A Timing Model for Synchronous Language Implementations in Simulink*

Timothy Bourke
School of CSE/National ICT Australia
University of NSW
Sydney 2052, Australia
tbourke@cse.unsw.edu.au

Arcot Sowmya
School of CSE, UNSW/National ICT Australia
Division of Engineering, Science and Technology
UNSW Asia, Singapore
sowmya@cse.unsw.edu.au

ABSTRACT

We describe a simple scheme for mapping synchronous language models, in the form of Boolean Mealy Machines, into timed automata. The mapping captures certain idealized implementation details that are ignored, or assumed away, by the synchronous paradigm. In this regard, the scheme may be compared with other approaches such as the AASAP semantics. However, our model addresses input latching and reaction triggering differently. Additionally, the focus is not on model-checking but rather on creating a semantic model for simulating synchronous controllers within Simulink.

The model considers both sample-driven and event-driven execution paradigms, and clarifies their similarities and differences. It provides a means of analyzing the timing behavior of small-scale embedded controllers. The integration of the timed automata models into Simulink is described and related work is discussed.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems

General Terms

Design

Keywords

Synchronous Languages, Simulink, Timed Automata

1. INTRODUCTION AND MOTIVATION

Embedded controllers interact with their environment in real time. Complicated sequential behaviors able to operate under a variety of conditions must be specified, simulated, verified and implemented. The MathWorks Matlab/Simulink software is popular for generating simulation runs of controllers within specific environments.

Simulink [12] is a platform for numerically simulating discrete, continuous and hybrid models over time. Models are constructed by connecting parameterized blocks with lines representing signals. The Stateflow block allows reactive controllers to be specified in a notation akin to Statecharts. The Simulink/Stateflow combination is increasingly used as a platform for *model-based design*, where code for embedded systems is generated automatically and directly from simulation models. Stateflow is practical and powerful, but the underlying model and semantics are less tractable than those of the synchronous languages.

The synchronous languages [2] combine domain specific programming notations with a rigorous and tractable execution model. The model is based on the simplification that computations on reactive controllers are so fast, relative to the arrival rate of inputs, that they may be considered instantaneous. In reality, computation time is finite and limited, and input and output signals must be latched.

We have developed [7] a prototype block for embedding synchronous language programs within Simulink. The block was *perfectly synchronous* in the sense that execution times were not modeled and outputs occurred in the same simulation step as, i.e. simultaneously with, triggering inputs. While one normally reasons about synchronous language programs under such assumptions, Simulink is often used to simulate the effect of timing details, thus it is interesting to examine how more detailed results may be obtained from synchronous language programs. In this paper we develop a formal model that accounts for some of the delays that occur in synchronous language implementations. The model specifies precisely how a Simulink block, or collection of blocks, should respond during a simulation. It also provides a means of analyzing details not directly addressed by the synchronous paradigm. Unlike the `AASAP` semantics [10], the model is not intended as a basis for model-checking.

2. OVERVIEW

This paper focuses on the imperative synchronous languages Argos and Esterel. Two execution schemes are generally accepted for these languages. The scheme chosen affects implementation timing behavior. Section 3 provides background information on the languages and execution schemes.

Section 4 describes and motivates the two parameters upon which our timing model is based. The model itself

*This research was fully funded by National ICT Australia. National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

EMSOFT'06, October 22–25, 2006, Seoul, Korea.

is specified as a mapping from a Boolean Mealy Machine—representing a synchronous program—and the two timing parameters, into a timed automaton. Section 5 begins by showing how a simple example is transformed and then defines the mapping formally.

The formal mapping greatly simplifies the development of a Simulink block (or model) with the given timing properties. This motivating problem is addressed in Section 6 where the intricacies and design choices involved are made clear by the timed automaton definition.

Finally, Section 7 discusses the similarities and differences between the approach described and others in the literature, most notably the `AASAP` semantics.

3. SYNCHRONOUS LANGUAGES

Our focus is the class of imperative synchronous languages exemplified by Argos and Esterel.

Argos [11] is a simplification of Statecharts with a truly synchronous, as opposed to microstep, execution model. Although valued extensions have been sketched, Argos is essentially limited to signals that are either true (present) or false (absent) at any instant. Argos provides four operators for composing programs from Mealy machines and each has a graphical representation: *synchronous product* for parallel composition, *refinement* for hierarchical structure, *encapsulation* for enforcing synchronization and hiding signals, and *inhibition* for selectively prohibiting the participation of subcomponents in reactions. The operators close over Mealy machines, but do not in general preserve determinism nor input-enabledness (often called *reactivity*) [11]. Watchdog timers are sometimes used in Argos programs and are written by adding an integer and signal name in square brackets to a state, e.g. `[5 SEC]`, and distinguishing a time-out transition. This notation is shorthand for an Argos subroutine that counts signal occurrences and triggers timeout transitions when required. Any Argos program can be transformed into a regular Mealy machine and vice versa.

Our prototype Simulink block [7] compiles and executes Argos programs. The graphical notation of Argos gives a useful comparison with Stateflow, the semantics makes compilation relatively straight-forward, and the execution model is easily understood from the state-based syntax. Esterel is more complicated to compile and the execution model requires more effort to follow. It does, however, have significant advantages, particularly because programs are structured in a domain-specific notation designed around key concepts of the specification of embedded controllers.

Esterel provides statements for emitting, detecting, and localizing signals. It includes operators for sequencing, looping, suspending, and parallelizing program components. Exceptions are used extensively to manage control flow. Statements either complete instantaneously, i.e. within the same reaction, or block until a later reaction. Any pure and valid Esterel program can be mapped to a Mealy Machine, using the behavioral semantics of [3]: each state corresponding to a program term, with behavioral transitions translated directly. Any Mealy Machine may be encoded as an Esterel program. We do not consider valued Esterel in this paper.

While Argos and pure Esterel are neither as powerful, nor as feature-rich as Stateflow their underlying model of computation is easier to understand and reason about—a distinct advantage for embedded applications where accuracy and correctness are paramount.

Implementations of synchronous programs loop continually: 1) reading inputs; 2) computing and emitting outputs; and 3) computing and storing the next state. There are two accepted execution schemes [2]:

- *sample-driven*: the program loop runs periodically.
- *event-driven*: input detection triggers the loop body.

The sample-driven mode is natural for clocked sequential circuits and also for control domain applications [8]. The event-driven scheme is more general (periodic clock ticks are but one triggering input) and perhaps harder to realize. An implementation could continuously poll for input signals (a degenerate form of sample-driven behavior) or the program body could be attached to an interrupt, function call, or a mechanism specific to a given operating system. The translation of Section 5 distinguishes between the two modes by employing the notion of urgency for event-driven reactions. Synchronous programs operate at a higher-level of abstraction than is typical for embedded control software. The execution schemes abstract over the details of triggering and an *interface* layer [5] treats input and output signals uniformly, ignoring issues of detection and latching. This simplifies the programming task, and, in many cases, makes programs more reusable. However, to simulate programs in continuous time it will be necessary to be more explicit about such details.

4. EXECUTION PARAMETERS

It will be useful to regard synchrony along two orthogonal dimensions: one *internal*, relating to the semantics of languages and models, the other *external* and concerned with the interaction of implementations and the environment.

The model of *perfect synchrony* shared by languages such as Esterel, Argos and Lustre, is 'perfect' in both dimensions. Internally, concurrent components communicate and change state simultaneously in a single step. This fact has challenged researchers to provide satisfying semantics despite seeming paradoxes [3]. It yields deterministic behavior with strong mathematical and physical foundations. Externally, the assumption allows a mapping between system function, in terms of input/output traces, and discrete instants of time.

For the synchronous languages the two views are inseparable and interrelated: the single step nature of the internals justifies an instantaneous, or at least very fast, transformation of inputs to outputs. Instantaneous reactions imply that internal events cannot be further ordered in time. Each reaction is assigned a single, consistent set of signal valuations. Data state changes depend only on the control state and inputs. This coupling, however, is no *fait accompli*.

Statecharts and Stateflow typify the *micro-step* [2] alternative for reasoning about internal behavior while maintaining the appearance of synchronous operation externally. At each chart awakening, inputs trigger sequences of internal transitions and logical signal emissions which culminate in outputs and a new chart state. Since events are processed one-by-one and may reoccur in a sense, notions of consistency are less applicable. The transformation of associated data state may depend on fine details of the evaluation order. This scheme avoids the difficulties of synchronous language compilation and, arguably, provides a clear sense of

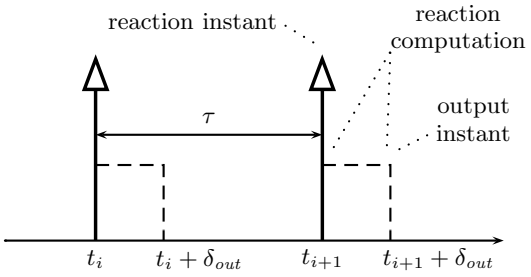


Figure 1: External timing parameters

cause and effect [17]. Unfortunately, it can also make reasoning about system behavior unnecessarily complicated, and lead to brittle, target-dependent behaviors. Surely, designing embedded controllers is already challenging enough!

Simulink is used by engineers to analyse the dynamic behavior of models containing both continuous and discrete elements. Numerical techniques are employed to permit a liberal mix of components and parameters within models, making it feasible to experiment with low-level controller detail and the idiosyncrasies of different environments. In this spirit, a block for simulating synchronous language programs might remain true to the synchronous semantics internally, but admit more choice when modeling externally observable behavior.

We have chosen to parameterize our Simulink block with two timing values:

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

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

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

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

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

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

The timing parameters are deliberate simplifications. δ_{out} encompasses the WCET (Worst Case Execution Time) for computing a reaction. τ captures other inherent limitations of the target platform. It may be non-trivial to acquire such bounds despite the general assertion that they are relatively easy to calculate for a synchronous program. Further, it is assumed that all reaction computations are of equal duration. More detailed alternatives are discussed in Section 7. The δ_{out} parameter implies that output signals always change value simultaneously at a fixed time after each reaction begins. This simplifying assumption could be met by an implementation if necessary.

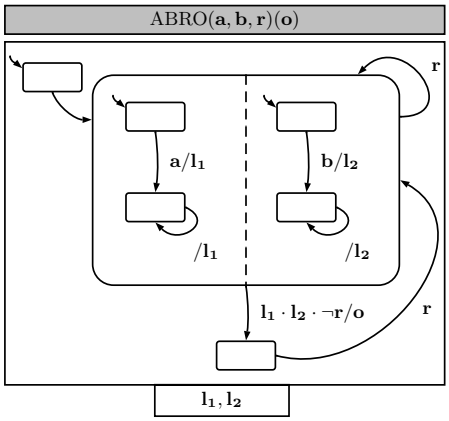
5. MAPPING TO TIMED AUTOMATA

5.1 Intuition

The semantics of the execution parameters, with respect to a given synchronous language program, are formalized by mapping from a Boolean Mealy Machine and the parameters themselves to a timed automaton. This makes the model unambiguous, leads to an accurate Simulink block implementation naturally (Section 6), and provides a precise basis for thinking about how a controller implementation operates over time. This section describes the process using a small motivating example before a formal account is given in Subsection 5.2.

Figures 2(a) and 2(b) show the ABRO program [4], in Argos and Esterel respectively. It has three input signals, a, b and r, and one output signal o. When both a and b have been received, in any order or even simultaneously, an o is emitted. The r signal resets the program prohibiting o from being emitted and forgetting any as or bs that may have already been received. The Argos version uses two local signals, l_1 and l_2, to emulate the behavior of the Esterel parallel construct ($\|$). Both programs map to the same Boolean Mealy Machine, shown in Figure 2(c).

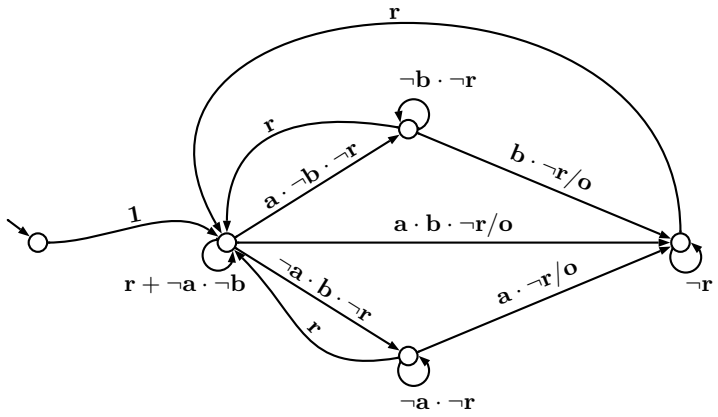
Given parameters τ and δ_{out}, and an execution mode, in this case sample-driven execution, the timed automaton of Figure 2(d) may be produced. The cube-like structures, positioned at each state of the original Boolean Mealy Machine, represent input latches. A single clock c constrains the timed behavior of the automaton. Latch transitions within the initial Boolean Mealy Machine state may happen at any time, but those in the other states may only occur when $0 < c \leq \tau$ (for technical reasons discussed in Subsection 5.2). The darker transition lines mark instants of reaction occurring precisely when $c = \tau$, except the initial reaction which happens immediately at $c = 0$. The destination of the reaction transitions depends on the latch contents and the original automaton, but it is always to an empty latch state. The clock c is reset on each reaction transition and thus measures the time elapsed since the last reaction. The right-most original state is split in two, control stays in the left-hand latch until the output o occurs then it shifts into the right-hand latch. The connecting transitions encode the emission of the output o when $c = \delta_{out}$ and they do not change the input latch contents.



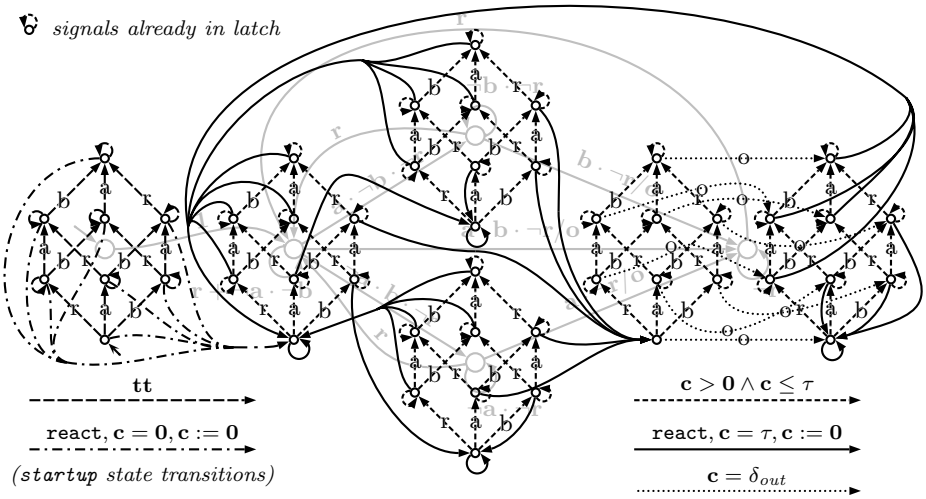
(a) Argos

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

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

end module
```

(b) Esterel



(c) Boolean Mealy Machine



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

Figure 2: Translation of ABRO to a sample-driven system

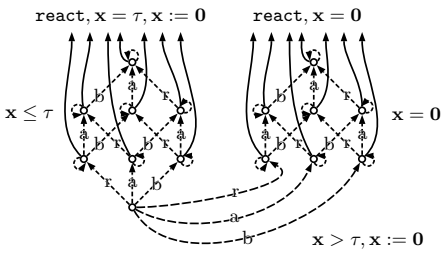


Figure 3: Urgent inputs (*trigger* = event)

When an event-driven mode is chosen, the translation works somewhat differently. If events are received while $c \leq \tau$, they are latched and a reaction occurs when $c = \tau$, as illustrated by the left-hand side of Figure 3. However, events occurring when $c > \tau$ trigger a switch to an *urgent* latch, the right-hand side of Figure 3, where further events may be captured before a reaction occurs in the same instant (time may not progress between the first such event and a reaction).

5.2 Details

The intricacies of the proposal in Section 4 are made more clear by a mapping from Boolean Mealy Machines, given certain timing and execution-mode parameters, into Timed Automata. Boolean Mealy Machines [11] are the basic semantic model for pure (no valued signals) Argos programs. Pure Esterel programs may be treated in a similar way.

Definition 5.1 *A* Boolean Mealy Machine $\langle S, s_0, I, O, T\rangle$ *is a 5-tuple where S is a set of states, $s_0 \in S$ is the initial state, I is a set of input signals, O is a set of output signals ($I \cap O = \emptyset$), and T is a transition relation. Each transition is labeled with a boolean formula over I, the set of such formulas being written $\mathrm{A}B(I)$, and a subset of O: $T \subseteq S \times \mathrm{A}B(I) \times \mathcal{P}(O) \times S$.*

The combination of program and implementation in dense time will be expressed using the timed transition systems of [1]. This formalism is a minimal and adequate framework for describing the mapping. The type of urgent action required by event-driven execution is also easily constructed.

Definition 5.2 *A* timed transition system $\langle \Sigma, L, L_0, C, E\rangle$ *is a 5-tuple where Σ is a finite alphabet, L is a set of control locations of which those in a subset L_0 are the initial control locations, C is a finite set of clocks, and E is a transition relation. Each transition is labeled with an element of Σ, a set of clocks to be reset, and a guard expression over clocks: $E \subseteq L \times \Sigma \times \mathcal{P}(C) \times \Phi(C) \times L$. Each $\delta \in \Phi(C)$ is formed from the grammar: $\delta := x \leq c \mid c \leq x \mid \neg\delta \mid \delta_1 \wedge \delta_2$, where $x \in C$ and the constants $c \in \mathbb{Q}$ (the rationals).*

Given a Boolean Mealy Machine fixed values for the parameters τ and δ_{out}, and a choice of either sample-driven or event-driven execution, a timed transition system incorporating the intuitions of Section 4 can be constructed. In the chosen notation, $\mathbb{B} = \{\mathsf{tt}, \mathsf{ff}\}$ and $\mathbb{Q}_0^+$ is the set of positive rationals including zero. The notation $l_s \xrightarrow[R]{\sigma,\ \phi} l_d$ is an abbreviation for the 5-tuple: $(l_s, \sigma, R, \phi, l_d) \in L \times \Sigma \times \mathcal{P}(C) \times \Phi(C) \times L$. The symbol $\dot\cup$ stands for disjoint union.

Definition 5.3 *For any set I, define a* valuation *with respect to $J \subseteq I$, $\nu_J : I \longrightarrow \mathbb{B}$, by:*

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

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

Definition 5.4 *Given a Boolean Mealy Machine, $A_B = \langle S, s_0, I, O, T\rangle$, and parameters $\tau \in \mathbb{Q}_0^+$, $\delta_{out} \in \mathbb{Q}_0^+$, and trigger $\in \{\mathsf{sample}, \mathsf{event}\}$, such that:*

(C1) $\delta_{out} \leq \tau$, and,

(C2) trigger $= \mathsf{event} \vee \tau > 0$,

the timed transition system $A^{trigger}_{\tau,\delta_{out}} = \langle \Sigma, L, L_0, C, E\rangle$ is defined:

- $\Sigma = I \,\dot\cup\, O \,\dot\cup\, \{\mathsf{react}\}$
- $L = S \,\dot\cup\, \{\mathsf{startup}\} \times \mathcal{P}(I) \times \mathcal{P}(O) \times \mathbb{B}$
- $L_0 = \{(\mathsf{startup}, \emptyset, \emptyset, \mathsf{ff})\}$
- $C = \{c\}$
- *E is the smallest set defined by the conjunction of:*

1. $\exists i \in I:\ J \cup \{i\} = J'$
   $\Longrightarrow (s, J, P, \mathsf{ff}) \xrightarrow[\emptyset]{i,\ c>0\ \wedge\ c\leq\tau} (s, J', P, \mathsf{ff}) \in E$
2. *trigger* $= \mathsf{event}$
   $\Longrightarrow (s, \emptyset, P, \mathsf{ff}) \xrightarrow[\{c\}]{i,\ c>\tau} (s, \{i\}, P, \mathsf{tt}) \in E$
3. $\exists i \in I:\ J \cup \{i\} = J'$
   $\Longrightarrow (s, J, P, \mathsf{tt}) \xrightarrow[\emptyset]{i,\ c=0} (s, J', P, \mathsf{tt}) \in E$
4. $\exists P \in \mathcal{P}(O):\ \exists m \in \mathrm{A}B(I):$
   $\wedge\ (trigger = \mathsf{sample} \vee J \neq \emptyset)$
   $\wedge\ (s, m, P, s') \in T$
   $\wedge\ \nu_J(m)$
   $\Longrightarrow (s, J, \emptyset, \mathsf{ff}) \xrightarrow[\{c\}]{\mathsf{react},\ c=\tau} (s', \emptyset, P, \mathsf{ff}) \in E$
5. $\exists P \in \mathcal{P}(O):\ \exists m \in \mathrm{A}B(I):$
   $\wedge\ (s, m, P, s') \in T$
   $\wedge\ \nu_J(m)$
   $\Longrightarrow (s, J, \emptyset, \mathsf{tt}) \xrightarrow[\emptyset]{\mathsf{react},\ c=0} (s', \emptyset, P, \mathsf{ff}) \in E$
6. $o \notin P$
   $\Longrightarrow (s, J, P \cup \{o\}, \mathsf{ff}) \xrightarrow[\emptyset]{o,\ c=\delta_{out}} (s, J, P, \mathsf{ff}) \in E$
7. *trigger* $= \mathsf{event}$
   $\Longrightarrow (\mathsf{startup}, \emptyset, \emptyset, \mathsf{ff}) \xrightarrow[\{c\}]{i,\ \mathsf{tt}} (s_0, \{i\}, \emptyset, \mathsf{tt}) \in E$
8. *trigger* $= \mathsf{sample}$
   $\wedge\ \exists i \in I:\ J \cup \{i\} = J'$
   $\Longrightarrow (\mathsf{startup}, J, \emptyset, \mathsf{ff}) \xrightarrow[\emptyset]{i,\ \mathsf{tt}} (\mathsf{startup}, J', \emptyset, \mathsf{ff}) \in E$
9. $\exists P \in \mathcal{P}(O):\ \exists m \in \mathrm{A}B(I):$
   $\wedge\ (trigger = \mathsf{sample})$
   $\wedge\ (s_0, m, P, s) \in T$
   $\wedge\ \nu_J(m)$
   $\Longrightarrow (\mathsf{startup}, J, \emptyset, \mathsf{ff}) \xrightarrow[\{c\}]{\mathsf{react},\ c=0} (s', \emptyset, P, \mathsf{ff}) \in E$

Essentially, the timed transition system of Definition 5.4 is derived from a Boolean Mealy Machine by adding latches for inputs and outputs, and a react symbol to mark the start of a reaction. A distinguished marker, startup, indicates that the program is in the initial state s_0, and a reaction has yet to occur; in which case transition timing is slightly different. The fourth component of the state tuple is a flag used to model actions that must occur *urgently*, after the techniques of [6]. The transitions coordinate latching and reacting while respecting the timing and trigger parameters. The end of a reaction is not marked by a special action but a similar effect can be obtained by adding a special output to each transition of the source Boolean Mealy Machine.

The conjuncts of the definition characterizing E can be understood:

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

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

The react symbol represents the beginning of a reaction. All inputs that have occurred since the last reaction are grouped into a single synchronous event. Inputs that occur afterward are latched for the next reaction, thus maintaining atomicity of reaction processing. Multiple input events may occur simultaneously with react, i.e. have the same time tag. There are essentially three ways of treating these events:

1. A *closed time guard* would have such inputs latched until the next reaction.
2. An *open time guard* considers all such inputs as part of the simultaneous reaction.
3. Although all inputs occur at the same time, some occur in sequence before react, and form part of the immediate reaction, and some afterward, and are latched for the next reaction.

Item 1 is problematic for event-driven systems where inputs may in fact cause simultaneous reactions. It may be reasonable for sample-driven systems and could be used to add feedback delays of sorts. For Item 3, the meaning of an order on events sharing the same time tag must be considered. For the present, this option is rejected, though further consideration may be warranted. Definition 5.4 respects Item 2 by forbidding—the $c > 0$ component of Conjunct 1, since c is reset at each reaction—timed words where simultaneous inputs occur after react. This choice necessitates some consideration of the initial state where $c = 0$ even though a reaction has yet to occur. Note that duplicate input events between reactions are effectively ignored.

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

Outputs are emitted simultaneously δ_{out} units after a reaction. While it would be possible to represent this emission with a single symbol, chosen from $\mathcal{P}(O)$, we prefer to represent each output individually and in arbitrary order, though all share the same time tag (Conjunct 6). Timed words where some outputs are not emitted are forbidden by constraining react transitions to those states where the output buffer component is empty (Conjuncts 4 and 5).

The system state before any reaction has occurred is distinguished by the startup element. For event-driven systems, Conjunct 7 ensures that the first event triggers a simultaneous reaction even if it occurs when $c = 0$. For sample-driven systems, Conjunct 8 permits input latching at time $c = 0$ and afterward. Conjunct 9 mandates that the first sample-driven reaction occurs at $t = 0$. An additional parameter specifying the time of the first sample-driven reaction could be introduced. Only the clock guard of the transition implied by Conjunct 9 need change. A simpler alternative would be to assume that the first reaction occurs at or after $c = \tau$. In this case, Conjuncts 7, 8 and 9 will be removed, L becomes $S \times \mathcal{P}(I) \times \mathcal{P}(O) \times \mathbb{B}$ and L_0 becomes $\{(s_0, \emptyset, \emptyset, \mathsf{ff}\}$.

Definition 5.4 is effectively an interface between a dense-time model where inputs and outputs are interleaved, though simultaneous events may have identical time tags, and the synchronous model where multiple, distinct inputs may be consumed at a reaction as a single event. Implementation details are modeled to some degree. The abstract program is effectively executed by evaluating the guards of potential transitions against the contents of the input latch, i.e. by $\nu_J(m)$ for each (s, m, O, s') matching the current state.

The mapping defined in Definition 5.4 simplifies several issues. Inputs are latched instantaneously as soon as they occur. Event detection is idealized: for instance, to detect changes it may be necessary to continuously sample an input value. Perfect timing is assumed, neither clock digitization nor drift are modeled. All reactions take an exact and equal amount of time to compute.

5.3 Liveness

Definition 5.4 gives rise to timed transition systems that specify safety constraints alone. For timed Büchi automata, a set of accepting states, $F \subseteq L$, must also be defined. Since reactions must occur infinitely often, and when they do occur the input latch is emptied, a natural choice is:

$$F = \{(s, \emptyset, \emptyset, \mathsf{ff}) \mid s \in S\}$$

For event-driven systems this constraint excludes timed words where inputs do not occur infinitely often (with non-converging time components).

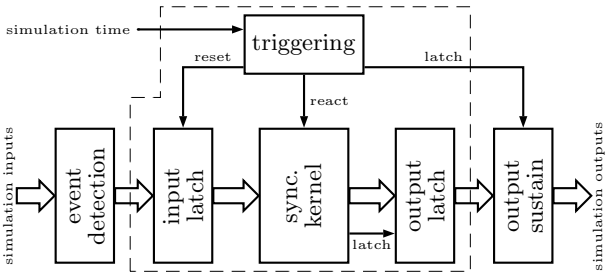


Figure 4: Simulation components

6. EMBEDDING WITHIN SIMULINK

We now describe how the timed transition system of Definition 5.4 relates to the operations of Simulink. There are at least three possible approaches:

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

2. Relate Simulink models, via informal descriptions of the tool, and test simulations, to another, better (mathematically) defined model of computation, as is done in [16]. The results of applying Definition 5.4 could then, for instance, be translated to Lustre subprograms.

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

We pursue the last, which provides sufficient guidance to implement a block or model, but does not directly describe interactions with other components nor the functioning of an entire model.

A realization in Simulink of the desired timing behavior could comprise the subcomponents depicted in Figure 4. The timed automata given by Definition 5.4 describe the *sync. kernel* that implements the original program, the *input latch* and *output latch*, and the *triggering* logic. Additionally, some *event detection* is necessary to translate from Simulink signal changes into discrete events, for example, polling signal values to detect rising or falling edges.

The pure signals used within Argos and Esterel to synchronize program components are subject to the perfectly synchronous semantics. Their effect outside of a synchronous kernel depends on a specific implementation. They might trigger an external event and could, for example, be communicated by Simulink's function-call triggering mechanism. They could generate a rising, falling, or pulse signal. Or they could be treated as valued, emitting a zero/false when absent and a one/true when present, though consecutive emissions might then be obscured. The *output sustain* subcomponent of Figure 4 models such details.

There is some merit in implementing each subcomponent of Figure 4 separately within Simulink, using a mix of built-in and custom blocks. Definition 5.4 would still function as a specification. In the following, we assume that all subcomponents except event detection and output sustain, i.e. everything inside the dashed region of Figure 4, is part of a single custom Simulink block. Event detection and output handling logic can then be constructed as required.

Simulink processes a model, a set of blocks and their interconnections, by increasing a global time parameter t in steps. At each step, blocks are polled in sequence to determine new signal and state values. When (if) a stable value is found for every signal, t is increased.

In particular [13, p. 3-36 and 3-37], a model is initialized before simulation by flattening the hierarchy of blocks, determining an execution order based on signal dependencies, and collecting details related to sampling. The simulation then proceeds in a loop where, for each iteration, termed a *major time step*, the value of t is fixed and the blocks are processed in sequence to determine signal values and the next value for t. There is another loop (of *minor time steps*) within each major time step for calculating the values of continuous signals and their derivatives.

Conceptually, each block represents, and must compute, three functions [13, p. 1-6]:

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

Given a simulation time t, state vector x and input vector u: f_0 determines the output values, f_u the next discrete state, and f_d the derivatives of continuous state elements. The latter function will not concern this paper further. The simulation engine can only sample a finite number of values of t. It is assumed that the functions f_o and f_u are constant between sample points, i.e. given consecutive sample times t_i and t_{i+1}:

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

The sample times chosen for a simulation run depend both on the blocks comprising a model and the global parameters. The parameters allow compromises between accuracy and speed when performing simulations, but, as noted in [16], they affect the semantics, in terms of signal traces over time, of a model. We now describe how a synchronous block specifies sample times and how it responds at other times, and assume that the simulation engine will not skip instants where the state changes.

The clock c of a timed automaton produced according to Definition 5.4 is tied to the simulation time t by tracking the clock value and previous sample time t_p as discrete state variables. At any sample instant, $c = x_c + (t - t_p)$, where x_c is the stored clock value. The clock value to be stored depends on whether a reaction has occurred:

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

where at any sample point, after processing inputs, *react* is calculated:

$$\mathit{react} = \begin{cases} c = \tau \ \vee\ t = 0 & \text{if } \mathit{trigger} = \mathsf{sample} \\ (c \ge \tau \ \vee\ \mathit{init}) \ \wedge\ J \neq \emptyset & \text{if } \mathit{trigger} = \mathsf{event} \end{cases}$$

where *init* is true only until the first reaction, and J represents the input latch contents.

The block must use *port based* [13, p. 7-20] inherited sample times to ensure that all input changes are detected and latched. *Block based* variable sample times [13, p. 7-17] are required to schedule output emissions and reactions.

The next required sample hit is calculated:

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

When $\delta_{out} = 0$ the outputs depend instantaneously upon the inputs and may thus cause *algebraic loops* (paths of instantaneous feedback) within the model. The block warns Simulink of this possibility.

7. RELATED WORK

Of the recent publications on the synchrony hypothesis with more timing detail, or synchronous languages mixed with Simulink, we choose three to discuss and compare in this section: the `AASAP` semantics [10], the Simulink to Lustre translation [9, 14], and the TAXYS framework [15].

7.1 AASAP Semantics

The Almost As Soon as Possible (`AASAP`) semantics [10] supports the verification and implementation of controllers modeled by timed automata. Two imperfections of real implementations are considered:

1. There is a delay between the occurrence of an event and its detection as an input.
2. Controller computations take time.

Both are modeled by a single parameter Δ. The delay between input occurrence and detection necessitates a form of latching which also tracks the age of an event.

The *program semantics* considers that controller clocks are digital and proposes an execution scheme: 1) read the time; 2) update input latches; 3) if possible, take a transition updating the state and possibly emitting an output. Δ is refined to two properties, Δ_L the loop execution delay and Δ_P the digital clock precision. The transition may be non-deterministic, allowing for more abstract specifications, whereas most synchronous language designers have focused on ensuring determinism for reasons of compilation and trouble-shooting.

The `AASAP` approach is different to the synchronous language paradigm exemplified by Esterel and Argos. The execution scheme is not necessarily sample-driven, since the period is bounded but not specified by Δ_L, though such an implementation would be a correct refinement. An `AASAP` controller is not necessarily event-driven, less so because of the constant polling, but more because it may act spontaneously by emitting outputs or taking internal steps.

The use of a digital clock contrasts with the notion of *multi-form time*, where event counters are used to measure intervals, promoted by the synchronous languages [4].

The biggest difference is in the treatment of events. The `AASAP` and related semantics queue input events and process them one-by-one, output events are also distinct, whereas a synchronous language event is more structured, consisting of bundled input and output signals.

Our model is similar to those of [10] in that it is parameterized by delay values representing platform limitations. It differs, however, in both intent and detail. Our approach focuses on generating simulation traces and provides no support for verification. The `AASAP` semantics gives a means for modeling, verifying and implementing controllers. We have focused on details related to the classical synchronous language execution schemes and address signal-bundling explicitly. Although input and output events are split, one of the former always precedes one of the latter. The model of Definition 5.4 ignores duplicate input events should they occur, whereas the `AASAP` semantics would detect such possibilities as a receptiveness problem in a given environment. Finally, the time to urgent action in the `AASAP` model depends on how long an individual input event has been latched, while in our model it depends on how much time has passed since the last reaction.

7.2 Simulink to Lustre

Although conceived as simulation software, Simulink is also used to design and generate implementations for embedded controllers. In [9] the authors show how to convert the discrete time components of Simulink models into the Lustre synchronous language. Controllers may then be verified using formal techniques and/or implemented using a certified compiler. An extension [14] handles aspects of the Stateflow tool.

Implementation concerns are also addressed by [9], where a time-triggered execution platform is assumed. Timing analysis and synthesis/scheduling makes use of annotations giving lower and upper bounds on task execution times.

We approach the application of synchronous languages to Simulink from a different perspective. Rather than convert a Simulink model into a synchronous program, we describe how to embed such a program within Simulink. Rather than utilizing the assumption of synchrony implicit to many Simulink blocks, we seek to model delays explicitly. It is conceivable that simulated controllers given in Argos or Esterel could be incorporated into the Simulink to Lustre translation scheme. The two execution parameters might then be passed on for timing analysis and compilation.

7.3 TAXYS framework

In [15] a notion of correctness between application software specified in logical time and a corresponding implementation as a real-time system is formally defined. TAXYS applies this general methodology to Esterel programs that are annotated with lower and upper bounds on the execution time of external functions. The running time of the control skeleton is ignored. Systems are verified in closed loop with an environment which is also specified in Esterel and annotated with clock constraints. Correctness is checked and a timing analysis performed by compiling both controller and environment models into timed automata, along with routines from KRONOS, to produce a verification engine.

The methodology specifies an execution platform involving multiple asynchronous tasks, a scheduler, and an *event-handler* that generalizes the event detection and input latch components of Section 6. An event-handler for synchronous languages is described, that bundles individual inputs into synchronous inputs based on separator events, others are given for interrupts and sampled signals.

The methodology of [15] is for verifying system properties. The model presented by this paper aims to address simulation issues, it does not require the addition of information to Argos or Esterel programs but instead that two parameters be estimated and given. It might be useful to enable simulations based on annotated TAXYS models, where the timing behavior would then be more dynamic and poten-

tially more accurate. Since Simulink traces a single path through a model, some mechanism for choosing between, or averaging, the lower and upper time bounds would be required.

8. CONCLUDING REMARKS

We have described an approach for simulating controllers specified in a synchronous language such as Argos or Esterel within Simulink. Simplified implementation details are modeled by two timing parameters. We have given a semantics in terms of timed automata that may in turn be embedded within Simulink.

9. REFERENCES

[1] R. Alur and D. L. Dill. Automata for modeling real-time systems. In *Proc. 17th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science, pages 322–335, Warwick, England, July 1990. Springer-Verlag.

[2] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proc. IEEE*, 91(1):64–83, Jan. 2003.

[3] G. Berry. *The Constructive Semantics of Pure Esterel*. `ftp://ftp-sop.inria.fr/meije/esterel/papers/constructiveness3.ps`, draft book, current version 3.0 edition, July 1999.

[4] G. Berry. *The Esterel v5 Language Primer*. Ecole des Mines and INRIA, version 5.92 edition, June 2000.

[5] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, Nov. 1992.

[6] S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In W. P. de Roever, H. Langmaack, and A. Pnueli, editors, *International Symp. Compositionality: The Significant Difference (COMPOS '97)*, volume 1536 of *Lecture Notes in Computer Science*, pages 103–129, Bad Malente, Germany, Sept. 1997. Springer-Verlag.

[7] T. Bourke and A. Sowmya. Formal models in industry standard tools: An Argos block within Simulink. In F. E. Tay, editor, *Int. J. Software Engineering and Knowledge Engineering: Selected Papers from the 2005 International Conference on Embedded and Hybrid Systems*, volume 15, pages 389–395, Singapore, Mar. 2005. World Scientific.

[8] P. Caspi. Embedded control: From asynchrony to synchrony and back. In T. A. Henzinger and C. M. Kirsch, editors, *Proc. 1st International Conference on Embedded Software (EMSOFT'01)*, volume 2211 of *Lecture Notes in Computer Science*, pages 80–99, Tahoe City, USA, Oct. 2001. Springer-Verlag.

[9] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *Proc. 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '03)*, pages 153–162. ACM, ACM Press, 2003.

[10] M. De Wulf, L. Doyen, and J.-F. Raskin. Almost ASAP semantics: from timed models to timed implementations. In *HSCC 04: Hybrid Systems — Computation and Control*, number 2993 in Lecture Notes in Computer Science, pages 296–310. Springer-Verlag, 2004.

[11] F. Maraninchi and Y. Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, 27(1–3):61–92, 2001.

[12] The Mathworks, Natick, MA, U.S.A. *Simulink — Using Simulink*, 5.1 edition, Sept. 2003. Release 13SP1.

[13] The Mathworks, Natick, MA, U.S.A. *Simulink — Writing S-Functions*, 5.1 edition, Sept. 2003. Release 13SP1.

[14] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a "safe" subset of Simulink/Stateflow into Lustre. In G. Buttazzo and S. Edwards, editors, *Proc. 4th ACM International Conference on Embedded Software (EMSOFT'04)*, pages 259 – 268, Pisa, Italy, Sept. 2004. ACM, ACM Press.

[15] J. Sifakis, S. Tripakis, and S. Yovine. Building models of real-time systems from application software. *Proc. IEEE*, 91(1):100–111, Jan. 2003.

[16] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating discrete-time Simulink to Lustre. *ACM Trans. Embedded Computing Systems*, 4(4):779–818, Nov. 2005.

[17] M. von der Beeck. A comparison of Statecharts variants. In H. Langmaack, W. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148. Springer-Verlag, Sept. 1994.