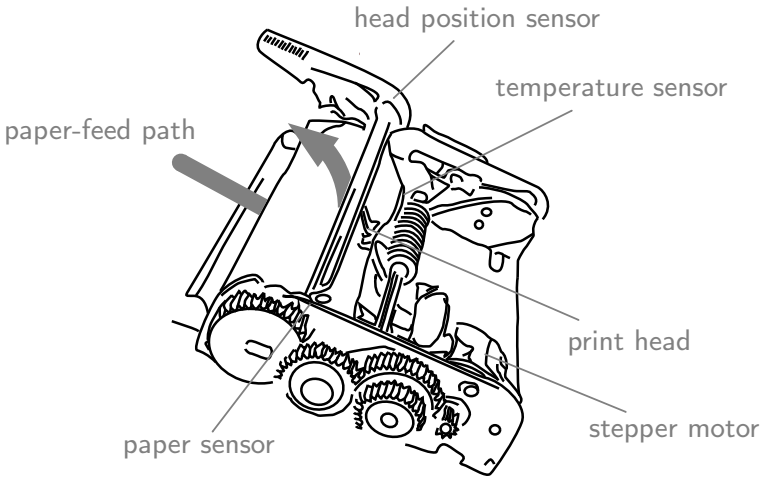


Fig. 1: Physical structure of the microprinter example

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 step is normally energized for 1.667ms, but if the motor is held for more than

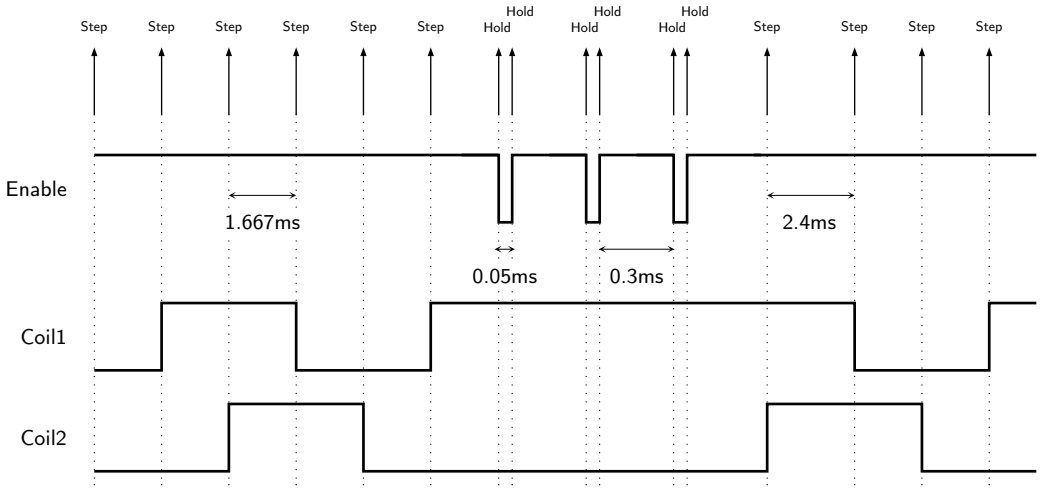


Fig. 2: Typical microprinter motor control signals

```
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

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** ... **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

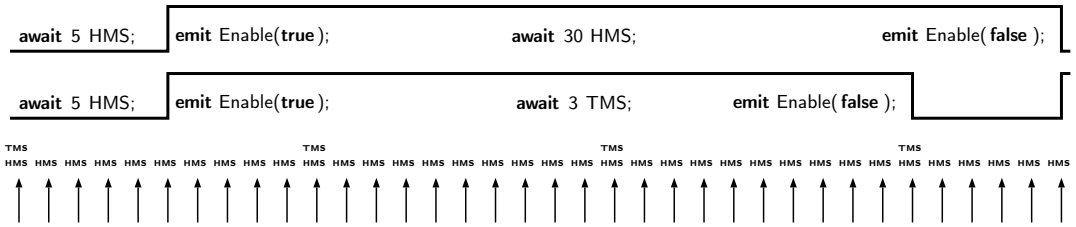


Fig. 4: Granularity of timing inputs

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:

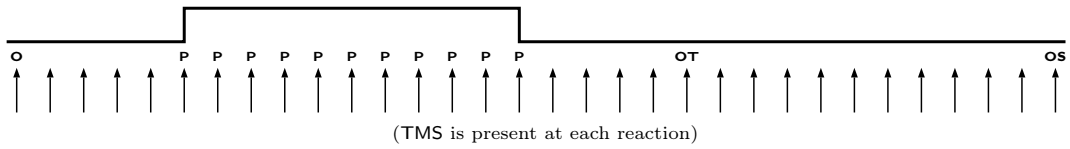


Fig. 5: Effect of suspend on delays

```
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.

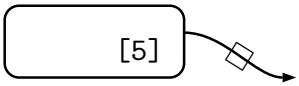
There are three main advantages over the timers described in the previous subsection. The delay is expressed as a single statement, which makes programs easier to read and simpler to analyze. The semantics of **exec** precisely defines its interaction with **abort**. The semantics also accounts for issues 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 $= x * 10\text{E}{-3}$
x m $= x * 60$	x us $= x * 10\text{E}{-6}$
x s $= x * 1$	x ns $= x * 10\text{E}{-9}$

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 $(s, \tau_{s_1}), (s, \tau_{s_2}) \in A \to \tau_{s_1} = \tau_{s_2}$.

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 $d \cdot \tau \geq 1$,

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

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:*

$$\begin{aligned}
D(\textsf{nothing}) &= 0 \\
D(\textsf{emit}\ s) &= 0 \\
D(\textsf{pause}) &= 0 \\
D(\textsf{delay}\ d) &= d \\
D(\textsf{present}\ s\ \textsf{then}\ p\ \textsf{else}\ q\ \textsf{end}) &= D'(p, q, ;) \\
D(\textsf{suspend}\ p\ \textsf{when}\ s\ \textsf{end}) &= D(p) \\
D(p\ ;\ q) &= D'(p, q, ;) \\
D(\textsf{loop}\ p\ \textsf{end}) &= D(p) \\
D(p\ \|\ q) &= D'(p, q, \|) \\
D(\textsf{trap}\ t\ \textsf{in}\ p\ \textsf{end}) &= D(p) \\
D(\textsf{exit}\ t) &= 0 \\
D(\textsf{signal}\ s\ \textsf{in}\ p\ \textsf{end}) &= D(p)
\end{aligned}$$

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:*

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

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.

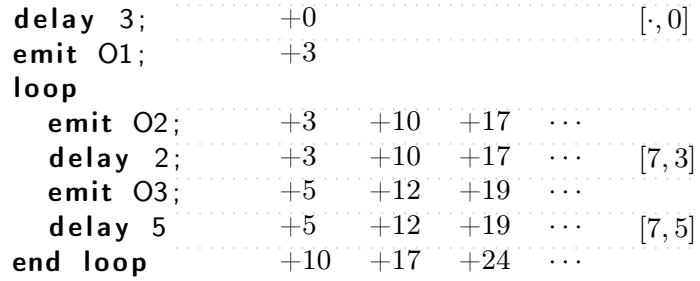


Fig. 7: Phase relationships in an Esterel+delay program

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-

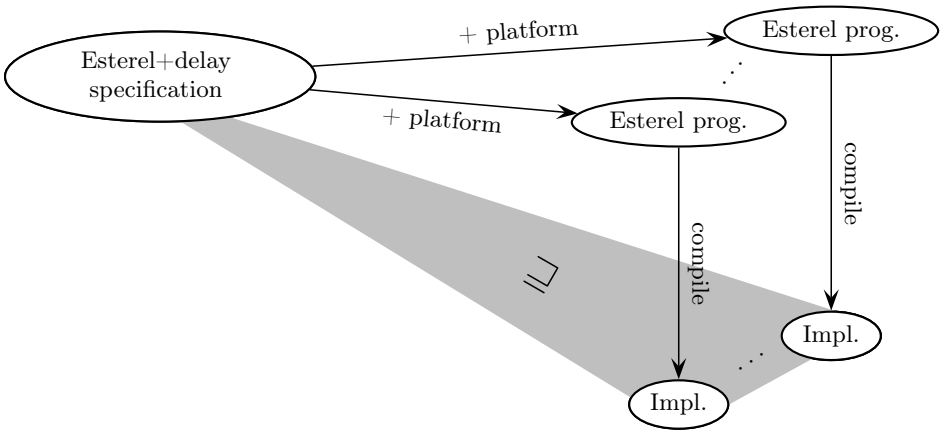


Fig. 8: Esterel+delay: artifacts and relations

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

Lusterel Reactive Streams: How to Schedule Asynchronous Data Flow into Synchronous Control Flow

Michael Mendler and Joaquín Aguado
Universität Bamberg

Many software systems for user interfaces, web services, business processes, games and hardware modelling, among others, are naturally composed of interactive processes. Moreover, it may be important to decompose applications in order to leverage the increased processing performance available on modern multi-core execution architectures. In these areas, an application typically consists of a set of connected sequential processes, where executions can be abstracted in two ways: (i) The *data-flow* view interprets inter-process communications as unbounded data streams changing over time and concentrates on functional relationships between streams. (ii) The *control-flow* abstraction perceives sets of connections at a particular time as interconnect states and analyses changes of such states. In addition, processing is often organised into nested or interleaved computation cycles (phases) that can be associated with clocks running under the Synchrony Hypothesis. The synchronous paradigm for programming reactive systems gave rise to languages such as Lustre and Signal for data flow or Esterel which is dedicated to control flow. These languages, which have been very successful for embedded systems, so far have been developed mainly separately and around domain-specific tools (e.g., SCADE, Esterel Studio, SpecTRM, Simulink, Stateflow, Rhapsody) for use in control systems, automotive or avionics applications.

Yet, there is no reason why this state-of-the-art technology should be restricted to embedded system domains. The Lusterel Project tries to demonstrate that using a functional language like Haskell, it is possible to take advantage of the synchronous metaphors for a wider range of market-relevant applications. Lusterel is a Haskell library under development which provides a shallow embedding of Lustre and Esterel-style programming combinators. It coherently unifies both views for interactive applications without blurring the distinction between data and control. This contrasts with the traditional view which takes data flow as primary and flattens control in terms of explicit absence values introduced into the data streams.

In this talk we show how standard Kahn-style data flow can be coded in Haskell so that it can be scheduled, and thus smoothly combined, with synchronous control flow. Our co-inductive coding supports a variety of scheduling schemes, such as Esterel's 0-synchronous execution, bounded-buffer synchronisation and multi-rate execution. Currently, the library only depends on the standard lazy lambda-calculus which provides sufficient operational structure to express single-threaded control flow. In the future we hope to extend this work to concurrent Haskell for multi-threaded execution.