A metamodel for the design of polychronous systems*

Christian Brunette[1] Jean-Pierre Talpin[1] Abdoulaye Gamatié[2] Thierry Gautier[1]
[1]INRIA, Unité de Recherche Rennes-Bretagne-Atlantique
Campus de Beaulieu, 35042 Rennes Cedex, France
[2]LIFL - UMR CNRS 8022
Bâtiment M3, 59655 Villeneuve d'Ascq Cedex, France

Abstract

This article presents the development of a metamodel and an open-source design environment for the synchronous language SIGNAL in the GME and Eclipse frameworks. This environment is intended to be used as a pivot modeling tool for a customized, aspect-oriented and application-driven, computer-aided engineering of embedded systems starting from multiple and heterogeneous initial specifications. The metamodel, called SIGNALMETA, is defined on top of the design workbench POLYCHRONY, which is dedicated to SIGNAL programming. Automated transformations are defined and implemented in order to produce, analyze, statically verify and model-check programs obtained from high-level models.

The proposed approach promotes model-driven engineering within a framework that strongly favors formal validation. It aims at significantly decreasing design costs while improving the quality of systems. We demonstrate the agility of this approach by considering the design of both control-oriented and avionic systems. We start with an implementation of core polychronous[1] data-flow concepts in GME and show the ease of its modular extension with application-specific concepts such as mode automata or integrated modular avionics concepts. This work is the first attempt to generalize the formal model of computation and the design philosophy of POLYCHRONY.

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.

*This work is partly funded by the RNTL project OpenEmbeDD and the ANR project TopCased

[1]From the Greek words *poly* and *chronos*: multiple and time

In this spirit, synchronous data-flow programming languages, such as Lustre [19], Lucid Synchrone [15] and SIGNAL [26], implement a model of computation in which time is abstracted by symbolic synchronization. The synchronous paradigm provides a notion of *deterministic concurrency* which facilitates the functional modeling and analysis of embedded systems. 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 [8], Statecharts [20], or SyncCharts [4].

1.1 Polychrony

While modeling a synchronous process or module can be easy, implementing a concurrent system by composing synchronous modular specifications is often hardened by the need of preserving global synchronizations in the model of the system. These synchronization artifacts need most of the time to be preserved, at least in part, in order to ensure functional correctness when, for instance, the presence or absence of an event can influence the behavior of the whole system.

This is why, in the particular case of the POLYCHRONY workbench, on which SIGNAL is based, time is represented by a partial order that expresses a symbolic abstraction of synchronization and scheduling relations on variables or signals. This model of computation offers a unique means to model high-level abstractions of *multi-clocked systems* in which each component owns a local activation clock (e.g., distributed real-time systems, systems on chip). We call this abstract model of time *polychrony*. Polychrony 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 of time.

Fig 1 abstractly depicts the design philosophy behind polychrony. The idea is to give a high-level description of a system, Fig. 1(b), that is abstract enough to enable a subsequent choice between a multi-clocked or a GALS (globally asynchronous locally synchronous) implementation, Fig. 1(a), or a centralized implementation, Fig. 1(c).

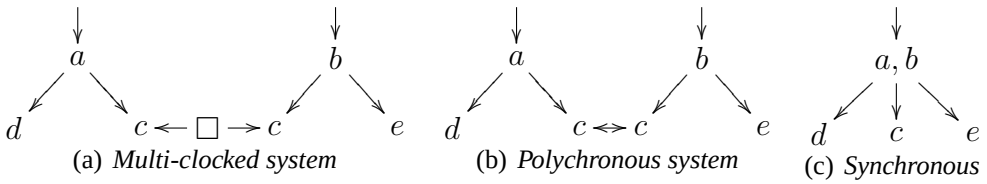


Figure 1: From synchrony to asynchrony.

A polychronous system, depicted in Fig. 1(b), consists of two concurrent sub-systems that receive control from signals a and b and conditionally communicate via a signal c or do something else (d or e). The system can be refined into a synchronous system, Fig. 1(c), by synchronizing the signals a and b. It can also be refined into a multi-clocked system, Fig. 1(a), by distributing a and b on two different locations and by implementing c by a FIFO communication channel.

1.2 Goal

The objective of the present article is to propose a metamodel that suits the polychronous model of computation by defining an open-source, modular and extensible platform targeting the model-driven engineering of embedded software. More specifically, our approach aims at bringing the tool POLYCHRONY in the context of model-driven engineering environments such as GME (Generic Modeling Environment [42]), Eclipse or UML in order to provide designers with better ergonomy and higher-level design abstraction facilities. Along the path to this goal, this article introduces an important milestone: the definition of a metamodel of POLYCHRONY referred to as SIGNAL-META. This metamodel allows a conceptually unified engineering of safety-critical embedded systems within GME. Taking advantage of this definition, we first extend SIGNALMETA with an inherited metamodel of polychronous mode automata suitable for control-oriented design. Then, we illustrate a customization of the resulting extended metamodel for the design of specific integrated modular avionic (IMA) systems.

1.3 Domain-Specific Concepts

Integrated Modular Avionics (IMA) [2] are the recent architectures proposed for avionic systems. In such architectures, several functions, possibly of different criticality levels, can share common computing resources. IMA has been defined as an alternative solution to federated architectures in which each function executes exclusively on its dedicated computer system. While federated architectures favor fault containment, they are very expensive because of high maintenance costs, huge power consumption, etc.

In IMA, error propagation is addressed by the partitioning of resources with respect to available time and memory capacities. A partition is a logical allocation unit resulting from a functional decomposition of the system. IMA platforms consist of modules grouped in cabinets throughout the aircraft. A module can contain several partitions that possibly belong to applications of different levels of criticality. Mechanisms are provided in order to prevent a partition from having "abnormal" access to the memory area of another partition. The processor is allocated to each partition for a fixed time window within a major time frame maintained by the module level operating system (OS). A partition cannot be distributed over multiple processors either in the same module or in different modules. Finally, partitions communicate asynchronously via logical ports and channels. Message exchanges rely on two transfer modes: sampling and queuing.

Partitions are composed of processes that represent the executive units. Processes run concurrently and execute functions associated with the partition in which they are contained. Each process is uniquely characterized by information, such as its period, priority, or deadline time, used by the partition level OS, which is responsible for the correct execution of processes within a partition. The scheduling policy for processes is priority preemptive. Communications between processes are achieved by three basic mechanisms. The bounded buffer allows to send and receive messages following a FIFO policy. The event permits the application to notify processes of the occurrence of a condition for which they may be waiting. The blackboard is used to display and read messages: no message queues are allowed, and any message written on a blackboard remains there until the message is either cleared or overwritten by a new instance of the message.

The APEX-ARINC 653 standard [1] defines an interface allowing IMA applications to access their underlying OS functionalities. This interface includes services for communication between partitions on the one hand and between processes on the other hand. It also provides services for process synchronization, and finally, partitions, processes, and time management services.

1.4 Example - Integrated Modular Avionics

The modeling of integrated modular avionics (IMA) architectures [2] is a typical case in which the multi-clocked synchronous model of computation and mixed data-flow/control-flow formalisms such as mode automata, are particularly well-suited for compositional modeling. This fact is shown in this paper by considering a real world application, called CMF, from the avionic domain. The role of the CMF application is to gather maintenance information received from an aircraft during the flight and treat this information. More precisely, its main functionalities can be summarized as follows:

- it calculates and sends to other partitions some general parameters as well as information corresponding to the maintenance phase (this is done cyclically);
- it gathers and treats maintenance messages received from other partitions during the flight phase;
- it indirectly communicates with a peripheral in order to allow an operator to visualize the computed maintenance information;
- it periodically checks the availability of maintenance data, and emits a report message.

The design of the CMF application consists of several processes that are grouped within a single IMA partition as shown in FIG. 2. This choice has been decided by our industrial partner. The CMF partition cooperates with other partitions within an IMA module, which is not represented here. The message exchanges that occur between the different partitions within this module are achieved via specific ports. The module itself is linked via an Ethernet bus to the peripheral that provides a Human-Machine Interface for the CMF application. In FIG. 2, we can distinguish the following elements that must be modeled:

- processes (executable units): AB_BITE, ACQ_GBx, MSG_SURV, MAT_SURV, MEMORIZATION, CYCLIC, DIAL_MAT, and CORRELATION;
- communication mechanisms: five buffers identified by Correlation, Bdm, Dial_mat, Bite, and Fault_msg; three events identified by Dial_correlation, Bdm_correlation, and Flight_trans;
- a synchronization mechanism: a semaphore Msg_sema allowing MSG_SURV and CORRELATION to get access to the shared resource RE_msg_table in mutual exclusion.

Section 5.2 shows, via the modeling of the CMF partition, how the multi-clocked mode automata extension of SIGNALMETA is customized in order to design avionic applications based on the IMA philosophy. From the resulting models, a corresponding SIGNAL program is automatically generated for formal validation with POLYCHRONY.

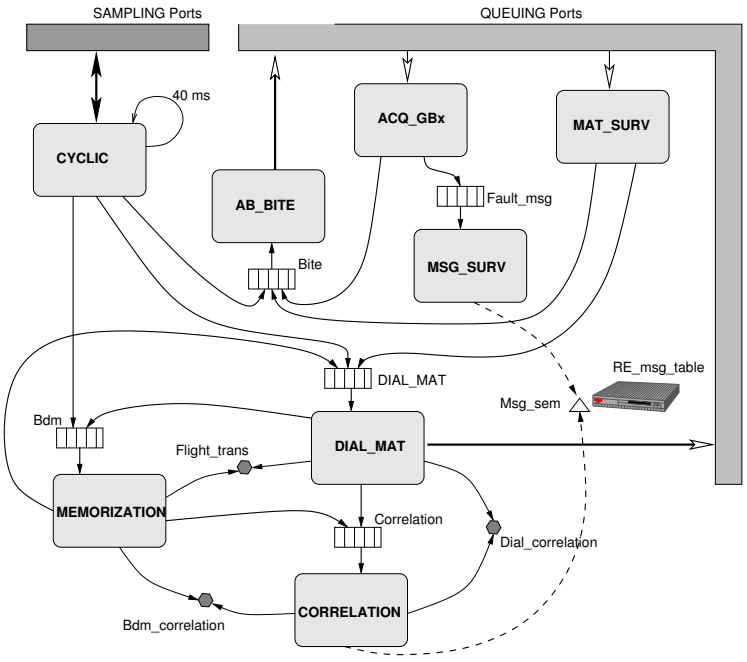


Figure 2: A typical avionic application.

1.5 Rationale

From the above observations, the SIGNALMETA metamodel proposed in this article aims to leverage current design practices in embedded system and model-driven engineering domains, with the benefits of the domain-specific technology of POLYCHRONY: a semantic model that enables a modular design approach for polychronous systems, useful formal validation techniques and tools, and automatic code generation towards different target languages (e.g. C, Java, C++) and architectures (monoprocessor and multiprocessor).

To achieve all this, substantial efforts are undertaken to equip SIGNALMETA with the required programmatic features. A working implementation is provided within the Generic Modeling Environment to make the use of the proposed approach very pragmatic. This is very important in order to encourage application engineers to adopt formal methods-based frameworks such as POLYCHRONY.

1.6 Outline of the paper

The remainder of the article is organized as follows. Section 2 introduces the basic features of the synchronous language SIGNAL. Then, Section 3 presents the GME environment, which is used to define SIGNALMETA, the metamodel associated with POLYCHRONY. In Sections 4 and 5, we extend SIGNALMETA with multi-clocked mode automata and of IMA concepts to use it for both control-oriented and avionic system designs. Section 6 presents the migration of SIGNALMETA from the GME environment to Eclipse. Discussions and related works are mentioned in Section 7 and ongoing and future work in Section 8. Finally, Section 9 gives concluding remarks.

2 The synchronous language SIGNAL

This section first introduces the syntax of SIGNAL, a declarative synchronous data-flow language, Section 2.1. Section 2.2 illustrates it by considering the specification of a simple counter. Section 2.3 presents the data-structure of multi-clocked data-flow graphs used for analysis and transformation of SIGNAL programs. Section 2.4 shows how to infer this information and Section 2.5 how to use it for code generation. Finally, a brief description of the POLYCHRONY workbench is given, Section 2.6. The appendix gives a denotational semantics of Signal in the polychronous model of computation [26].

2.1 Abstract syntax

A signal x consists of an infinite flow of typed values (e.g. Boolean, integer, real) that is discretely sampled according to the pace of its clock, denoted by $\hat{x}$. An equation, denoted by $x = y\, f\, z$, relates the values and clocks of signals x, y, z. The synchronous composition $p \| q$ of processes p and q consists of simultaneously considering a solution of the equations in p and q at all times. The whole abstract syntax[2] can be summarized as follows:

$$p ::= x = y\, f\, z \mid p \| q \mid p/x.$$

SIGNAL defines four primitive kinds of equations, that are generically denoted by the abstract syntax $x = y\, f\, z$:

- A functional equation $x = y\, f\, z$ defines an arithmetic or Boolean relation f between its operands y, z, and a result x.
- A delay equation $x = y \,\mathsf{pre}\, v$ 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 (then, they hold some values) or simultaneously absent (then, they hold no value) at all times.
- A sampling $x = y \,\mathsf{when}\, z$ 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 $x = y \,\mathsf{default}\, z$ defines x by y when y is present and by z otherwise. In a merge equation, the output signal x is present iff either of the input signals y or z is present.
- Hiding p/x limits the lexical scope of the signal named x to the process p.

We notice that neither of the sampling and merge combinator synchronize the input and output signals x, y, z. We call them polychronous operators. They simply define partially ordered clocks in the aim of preserving the natural synchronization and scheduling structure of the intended specification, and let analysis and code generation algorithms schedule and compile these

[2] We shall distinguish the composition operator $\|$ from the symbol $|$ which expresses alternative in syntactic rules.

specification according to the target architecture specification. This feature differentiates SIGNAL from related synchronous programming languages as a higher-level, more abstract, specification formalism to compositionally model an embedded system starting from partial equations and in a constraint-based manner.

2.2 Example - From specification to implementation

This claim is best illustrated by the specification of a simple counter. One might wish to define it in a (slightly exaggerated) modular or compositional manner by viewing it as the composition of two separate equations. The count is defined by the value stored in the variable val which is defined to be either 0 if the counter is reset or by an increment of the count. Notice that the clock of the process is defined by that of the output count, and that it is not related to that of the reset.

$$\mathsf{count} = \mathsf{val}\ \mathsf{pre}\ 0 \,|\, \mathsf{val} = 0\ \mathsf{when}\ \mathsf{reset}\ \mathsf{default}\ \mathsf{count} + 1$$

To generate deterministic sequential code starting from this specification, one needs to synthesize a a timing model in which every calculation is paced by a clock tick. This can be done by composing the counter with the specification of this structure: $\hat{\mathsf{count}} = \hat{\mathsf{reset}} = \hat{\mathsf{tick}}$.

2.3 Clock and scheduling relations

SIGNAL supports a representation of the control-flow and the data-flow of multi-clocked specifications for the purpose of analysis, transformation, and code generation. In this structure, Fig 3, a clock c denotes a set of logical instants that defines a discrete sample of time. Here, a clock is not necessarily a set of points in real-time. A logical instant is rather assimilated as a reaction in the execution of a reactive system [21].

Such a reaction may consist, for example, in reading inputs, processing them and writing outputs. These actions are therefore simultaneous from the viewpoint of the logical instant associated with the reaction in which they take place. The signals involved in the actions are said to be present at the corresponding logical instant. From these observations, a clock can be seen as a set of reactions. In SIGNAL, a data-flow relation is only executed at instants where the involved signals are present.

The clock $\hat{x}$ of a signal x denotes the set of instants at which the signal x is present. In the specific case of Boolean signals x, the clock $[x]$ (resp. $[\neg x]$) denotes the set of instants at which x is present and holds the value true (resp. false). Because clocks are sets, the usual operations on sets also apply to clocks. For instance, the clock union of two signals x and y consists of the set of instants at which either x or y is present.

In the same way, clock intersection and difference can be defined as usually for sets. All these operations are provided in SIGNAL. A clock expression e is a Boolean property. It consists of the empty clock, denoted by 0 to mean the empty set of instants, of signal clocks c, and of the conjunction, disjunction, and complement of clock expressions $e_1 \wedge e_2$ and $e_1 \vee e_2$, and $e_1 \setminus e_2$.

SIGNAL programs are implicitly or explicitly related by synchronization and scheduling relations, denoted by g. A scheduling relation $a \rightarrow^c b$ specifies that, at instants of the clock c, the

calculation of the node b, which is either a signal or a clock, is scheduled after that of the node a. A clock relation $c = e$ specifies that the clock c is present iff the clock expression e is true. The set of scheduling relations associated with a program forms a graph. Such graphs g and h are subject to the same composition $g \| h$ and scoping g/x as processes:

$$\begin{array}{rcll} c & ::= & \hat{x} \,|\, [x] \,|\, [\neg x] & \text{(signal clock)} \\ e & ::= & 0 \,|\, c \,|\, e_1 \wedge e_2 \,|\, e_1 \vee e_2 \,|\, e_1 \setminus e_2 & \text{(clock expression)} \\ a, b & ::= & x \,|\, \hat{x} & \text{(graph node)} \\ g, h & ::= & c = e \,|\, a \rightarrow^{c} b \,|\, (g \| h) \,|\, g/x & \text{(scheduling graph)} \end{array}$$

Figure 3: A syntax for clock relations and scheduling graphs.

2.4 Clock inference

Any SIGNAL process p corresponds to a system of implicit clock and scheduling relations g that denote its implied timing and scheduling structure, Fig. 4. It is specified by the inference system $p : g$. It can be interpreted as a logical relation saying that process p has graph g. The rules for composition $p \| q$ and restriction p/x are defined by induction on the deductions $p : g$ and $q : h$ made on the sub-terms of the expressions.

In a delay equation $x = y \,\mathsf{pre}\, v$, the input and output signals are synchronous, written $\hat{x} = \hat{y}$, and do not have any scheduling relation. In a merge equation $x = y \,\mathsf{default}\, z$ the output signal x is present if either of the input signals y, z are (i.e., $\hat{x}$ is equal to the union of $\hat{y}$ and $\hat{z}$). Since the merge operation is deterministic, y is assumed in SIGNAL to have priority on z when both are present. For this reason, the input y is scheduled before the output x when it is present, written $y \rightarrow^{\hat{y}} x$, and otherwise z is, written $z \rightarrow^{\hat{z}\setminus\hat{y}} x$.

In a sampling equation $x = y \,\mathsf{when}\, z$, the clock of the output signal x is defined by that of the input signal y at the sampling condition $[z]$ (i.e., $\hat{x}$ is equal to the intersection of $\hat{y}$ and $[z]$). The input y is scheduled before the output x when both $\hat{y}$ and $[z]$ hold, written $y \rightarrow^{\hat{x}} x$. A functional equation $x = y \, f \, z$ synchronizes and serializes its input and output signals.

$$\frac{p : g \quad q : h}{p \| q : g \| h} \qquad \frac{p : g}{p/x : g/x}$$

$$\begin{array}{rcl} x = y \,\mathsf{pre}\, v & : & (\hat{x} = \hat{y}) \\ x = y \,\mathsf{default}\, z & : & (\hat{x} = \hat{y} \vee \hat{z} \,\|\, y \rightarrow^{\hat{y}} x \,\|\, z \rightarrow^{\hat{z}\setminus\hat{y}} x) \\ x = y \,\mathsf{when}\, z & : & (\hat{x} = \hat{y} \wedge [z] \,\|\, y \rightarrow^{\hat{x}} x) \\ x = y \, f \, z & : & (\hat{x} = \hat{y} = \hat{z} \,\|\, y \rightarrow^{\hat{x}} x \,\|\, z \rightarrow^{\hat{x}} x) \end{array}$$

Figure 4: Inference of clock and scheduling relations.

2.5 Example - Compiling a buffering protocol

To outline the sequential code generation process in SIGNAL, we consider the specification and analysis of a one-place buffer that is very similar in purpose and behavior as the FIFO buffers of

the CMF application in Fig. 2. Our buffer uses two functionalities, alternate and current:

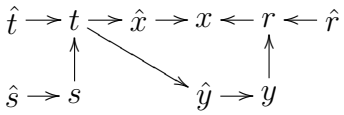
$$x = \mathsf{buffer}(y) \stackrel{\text{def}}{=} \left(\; x = \mathsf{current}(y) \,|\, \mathsf{alternate}(x, y) \; \right)$$

The process alternate synchronizes the signals x and y to the true and false values of an alternating Boolean signal t. The process current stores the values of an input signal y and sends them along the output signal x upon request:

$$\mathsf{alternate}(x, y) \stackrel{\text{def}}{=} \left(\begin{array}{l} s = t \,\mathsf{pre}\, \mathsf{true} \\ \mid\; t = \,\mathsf{not}\, s \\ \mid\; \hat{x} = [t] \\ \mid\; \hat{y} = [\neg t] \end{array} \right) /st \qquad x = \mathsf{current}(y) \stackrel{\text{def}}{=} \left(\begin{array}{l} r = y \,\mathsf{default}\, u \\ \mid\; u = (r \,\mathsf{pre}\, \mathsf{false}\,) \\ \mid\; x = r \,\mathsf{when}\, \hat{x} \\ \mid\; \hat{r} = \hat{x} \vee \hat{y} \end{array} \right) /ru$$

The process buffer has three clock equivalence classes, each represented by equalities between clocks. The clocks $\hat{s}$ and $\hat{t}$ are synchronous and define the master clock equivalence class of buffer. Also note that $\hat{r} = \hat{x} \vee \hat{y} = \hat{t}$. The two other equivalence classes, $\hat{x}$ and $\hat{y}$, correspond to the true and false values of the Boolean flip-flop variable t (the intermediate signal u is discarded). Program analysis defines scheduling relations between the input and output signals of process buffer. Notice that t is needed to compute $\hat{x}, \hat{y}$:

$$\begin{array}{rcl} \hat{r} & = & \hat{s} = \hat{t} = \hat{u} = \hat{x} \vee \hat{y} \\ \hat{x} & = & [t] \\ \hat{y} & = & [\neg t] \end{array}$$



In the SIGNAL compiler, the above clock equivalence classes define three nodes in the control-flow graph of the generated code. This code describes a cyclic execution in which every cycle inputs are read and processed, and outputs are produced. For illustration, let us consider the following sketch of code associated with process buffer, generated in C, which corresponds to what is executed every cycle on a monoprocessor architecture:

Here, there has been a code expansion of processes alternate and current. So, they are no longer explicitly visible. At each execution cycle, `t` is calculated based on `s`, which has been initialized outside the scope of `the buffer_iterate` block. At the sub-clock `t=0`, the value of signal `y` is read from the input interface (using function `r_buffer_y`). At the sub-clock `t=1` the value of signal `x` is written on output interface (using function `w_buffer_x`). Finally, `s` is updated using `t`. The sequence of instructions follows the scheduling relations determined during clock inference.

```
buffer_iterate () {
 t = !s;
 if (!t && !r_buffer_y(&y))
   { return FALSE; }
 if (t)
   { w_buffer_x(x = y); }
 s = t;
 return TRUE;
}
```

2.6 The POLYCHRONY workbench

The POLYCHRONY workbench [43] is the integrated development environment supporting the data-flow notation SIGNAL. It offers several tools including the SIGNAL batch compiler that provides a set of functionalities, such as program transformations, optimizations, formal verification, and code generation. It includes the SIGALI model checker that enables both verification and controller synthesis [29].

In POLYCHRONY, the design approach 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, additional timing relations are provided to gradually refine the synchronization and scheduling structure of the system. Finally, the correctness of the 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 provide a functionally correct deployment on the target architecture.

3 The SIGNAL metamodel

This section presents the definition of the SIGNAL metamodel, SIGNALMETA [13], using the Generic Modeling Environment (GME). In the following, Section 3.1 gives a brief overview of the GME. Then, Section 3.2.1 introduces the basic concepts of SIGNALMETA. Then, Section 3.2.2 presents the aspects associated with the metamodel.

3.1 The Generic Modeling Environment

GME is a configurable UML-based toolkit that supports the creation of domain-specific modeling and program synthesis environments [23] [42]. It proposes a metamodeling framework that supports the definition of *modeling paradigms*, which are type systems that describe the roles and relationships between elements in a particular domain. It also includes all relationships between those concepts, their organization, and all rules governing the construction of models. In the following, note that the notation of GME-specific concepts begins with an uppercase.

3.1.1 Main features

Within GME, a new modeling paradigm has to be described using the MetaGME paradigm, available in the environment. The modeling paradigm is specified in terms of usual UML class diagrams, which are built with some predefined stereotypes of MetaGME [23]: Atom, Model, Set, Reference, and Connection. These stereotypes are all first class objects (or FCOs for short). Atoms are elementary objects in the sense that they cannot contain parts. Models are compound objects that can have parts and inner structure. Sets are used to specify a relationship among a group of objects that belong to the same Model. References are similar to pointers in object-oriented programming languages.

There are different kinds of relationships, which can be expressed between classes that use the above stereotypes. Containment relation is characterized on the class diagram by a link ending with a diamond on the container side. Such a link is used in FIG. 5 for example between the *Input* atom and the *InterfaceDefinition* Model. Inheritance relations are represented as in UML. All the other relationships are specified by classes that use a Connection stereotype.

In FIG. 5, some FCOs use a stereotype suffixed by "Proxy" (e.g. Module). These stereotypes are references to other FCOs declared in different paradigm *sheets* within the metamodel. Sheets allow one to define in a modular way complex metamodels by describing separately their different parts. Attributes can be added to classes. They are associated with various types: EnumAttribute (a finite list of choices), FieldAttribute (a typed text field, e.g., string, integer, or double), and BooleanAttribute. GME provides a means to express the visibility of FCOs within a model through the notion of Aspect (i.e. one can decide which parts of the descriptions are visible depending on their associated aspects).

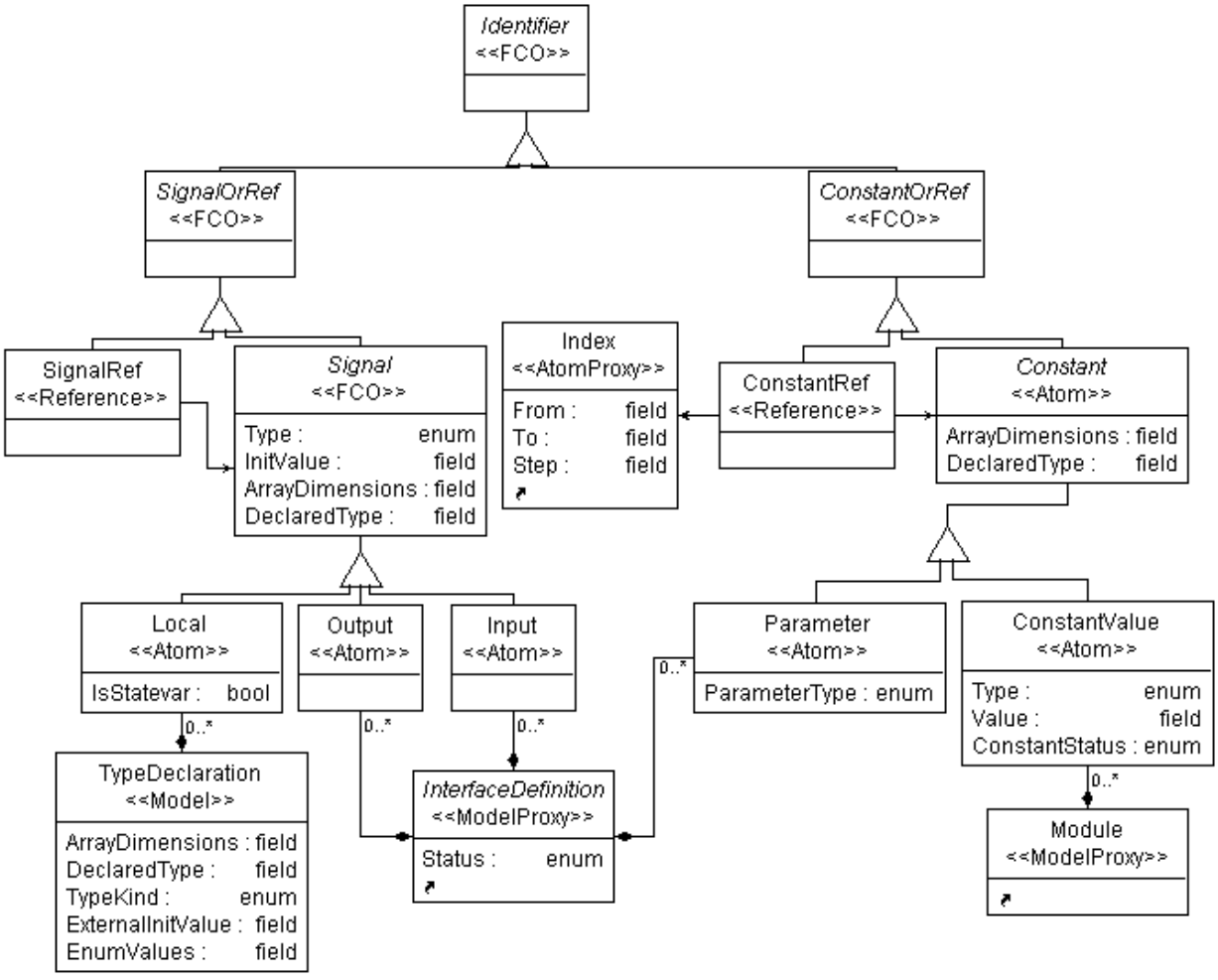


Figure 5: SIGNALMETA's Identifier class diagram

Finally, OCL constraints can be added to class diagrams in order to check *parametric* proper-

ties, which depend on the values given during the edition of a model (e.g. some attribute values may restrict the number of possible connections to an FCO). The specified constraints are associated with events. Whenever these events occur, their corresponding constraints are checked. Typical actions achieved during model edition such as creation, connection, or value modification are all associated with such events.

3.1.2 The MetaGME Interpreter

The above features represent the basic building blocks in GME, which are used to define modeling paradigms. Every paradigm is associated with a paradigm file, produced automatically. GME uses this file to configure its environment for the creation of models using a new paradigm. This is achieved by the MetaGME *Interpreter*, a plug-in accessible via the GME Graphical User Interface (GUI). The MetaGME interpreter first checks the correctness of the metamodel, then generates the paradigm file, which is saved in GME.

More generally, model interpreters are high-level code associated with a given modeling paradigm, used to translate information found in the graphical models into specific formats (executable code, data streams, etc.). Similarly to the MetaGME Interpreter, other Interpreters can be developed and plugged into the GME environment. The role of such an Interpreter consists in interacting with the graphical designs. To achieve the connection between the Interpreter and GME, an executable module is provided with the GME distribution, which enables the generation of the Interpreter skeleton. It can be generated in C/C++ or JAVA. In C++, the skeleton is written using the low-level COM language or the "Builder Object Network" (BON) API [23].

3.2 Modeling SIGNAL in GME

SIGNALMETA, the metamodel of the SIGNAL language in GME, describes all the syntactic elements defined in SIGNALv4 [10]. It is composed of several paradigm sheets that define relations between signals, operators, and process frame.

3.2.1 Basic notions

The paradigm sheet associated with *Identifiers*, represented in FIG. 5, defines Atoms for the different kinds of signals (*Input*, *Output*, and *Local*), and constants (*ConstantValue* and *Parameter*). All these Atoms have several attributes including their types, which are an enumeration of all intrinsic types of SIGNAL.

The *DeclaredType* attribute may represent a type imported from a SIGNAL library or a type declared in GME. The declaration of a new type is done via the *TypeDeclaration* Model. The associated attribute *TypeKind* indicates if this type is an enumeration, a structure or a process model. Note that a type declaration depends on its nature. For instance, a structure type is specified by adding an ordered list of *Local* Atoms in the *TypeDeclaration* Model, while in the case of an enumerated type, all its possible values are specified in the *EnumValues* attribute.

To use in a Model some signals (resp. some constants or indexes of an iteration) declared in an upper-level Model, one can use a *SignalRef* (resp. a *ConstantRef*) with the same name.

Another way for different Model levels to communicate is to use *Input/Output/Parameter* Atoms. These kinds of Atoms are declared as ports in GME. This means that they are visible in the Model where they are added and in the upper-level Model, so that one can connect them to upper Atoms. *Input/Output/Parameter* Atoms can be added in all Models that inherit from the *InterfaceDefinition* abstract Model.

FIG. 6 shows the graphical representation of these Atoms in SIGNALMETA. The second line represents some SIGNAL operators. The main constituent Models of SIGNALMETA are *ModelDeclaration*, *SubProcess*, and *Module*. A *ModelDeclaration* corresponds to a SIGNAL process model, which can be either an action, a function, a node, or a process. This characterization is done via a *ModelDeclaration* attribute. A *ModelDeclaration* consists of a container in which are declared *Input/Output/Local* signals, static *Parameters*, included *ModelDeclaration* and *TypeDeclaration* Models, and in which one can add FCOs corresponding to SIGNAL operators to express relations between signals. Finally, the *Module* Model is a library of *ModelDeclaration*, *TypeDeclaration*, and *ConstantValue* FCOs.

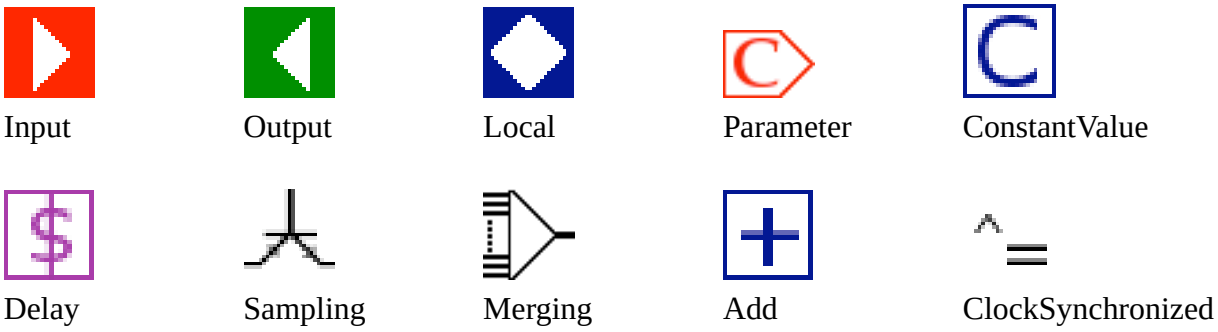


Figure 6: Some concepts of SIGNALMETA and their associated icons.

Another interesting point is the way to represent SIGNAL process model instantiations. GME provides means to create Model instances. Such an instance is a deep copy of the Model in which no FCO can be added or removed, but in which attribute values can be modified. As a result, this can lead to some state incoherence between a Model and its instances. In SIGNALMETA, we consider a new notion of instance by defining a Reference, called *ModelInstance*, which offers exactly the same view as a Model without creating a deep copy (a deep copy results from the duplication of an object along with the objects to which it refers).

FIG. 7 represents all relations between SIGNALMETA concepts, except clock relations (described in another paradigm sheet). Among them, we can highlight *Definition* whose destination is a *Signal* or a *SignalRef* (gathered in the *SignalOrRef* abstract concept - see FIG. 5), and which allows to specify the definition of a signal. For a given signal, such a Connection can be used only once. SIGNAL offers a means, called partial definition, to avoid the syntactic single assignment rule for the definition of a signal, even if semantically, this single assignment rule applies. Similarly, SIGNALMETA offers the *PartialDefinition* Connection to be able to define, in different Models, the different parts of the signal definition.

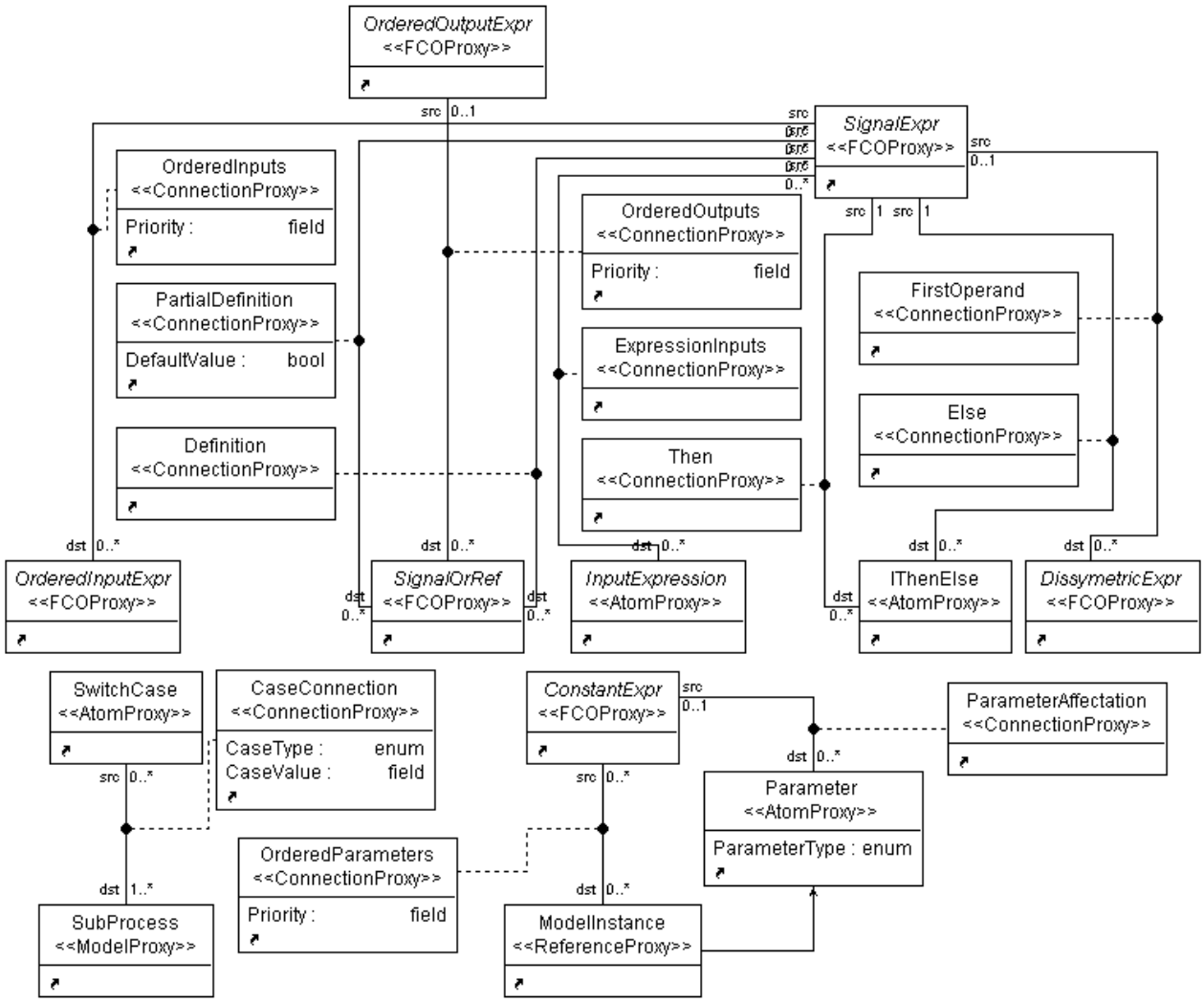


Figure 7: SIGNALMETA's "Expression Connection" paradigm sheet

3.2.2 Aspects

The concepts of SIGNALMETA are organized in four Aspects: *Interface*, *Dataflow*, *ClockAndDependence*, and *Library*. The *Interface* Aspect is dedicated to represent input/output signals of a *ModelDeclaration* and its static parameters. Moreover, *Specifications* can be added to describe clock and dependence relations between these signals. Signals and parameters are ordered according to their position in this Aspect.

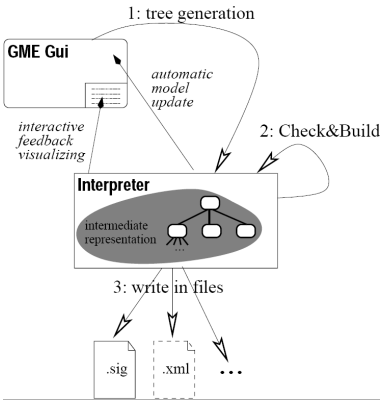
The *Dataflow* Aspect is dedicated to the specification of computations and data flows within a process, whereas the *ClockAndDependence* Aspect contains all clock relations and dependencies between signals. Thus, the latter contains mainly clock constraint and relation operators (e.g. *ClockSynchronized*, *ClockUnion*), the *Dependence* Atom, signals (or references of signals), and all Connections to link them. The *Dataflow* Aspect can contain all other SIGNAL operators.

This separation of concerns makes the modeling more readable [22]. Indeed, Connections in

the *Dataflow* Aspect represent data flows, while they represent only relations in the *Clock Relation* Aspect. However, this separation between the data-flow and the control parts is not so obvious in SIGNAL. Actually, SIGNAL primitives, such as the delay and arithmetic operators, synchronize automatically their inputs and their outputs. Thus, operators in the *Dataflow* Aspect also express part of the control dimension of the process. Finally, a *Library* Aspect is dedicated to SIGNAL modules, which consist of libraries of constants, types, and process model declarations.

3.3 Model Interpretation

Models described with SIGNALMETA may be transformed into corresponding SIGNAL programs so as to use the functionalities of POLYCHRONY. Such a transformation is achieved by a new GME "Interpreter". This Interpreter acts similarly for SIGNALMETA models as the MetaGME Interpreter for MetaGME metamodels.



The different steps of the "interpretation" can be described as follows: (i) creating the tree structure of the SIGNAL programs corresponding to the graphical representation; (ii) generating the SIGNAL equation for each level of this intermediate representation; (iii) and finally generating SIGNAL code.

3.4 Example - Modeling a watchdog protocol

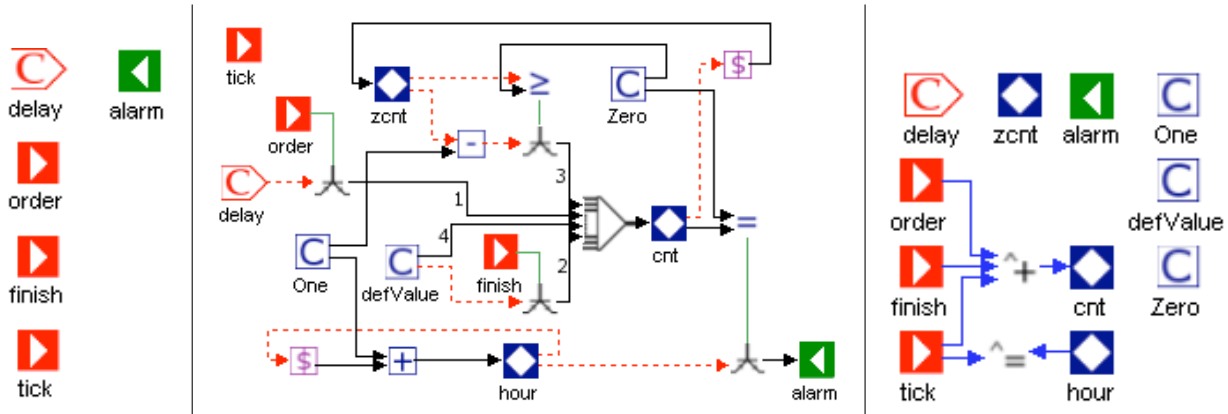
Another typical building block to help modeling the behavior of the CMF application in Fig. 2 is a watchdog protocol. The watchdog is a perfect candidate to exemplify aspect oriented modeling with SIGNALMETA. Its goal is to control that some action is executed within a given `delay`. At all times, the action emits an `order` signal when it begins its execution, and a `finish` event when it terminates. If the job is not finished in time, the watchdog must emit an `alarm` signal to indicate at what time an error occurs. Moreover, if a new `order` occurs while the previous one is not finished, the time counting restarts from zero. A `finish` signal out of delay, or not related to an `order`, is ignored.

The `Watchdog` process can be specified in GME as shown in FIG. 8. FIG. 8(a) represents the *Interface* Aspect in which are described the input/output signals and the static parameters of the process. Thus, one has to drag and drop an *Input* Atom for the `order` and `finish` signals, and an *Output* Atom for the `alarm`.

In order to count the time, another input signal called `tick`, which is provided at regular intervals of instants, is added to the *Interface* Aspect to represent each tick of a clock. Finally, the `delay` to process an order is expressed as a number of `ticks` by a *Parameter* Atom. The types of these signals/parameters are specified in the attributes of the corresponding Atom.

In the Dataflow Aspect (FIG. 8(b)), three local signals are declared: `hour`, `cnt`, and `zcnt`. The `hour` signal represents the internal clock to count the time. The `cnt` signal works as a countdown before emitting an alarm: when `cnt` is 0, the alarm is emitted with the value of `hour`. The value of `cnt` is fixed, by order of priority, to: (i) `delay` when an order is emitted, (ii) `defValue` when

`finish` is emitted, (iii) the previous value of `cnt` contained by `zcnt` decremented by one, or finally to (iv) the constant `defValue`.



(a) *Interface* (b) *Dataflow* (c) *ClockAndDependence*

Figure 8: A SIGNALMETA model of the watchdog process.

This order is fixed using the *Priority* attribute of all incoming Connections on the *Merging* Atom. In FIG. 8(b), dashed arrows connect their source FCO as first operand of the destination FCO, plain links connect a Boolean expression to an *Extraction* Atom, plain arrows whose destination is a *Signal* Atom correspond to a *Definition* of this signal, and finally other plain arrows are specific *Connection*s for each operator (cf. FIG. 7). In the Clock Aspect (FIG. 8(c)), `hour` and `tick` are synchronized using the *ClockSynchronized* Atom. This leads `hour` to be incremented at each `tick`. Moreover, `cnt` has to be present each time one of the input signals is present. This is expressed with the *ClockUnion* Atom whose result is assigned to `cnt`.

```
1.  process Watchdog =
2.     { integer delay; }
3.     ( ? integer order; event finish, tick;
4.       ! integer alarm; )
5.     (| alarm := (hour when (cnt = Zero))
6.      | hour := (One + (hour$(1) init (0) ))
7.      | cnt ^= (order ^+ tick ^+ finish)
8.      | cnt := ((delay when ^order) default (defValue when finish)
9.               default ((zcnt - One) when (zcnt >= Zero))
10.              default defValue)
11.     | zcnt := (cnt$(1) init (-1) )
12.     | tick ^= hour
13.     |)
14.    where
15.      constant integer One = (1), defValue = (-1), Zero = (0);
16.      integer hour, cnt, zcnt;
17.    end; %process Watchdog%
```

Figure 9: Code generated for the watchdog SIGNALMETA model.

It is important to notice that the purpose of SIGNALMETA goes beyond the definition of a graphical user interface. It makes available an expressive formal model and its related tool-suite for the design of multi-clocked embedded systems. This is a main difference with existing popular frameworks in embedded system domain such as Matlab/Simulink [39] or Esterel Studio [45], which also propose attractive environments for the same goals. Furthermore, by adopting a meta-modeling approach, SIGNALMETA adheres more to the model-driven engineering philosophy.

The code of FIG. 9 corresponds to the application of the SIGNALMETA Interpreter (cf. Section 3.3) on the watchdog process described in FIG. 8. Note that, in the concrete SIGNAL syntax, `y$ init v` is the notation for `y pre v`; the statement `x^=y` stands for $\hat{x} = \hat{y}$; the operator `^+` designates the union of clocks. The interpreter generates files using the SIGNAL syntax. However, it is possible to specialize it to construct equations using, for example, XML syntax.

4 Application to control-oriented design

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 [39], which supports the Stateflow notation to describe modes in event-driven and continuous systems. Similarly, Ptolemy [25] allows for the hierarchical and modular specification of finite state machines hosting heterogeneous models of computation. Worth noticing is Hyscharts [6], 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. [28] to gather advantages of declarative and imperative approaches. The authors therefore extended the functionality-oriented data-flow paradigm of Lustre with the capability to model transition systems easily. Similar variants and extensions of the same approach to mix multiple programming paradigms or heterogeneous models of computation [25] [27] have been proposed until recently, the latest advance being the combination of stream functions with automata [14]. Nowadays, commercial tool-sets such as the Esterel Studio's Scade or Matlab/Simulink's Stateflow are largely inspired by similar concepts.

4.1 Principle

We extend SIGNALMETA with polychronous mode automata [35] so as to enable the design of control-oriented applications. To fit it within the polychronous design methodology presented in Section 1.1, we shall propose high-level and abstract modeling or programming means to compositionally model functionalities (the modes) and control (automata) in a system. Therefore, we choose to give automata an implicit clock that is defined by that of its transitions and of its modes.

4.2 Example - a flip-flop switch

To illustrate this paradigm, our first example is once again one of the typical building blocks found in embedded software such as the CMF of Fig. 2. It implements a flip-flop or crossbar switch

controlling the activation of two sub-systems. The mode automaton of the switch, described in Fig. 12, consists of two modes 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 converse, upon an occurrence of r.

To elaborate on our principle, one notices that the activity of the automaton is triggered by several possible events. Transitions are triggered by occurrences of the input event r. In either of the modes flip and flop, two concurrent and independent activities are performed upon each occurrence of the signals y_1 and y_2.

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

Our switch typically has a polychronous behavior. As such, and in the same spirit as in Sec.1.1, we might want to refine and compile it into a sequential program, but we could also deploy it on a distributed architecture (and calculate x_1 and x_2 on two separate locations) to implement a multi-clocked system. Its above specification allows us to perform both implementation choices in an automated and correct manner.

4.3 Formal syntax

We define a formal syntax for mode automata and specify the function that maps objects in this syntax to Signal equations. To express mode automata, we consider four additional elements. First, $\mathsf{init}\, s$ specifies the initial state (mode) of an automaton a and the expression $s : p$ labels the behavior p by the mode s.

A weakly preemptive transition $e \Rightarrow s \rightarrow t$ defines the clock or guard e of the transition from mode s (the present state) to mode t (the next state). By contrast, a strongly preemptive $e \Rightarrow s \twoheadrightarrow t$ immediately transfers control from mode s to mode t upon the condition e (most likely a condition on input signals such as an alarm). Automata a and b naturally compose synchronously with $a \,|\, b$.

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

4.4 Formalization of the transformation

The main technical difficulty is to interpret mode automata by data-flow equations. To tackle it, we use a feature of the Signal compiler enabling multiple signal definitions using partial equations (noted ::= in the example of figure 13). The principle of partial equations is to allow the notation

$$x = y_1 \,\mathsf{when}\, z_1 \,|\, \ldots \,|\, x = y_n \,\mathsf{when}\, z_n$$

to stand for the equation $x = y_1 \,\mathsf{when}\, z_1 \,\mathsf{default}\, \ldots \,\mathsf{default}\, y_n \,\mathsf{when}\, z_n$ provided that the conditions $z_{1..n}$ which guard the definitions $y_{1..n}$ are exclusive, i.e., that $[z_i] \wedge [z_j] = 0$ holds for all $0 < i \neq j \leq n$. We shall also make use of the converse notation $p \,\mathsf{when}\, c$ to condition (all equations in)

a process p by a clock c (that defines its guard). A conditioned process p when c translates into a regular process q by using the following structural translation rules:

$$(p \,|\, q) \text{ when } c = (p \text{ when } c) \,|\, (q \text{ when } c)$$

Now that all formal syntactic elements are defined, we specify the function that translates a mode automaton by the composition of data-flow equations.

The top-level rule $\mathcal{C}[\![a]\!]$ defines the current state of a by a signal x. Its next value is defined by the signal x'. The clock of the mode automaton is noted $\hat{x}$. It is defined by the clock expression $e_x = \bigvee_{y \in \mathrm{def}(a)} \hat{y}$ to mean that if at least one signal y defined by the automaton (i.e. s.t. $y \in \mathrm{def}(a)$) is active then so is the automaton. The auxiliary function $\mathcal{C}^x[\![]\!]$ structurally decomposes the translation of all declarations in an automaton with respect to the name x of the automaton's state.

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

Figure 10: Interpretation of mode automata by data-flow equations.

The declaration of the initial state $\mathsf{init}\, s_0$ of the automaton corresponds to the partial definition of x, the current state of the automaton, by its next value x' and initially by s_0. This is the default definition of the state x. It defines the behavior of stuttering: when no transition takes place, the automaton should remain in the same state. The condition of that behavior is defined by the clock $\hat{x} \setminus f_x$. Clock $\hat{x}$ means that the automaton is active. Clock f_x means that some transition takes place. It is defined by the union $f_x = \bigvee_{e \in \mathrm{lab}(a)} e$ of all clock expressions e labelling a transition $e \Rightarrow s \twoheadrightarrow t$ in a.

The rule for a weakly preemptive transition $c \Rightarrow s \to t$ defines the next state x' by t if the current state s is x and the condition c holds. The rule for a strongly preemptive transition $\mathcal{C}^x[\![c \Rightarrow s \twoheadrightarrow t]\!]$ 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 $[x = s]$. A clock expression noted $[x = s]$ can be regarded as the clock $[y]$ where the signal y is defined by the equation $y = \mathsf{eq}\,(x, s)$ meaning that x is present and equals s.

4.5 Extending SIGNALMETA with modes

To manage mode automata, we extend SIGNALMETA with a new class diagram represented in FIG. 11. An *Automaton* is a Model composed of states, transitions, local signals, and *StateObservers*. We put an emphasis on simplicity both for the specification (half a page, FIG. 11) and for the implementation of these mode automata.

As for classical Statecharts [20], or SyncCharts [4], 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 SIGNALMETA Models (represented in the class diagram by the *ModelsWithDataflow* Model). Thus, mode automata can be composed of SIGNAL programs or of sub-mode automata. *State* inherits from the *ModelsWithDataflow* Model to be able to describe SIGNAL equations. Finally, the *InitState* Atom is intended to be connected to the initial state of the *Automaton*.

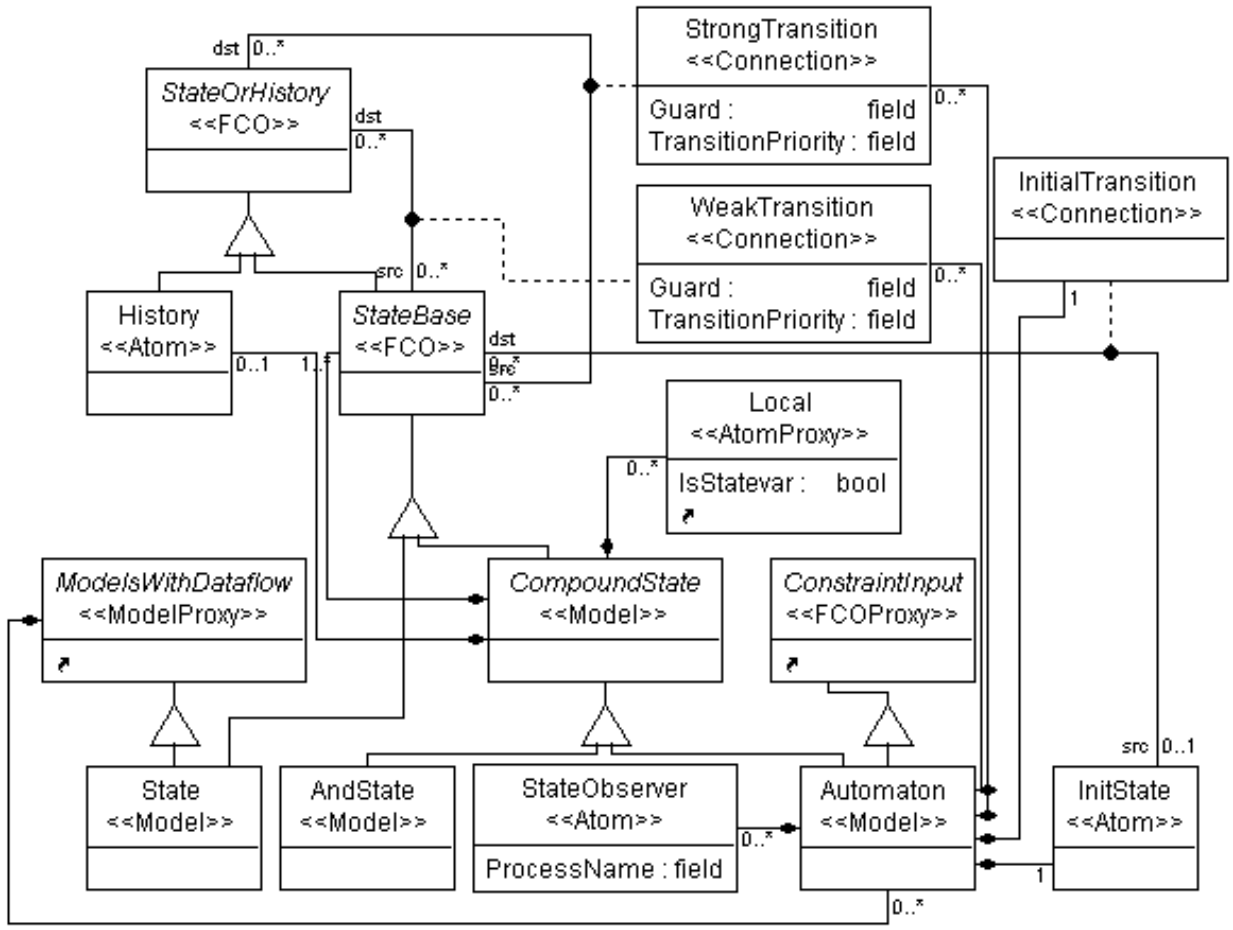


Figure 11: Extension of the SIGNAL metamodel taking into account mode automata.

The automata transitions are represented as Connections in the metamodel. 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.

Contrarily to SyncCharts [4], 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 [14]. 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. 12(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*, to re-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 *WeakTransition*s (resp. *StrongTransition*s) 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*.

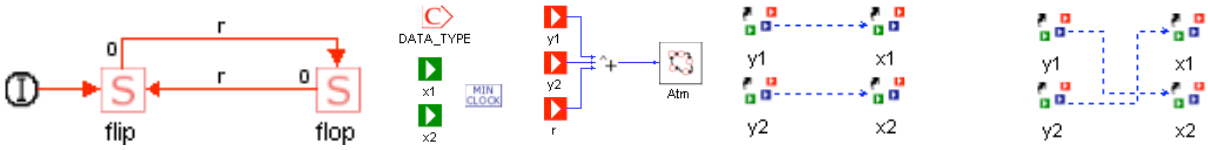
To observe the state of an automaton, we add a *StateObserver* Atom, which allows to call a process having the current state of the automaton as input signal. The name of this process is provided through the attribute *ProcessName*. If this attribute is not defined, the current state is written on the 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.

4.6 Example - Modeling the flip flop switch

We exemplify the modeling of a mode automaton in GME by reconsidering the example of Sec. 4.2. The model of the switch consists of an interface, an automaton and two modes. 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 upon the internal state s of the switch. The state is toggled using the reset signal r.

FIG. 12(a) represents 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. 12(c) (resp. 12(d)). 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.



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

Figure 12: Refinement of the `Atm` *Automaton* in GME.

In FIG. 12(b), we have generated code for sequential execution of the automaton. The clock of `Atm` is defined as 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: SIGNAL is a single assignment language, however it allows one to define several assignments to the same signal in a program (see lines `14` and `15` in FIG. 13), provided that these assignments are exclusive in time, i.e. at any time only one assignment is at most valid. So, 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.

```
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 13: The SIGNAL code generated by the Interpreter from the model of the switch.

4.7 Implementation in GME

The GME Interpreter used to analyze SIGNALMETA Models and produce the corresponding SIGNAL programs is extended to produce the SIGNAL equations corresponding to mode automata descriptions. The transformation, illustrated by the code (cf. 13), 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 *StrongTransition*s (lines 5-6).
- If the *Automaton* contains *CompoundState*s (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 *WeakTransition*s (resp. *StrongTransition*s) are conditioned by the event of the corresponding transition. The default values 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 *WeakTransition*, 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 defined by the value of `currentState` (lines 9-11).

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. 12(b). 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 SIGNALMETA Model.

5 Application to avionics system design

The idea of the specialized avionics model-driven environment is illustrated in FIG. 14. Two description layers are distinguished. The first one (on the top) is entirely object-oriented. It encompasses the MIMAD paradigm defined within GME. The other one (on the bottom) is dedicated to domain specific technologies. Here, we consider the POLYCHRONY environment. However, one can observe that the approach is extensible to other technologies or models of computation that offer specific functionalities to the UML layer.

As GME allows to import and export XML files, information exchange between the layers can rely on this intermediate format. This favors a high flexibility and interoperability of the approach. GME also proposes specific facilities that enable to connect new environments to its associated platform. This latter possibility permits to implement the code generation directly from GME models without exporting them in XML. It also facilitates the interactive dialog between GME and the connected environments.

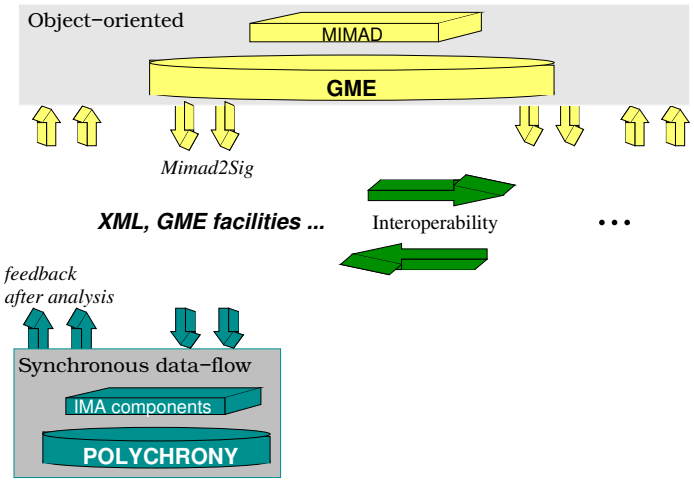


Figure 14: Connection of MIMAD to POLYCHRONY.

The object-oriented layer aims at providing a user with a graphical framework allowing to model applications using the components offered in MIMAD. Application architectures can be easily described by just selecting components via drag and drop. Component parameters can be specified (e.g. period and deadline information for an IMA process model). The resulting model is transformed into SIGNAL (referred to as **Mimad2Sig** in FIG. 14) based on the intermediate representation (e.g. XML files).

In the synchronous data-flow layer, the intermediate description obtained from the upper layer is used to generate a corresponding SIGNAL model. This is achieved by using the IMA-based components already defined in POLYCHRONY [17]. Thereon, the formal analysis and transformations techniques available in POLYCHRONY can be applied to the generated SIGNAL specification.

Finally, a feedback is sent to the object-oriented layer to notify the user about possible incoherence in initial descriptions. In this way, a user can quite easily design applications based on the IMA modeling approach proposed in POLYCHRONY.

5.1 Definition of MIMAD in GME

The description of an IMA system in GME is done in a modular way. Each level of the MIMAD paradigm is defined by a class diagram and inherits from the *InterfaceDefinition* Model of SIGNALMETA. This means that *ImaSystem*, *ImaModule*, *ImaPartition*, and *ImaProcess* Models (see FIG. 15) can contain *Input*, *Output*, and *Parameter* Atoms.

ImaPartition also includes a *PartitionLevelOS* Atom, which allows to specify the scheduling policy. The most complex class diagram is the *ImaProcess* Model shown in FIG. 15. It contains *Block* References, which refer to APEX-ARINC services or other functions defined by the user in a *ModelDeclaration* Model. The control and computation parts of an *ImaProcess* Model are separated into two Aspects. In the computation part, Connections between inputs and outputs of *Block*s are explicitly described. The control part is represented by a mode automaton (see Section 4) in which states are the *Block*s specified in the other Aspect[3].

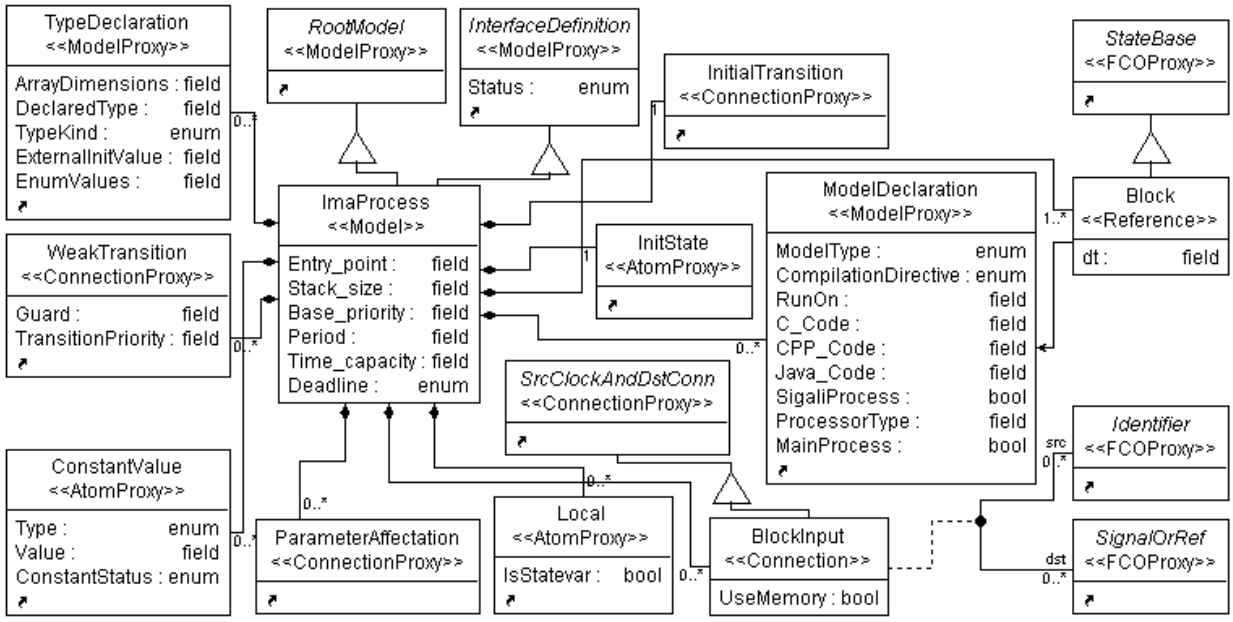


Figure 15: The *ImaProcess* paradigm sheet.

To complete the metamodel, we have defined a GME library, which contains a *ModelDeclaration* for each APEX-ARINC service. These services are required to describe communications and synchronization between processes and partitions, time management, scheduling issues. The overall metamodel results from the above component models: IMA architectural elements (process, partition, etc.) and APEX-ARINC services.

[3]This separation of concerns is illustrated in FIG. 17 and 18 for the CORRELATION process of the CMF avionic application (Section 5.2)

5.2 Example - Modeling of an avionic application

The use of MIMAD to model avionic applications is illustrated by considering the CMF application introduced in Section 1.4. This application has been specified and already modeled in SIGNAL [18]. For more details on the adopted approach to model such an application in SIGNAL, the reader may refer to the mentioned paper. Here, the given illustration mainly shows how the same approach is described using the MIMAD extension of SIGNALMETA. A partition is composed of two Aspects. The first one is the *Interface* Aspect of SIGNALMETA. The second Aspect (*ImaAspect*) specifies the architecture of applications based on IMA concepts. The elements of the *ImaAspect* are the following (see FIG. 16, left frame): the partition-level OS, processes and mechanisms required for communication and synchronization between processes.

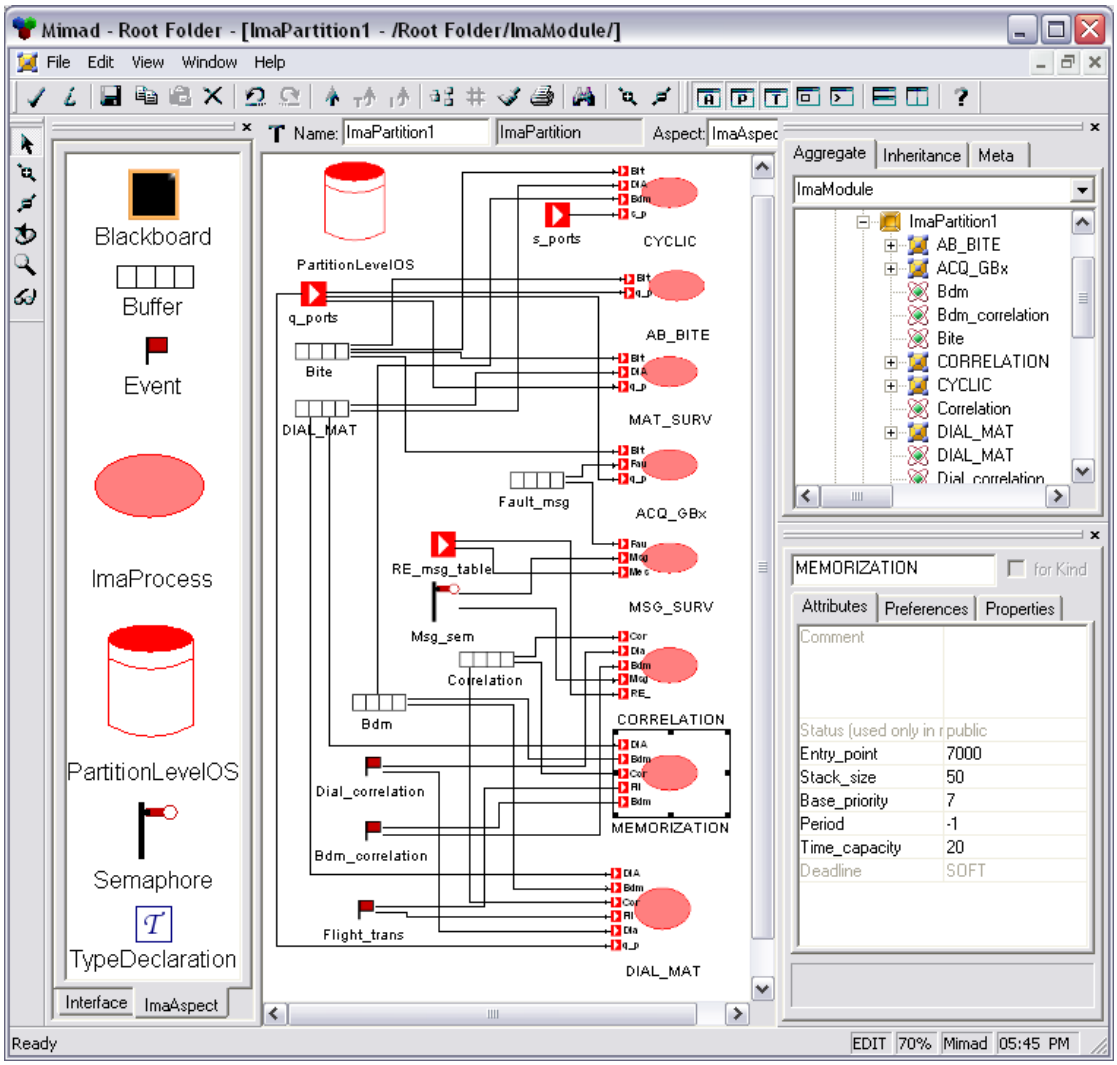


Figure 16: A MIMAD model of CMF.

5.2.1 Partition Level Design

The *PartitionLevelOS* is represented by an Atom whose attribute specifies the scheduling policy adopted by the partition (e.g. *Rate Monotonic Scheduling, Earliest Deadline First*). We observe at this stage that the presence of this element is more for structural and visual convenience. However, the scheduling information it carries will be necessary in the resulting executable description of the application after transformations. An IMA process is described by a Model whose attributes are used by the partition level OS for process creation and management (Section 5.2.2 focuses in a more detailed way on the modeling of IMA processes).

There are four kinds of inter-process communication and synchronization mechanisms: Blackboards, Buffers, Events, and Semaphores. The attributes of their associated models are those needed by the partition level OS for their creation. The Model of the CMF partition is shown in FIG. 16. The partition contains eight IMA processes, several communication and synchronization mechanisms, and the partition level OS. The global parameters managed by the partition are represented by the input `RE_msg_table`. Finally, the inputs `s_port` and `q_port` respectively identify sampling and queuing ports used for the communications with the other partitions

5.2.2 Process Level Design

The SIGNAL model of an IMA process consists of two sub-parts [17]: a *control* part that selects a subset of actions, called block, to be executed, and a *compute* part, which is composed of blocks. With MIMAD, IMA processes are designed using three aspects: the Interface as in the partition level, the ImaAspect, which includes the computation subpart, and the ImaProcessControl, containing the control flow of the process. We focus on the CORRELATION process in CMF to illustrate the process level design. The role of this process is to treat the failure messages and to produce the maintenance information that must be retrieved by an operator.

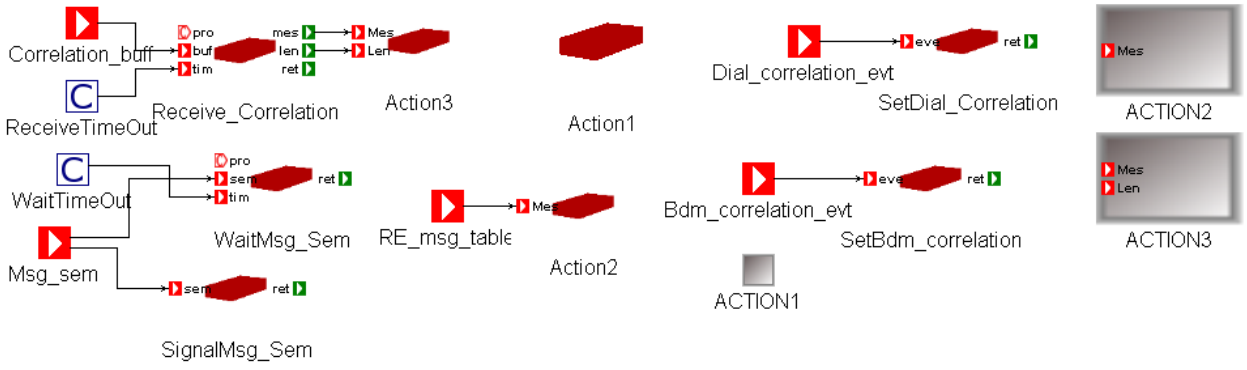


Figure 17: A SIGNALMETA model of process CORRELATION - Computation (*ImaAspect*)

The computation part of the process model shown in FIG. 17 consists of a data-flow graph. It contains Blocks of data-flow equations. These Blocks are References to *ModelDeclaration* specified in the same Aspect or in some library (e.g. APEX-ARINC services). It also contains constant values, local signals, and type declarations. The connections between the Interface and the Blocks

on the one hand, and between Blocks on the other hand are also specified in this Aspect. The control part of the process is described with the mode automaton depicted by FIG. 18. Each time the process CORRELATION is active, the current state of the automaton indicates which Block in the computational part is executed.

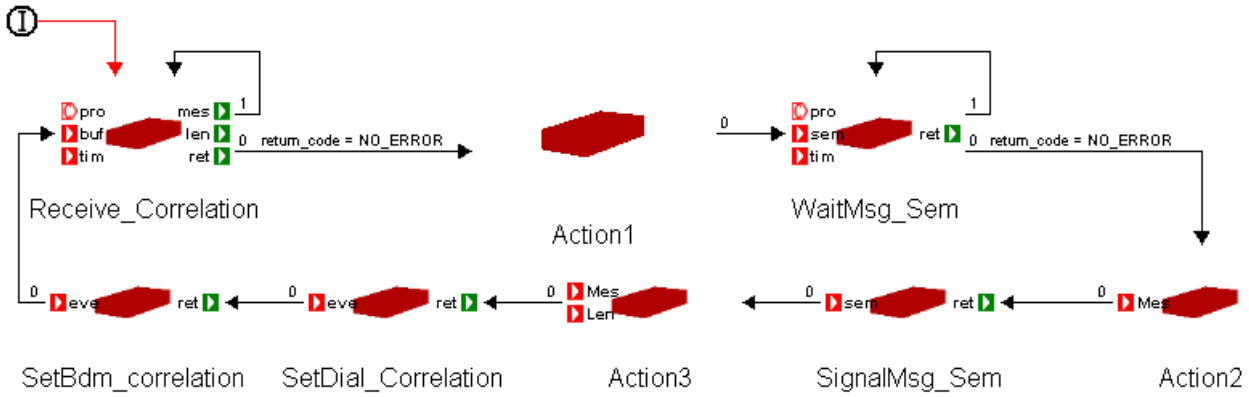


Figure 18: A SIGNALMETA model of process CORRELATION - Control (*ImaProcessControl*)

From these MIMAD descriptions, a corresponding SIGNAL program can be automatically generated. The functionalities of POLYCHRONY can be used in order to formally analyze the application model. The reader will find further details on the code generation process in [12]. We have already addressed the temporal analysis of the hand coded SIGNAL program corresponding to the CMF application in [18]. We showed how the scalability issue is dealt with for such huge applications by proposing a modular approach.

6 Migration to Eclipse

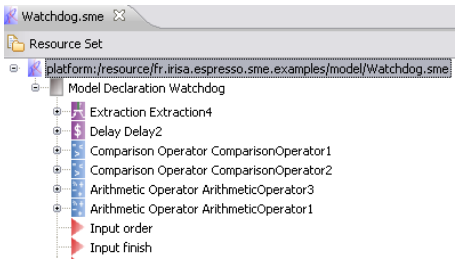
With the objective of integrating the environment presented so far into the open-source platforms TopCased [36] and OpenEmbeDD [44], Eclipse plug-ins have been created. Section 6.1 presents the editors built around the SME metamodel, which corresponds to the SIGNALMETA metamodel under Eclipse. Section 6.2 describes how these editors are connected to the POLYCHRONY compiler to make it available from the Eclipse environment.

6.1 Eclipse editors for POLYCHRONY

The Eclipse plug-ins rely on the Eclipse Modeling Framework (EMF) [41]. From a metamodel, this framework generates a plug-in allowing one to visualize and create models, conform to this metamodel, as a tree leading to what is referred to as a *reflexive* editor. For that, the metamodel must be specified using the Ecore metametamodel language provided with EMF. To build this metamodel, some preliminary work has been done to realize a bridge between GME and EMF. This work, described in [11], uses the ATL tool [5] to transform MetaGME metamodels into Ecore metamodels. SIGNALMETA has been used as a test case for this transformation, which gave a first

metamodel for Eclipse. However, the Ecore language is more abstract than the MetaGME one, since one manipulates only generic metaclasses and relations.

As a consequence, while all concepts and their associated relations are kept during the automatic transformation, the graphical and Aspect information are lost. Moreover, the abstractness of the Ecore language has been exploited to model more precisely a few concepts that would be difficult to represent graphically in GME. In this way, the initial metamodel obtained through the ATL transformation has been modified to obtain SME.



Around the SME metamodel, the SME reflexive editor has been built using the EMF facilities. To obtain a graphical modeler as in GME, we have used the modeling facilities of TopCased. A set of graphical modeler plug-ins has been generated from a file where each concept of an Ecore metamodel is mapped to graphical information.

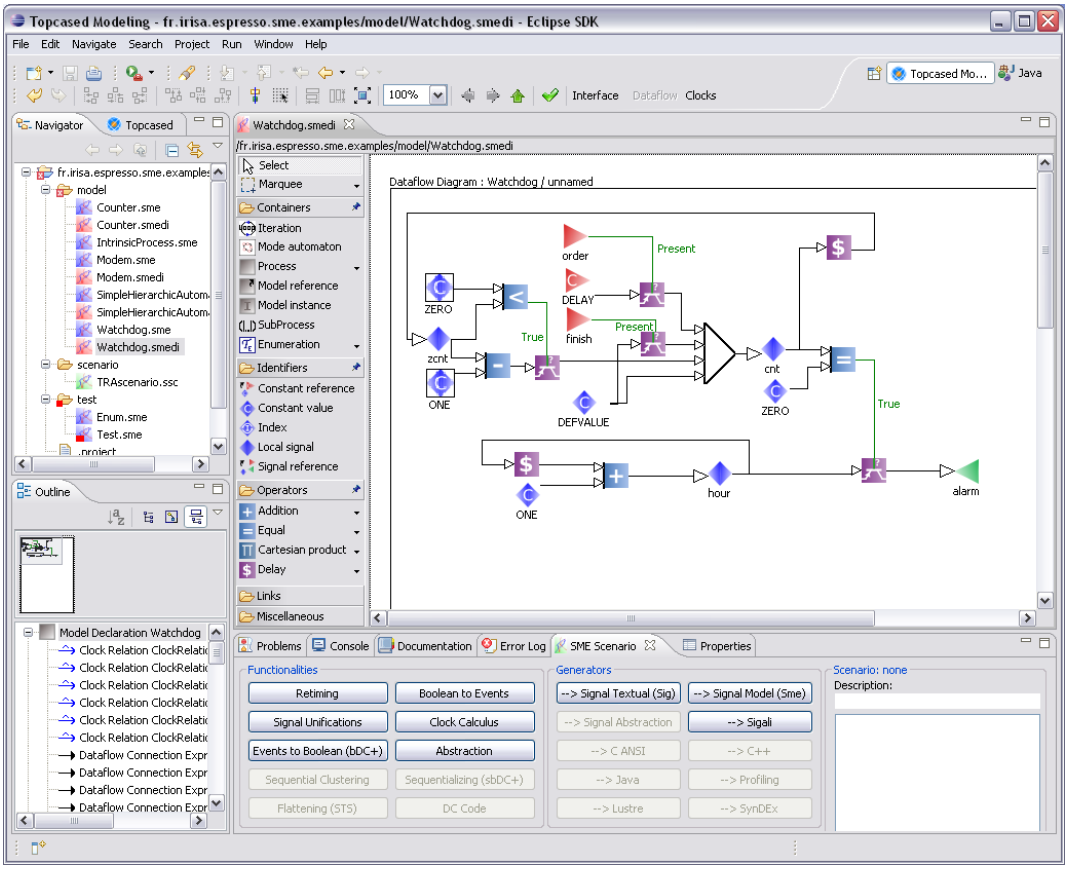


Figure 19: SME graphical modeler.

This graphical environment is represented in FIG. 19. The notion of *Aspect*, which is present in GME, has been reproduced so as to obtain the same approach for modeling. For this purpose,

the modeling is divided into several diagrams that correspond to the Aspects defined in GME:

- for a process, one diagram to model its interface, one for the computation part, and one for all explicit clock relations and dependencies,
- for a module, one diagram to describe a library,
- one diagram for mode automata extension.

6.2 Connection to the POLYCHRONY compiler

The SME plug-ins go further than SIGNALMETA for the connection to POLYCHRONY. We deeply connect the reflexive editor and the graphical environment with the compiler in order to dynamically check the correctness of the model. Our main goal for this connection is to obtain a traceability between the SME models and the results returned by the compiler. Thus, it becomes possible to indicate directly on the source model the compilation errors.

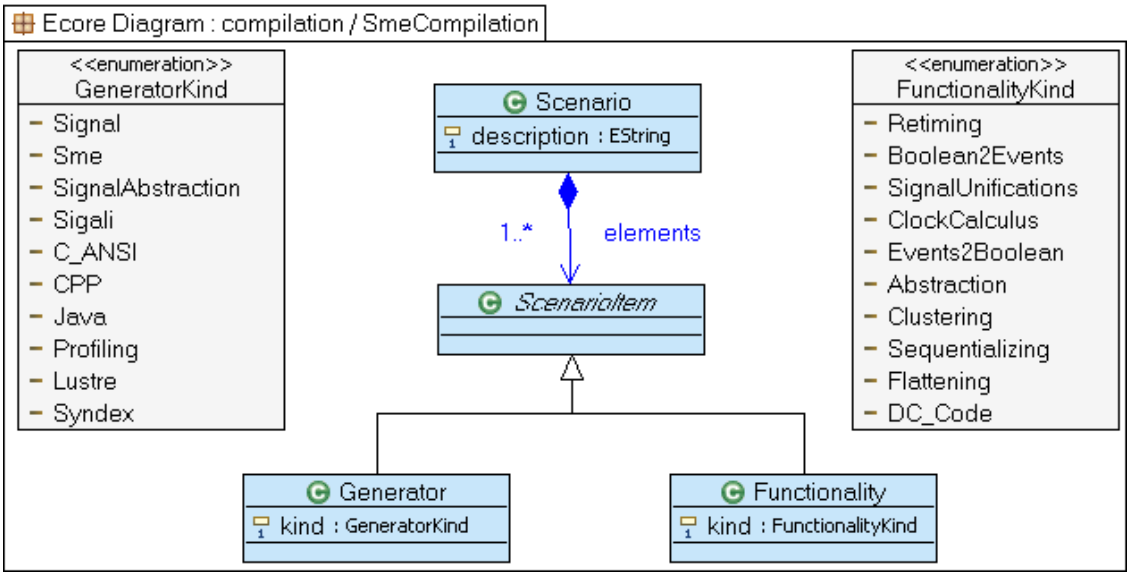


Figure 20: SME compilation metamodel.

To build this connection, the POLYCHRONY services have been described in another simple metamodel (see FIG. 20). This allows users to create compilation scenarios by defining the order in which the different functionalities and generators are called. From this metamodel, a reflexive editor has been generated as for the SME one, and a specific view, shown under the diagram in FIG. 19, has been added. This view favors services of interactive "intelligent" compilation: it allows to build a compilation scenario by selecting the different services of the compiler. But it enables progressively functionalities and generators when they are applicable or disables them when they are forbidden or useless.

The second issue about the connection between the SME Eclipse editors and the POLYCHRONY compiler consists of a Java/C interface to communicate with the compiler through native libraries

(for Linux, Windows, and MacOS X/Intel). The principle of the communication (represented on figure 21) is the following:

1. First, the SME model is transformed into the abstract syntax tree (AST) representation inside the compiler. A SME model parser that makes this translation has been developed. At this step, the parser can report all errors concerning the well-formedness of the model and the parsing errors (for attributes that contain SIGNAL syntax).

2. Then, this AST representation is transformed into an internal graph structure. This step consists in resolving all references specified inside the AST, checking the type errors, and making explicit all implicit clock constraints and clock relations. This means that new signals, which do not exist in the source model, are created. The traceability consists in linking each new signal to the corresponding original AST object.

3. The third step consists in applying to this graph the different POLYCHRONY services specified in the compilation scenario (clock resolution, code generation...). These operations also modify the graph obtained at the previous step.

4. Finally, by analyzing the graph obtained after these transformations, errors can be reported in the graphical part. This part is partially implemented.

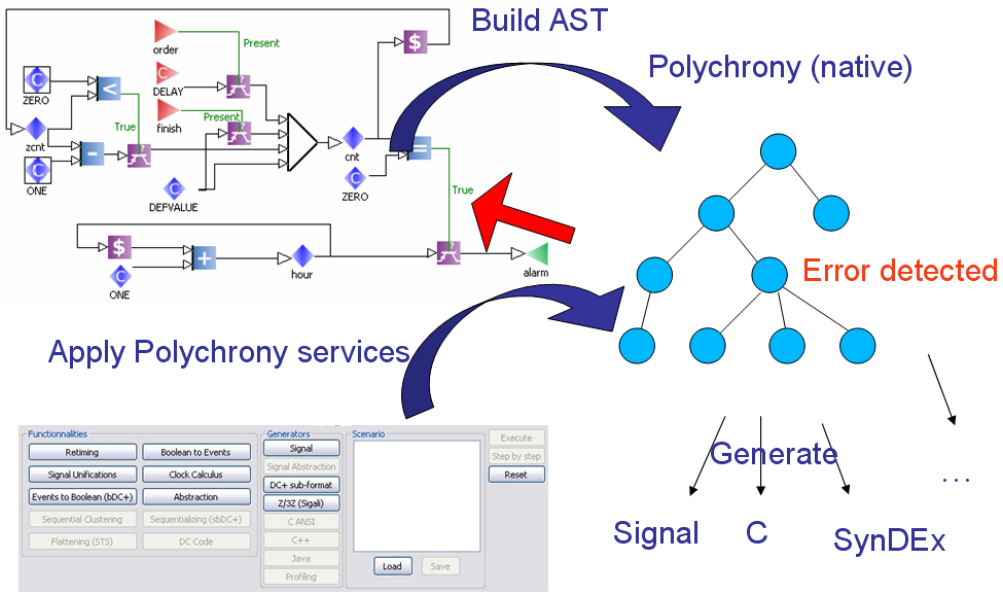


Figure 21: Interaction with Polychrony.

All these tools are packaged inside Eclipse plug-ins available in the current OpenEmbeDD platform and in the TopCased distribution. In order to easily reuse all existing programs and libraries proposed in the SIGNAL distribution, we have also developed a transformation of SIGNAL textual files into SME model files.

7 Discussions and related work

We shall qualitatively assess our approach with respect to the three main aspects addressed during the presentation: model-driven engineering, control-oriented and avionic system design. We also mention a few words on closely related studies.

7.1 On model-driven engineering

The approach presented in this article is the first attempt to open the use of the formal methods supported by POLYCHRONY in the context of a model-driven engineering. The synchronous multi-clock language SIGNAL is tailored via metamodeling to precisely match its semantics and syntax, and to extend its features with new constructs (such as mode automata).

Indeed, model-driven software development is based on a number of common concepts such as XMI, OCL and UML, that can be mapped to different environments. In this case, we have chosen GME to develop our metamodel because it enables rapid creation of metamodels, and it offers customized modeling environments for free. Moreover, GME promotes a generic description of models, which favors a relative independence from specific modeling platforms. The higher the abstraction level is, the more adaptable to various operational environments the specified concepts will be.

SIGNALMETA plays the role of a new graphical front-end for POLYCHRONY. It enables to automatically transform graphical specifications into SIGNAL programs based on its associated interpreter. Structural information of the graphical specification, such as cyclic definitions or the well-formedness of designed models, can be checked. Type consistency and clock constraints are addressed at a lower level with the SIGNAL compiler. Future improvements of the environment aim at automatically enriching higher level description with results of the analysis achieved in lower levels. These results could be displayed graphically and dynamically during the modeling process so as to facilitate the earlier design exploration for users.

Finally, our metamodel has been used as an experiment towards the development of the general-purpose UML profile for modeling real-time and embedded systems, called MARTE [38], which has been adopted by the OMG. The insights gained from the definition of SIGNALMETA have been very useful to us during our participation to the working group for the definition of the MARTE standard.

7.2 On polychronous mode automata

Mode automata were originally proposed by Maraninchi et al. [28] to gather advantages of declarative and imperative approaches to synchronous programming. They have been recently combined with stream functions in Lucid Synchrone in [14].

In a previous work, the introduction of preemption mechanisms in the multi-clocked data-flow formalism SIGNAL was studied by Rutten et al. [33]. 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 one of the issues addressed in this article.

The main advantage of the polychronous approach over previous installments of mode automata principles lies in the capabilities gained for rapid prototyping: not only functionalities and components may 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 synthesize the foreseen implementation in a correct-by-construction manner. SIGNALMETA offers such a capability through its extension with mode automata for control-oriented applications.

7.3 On polychrony versus multi-clocked synchrony

To this very last respect, and in light of previous examples, one observes that polychrony differs very much from related synchronous, multi-clocked or GALS[4] models of computation (MoC), proposed in related works.

In a synchronous MoC such as that of Scade [45], the specification of the system is paced by an input clock which governs the execution of the system and of all its components. The reaction of all synchronous components are synchronized to a tick of that very clock that paces execution across the system. The specified timing model is the implemented timing model. In the polychronous MoC, the specified timing model is a partial order and the implemented timing model can automatically be synthesized from this partial order.

In a multi-clocked MoC, e.g. multi-clocked Esterel [9] or in a GALS MoC, e.g. communicating reactive processes or CRP [7], the timing structure of the system implements the actual distribution of its clocks across the system. Each reactive process owns an execution clock. The communication across clock domains supports concurrency.

In the polychronous MoC, two processes $A \,|\, B$ communicating via a signal x are partially synchronized: clock $\hat{x}$ is necessarily included in $\hat{A}$ and in $\hat{B}$, but no relation between $\hat{A}$ and $\hat{B}$ is supposed to pre-exist. Hence, the specified structure (modules) and the implemented structure (architecture) are not necessarily the same.

The polychronous MoC allows one to refine such a specification in order to either synchronize A and B, for sequential execution, or to partition them, by adding some extra protocol to support its actual distribution. Both refinements define strictly more precise specification (i.e. characterized by a reduced set of possible behaviors) and can be performed by the user with the assistance of automation algorithms.

7.4 On modular avionics design

The central feature of the MIMAD extension of SIGNALMETA is to allow integrated modular avionic system designers and engineers to describe both the system architecture and functionalities based on platform-independent models, within a component-oriented design framework. MIMAD proposes an open modeling framework that ideally complements general-purpose UML profiles such as the AADL [37] or MARTE [38] with an application-domain-specific model of computation suitable for the trusted avionics architecture design.

[4] globally asynchronous locally synchronous

MIMAD is extensible with heterogeneous domain-specific tools for the analysis of properties that are foreign to the polychronous model of computation, e.g., timed automata and temporal property verifiers such as UPPAAL.

Another interesting characteristic of MIMAD is that its visual components, which allow a user to describe IMA concepts, serve as a learning framework for IMA design as well as synchronous programming. There are a few studies that also aim at similar goals with GME, but in different domains. In [32], the facilities of GME are applied to define a visual language dedicated to the description of instruction sets and generating decoders, while in [31], authors define a visual modeling environment for complex embedded systems, where multiple models of computation together with their interaction are captured. In [34], GME is merely used to teach the design of domain-specific modeling environments.

7.5 Overall assessment

The SIGNALMETA environment inherits from the *usability* and *portability* inherent to GME. This is very interesting from a practical point of view. In addition, we can mention the following highly desirable criteria according to which SIGNALMETA can be also distinguished from similar approaches.

Reusability. The GME environment enables to define SIGNALMETA models and to store them as XML files in a repository. These models could be further reused in different contexts, allowing the user to reduce costs in time as promoted in MDE.

Analyzability. The key properties of application models designed with SIGNALMETA are those addressed by tools available in the lower layers (see FIG. 14): functional properties (e.g. safety, liveness) and non functional properties (e.g. response times). The SIGNAL code generated from SIGNALMETA can be analyzed using tools available in POLYCHRONY. Static properties are checked with the compiler and dynamic properties with the SIGALI tool [29].

Scalability. GME plays an important role in the scalability of SIGNALMETA. Indeed, it enables modular designs so that the designer becomes able to model large scale applications in an incremental way. However, one must take care of the code generation process for lower layers, from GME Models, especially when the application size is important. The solution adopted in order to overcome this problem consists in a modular generation approach. The current version of GME enables to select and generate sub-parts of a model. Afterward, they can be stored in repositories without re-generating them when reused.

Regarding the scalability of the programs that are analyzable with the SIGNAL compiler, a detailed experiment can be found in [3]. In the presented case studies, the authors give the time and memory limits computed during the analysis, depending on the size of each considered application. These limits obviously hold for the analysis of SIGNAL programs generated from SIGNALMETA and its extensions. In the particular case of the CMF application illustrated in Fig. 16, some abstractions have been necessary in order to make its temporal analysis possible [18].

Flexibility. Starting from the core, open-source, implementation of SIGNALMETA, we demonstrated the facility of designing model extensions for application-specific purposes. Our first extension was to incorporate a model of polychronous mode automata. Such automata describe the control of a system in a relational manner, in the spirit of the original SIGNAL language. In each

state of the automaton, SIGNAL equations are built in SIGNALMETA. The second extension, called MIMAD, introduce specific modeling concepts for the design of avionics systems based on the Integrated Modular Avionics (IMA) architecture. We also reported extensions to simulate AADL specifications [30] or to design a domain-specific language for space software [46].

Interoperability. SIGNALMETA promotes interoperability by exploiting the possibility of generating, from GME descriptions, XML files as intermediate representation (see FIG. 14). This feature has been clearly shown in the MIMAD extension. An example of closed framework is the AMMA infrastructure developed on top of EMF [5]. It achieves interoperability between different environments by extending the facilities of EMF.

GME or Eclipse. The current SME plug-in suites built within the Eclipse environment is a more achieved tool than SIGNALMETA in the sense that the graphical modeling is more accurate and that the editors are connected directly to POLYCHRONY. The advantage of using GME is that a graphical environment conform to a metamodel is obtained more quickly. Thus, one can easily build its metamodel incrementally, and observe rapidly the result on a graphical model.

The advantage of using Eclipse comes from the separation of the metamodel concepts from the graphical notation. So the specification of the metamodel can be more accurate without thinking of the way the models will be represented graphically.

Due to all above features of SIGNALMETA, we believe that it brings several interesting answers to the high demand of pragmatic frameworks where formal methods can be used for the safe development of embedded systems. In particular, the central role played by metamodeling to achieve the answers has been largely exhibited.

8 Ongoing and future works

The SME plugins are primarily designed to provide the model transformation and code generation services of the Polychrony tool-chain. These services are best suited for being integrated with visual modeling standards, such as the UML profile MARTE [38], or customized for application-specific purposes, as in the Synoptic DSL [46], or interfaced with the Eclipse plugins of related tool-sets, such as Syndex [40].

Virtual prototyping As an example, in [30], we present a model transformation tool that uses SME to produce simulation code for the purpose of virtual prototyping AADL architectures [37]. The tool accepts an AADL specification of a distributed embedded system, written using the Topcased-AADL plugin [47], and uses the model transformation language ATL [5] to produce a semantically equivalent, distributed, specification of the architecture in POLYCHRONY.

Domain-specific language design In the RNTL project Spacify [46], led by CNES and ONERA, we are developing a domain-specific programming environment called Synoptic for the development of embedded software in mission-critical spatial applications. The design of Synoptic starts from the modeling concepts that engineers actually use, borrowed from Simulink, StateFlow or AADL, to specialize and enhance these modeling concepts with features that are specific to the

domain of space applications. In this framework, the SME plugins are used as an implementation platform, which hosts the chosen model of computation and the modular and distributed code generation services.

An open-source embedded system design platform Finally, in the ANR project OpenEmbeDD [44], INRIA is developing an open-source platform for model-driven embedded system design. In this context, we are further developing the SME plugin to inter-operate it with the Syndex tool [40], which provides a back-end infrastructure to generate target-specific, real-time executives.

All these experiments put forward the polychronous model of computation of SME to host prototyping, formal verification, code generation from high-level and heterogeneous visual formalisms. So far, the SME plugins have proved agility for its rapid adaptation and inter-operation within large and industry-scale development platforms.

Our aim is to further promote polychrony as a pivot model of computation for the modular and compositional capture of heterogeneous high-level specifications. In that aim, Polychrony offers semantic-preserving refinement and transformation algorithms tailored by optimizing modular, sequential and distributed code generation services.

9 Conclusions

We have presented a metamodel, SIGNALMETA defined in the GME tool, for the design of multi-clocked systems on top of the POLYCHRONY workbench, which relies on the synchronous language SIGNAL. We have also shown how this metamodel can easily be extended so as to provide designers with adequate concepts for the design of both control-oriented and avionic systems. This work promotes MDE in a formal framework. It therefore allows us to take advantage of key features inherent to MDE such as flexibility and reusability, while being able to address validation issues using available tools.

The definition of the extension of SIGNALMETA for the design of control-oriented applications is based on a model of multi-clocked mode automata. A salient feature of the presentation of this extension 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.

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).

The modeling paradigm for integrated modular avionics (IMA) design, presented in this article, best demonstrates the combined need for a model of computation featuring multi-clocked

synchrony through partially ordered clock relations, as well as the need for combining data-flow and control-flow in order to achieve compositional and ergonomic graphical modeling capabilities[5].

The IMA further illustrates the usefulness of SIGNALMETA and its associated extensions to provide developers with a practical and component-based modeling framework that favors rapid prototyping for design exploration. This is very interesting in the perspective of answering the growing industry demand for higher levels of abstraction in system design process. Moreover, the results obtained from our previous studies on IMA architectures using the synchronous technology available in POLYCHRONY have been fully reused.

The SME environment that is the result of the migration of SIGNALMETA into the Eclipse environment, has also been described. Integrated in the OpenEmbeDD and TopCased platforms, SME communicates with a set of other tools to model, verify, and build embedded systems.

Finally, we can mention the substantial advantage of metamodels, such as SIGNALMETA and SME, in that they allow the abstraction and description of different critical aspects of embedded systems in a structured, flexible and formal way.

References

[1] Airlines Electronic Engineering Committee, "ARINC Specification 653: Avionics Application Software Standard Interface," Aeronautical radio, Inc., Annapolis, Maryland, January 1997.

[2] Airlines Electronic Engineering Committee, "ARINC Report 651-1: Design Guidance for Integrated Modular Avionics," Aeronautical radio, Inc., Annapolis, Maryland, November 1997.

[3] , T.P. Amagbegnon, L. Besnard and P. Le Guernic, "Arborescent Canonical Form of Boolean Expressions", INRIA, Research Rep. RR-2290, June 1994.

[4] C. André, "Representation and analysis of reactive behaviors: A synchronous approach," in *Computational Engineering in Systems Applications (CESA)*. Lille (F): IEEE-SMC, July 1996, pp. 19–29.

[5] ATLAS Transformation Language, `http://www.eclipse.org/m2m/atl`. The Eclipse Consortium, 2008.

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

[7] G. Berry, S. Ramesh, R. K. Shyamasundar. "Communicating reactive processes". 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 1993.

[8] G. Berry. "The Foundations of Esterel", *Proof, Language and Interaction: Essays in Honour of Robin Milner, MIT Press*, 2000.

[5] In the same spirit, a commercial implementation of POLYCHRONY (RT-Builder, commercialized by Geensys) allows for the real-time, hardware-in-the-loop, simulation of all electronic equipments connected around a CAN bus on-board a commercial car.

[9] G. Berry, H. Sentovich. “Multiclock Esterel”. 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods. Lecture Notes In Computer Science, v. 2144. Springer, 2001.

[10] L. Besnard, T. Gautier, and P. Le Guernic. SIGNAL v4 Reference Manual, `www.irisa.fr/espresso/Polychrony/doc/document/V4_def.pdf`. INRIA, 2008.

[11] J. Bézivin, C. Brunette, R. Chevrel, F. Jouault, and I. Kurtev, “Bridging the Generic Modeling Environment and the Eclipse Modeling Framework,” in *Proceedings of the 4th workshop in Best Practices for Model Driven Software Development, OOPSLA*, October 2005.

[12] C. Brunette, R. Delamare, A. Gamatié, T. Gautier, and J.-P. Talpin, “A Modeling Paradigm for Integrated Modular Avionic Design,” INRIA, Research Rep. RR-5715, October 2005.

[13] 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.

[14] J.-L. Colaço, B. Pagano, and M. Pouzet, “A conservative extension of synchronous data-flow with state machines,” in *EMSOFT ’05: Proceedings of the 5th ACM International Conference on Embedded Software*. New York, NY, USA: ACM Press, 2005, pp. 173–182.

[15] 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.

[16] A. Gamatié and T. Gautier, “Synchronous Modeling of Modular Avionics Architectures using the SIGNAL Language,” *INRIA research report*, no. 4678, December 2002.

[17] A. Gamatié and T. Gautier, “Synchronous Modeling of Avionics Applications using the SIGNAL Language.” in *RTAS ’03: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*. Washington D.C., USA: IEEE Computer Society, May 2003, p. 144.

[18] A. Gamatié, T. Gautier, and P. Le Guernic, “An Example of Synchronous Design of Embedded Real-Time Systems based on IMA,” in *Proceedings of the 10th International Conference on Real-time and Embedded Computing Systems and Applications (RTCSA’2004)*, vol. 88. Gothenburg, Sweden. August 2004.

[19] N. Halbwachs. “A Synchronous Language at Work: the Story of Lustre”, *Proceedings of the third ACM-IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE’2005*, Verona, Italy, july 2005.

[20] D. Harel, “Statecharts: A visual formalism for complex systems,” *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, June 1987.

[21] D. Harel and A. Pnueli, “On the Development of Reactive Systems,” in *Logics and Models of Concurrent Systems (K. R. Apt, ed.), NATO ASI Series*, Vol. F-13, pp. 477-498, Springer-Verlag, New York, 1985.

[22] E. K. Jackson and J. Sztipanovits, “Using separation of concerns for embedded systems design,” in *EMSOFT ’05: Proceedings of the 5th ACM international conference on Embedded software*. New York, NY, USA: ACM Press, 2005, pp. 25–34.

[23] A. Ledeczi, M. Maroti, and P. Volgyesi, "The Generic Modeling Environment," in *Proceedings of the IEEE Workshop on Intelligent Signal Processing (WISP'01)*, May 2001.

[24] Lee, E. A., Sangiovanni-Vincentelli, A. "A framework for comparing models of computation". In *IEEE transactions on computer-aided design*, v. 17, n. 12. IEEE Press, December 1998.

[25] E.A. Lee "Overview of the Ptolemy Project," *Technical Memorandum No. UCB/ERL M03/25, University of California, Berkeley, CA, 94720*, USA, July 2, 2003.

[26] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann, "POLYCHRONY for system design," *Journal of Circuits, Systems, and Computers - Special Issue: Application Specific Hardware Design*, vol. 12, no. 3, pp. 261–303, december 2003.

[27] 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.

[28] F. Maraninchi and Y. Rémond, "Mode-automata: a new domain-specific construct for the development of safe critical systems," *Science of Computer Programming*, vol. 46, no. 3, pp. 219–254, 2003.

[29] 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.

[30] Ma, Y., Talpin, J.-P.,Gautier, T. Virtual prototyping AADL architectures in a polychronous model of computation. ACM-IEEE Conference on Methods and Models for Codesign. IEEE, June 2008.

[31] D. Mathaikutty, H. Patel, S. Shukla and A. Jantsch, "EWD: A Metamodeling Driven Customizable Multi-MoC System Modeling Environment," *Virginia Polytechnic Institute and State University research report*, no. 2004-20, 2004.

[32] T. Meyerowitz, J. Sprinkle and A. Sangiovanni-Vincentelli, "A Visual Language for Describing Instruction Sets and Generating Decoders," *4th ACM OOPSLA Workshop on Domain Specific Modeling, Vancouver, BC*, pp. 23–32, October 2004.

[33] E. Rutten and F. Martinez, "SIGNAL GTI: implementing task preemption and time intervals in the synchronous data flow language signal," in *Proceedings of the 7th Euromicro Workshop on Real-Time Systems*. IEEE Press, 1995.

[34] J. Sprinkle, J. Davis and G. Nordstrom, "A Paradigm for Teaching Modeling Environment Design," *OOPSLA'04 Educators Symposium (Poster Session), Vancouver, BC*, pp. 24–28, October 2004.

[35] J.-P. Talpin, C. Brunette, T. Gautier, A. Gamatié. Polychronous mode automata. International Conference on Embedded Software. ACM Press, 2006.

[36] 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. More information on `http://www.topcased.org`.

[37] The SAE AADL Information Site, http://www.aadl.info. Society of Automotive Engineers Standard, 2008

[38] The OMG. UML Profile for modeling and analysis of real-time and embedded systems (MARTE). OMG document realtime/05-02-06, 2006.

[39] The Mathworks, Matlab's Simulink and Stateflow, http://www.mathworks.com, 2008.

[40] Syndex - System-level CAD software for distributed real-time system design, www-rocq.inria.fr/syndex. INRIA, 2008.

[41] The Eclipse Modeling Framework, http://www.eclipse.org/emf. The Eclipse Consortium, 2008.

[42] The Generic Modeling Environment (GME), http://www.isis.vanderbilt.edu/Projects/gme/. Vanderbilt University, 2008.

[43] The POLYCHRONY website, http://www.irisa.fr/espresso/Polychrony. INRIA, 2008.

[44] The RNTL project OpenEmbeDD, http://www.openembedd.org. INRIA, 2008.

[45] The Scade Suite, http://www.esterel-technologies.com/products/scade-suite. Esterel technologies, 2008.

[46] The Spacify project, http://spacify.gforge.enseeiht.fr. CNES, 2008.

[47] The TopCased project, http://www.topcased.org. Airbus, 2008.

A Model of computation

We describe the semantics of SIGNAL in the polychronous model of computation (see [26] for more detail). In this model, symbolic tags t or u denote periods in time during which execution takes place. Time is defined by a partial order relation $\leq$ on tags: $t \leq u$ stipulates that t occurs before u. A chain is a totally ordered set of tags. It corresponds to the clock of a signal: it samples its values over a series of totally related tags. The domains for events, signals, behaviors and processes are defined as follows:

- an *event* is a pair consisting of a tag $t \in \mathbb{T}$ and a value $v \in \mathbb{V}$,
- a *signal* is a function from a *chain* of tags to a set of values,
- a *behavior* b is a function from a set of signal names to signals,
- a *process* p is a set of behaviors that have the same domain.

Notations We write $\mathcal{T}(s)$ for the chain of tags of a signal s and $\min s$ and $\max s$ for its minimal and maximal tag. We write $\mathcal{V}(b)$ for the domain of a behavior b (a set of signal names). The restriction of a behavior b to X is noted $b|_X$ (i.e. $\mathcal{V}(b|_X) = X$). Its complementary $b_{/X}$ satisfies $b = b|_X \uplus b_{/X}$ (i.e. $\mathcal{V}(b_{/X}) = \mathcal{V}(b) \setminus X$).

Synchrony The synchronization of a behavior b with a behavior c is noted $b \leq c$ and is defined as the effect of "stretching" its timing structure. A behavior c is a *stretching* of a behavior b, written $b \leq c$, iff $\mathcal{V}(b) = \mathcal{V}(c)$ and there exists a bijection f on tags s.t.

$\forall t, u, t \leq f(t) \wedge (t < u \Leftrightarrow f(t) < f(u))$

$\forall x \in \mathcal{V}(b), \mathcal{T}(c(x)) = f(\mathcal{T}(b(x))) \wedge \forall t \in \mathcal{T}(b(x)), b(x)(t) = c(x)(f(t))$

b and c are *clock-equivalent*, written $b \sim c$, iff there exists a behavior d s.t. $d \leq b$ and $d \leq c$. The synchronous composition $p \,|\, q$ of two processes p and q is defined by combining behaviors $b \in p$ and $c \in q$ that are identical on $I = \mathcal{V}(p) \cap \mathcal{V}(q)$, the interface between p and q.

$$p \,|\, q = \{b \cup c \,|\, (b, c) \in p \times q \ \wedge \ b|_I = c|_I \ \wedge \ I = \mathcal{V}(p) \cap \mathcal{V}(q)\}$$

Asynchrony Desynchronization is defined as the effect of "relaxing" the timing structure of a behavior: a behavior c is a *relaxation* of b, written $b \sqsubseteq c$, iff $\mathcal{V}(b) = \mathcal{V}(c)$ and, for all $x \in \mathcal{V}(b)$, $b|_x \leq c|_x$. Two behaviors b and c are *flow-equivalent*, written $b \approx c$, iff there exists a behavior d s.t. $b \sqsupseteq d \sqsubseteq c$. The asynchronous composition $p \parallel q$ of two processes p and q is defined by the set of behaviors d that are flow-equivalent to behaviors $b \in p$ and $c \in q$ along the interface $I = \mathcal{V}(p) \cap \mathcal{V}(q)$.

$$p \parallel q = \left\{d \,|\, (b, c) \in p \times q \ \wedge \ b_{/I} \cup c_{/I} \leq d_{/I} \ \wedge \ b|_I \sqsubseteq d|_I \sqsupseteq c|_I \ \wedge \ I = \mathcal{V}(p) \cap \mathcal{V}(q)\right\}$$

B Semantics of Signal

The semantics $[\![P]\!]$ of a Signal process P is a set of behaviors that are inductively defined by the concatenation of reactions. Initially, we assume that $\emptyset|_{\mathcal{V}(p)} \in [\![P]\!]$. A reaction r is a behavior with (at most) one time tag t. We write $\mathcal{T}(r)$ for the tag of a non empty reaction r. An empty reaction of the signals X is noted $\emptyset|_X$. The empty signal is noted $\emptyset$. A reaction r is concatenable to a behavior b iff $\mathcal{V}(b) = \mathcal{V}(r)$, and, for all $x \in \mathcal{V}(b)$, $\max(b(x)) < \mathcal{T}(r(x))$. If so, concatenating r to b is defined by

$$\forall x \in \mathcal{V}(b), \forall u \in \mathcal{T}(b) \cup \mathcal{T}(r), (b \cdot r)(x)(u) = \text{if } u \in \mathcal{T}(r(x)) \text{ then } r(x)(u) \text{ else } b(x)(u)$$

Meaning of equations The semantics of a delay $x = y \,\mathsf{pre}\, v$ is defined by appending a reaction r of tag t to a behavior b. It initially defines x by the value v (when b is empty) and then by the previous value of y (i.e. $b(y)(u)$ where u is the maximal tag of b).

$$[\![x = y \,\mathsf{pre}\, v]\!] = \left\{ b \cdot r \,\middle|\, \begin{array}{l} b \in [\![x = y \,\mathsf{when}\, z]\!], \\ u = \max(\mathcal{T}(b(y))), \ r(x) = \\ t = \mathcal{T}(r), \end{array} \,\middle|\, \begin{array}{ll} t \mapsto b(y)(u), & r(y) \neq \emptyset \wedge b \neq \emptyset_{xy} \\ t \mapsto v, & r(y) \neq \emptyset \wedge b = \emptyset_{xy} \\ \emptyset, & r(y) = \emptyset \wedge b = \emptyset_{xy} \end{array} \right\}$$

Similarly, the semantics of a sampling $x = y \,\mathsf{when}\, z$ defines x by y when z is true.

$$[\![x = y \,\mathsf{when}\, z]\!] = \left\{ b \cdot r \,\middle|\, \begin{array}{l} b \in [\![x = y \,\mathsf{when}\, z]\!], \\ u = \max(\mathcal{T}(b(y))), \ r(x) = \\ t = \mathcal{T}(r), \end{array} \,\middle|\, \begin{array}{ll} r(y), & r(z)(t) = \mathsf{true} \\ \emptyset, & r(z)(t) = \mathsf{false} \\ \emptyset, & r(z) = \emptyset \end{array} \right\}$$

Finally, $x = y \,\mathsf{default}\, z$ defines x by y when y is present and by z otherwise.

$$[\![x = y \,\mathsf{default}\, z]\!] = \left\{ b \cdot r \,\middle|\, b \in [\![x = y \,\mathsf{default}\, z]\!], r(x) = \begin{vmatrix} r(y), & r(y) \neq \emptyset \\ r(z), & r(y) = \emptyset \end{vmatrix} \right\}$$

The meaning of the synchronous composition $P \,|\, Q$ is the synchronous composition $[\![P \,|\, Q]\!] = [\![P]\!] \,|\, [\![Q]\!]$ of the meaning of P and Q. The meaning of restriction is defined by $[\![P/x]\!] = \{c \,|\, b \in [\![P]\!] \wedge c \leq (b_{/x})\}$.

Meaning of clocks The meaning $[\![e]\!]_b$ of a clock e is defined with respect to a given behavior b and consists of the set of tags satisfied by the proposition e in the behavior b. The meaning of the clock $x = v$ (resp. $x = y$) in b is the set of tags $t \in \mathcal{T}(b(x))$ (resp. $t \in \mathcal{T}(b(x)) \cap \mathcal{T}(b(y))$) such that $b(x)(t) = v$ (resp. $b(x)(t = b(y)(t))$. In particular, $[\![\hat{x}]\!]_b = \mathcal{T}(b(x))$ and $[\![[x]]\!]_b = [\![x = \mathsf{true}\,]\!]_b$. The meaning of a conjunction $e \wedge f$ (resp. disjunction $e \vee f$ and difference $e \setminus f$) is the intersection (resp. union and difference) of the meaning of e and f. Clock 0 has no tags.

$$\begin{array}{rlr} [\![1]\!]_b{=}\mathcal{T}(b) & [\![0]\!]_b = \emptyset & [\![e \wedge f]\!]_b{=}[\![e]\!]_b \cap [\![f]\!]_b \\ [\![x = v]\!]_b{=}\{t \in \mathcal{T}(b(x)) \,|\, b(x)(t) = v\} & & [\![e \vee f]\!]_b{=}[\![e]\!]_b \cup [\![f]\!]_b \\ [\![x = y]\!]_b{=}\{t \in \mathcal{T}(b(x)) \cap \mathcal{T}(b(y)) \,|\, b(x)(t) = b(y)(t)\} & & [\![e \setminus f]\!]_b{=}b[\![e]\!]_b \setminus [\![f]\!]_b \end{array}$$

Meaning of automata One simple way to define the semantics of polychronous mode automata is to interpret its translation by extending the function $[\![\cdot]\!]$ as below.

$$\begin{array}{rcl} [\![a]\!] & = & ([\![a]\!]^x \,|\, [\![(\hat{x} = \hat{x}') \,|\, (\hat{x} = e_x)]\!])/xx' \\ [\![a \,|\, b]\!]^x & = & [\![a]\!]^x \,|\, [\![b]\!]^x \\ [\![\,\mathsf{init}\, s_0]\!]^x & = & [\![x = (x' \,\mathsf{pre}\, s_0) \,\mathsf{when}\, (\hat{x} \setminus f_x)]\!] \\ [\![c \Rightarrow s \rightarrow t]\!]^x & = & [\![x' = t \,\mathsf{when}\, ([x = s] \wedge c)]\!] \\ [\![c \Rightarrow s \twoheadrightarrow t]\!]^x & = & [\![x = t \,\mathsf{when}\, ([(x' \,\mathsf{pre}\, s_0) = s] \wedge c)]\!] \\ [\![s : p]\!]^x & = & [\![p \,\mathsf{when}\, [x = s]]\!] \end{array}$$

The activity clock $e_x = \bigvee_{y \in \mathrm{def}(a)} \hat{y}$ ticks if at least one signal y is active in the automaton a. The transition clock $f_x = \bigvee_{e \in \mathrm{lab}(a)} e$ is the union of all events e that trigger a transition $e \Rightarrow s \twoheadrightarrow t$ in a.