Polychronous mode automata

Jean-Pierre Talpin
IRISA/INRIA-Rennes
Campus de Beaulieu
F-35042 Rennes, France
Jean-Pierre.Talpin@irisa.fr

Christian Brunette
IRISA/INRIA-Rennes
Campus de Beaulieu
F-35042 Rennes, France
Christian.Brunette@irisa.fr

Thierry Gautier
IRISA/INRIA-Rennes
Campus de Beaulieu
F-35042 Rennes, France
Thierry.Gautier@irisa.fr

Abdoulaye Gamatié
INRIA Futurs
6b Av. Pierre et Marie Curie
59260 Lezennes, France
Abdoulaye.Gamatie@lifl.fr

ABSTRACT

Among related synchronous programming principles, the model of computation of the POLYCHRONY workbench stands out by its capability to give high-level description of systems where each component owns a local activation clock (such as, typically, distributed real-time systems or systems on a chip). In order to bring the modeling capability of POLYCHRONY to the context of a model-driven engineering toolset for embedded system design, we define a diagramic notation composed of mode automata and data-flow equations on top of the multi-clocked synchronous model of computation supported by the POLYCHRONY workbench. We demonstrate the agility of this paradigm by considering the example of an integrated modular avionics application. Our presentation features the formalization and use of model transformation techniques of the GME environment to embed the extension of POLYCHRONY's meta-model with mode automata.

Categories and Subject Descriptors

D.3 [**Programming Languages**]: Formal Definition and Theory

General Terms

Design, Languages, Theory

1. INTRODUCTION

Inspired by concepts and practices borrowed from digital circuit design and automatic control, the *synchronous hypothesis* has been proposed in the late '80s to facilitate the specification and analysis of control-dominated systems. Nowadays, synchronous languages are commonly used in the European industry, especially in avionics, to rapidly prototype, simulate, verify embedded software applications.

In this spirit, synchronous data-flow programming languages, such as LUSTRE [11], Lucid Synchrone [9] and SIGNAL [15], implement a model of computation in which time is abstracted by symbolic synchronization and scheduling relations to facilitate behavioral reasoning and functional correctness verification. While block diagrammatic modeling concepts are best suited for data-flow dominated applications, control-dominated processes may sometimes be preferably modeled using imperative formalisms, such as Esterel [3], Statecharts [12], or SyncCharts [1].

1.1 Design methodology

In the particular case of the POLYCHRONY workbench, on which SIGNAL is based, time is represented by *partially ordered* synchronization and scheduling relations, to provide an additional capability to model high-level abstractions of system paced by multiple clocks: globally asynchronous systems. This gives the opportunity to seamlessly model heterogeneous and complex distributed embedded systems at a high level of abstraction, while reasoning within a simple and formally defined mathematical model.

In POLYCHRONY, design proceeds in a compositional and refinement-based manner. It first consists of considering a weakly timed data-flow model of the system under consideration. Then, partial timing relations are provided to gradually refine the synchronization and scheduling structure of the application.

Finally, the correctness of refined specification is checked with respect to initial requirement specifications. That way, SIGNAL favors the progressive design of systems that are correct by construction using well-defined model transformations that preserve the intended semantics of early requirement specifications and that provide a functionally correct deployment on the target architecture.

1.2 Model-driven design framework

Taking advantage of recent works extending POLYCHRONY with a meta-model, Signal-Meta [6], in the model-driven engineering framework of GME (Generic modeling environment [14]), we position our problem as extending the meta-model on which SIGNAL is based with an inherited meta-

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

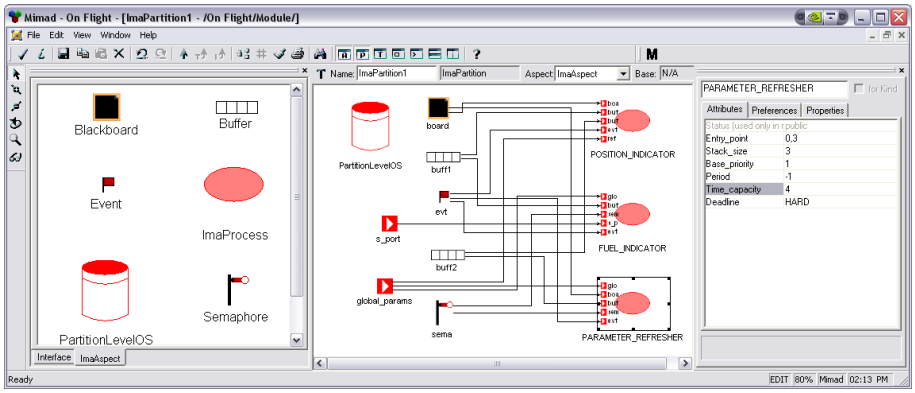
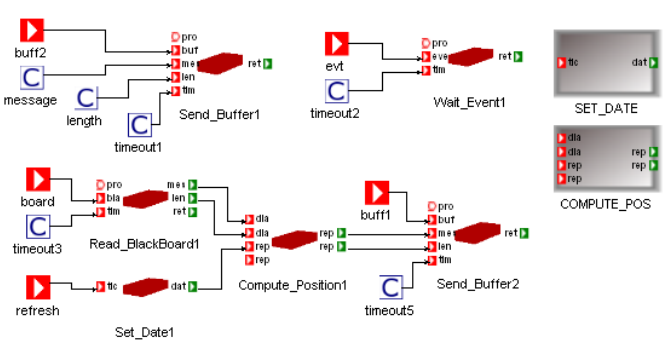
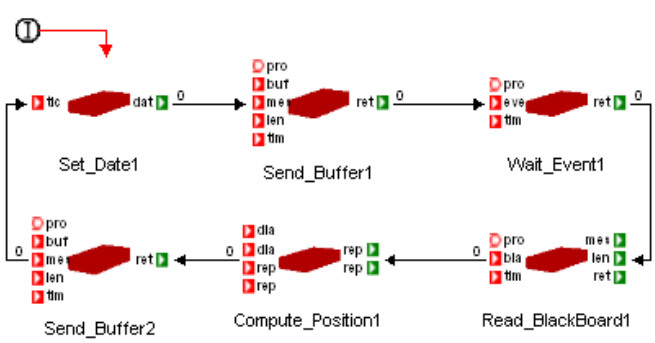


Figure 1: A MIMAD model of On_flight.



(a) Computation part (*ImaAspect*)



(b) Control part (*ImaProcessControl*)

Figure 2: The Position_indicator process.

model of multi-clocked mode automata to finally demonstrate how the latter can be translated into the former by operating a model transformation. We put an emphasis on simplicity both for the specification (one third of a page, Fig. 4) and for the formalization (five rules, Section 5.3) of mode automata. The framework of mode automata presented in this article was specified and implemented in the matter of one month, thanks to the facilities offered by the GME environment. It is currently being ported to Eclipse [5].

1.3 A modeling paradigm

The modeling of integrated modular avionics (IMA) architectures are a typical case in which both the polychronous model of computation and the need for mixed data-flow and control-flow formalisms (as offered by mode automata) are particularly well-suited.

As an example, consider the following diagram, Fig. 1, from the Signal-Meta environment[1]. It represents a simple avionic application within GME. Its main function consists of computing the current position of an airplane and its fuel level and reporting that information. It is decomposed into three processes:

- **Position_indicator** produces information about the current position of the aircraft.
- **Fuel_indicator** produces information about the level of kerosene in the aircraft.
- **Parameter_refresher** refreshes the parameters used by other processes.

To illustrate the use of mode automata at the process-level of this application, we focus on **Position_indicator**, Fig. 2(a)-2(b). It is composed of two main aspects: the ImaAspect includes the computational part and the ImaProcessControl contains the control flow part. The computational part (see Fig. 2(a)) consists of a data-flow graph. It contains Blocks of data-flow equations.

The control-flow part is best described in an imperative manner by a mode automaton, shown in Fig. 2(b). Each time the partition is active, the current state of the automaton indicates which of the Blocks in the computational part is executed. From the above descriptions, a corresponding SIGNAL program is automatically generated allowing one to use the functionalities of POLYCHRONY to formally analyze, transform and verify the application model.

1.4 Overview

The scope of this article is to present the definition of polychronous mode automata within the model-driven engineering framework Signal-Meta. It consists of an extension of the synchronous data-flow formalism SIGNAL with multi-clocked mode automata. To this end, the remainder of this paper is organized as follows.

Section 2 firsts presents related works. Section 3 gives an informal presentation of the SIGNAL formalism and of its extension. Section 4 outlines the meta-model of SIGNAL, defines its extension with mode automata and outlines the

[1]Signal-Meta is the model-driven front-end of POLYCHRONY designed with GME

use of GME to define the transformation of mode automata into SIGNAL. Section 5 formalizes the model transformation by considering the intermediate representation of SIGNAL. Section 6 provides operational semantics of mode automata framework. Concluding remarks are given in Section 7.

2. RELATED WORKS

The hierarchical combination of heterogeneous programming models is a notion whose introduction dates back to early models and formalisms for the specification of hybrid discrete/continuous systems.

The most common example is Matlab [18], which supports the Stateflow notation to describe modes in event-driven and continuous systems. Similarly, Ptolemy [7] allows for the hierarchical and modular specification of finite state machines hosting heterogeneous models of computation. Worth noticing is Hyscharts [2], which integrates discrete and continuous modeling capabilities within the same model-driven engineering framework.

In the same vein, mode automata were originally proposed by Maraninchi et al. [16] to gathering advantages of declarative and imperative approaches to synchronous programming and extend the functionality-oriented data-flow paradigm of Lustre with the capability to model transition systems easily and provide an additional imperative flavor. Similar variants and extensions of the same approach to mix multiple programming paradigms or heterogeneous models of computation [7, 8] have been proposed until recently, the latest advance being the combination of stream functions with automata [10]. Nowadays, commercial toolsets such as the Esterel Studio's Scade or Matlab/Simulink's Stateflow are largely inspired by similar concepts.

In previous work, the introduction of preemption mechanism in the multi-clocked data-flow formalism SIGNAL was previously studied by Rutten et al. [21]. This was done by associating data-flow processes with symbolic activation periods. However, no attempt has been made to extend mode automata with the capability to model multi-clocked systems, which is the aim of this article.

The main advantage of the multi-clocked approach over previous installments of mode automata principles lies in the capabilities gained for rapid prototyping: not only may functionalities and components be abstracted with multi-clocked specifications but mode describing early control requirements may then allow rapid prototyping of the system, while offering automated program transformation and code generation facilities to synthesis the foreseen implementation in a correct-by-construction manner.

3. POLYCHRONY

We position the problem by considering partially synchronized (or polychronous) specifications using the data-flow formalism SIGNAL [15].

3.1 Polychronous data-flow equations

A SIGNAL process consists of the simultaneous composition of equations on signals. A signal consists of an infinite flow of values that is discretely sampled according to the pace of its clock, noted $\hat{x}$. An equation partially relates signals with respect to an abstract timing model. SIGNAL defines the following primitive constructs:

- A functional equation `x = f(y,z)` defines an arithmetic or boolean relation `f` between its operands `y,z` and the result `x`.
- A delay equation `x = y pre v` initially defines the signal `x` by the value `v` and then by the value of the signal `y` from the previous execution of the equation. In a delay equation, the signals `x` and `y` are assumed to be synchronous, i.e. either simultaneously present or simultaneously absent at all times.
- A sampling `x = y when z` defines `x` by `y` when `z` is true and both `y` and `z` are present. In a sampling equation, the output signal `x` is present iff both input signals `y` and `z` are present and `z` holds the value *true*.
- A merge `x = y default z` defines `x` by `y` when `y` is present and by `z` otherwise. In a merge equation, the output signal is present iff either of the input signals `y` or `z` is present.
- The synchronous composition `(| P | Q |)` of the processes `P` and `Q` consists of simultaneously considering a solution of the equations in `P` and `Q` at any time.
- The equation `P / x` restricts the lexical scope of a signal `x` to a process `P`.

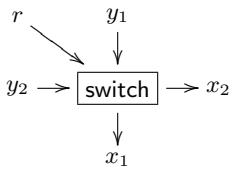
3.2 Mode automata

To express mode automata, we consider an extension of SIGNAL which comprises the following base syntactic elements. $\mathsf{init}\, s$ specifies the initial state (mode) of an automaton a. $s : p$ associates the behavior p with the mode s. $e \Rightarrow s \rightarrow t$ gives the clock e (or guard) of the next transition, from mode s to mode t, while $e \Rightarrow s \twoheadrightarrow t$ immediately transits from mode s to mode t upon the condition e (most likely a condition on input signals such as an alarm). The support of both weak preemption, noted $e \Rightarrow s \rightarrow t$, and strong preemption, noted $e \Rightarrow s \twoheadrightarrow t$, greatly enhance modeling capabilities to facilitate design. Synchronous composition of automata is noted $a \,|\, b$.

$$a, b ::= \mathsf{init}\, s \mid (s : p) \mid (e \Rightarrow s \rightarrow t) \mid (e \Rightarrow s \twoheadrightarrow t) \mid a \,|\, b$$

3.3 Example of a crossbar switch

To support the presentation of our modeling techniques, we consider the example of a simple crossbar switch. Its interface is composed of two input data signals y_1 and y_2 and a reset input signal r.



Data signals are routed along the output data signals x_1 and x_2 depending upon the internal state s of the switch. The state is toggled using the reset signal by the functionality $s = \mathsf{toggle}(r)$. Data is routed along an output signal x from two possible input sources y_1 or y_2 depending on the value of s by two instances of the functionality $x = \mathsf{route}(s, y_i, y_j)$ with $i \neq j$ and $i, j \in \{1, 2\}$.

$$(x_1, x_2) = \mathsf{switch}(y_1, y_2, r) \stackrel{\text{def}}{=} \left(\begin{array}{c} s = \mathsf{toggle}(r) \\ \mid x_1 = \mathsf{route}(s, y_1, y_2) \\ \mid x_2 = \mathsf{route}(s, y_2, y_1) \end{array} \right) / s$$

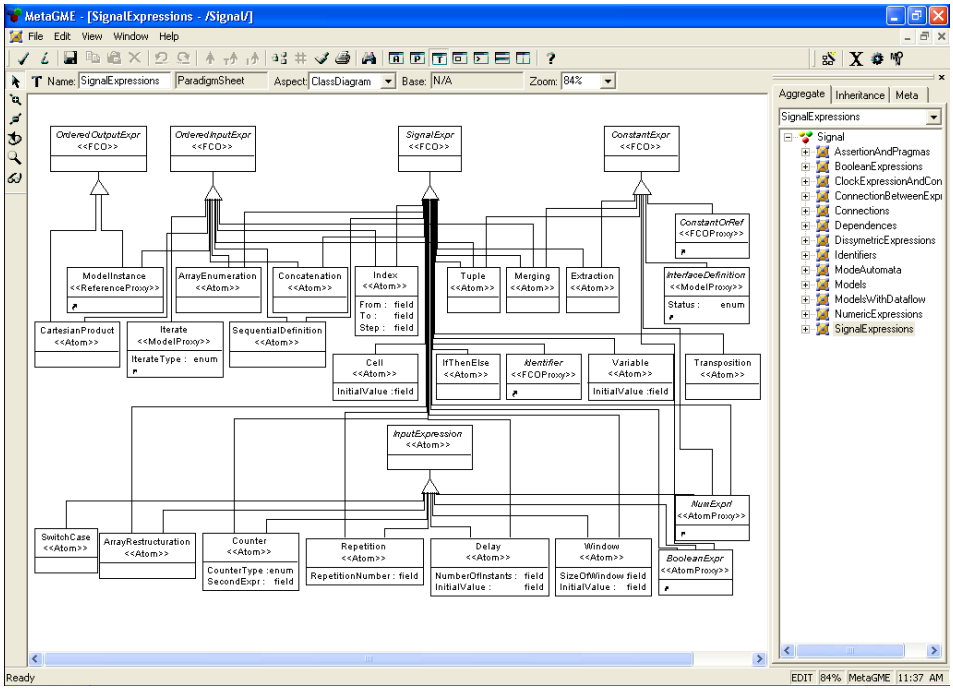


Figure 3: The SIGNAL meta-model in GME.

The subprocess toggle defines the state of the switch by the signal s. If the reset signal r is present and true, then the next state t is defined by the negation of current state s and otherwise by s.

$$s = \mathsf{toggle}(r) \stackrel{\text{def}}{=} (s = t \text{ pre true } | t = \text{ not } s \text{ when } r \text{ default } s)\,/t$$

The subprocess route selects which of the values of its input signals y_1 or y_2 to send along its output signals $x_i, i \in \{1,2\}$ depending on the boolean signal s. If s is present and true, it chooses y_i and else, if s is present and false, it chooses $y_{j\neq i}$.

Remember that SIGNAL equations partially synchronize input and output signals. In the route process, this implies that none of the signals y_1, y_2 and s are synchronized, and that the output signal $x_i, i \in \{1,2\}$ is present iff either of y_i are present and s true or $y_{j\neq i}$ is present and s false.

$$x_i = \mathsf{route}(s, y_i, y_j) \stackrel{\text{def}}{=} x_i = (y_i \text{ when } s) \text{ default } (y_j \text{ when not } s), \quad \forall 0 < i \neq j \leq 2$$

The switch is a typical example of specification where an imperative automata-like structure is superimposed on a native data-flow structure gives a shorter and more intuitive description of the system's behavior.

The mode automaton of the switch consists of two states flip and flop, in which routing is performed from $y_{1,2}$ to either $x_{1,2}$ or $x_{2,1}$ depending on the current mode of the automaton. The mode toggles from flip to flop, or conversely, when the event r occurs.

$$(x_1, x_2) = \mathsf{switch}(y_1, y_2, r) \stackrel{\text{def}}{=} \left(\begin{array}{l} \text{init flip } : (x_1 = y_1 \,|\, x_2 = y_2) \\ \quad | \text{ flop} : (x_1 = y_2 \,|\, x_2 = y_1) \\ \quad | \; r \Rightarrow \text{flip} \twoheadrightarrow \text{flop} \\ \quad | \; r \Rightarrow \text{flop} \twoheadrightarrow \text{flip} \end{array}\right)$$

4. A META-MODELING APPROACH

To develop our meta-modeling approach, we have used the GME environment [14], Fig 3. GME is a configurable UML-based toolkit that supports the creation of domain-specific modeling and program synthesis environments. GME uses meta-models to describe modeling paradigms for specific domains. The modeling paradigm of a given application domain consists of the basic concepts that represent its intended meaning from a syntactic and relational viewpoint.

4.1 The SIGNAL meta-model

The definition of a meta-model in GME is realized using the MetaGME modeling paradigm. First, modeling paradigm concepts are described in an UML class diagram. To achieve it, MetaGME offers some predefined UML-stereotypes [13], among which FCO, Atom, Model, and Connection. FCO (First Class Object) constitutes the basic stereotype in the sense that all the other stereotypes inherit from it. It is used for expressing abstract concepts. Atoms are elementary objects that cannot include any sub-part. On the contrary, models may be composed of several FCOs.

Containment and Inheritance relations are represented as in UML. All the other types of relations are specified through Connections. Some of these stereotypes are used in the class diagram represented in Fig. 4. For the SIGNAL meta-model, called Signal-Meta [6], class diagrams describe all syntactic elements defined in SIGNAL v4 [4]. Among these concepts, there are an Atom for each SIGNAL operator (e.g. numeric, clock relations, constraints), a Model for each SIGNAL "container" (e.g. process declaration, module), and a Connection for each relation between SIGNAL operators (e.g. definition, dependence).

With these class diagrams, GME provides a mean to express the visibility of FCOs within a model through the notion of Aspect (i.e. one can decide which parts of the de-

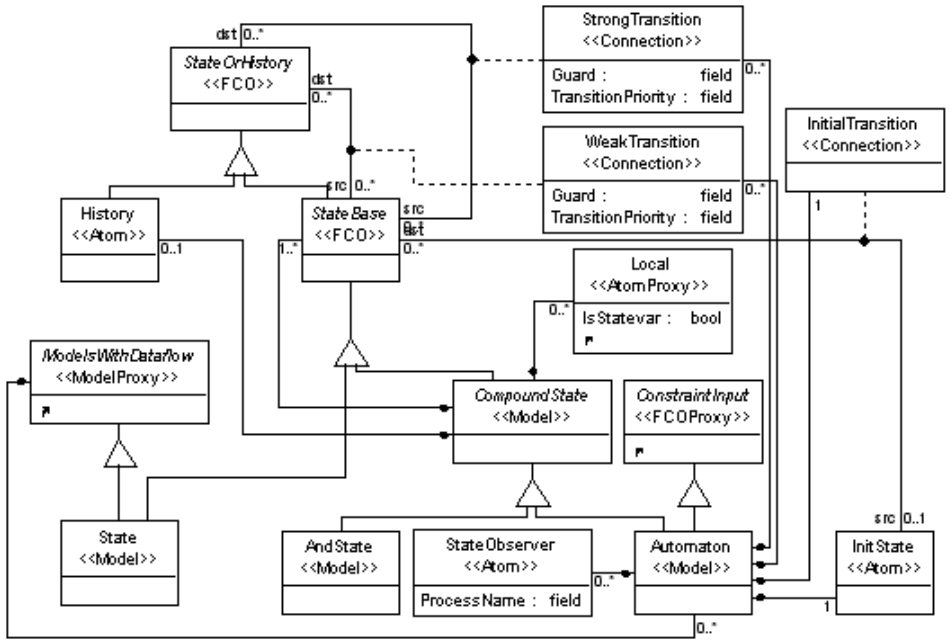


Figure 4: Extension of the SIGNAL meta-model with mode automata.

scriptions are visible depending on their associated aspects). Moreover, it is possible to restrict the use of some FCOs (add/remove in/from a Model) to a specific Aspect, even if these FCOs are visible in other Aspects. Signal-Meta includes two main Aspects: *Computation part* and *Clock and Dependence Relations*. The first one manages all data-flow relations of the Model, and the second one, all clock relations between signals.

Finally, OCL Constraints can be added to class diagrams in order to check some "parametric" properties on a model designed with this paradigm (e.g. the type of connections linked to an FCO according to the value of a FCO attribute). A parametric property depends on values given during the edition of a model. In Signal-Meta, OCL constraints check for example the validity of attribute values, such as the uniqueness of names inside a Model.

4.2 Refinement of the meta-model with modes

To manage mode automata, we extend Signal-Meta with a new class diagram represented in Fig. 4. An *Automaton* is a Model composed of states, transitions, local signals, and *StateObservers*. As for classical Statecharts [12], or SyncCharts [1], there are three kinds of states: *AndState*, *Automaton*, and *State*. The two former are Models composed of other states (*CompoundState*), whereas the latter is a terminal state describing SIGNAL equations. An *AndState* consists of several states composed in parallel.

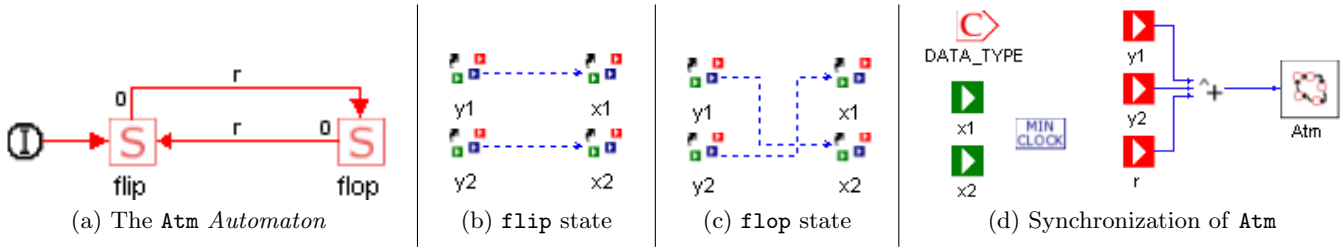
An *Automaton* can be added to another *Automaton* as a state (to create hierarchical automata), or to one of the Signal-Meta Models (represented in the class diagram by the *ModelsWithDataflow* Model). Thus, mode automata can be composed of SIGNAL programs or of sub-mode automata. This abstract concept represents Models including the two Aspects mentioned in the previous section and all operators described in Signal-Meta. *State* inherits from this Model to be able to describe SIGNAL equations. Finally, the *InitState* Atom is intended to be connected to the initial state of the *Automaton*.

The automata transitions are represented as Connections in the meta-model. Two kinds of transitions are considered: *StrongTransition*, and *WeakTransition*. *StrongTransition*s are used to compute the current state of the *Automaton* (before entering the state), whereas *WeakTransition*s are used to compute the state for the next instant.

More precisely, the guards of the *WeakTransition*s are evaluated to estimate the state for the next instant, and the guards of the *StrongTransition*s whose source is the state estimated at the previous instant, are evaluated to determine the current state. However, note that for each *Automaton*, at most one *StrongTransition* can be taken at each instant. To distinguish both kinds of transitions, a *StrongTransition* is denoted by light arrow in the graphical representation (see Fig. 5(a)), whereas *WeakTransition* is represented by a bold one.

Contrarily to SyncCharts [1], in which as many transitions as possible can be taken, in our model, at most two transitions can be taken during one reaction: one *StrongTransition* and/or one *WeakTransition*. It is also the case in [10]. This guarantees that there is no infinite loop when determining the current state of an automaton. For example, the determination of the current state for the `Atm` *Automaton* represented in Fig. 5(a) when the event r is emitted would be impossible if we allowed to take as many transitions as possible. Note also that the guard of a *StrongTransition* should not depend on signals defined in the state connected to this transition.

Both kinds of transitions link, inside an *Automaton*, a state to another one, or to the *History* Atom of one of the *CompoundState* sub-state of this *Automaton*. If the transition taken to arrive at a *CompoundState* is connected to the state itself, this *CompoundState* is automatically reinitialized. This reinitialization corresponds, for an *Automaton*,



(a) The `Atm` *Automaton* (b) `flip` state (c) `flop` state (d) Synchronization of `Atm`

Figure 5: Description of the `Atm` *Automaton* in GME.

to execute it from its initial state, and for an *Andstate*, to reinitialize all its sub-states. On the contrary, the *CompoundState* retains its previous state if the transition is connected to its *History*.

Each kind of transition has two attributes: *Guard*, in which the guard of the transition is expressed, and *TransitionPriority*, in which an integer expresses the priority of this transition among all transitions of the same kind (*WeakTransition* or *StrongTransition*) with the same source state. The smaller the value associated with the transition is, the higher the priority of the transition is. Thus, we can guarantee the determinism of the automaton. An OCL constraint checks that for each state, all outgoing *WeakTransitions* (resp. *StrongTransitions*) have different priorities. A third kind of Connection (*InitialTransition*) has been added to link the *InitState* of an *Automaton* to any state that corresponds to the initial one. There can be only one such Connection in an *Automaton*.

To observe the state of an automaton, we add a *StateObserver* Atom, which allows to call a process having the current state of the automaton as input signal. The name of this process is provided through the attribute *ProcessName*. If this attribute is not defined, the current state is written on the standard output. Basically, the clock of an automaton depends on the clocks of the signals used in all its transitions and states. Alternatively, the clock of an automaton can be explicitly specified. In the meta-model, this is expressed by the inheritance of *Automaton* from *ConstraintInput*.

4.3 Modeling of the crossbar switch

We illustrate the use of the mode automata extension in the example of the switch. Fig. 5(a) represents the modeling of the mode automaton of the switch in GME. `Atm` contains two terminal states (`flip` and `flop`). *StrongTransitions* are guarded by the value of the event `r`, as labeled on the middle of transitions. The 0 indicates the transition priority (it can be omitted here). The content of `flip` (resp. `flop`) state is represented in Fig. 5(b) (resp. 5(c)). In these figures, dotted arrows correspond to partial definitions in SIGNAL. `x1`, `x2`, `y1`, `y2` are references to signals from an upper Model.

The upper Model is that of the `switch`, and `Atm` and all the signals it uses are declared there. In this Model, `y1`, `y2`, and `r` are input signals, and `x1` and `x2` are output signals. In Fig. 5(d), the clock of `Atm` is fixed to the union of the clocks of `y1`, `y2`, and `r`. The clocks of `x1` and `x2` have to be specified explicitly because they are defined using partial definitions: a *MinClock* operator is used to define the clock of `x1` and `x2` as the union of clocks of their partial definitions. The `DATA_TYPE` parameter is used to associate a generic type with input and output signals.

4.4 Implementation in GME

GME offers different means to extend its environment with tools, such as the **MetaGME Interpreter**, which, like a compiler, checks the correctness of the meta-model, generates the paradigm file, and registers it into GME. This file is then used by GME to configure its environment for the newly defined paradigm.

In a similar way as the MetaGME Interpreter, we have developed a GME Interpreter to analyze Signal-Meta Models and produce the corresponding SIGNAL programs. We extend this Interpreter to produce the SIGNAL equations corresponding to mode automata descriptions. The code in Fig. 6 is that generated by the Interpreter for the switch example specified in Fig. 5 (note that in the concrete SIGNAL syntax: `y$ init v` is the real notation for `y pre v`; `x^=y` stands for $\hat{x} = \hat{y}$; `^+` designates the union of clocks; `x:=...` and `x:=...` represent respectively a partial definition and a complete definition of `x`). The transformation works as follows. For each automaton:

- One enumeration type is built (line 21). Each value of the enumeration is the name of a state (the uniqueness of names is checked).
- Four signals of this type are created. They correspond to the current state (`currentState`), the previous state (`previousState`), the next state (`nextState`) of the *Automaton* (lines 22-23) and its previous value (`zNextState`).
- an event is created for each transition of the *Automaton* (line 20). For a *WeakTransition* (resp. *StrongTransition*), this event is present when its guard is *true* and when the `currentState` (resp. `zNextState`) is equal to the source state of the transition. In this example, we have only *StrongTransitions* (lines 5-6).
- If the *Automaton* contains *CompoundStates* (it is not the case in our example), then two boolean signals are added: `history`, and `nextHistory`. They are *true* if the *StrongTransition* (resp. *WeakTransition*) taken to determine the `currentState` (resp. `nextState`) is connected to the *History* Atom of the destination *CompoundState*.
- The `previousState` and `zNextState` are defined respectively by the last value of `currentState` (line 12) and `nextState` (line 13).
- To define the `nextState` (line 8) (resp. `currentState` (lines 9-11)), the destinations of all *WeakTransitions* (resp. *StrongTransitions*) are conditioned by the event of the corresponding transition. The default values

```
1.  process Switch =
2.    { type DATA_TYPE; }
3.    ( ? DATA_TYPE y1,y2; event r; ! DATA_TYPE x1, x2; )
4.    (| min_clock(x2) | min_clock(x1)
5.     | %Atm%(| __ST_0_flop_To_flip := when (r) when (_Atm_0_zNextState = #flop)
6.             | __ST_1_flip_To_flop := when (r) when (_Atm_0_zNextState = #flip)
7.             | _Atm_0_currentState ^= (y1 ^+ y2 ^+ r)
8.             | _Atm_0_nextState := _Atm_0_currentState
9.             | _Atm_0_currentState := #flip when __ST_0_flop_To_flip
10.                                   default #flop when __ST_1_flip_To_flop
11.                                   default _Atm_0_zNextState
12.            | _Atm_0_previousState := _Atm_0_currentState$ init #flip
13.            | _Atm_0_zNextState := _Atm_0_nextState$ init #flip
14.            | case _Atm_0_currentState in
15.              {#flop}: (| x2 ::= y1 | x1 ::= y2 |)
16.              {#flip}: (| x2 ::= y2 | x1 ::= y1 |)
17.              end
18.            |)
19.           where
20.              event __ST_0_flop_To_flip, __ST_1_flip_To_flop;
21.              type _Atm_0_type = enum(flop, flip);
22.              _Atm_0_type _Atm_0_currentState, _Atm_0_previousState;
23.              _Atm_0_type _Atm_0_nextState, _Atm_0_zNextState;
24.           end
25.   |); % process Switch
```

Figure 6: The code generated from the `switch` model by the SIGNAL Interpreter

of the `nextState` and the `currentState` are respectively the `currentState` and the `zNextState`. If the *Automaton* is a sub-state of another one, the `currentState` is defined by the initial state of this *Automaton* if the `history` signal of the upper level *Automaton* is false. In our example, there is no *Weak-Transition*, thus `nextState` is always defined by the `currentState`. Note that the order of the transitions is not important, except for states with several outgoing transitions. In this case, transitions are ordered according to their priority.

- Mode changes are expressed according to the value of `currentState` (lines 14-17).

In a given *Automaton*, the clock of `currentState` is synchronized to that of `nextState`. Nonetheless, it may be defined by that of another *Automaton*. At the top-level, the clock of `currentState` is synchronized (line 7) only if there is some explicit synchronization in the Model, such as the Connection to `Atm` on the right of Fig. 5(d). For *AndState*s, the Interpreter has just to compose the equations of all sub-states. Finally, for *State*s, equations are produced as for any Signal-Meta Model [6].

5. FORMALIZATION

We use the POLYCHRONY workbench to perform formal verification (model checking and controller synthesis are provided with the Sigali tool [17]) and sequential and distributed code generation (in C, C++ or Java) starting from models with mode automata. Taking advantage of the meta-modeling framework provided by GME, we define the necessary generation of Signal code from the meta-model for mode automata.

5.1 An intermediate representation

The data-flow synchronous formalism SIGNAL supports an intermediate representation of multi-clocked specifications that exposes control and data-flow properties for the purpose of analysis and transformation. In this structure, noted G, a node g is a data-flow relation that partially defines a clock or a signal. A signal node $c \Rightarrow x = f(y,z)$ partially defines x by $f(y,z)$ at the clock c. A clock node $\hat{x} = e$ defines a relation between two particular signals or events called clocks.

$$\begin{array}{rcll} G,H & ::= & g \,|\, (G \,\|\, H) \,|\, G/x & \text{(graph)} \\ g,h & ::= & \hat{x} = e \,|\, c \Rightarrow x = f(y,z) & \text{(nodes)} \end{array}$$

A clock c expresses a discrete sample of time by a set of instants. It defines the condition upon which (or the time at which) a data-flow relation is executed. The clock $\hat{x}$ means that the signal x is present (its value is available). The clocks $[x]$ and $[\neg x]$ mean that x is present and is true (resp. false). A clock expression e is a boolean expression and 0 is the clock that means never (or the empty set of instant).

$$\begin{array}{rcll} c & ::= & \hat{x} \,|\, [x] \,|\, [\neg x] & \text{(clock)} \\ e & ::= & 0 \,|\, c \,|\, e_1 \setminus e_2, \,|\, e_1 \vee e_2 \,|\, e_1 \wedge e_2 & \text{(expression)} \end{array}$$

The decomposition of a process into the synchronous composition of clock and signal nodes is defined by induction on the structure of p. Each equation is decomposed into a data-flow function and is guarded by a condition, that is usually the clock $\hat{x}$ of the output signal.

$$\begin{array}{rcl} \mathcal{G}[\![x = y \,\mathsf{pre}\, v]\!] & \stackrel{\text{def}}{=} & (\hat{x} \Rightarrow x = y \,\mathsf{pre}\, v) \,|\, (\hat{x} = \hat{y}) \\ \mathcal{G}[\![x = y \,\mathsf{when}\, z]\!] & \stackrel{\text{def}}{=} & (\hat{x} \Rightarrow x = y) \,|\, (\hat{x} = \hat{y} \wedge [z]) \\ \mathcal{G}[\![x = y \,\mathsf{default}\, z]\!] & \stackrel{\text{def}}{=} & (\hat{y} \Rightarrow x = y) \\ & & \,|\, (\hat{z} \setminus \hat{y} \Rightarrow x = z) \\ & & \,|\, (\hat{x} = \hat{y} \vee \hat{z}) \\ \mathcal{G}[\![p \,|\, q]\!] & \stackrel{\text{def}}{=} & \mathcal{G}[\![p]\!] \,|\, \mathcal{G}[\![q]\!] \\ \mathcal{G}[\![p/x]\!] & \stackrel{\text{def}}{=} & \mathcal{G}[\![p]\!]/x \end{array}$$

5.2 Application to the crossbar switch

Let us construct the graph of the crossbar switch. It can modularly be defined by one instance of the toggle function-

ality and two instances of the router. Each functionality is decomposed into a set of guarded data-flow relations: its signal nodes, and its specific timing model, expressed by clock nodes.

$$\begin{array}{rcl}
\mathcal{G}[\![\mathsf{switch}]\!] & \stackrel{\text{def}}{=} & (\mathcal{G}[\![\mathsf{toggle}]\!] \,|\, \mathcal{G}[\![\mathsf{route}_1]\!] \,|\, \mathcal{G}[\![\mathsf{route}_2]\!])\,/st \\
\mathcal{G}[\![\mathsf{toggle}]\!] & \stackrel{\text{def}}{=} & (\hat{s} \Rightarrow s = t \,\mathsf{pre}\, \mathsf{true}\,) \\
 & & |\,([r] \Rightarrow t = \,\mathsf{not}\, s) \\
 & & |\,(\hat{t} \setminus r \Rightarrow t = s) \,|\, (\hat{t} = \hat{s}) \\
\mathcal{G}[\![\mathsf{route}_i]\!] & \stackrel{\text{def}}{=} & ([s] \Rightarrow x_i = y_i) \\
 & & |\,([\neg s] \Rightarrow x_i = y_j) \qquad 0 < i \neq j \leq 2
\end{array}$$

5.3 Compilation of mode automata

The compilation of a mode automaton into multi-clocked data-flow equations consists of its structural translation into partial equations modeling guarded commands and of the addition of the necessary synchronization relations described by clock equations. The top level rule $\mathcal{C}[\![a]\!]$ defines the current state of a, represented by a signal x (its next value being synchronously carried by the x').

The clock of the mode automaton is hence $\hat{x}$. It is synchronized to the clock expression e_x, the activity clock of the automaton: if at least one signal y defined by the automaton has an active clock $\hat{y}$, the automaton is activated to compute it and to possibly perform some transition.

The rule $\mathcal{C}^x[\![\,\mathsf{init}\, s_0]\!]$ defines x initially by the initial state s and then by the previous value of the next state x' unless one of the conditions c of strongly preemptive transitions prevails. The rule $\mathcal{C}^x[\![c \Rightarrow s \to t]\!]$ defines the next state x' by t if the current state s is x and the condition c holds.

The rule $\mathcal{C}^x[\![c \Rightarrow s \twoheadrightarrow t]\!]$ defines the current state x by t when the condition c holds upon entering state s (i.e. when the previous value of the next state x' is s). The rule $\mathcal{C}^x[\![s : p]\!]$ defines a mode s by guarding the process p with the condition $[x = s]$. The condition $[x = s]$ can equally be regarded as the clock $[y]$ where the signal y is defined by the equation $y = \mathsf{eq}\,(x, s)$.

$$\begin{array}{rcl}
\mathcal{C}[\![a]\!] & \stackrel{\text{def}}{=} & (\mathcal{C}^x[\![a]\!] \,|\, (\hat{x} = \hat{x}') \,|\, (\hat{x} = e_x))\,/x, x' \\
\text{with } e_x & \stackrel{\text{def}}{=} & \bigvee_{y \in \mathrm{defs}(a)} \hat{y} \\
\mathcal{C}^x[\![\,\mathsf{init}\, s_0]\!] & \stackrel{\text{def}}{=} & \hat{x} \setminus f_x \Rightarrow x = x' \,\mathsf{pre}\, s_0 \\
\text{with } f_x & \stackrel{\text{def}}{=} & \bigvee_{(c \Rightarrow s \twoheadrightarrow p) \in a} c \\
\mathcal{C}^x[\![c \Rightarrow s \to t]\!] & \stackrel{\text{def}}{=} & [x = s] \wedge c \Rightarrow x' = t \\
\mathcal{C}^x[\![s : p]\!] & \stackrel{\text{def}}{=} & [x = s] \Rightarrow \mathcal{G}[\![p]\!] \\
\mathcal{C}^x[\![c \Rightarrow s \twoheadrightarrow t]\!] & \stackrel{\text{def}}{=} & [(x' \,\mathsf{pre}\, s_0) = s] \wedge c \Rightarrow x = t \\
\mathcal{C}^x[\![a \,|\, b]\!] & \stackrel{\text{def}}{=} & \mathcal{C}^x[\![a]\!] \,|\, \mathcal{C}^x[\![b]\!]
\end{array}$$

The notation $[x = s] \Rightarrow G$ conditions G, the behavior of an automaton in the mode s, by the condition $[x = s]$. It can be decomposed into a set of core Signal equations by application of the following translation rules:

$$\begin{array}{rcl}
c \Rightarrow (G \,|\, H) & \stackrel{\text{def}}{=} & (c \Rightarrow G) \,|\, (c \Rightarrow H) \\
c \Rightarrow (\hat{x} = e) & \stackrel{\text{def}}{=} & (c \wedge \hat{x}) = (c \wedge e) \\
c \Rightarrow (G/x) & \stackrel{\text{def}}{=} & (c \Rightarrow G)/x,\ x \notin \mathrm{vars}(c) \\
c \Rightarrow (d \Rightarrow x = f(y_{1..n})) & \stackrel{\text{def}}{=} & (c \wedge d) \Rightarrow x = f(y_{1..n})
\end{array}$$

6. SEMANTICS OF MODE AUTOMATA

We complete the formalization of our extension to the Signal meta-model by the definition of the operational semantics of polychronous automata. It starts with the exposition of a micro-step automata theory and continues with the specification of the micro-step automata admitted by polychronous modes.

6.1 Micro-step automata

We first consider the theory of synchronous micro-step automata proposed by Potop et al. [19]. As already demonstrated for SIGNAL in [22], this framework accurately renders concurrency and causality for synchronous (multi-clocked) specifications.

Micro-step automata communicate through signals $x \in X$. The *labels* $l \in L_X$ generated by the set of names X are represented by a partial map of *domain* from a set of signals X noted $\mathrm{vars}(l)$ to a set of values $V^\perp = V \cup \{\perp\}$. The label $\perp$ denotes the *absence* of communication during a transition of the automaton. We write $\mathrm{supp}(l) = \{x \in X \,|\, l(x) \neq \perp\}$ for the *support* of a label l and $\perp_X$ for the empty support. We write $l' \leq l$ iff there exists l'' disjoint from l' and such that $l = l' \cup l''$.

An *automaton* $A = (s^0, S, X, \to)$ is defined by an initial state s^0, a finite set of states S noted s or $x = v$, labels L_X and by a transition relation $\to$ on $S \times L_X \times S$. The *product* $A_1 \otimes A_2$ of $A_i = (s_i^0, S_i, X_i, \to_i)$ for $0 < i \leq 2$ is defined by $((s_1^0, s_2^0), S_1 \times S_2, X_1 \cup X_2, \to)$ where $(s_1, s_2) \to^l (s_1', s_2')$ iff $s_i \to^{l|_{X_i}} s_i'$ for $0 < i \leq 2$ and $l|_{X_i}$ the projection of l on X_i. An automaton $A = (s^0, S, X, \to)$ is *concurrent* iff $s \to^\perp s$ for all $s \in S$ and if $s \to^l s'$ and $l' \leq l$ then there exists $s'' \in S$ such that $s \to^{l'} s''$ and $s'' \to^{l \setminus l'} s'$.

A *synchronous* automaton $A = (s^0, S, X, c, \to)$, of clock $c \in X$, consists of a concurrent automaton $(s^0, S, X, \to)$ s.t.

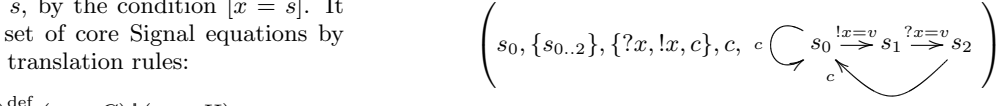
1. if $s \to^l s$ then $l = c$ or $c \notin l$: a clock transition always happens alone.

2. if $s^0 \to^c s^0$ and $s \to^c s'$ then $s' \to^c s'$: a clock transition can stutter.

3. if $s_{i-1} \to^{l_i} s_i, \forall i \in]0, n]$ and $l_i \neq c$ for $i < n$ and $l_n = c$, then $\mathrm{vars}(l_i) \cap \mathrm{vars}(l_j) = \emptyset$ for all $0 < i \neq j < n$: a reaction is composed of transitions on disjoint supports.

The composition of automata is defined by synchronized product and synchronous communication using 1-place synchronous FIFO buffers. The synchronous FIFO of clock c and channel x is noted $\mathrm{SFIFO}(x, c)$. It serializes the emission event $!x = v$ followed by the receipt event $?x = v$ within the same transition (the clock tick c occurs afterwards).

$$\mathrm{SFIFO}(x, c) \stackrel{\text{def}}{=}$$



Let $A_i = (s_i^0, S_i, X_i, c_i, \to_i)_{i=1,2}$ be two synchronous automata and c a clock and write $A[c_2/c_1]$ for the substitution of c_1 by c_2 in A. The synchronous composition $A_1 \,|^c A_2$ is defined by the product of A_1, A_2, and a series of synchronous FIFO buffers $\mathrm{SFIFO}(x, c)$ that are all synchronized on the same clock c.

$$A_1 \,|^c A_2 \stackrel{\text{def}}{=} (A_1[c/c_1]) \otimes (A_2[c/c_2]) \otimes \bigotimes_{x \in (\mathrm{vars}(X_1) \cap \mathrm{vars}(X_2))} \mathrm{SFIFO}(x, c)$$

$$\mathcal{A}[\![a]\!] \stackrel{\text{def}}{=} \left(s_a, S_a \bigcup_{(s:p)\in a} S_p, \text{vars}(a), \tau, \bigcup_{(s:p)\in a} \left(T_p[s/s_p] \bigcup_{u\in \text{final}\,(T_p)\setminus s_p}^{(c\Rightarrow s\to t)\in a} \left(\left(T_c^{u,r} \cup r \xrightarrow{\tau} t \right) /r \right) \bigcup_{u\in \text{init}\,(T_p)\setminus s_p}^{(c\Rightarrow s\to t)\in a} T_c^{u,t} \right) \right)$$

Figure 7: Semantics of a mode automaton

6.2 Micro-step semantics of SIGNAL

Micro-step automata provide a simple and expressive operational framework to formalize the semantics of multi-clocked specifications.

Clocks

A clock expression e corresponds to a transition system $T_e^{s,t}$ from s to t which evaluates the presence of signals in accordance to e.

$$T_c^{s,t} \stackrel{\text{def}}{=} \left(s \xrightarrow{l_c} t \right) \qquad T_{c\wedge d}^{s,t} \stackrel{\text{def}}{=} \left(\begin{array}{ccccc} & l_c & s' & l_d & \\ s & & & & t \\ & l_d & t' & l_c & \end{array} \right) /s't'$$

$$T_{c\vee d}^{s,t} \stackrel{\text{def}}{=} \left(T_{c\wedge d}^{s,t} \cup T_c^{s,t} \cup T_d^{s,t} \right)$$

We write l_c for the label l that corresponds to the clock c and canonically denote v_x the generic value of the signal x : $l_{\hat{x}} \stackrel{\text{def}}{=} (?x = v_x)$, $l_x \stackrel{\text{def}}{=} (?x = 1)$ and $l_{\neg x} \stackrel{\text{def}}{=} (?x = 0)$

Relations

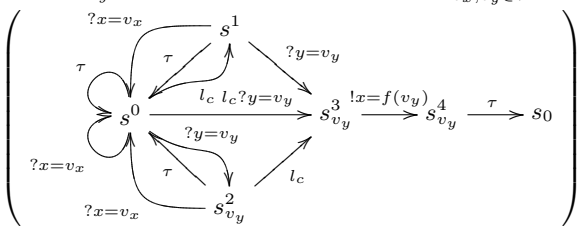
A synchronization relation $\hat{x} = e$ accepts the events $\hat{x}$ and e in any order, or none of them, and then performs a clock transition c. Hence, the conditions expressed by $\hat{x}$ and e need to occur at the same time.

$$\mathcal{A}[\![\hat{x} = e]\!] \stackrel{\text{def}}{=} \left(s, \{s,t\}, \{c,x\} \cup \text{vars}(e), c, \left(t \xrightarrow{c} s \right) \bigcup_{v_y\in V \mid y\in\text{vars}(e)}^{v_x\in V} T_{\hat{x}\wedge e}^{s,t} \right)$$

Equations

A partial equation $c \Rightarrow x = f(y)$ synchronizes x with the value of $f(y)$ at the clock c. But x may also be present when either c or y is absent. Therefore, the automaton requires x to be emitted with the value $f(v_y)$ only after the events y and c have occurred. If at least one of either c or y is present, then x may or may not be present with some value u computed by another partial equation. The semantics (combinatorially) generalizes to the case of $c \Rightarrow x = f(y_{1..n})$ with $n \geq 0$.

$$\mathcal{A}[\![c \Rightarrow x = f(y)]\!] \stackrel{\text{def}}{=} s^0, \{s^{0..1}, s_{v_y}^{2..4} \mid v_y \in V\}, \{x,y\} \cup \text{vars}(d), \tau, \bigcup_{v_x,v_y\in V}^{v_z\in V \mid z\in\text{vars}(c)}$$

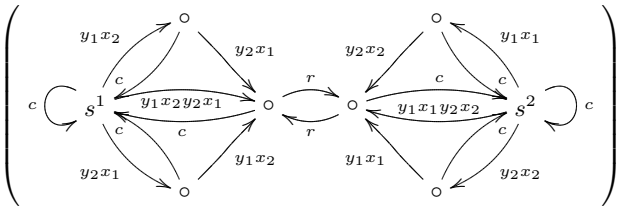


Structuring constructs

Composition $p \,|\, q$ *and restriction* p/x are defined by structural induction starting from the previous axioms with

$$\mathcal{A}[\![p \,|\, q]\!] \stackrel{\text{def}}{=} \mathcal{A}[\![p]\!] \,|^c\, \mathcal{A}[\![q]\!] \qquad \mathcal{A}[\![p/x]\!] \stackrel{\text{def}}{=} (\mathcal{A}[\![p]\!])/x$$

EXAMPLE 1. *Consider the transition system for the* switch *process (the notation* $y_j x_i$ *stands for two steps* $\circ \to^{?y_j=v} \circ \to^{!x_i=v} \circ$*). The switch automaton consists of two mirrored structures that allow for concurrently receiving* y_1 *and* y_2 *and transmitting them along* x_1 *or* x_2 *according to the mode* s_1 *or* s_2*, toggled using the signal* r*.*



6.3 Operational semantics of mode automata

The operational semantics of a mode automaton is described using one equation, Fig. 7, to define the micro-step automaton $\mathcal{A}[\![a]\!]$ corresponding to the mode declaration a. To this end, a mode automaton a is considered as a set of synchronously composed modes and transitions. Hence, we write $(s : p) \in a$ and $(c \Rightarrow s \to t) \in a$ for the modes and transitions it contains.

The semantics of a mode automaton a consists of a transition system that is the union of the transition systems of all modes $(s : p) \in a$. The transition system of a mode $(s : p)$ consists of T_p (that of the process p) where s_p (the initial state) is substituted by s (the mode state). For all mode transitions $c \Rightarrow s \to t \in a$, the transition system is completed with the transitions from the final states u of T_p to the mode state t.

We write $\text{init}\,(T)$ and $\text{final}\,(T) = \{t \mid t \to^\tau s \in T\}$ for the initial and final states of T (the sources and sinks of clock transitions τ in T) and $S_a = \{s \mid (s : p) \in a\}$ for the states of a. As usual, s_a denotes the initial state of a and, referring to automaton $\mathcal{A}[\![p]\!]$ of a process p, s_p its initial state, S_p its states, X_p its variables and T_p its transition system.

EXAMPLE 2. *In the case of the switch, this amounts to superimposing two transitions of condition* r *to the transition systems of the flip and flop modes.*

$$T_{\text{switch}} = \left(T_{\text{flip}} \circ \underset{r}{\overset{r}{\rightleftarrows}} \circ T_{\text{flop}} \right)$$

7. CONCLUSIONS

We have presented a model of multi-clocked mode automata defined by extending the meta-model of the synchronous data-flow specification formalism SIGNAL in the tool GME. A salient feature of our presentation is the simplicity incurred by the separation of concerns between data-flow (that expresses structure) and control-flow (that expresses a timing model) that is characteristic to the design methodology of SIGNAL.

From a user point of view, this simplicity translates into the ease of hierarchically and modularly combining data-flow blocks and imperative modes and significantly accelerates specification by making its structure closer to design intuitions. an example is the 28 lines long encoding of state transitions in the crossbar switch, Fig 6, as opposed to its 4 lines specification, end of Section 3. The same remark applies and scales to the more realistic on-flight example, Fig 2(b), by simplifying the specification of the mode transitions using implicit states.

While the specification of mode automata in related works requires a primary address on the semantics and on compilation of control, the use of SIGNAL as a foundation allows to transfer this specific issue to its analysis and code generation engine POLYCHRONY. Furthermore, it exposes the semantics and transformation of mode automata in a much simpler way by making use of clearly separated concerns expressed by guarded commands (data-flow relations) and by clock equations (control-flow relations).

8. REFERENCES

[1] C. André. Representation and analysis of reactive behaviors: a synchronous approach. In Computational Engineering in Systems Applications. IEEE, 1996.

[2] K. Bender, M. Broy, I. Peter, A. Pretschner, T. Stauner. Model-based development of hybrid systems: specification, simulation, test case generation. In *Modelling, Analysis, and Design of Hybrid Systems*. Lectures notes in computer science, 279. Springer, 2002

[3] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. Science of Computer Programming, v. 19(2). Elsevier, 1992.

[4] L. Besnard, T. Gautier, and P. Le Guernic. SIGNAL V4: reference manual. `http://www.irisa.fr/espresso/Polychrony/doc/document/V4_def.pdf`

[5] J. Bézivin, C. Brunette, R. Chevrel, F. Jouault, and I. Kurtev Bridging the Generic Modeling Environment (GME) and the Eclipse Modeling Framework (EMF) 4th Workshop on Best Practices for Model Driven Software Development, OOPSLA, San Diego, 2005.

[6] C. Brunette, J.-P. Talpin, L. Besnard, and T. Gautier. Modeling multi-clocked data-flow programs in the Generic Modeling Environment. Synchronous Languages, Applications, and Programming (SLAP'06). Elsevier, 2006.

[7] J. T. Buck, S. Ha, E. A. Lee and D. G. Messerschmitt. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. In International Journal of Computer Simulation, special issue on Simulation Software Development. v. 4, pp. 155-182. Ablex, 1994.

[8] X. Liu, J. Liu, J. Eker, and E. A. Lee. Heterogeneous Modeling and Design of Control Systems. In Software-Enabled Control: Information Technology for Dynamical Systems. IEEE Press, 2002.

[9] J.-L. Colaço, A. Girault, G. Hamon, and M. Pouzet. Towards a higher-order synchronous dataflow language. In Embedded Software Conference, Springer Verlag lectures notes in computer science, 2004.

[10] J.-L. Colaço, B. Pagano, and M. Pouzet. A conservative extension of synchronous data-flow with state machines. In Embedded Software Conference. ACM Press, 2005.

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. IEEE, vol.79(9), pages 1305-1320. Septembre, 1991.

[12] D. Harel. Statecharts: a visual approach to complex systems. Science of Computer Programming. Elsevier, 1987.

[13] ISIS, Vanderbilt Uni. GME User Man. `www.isis.vanderbilt.edu/Projects/gme`.

[14] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The Generic Modeling Environment. *Workshop on Intelligent Signal Processing*. IEEE, May 2001.

[15] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. POLYCHRONY for system design. *Journal of Circuits, Systems and Computers*. World Scientific, 2003.

[16] F. Maraninchi, and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. Science of Computer Programming, v. 46(3). Elsevier, 2003.

[17] H. Marchand, E. Rutten, M. Le Borgne, M. Samaan. Formal verification of programs specified with SIGNAL : application to a power transformer station controller. Science of Computer Programming, v. 41(1). Elsevier, 2001.

[18] The Mathworks. Matlab's Simulink and Stateflow. `http://www.mathworks.com`

[19] D. Potop-Butucaru and B. Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specification. In *Application of Concurrency to System Design*. IEEE Press, 2005.

[20] D. Potop, and B. Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. In Applications of Concurrency to System Design. IEEE Press, 2005.

[21] E. Rutten, F. Martinez. Signal GTI: implementing task preemption and time intervals in the synchronous data flow language Signal. In Euromicro Workshop on Real-Time Systems. IEEE Press, 1995.

[22] J.-P. Talpin, D. Potop-Butucaru, J. Ouy, B. Caillaud. From multi-clocked synchronous processes to latency-insensitive modules. International Conference on Embedded Software. ACM Press, 2005.

[23] Vernadat, F., Percebois, C., Farail, P., Vingerhoeds, R., Rossignol, A., Talpin, J.-P., and Chemouil, D. The Topcased project - a toolkit in open-source for critical application and system development. International Space System Engineering Conference - Data Systems in Aerospace. Eurospace, 2006.