

Modeling multi-clocked data-flow programs using the Generic Modeling Environment

Christian Brunette, Jean-Pierre Talpin, Loïc Besnard, Thierry Gautier

▶ **To cite this version:**

Christian Brunette, Jean-Pierre Talpin, Loïc Besnard, Thierry Gautier. Modeling multi-clocked data-flow programs using the Generic Modeling Environment. Synchronous Languages, Applications, and Programming, Mar 2006, Vienna, Austria. pp.SLAP 2006. hal-00541310

HAL Id: hal-00541310
https://hal.science/hal-00541310

Submitted on 30 Nov 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Modeling multi-clocked data-flow programs using the Generic Modeling Environment

Christian Brunette [1] Jean-Pierre Talpin [2] Loïc Besnard [3]
Thierry Gautier [4]

IRISA-INRIA-CNRS
Campus Beaulieu
F-35042 RENNES Cedex, FRANCE

Abstract

This paper presents Signal-Meta, the metamodel designed for the synchronous data-flow language SIGNAL. It relies on the Generic Modeling Environment (GME), a configurable object-oriented toolkit that supports the creation of domain-specific modeling and program synthesis environments. The graphical description constitutes the base to build multi-clock environments, and a good front-end for the POLYCHRONY platform. To complete this front-end, we develop a tool that transforms the graphical Signal-Meta specifications to the SIGNAL code. This modeling paradigm constitutes a first work for generalizing the use of formal methods proposed by POLYCHRONY.

Key words: Metamodeling, SIGNAL, GME, synchronous languages, model transformation

1 Introduction

The *synchronous hypothesis* has been proposed in the late '80s and extensively used ever since to facilitate design of control-dominated systems. Nowadays synchronous languages are commonly used in the European industry, especially in avionics, to rapidly prototype, simulate, verify and synthesize embedded software for mission critical applications. However, synchronous programming languages, such as LUSTRE, LUCID SYNCHRONE, ESTEREL, SIGNAL are most commonly regarded as "domain-specific" languages, as their usage is mostly restricted to aid highly-trained engineers to design mission-critical systems.

In the aim of bringing synchronous technologies to a vaster community aware of model-driven engineering, we have developed a simple and highly extensible

[1] Email: Christian.Brunette@irisa.fr
[2] Email: Jean-Pierre.Talpin@irisa.fr
[3] Email: Loic.Besnard@irisa.fr
[4] Email: Thierry.Gautier@irisa.fr

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

interface to the POLYCHRONY workbench, that implements the multi-clocked synchronous data-flow language SIGNAL, with the Generic Modeling Environment (or GME) [14]. This interface is the medium to experiments with relating the polychronous model of computation of the workbench with more ergonomic and diagrammatic notations such as data-flow diagrams, UML state diagrams and the combination of both as mode automata [15]. The aim of this experiment is to find the simplest and most ergonomic representation of a formal model of time such as that of the POLYCHRONY workbench in the forthcoming real-time profiles for more vastly known notations such as the UML state diagrams. This work, based on a syntactic approach, constitutes a first study concerning the generalization of the use of formal methods offered by POLYCHRONY. Other projects (e.g. TOPCASED [18] or OPENEMBEDD) will pursue this study.

The remainder is organized as follows. Sections 2 and 3 first introduce respectively the SIGNAL language and the Generic Modeling Environment. Section 4 describes Signal-Meta, the metamodel of SIGNAL specified in GME. Section 5 illustrates how to use this metamodel through the description of an example. Section 6 presents the component added to GME to transform the graphical specifications into a SIGNAL code. The adopted approach is discussed in Section 7 and finally, conclusions and future works are given in Section 8.

2 The synchronous language SIGNAL and POLYCHRONY

Among other synchronous languages, the POLYCHRONY workbench implements an original model of time as *partially ordered* synchronization and scheduling relations, to provide the ability to model high-level abstractions of systems paced by multiple clocks: locally synchronous and globally asynchronous systems. It provides a flexible way to model heterogeneous and complex distributed embedded systems at a high level of abstraction, while reasoning within a simple and formally defined mathematical model.

In POLYCHRONY, design proceeds in a compositional and refinement-based manner by first considering a weakly timed data-flow model of the system under consideration and then provides expressive timing relations to gradually refine its synchronization and scheduling structure to finally check correctness of the assembled components using assumption/guarantee reasoning. SIGNAL favors the progressive design of correct by construction systems by means of well-defined model transformations, that preserve the intended semantics of early requirement specifications to eventually provide a functionally correct deployment on the target architecture.

The POLYCHRONY IDE, available from [9], offers several tools including the SIGNAL batch compiler that provides a set of functionalities, such as program transformations, optimizations, formal verification, and code generation. POLYCHRONY includes the SIGALI model checker [16], which enables both verification and controller synthesis, and it also includes a graphical user interface for SIGNAL.

The SIGNAL language handles unbounded series of typed values $(x_t)_{t\in\mathbb{N}}$, called *signals*, denoted as x and implicitly indexed by discrete time. At a given instant, a signal may be present, at which point it holds a value; or absent. The set of instants

where a signal x is present is called its *clock*. It is noted as $\hat{x}$. Signals that have the same clock are said to be *synchronous*. In SIGNAL [13], a process is a system of equations over signals that specifies relations between values and clocks of the involved signals. SIGNAL relies on 6 primitive constructs:

- An equation `y := f(x)` describes a relation (e.g. arithmetic, boolean) between a sequence of operands `x` and a sequence of results `y` by a function `f`.
- A delay equation `x := y$n init v` initially defines the signal `x` by the value `v` and then by the previous value of the signal `y`. The signals `x` and `y` are assumed to be synchronous.
- A sampling `x := y when z` defines `x` by `y` when `z` is true and both `y` and `z` are present. `x` is present iff both `y` and `z` are present and `z` holds the value $true$.
- A merge `x := y default z` defines `x` by `y` when `y` is present and by `z` otherwise. `x` is present iff either `y` or `z` is present.
- The synchronous composition `(| P | Q |)` of the processes `P` and `Q` consists of simultaneously considering a solution of the equations in `P` and `Q` at any time.
- The equation `P where x` restricts the lexical scope of a signal `x` to a process `P`.

These primitives are of sufficient expressive power to derive other constructs for comfort and structuring: the clock synchronization operator ($\hat{}=$) for example. The equation $x \hat{}= y$ synchronizes the clocks of signals x and y. It corresponds using SIGNAL's primitives to $(|\, h := (\hat{x} = \hat{y})|)$ where h.

SIGNAL provides a process model in which any SIGNAL process may be "encapsulated" (see an example in FIG. 6). Different categories of process models are syntactically distinguished: these are actions, functions, nodes, and processes. This process frame allows to abstract a process to an interface, so that the process can be used afterwards as a black box through its interface. This interface describes parameters, input-output signals and clock and dependence relations between them. A process model also enables the definition of sub-processes. Sub-processes that are specified by an interface without any internal behavior are considered as external (they may be separately compiled processes or physical components). On the other hand, SIGNAL allows to import external modules (e.g. C++ functions). Finally, put together, all these features of the language favor modularity and re-usability.

3 GME

GME is a configurable UML-based toolkit that supports the creation of domain-specific modeling and program synthesis environments [1]. It is developed by the ISIS institute at Vanderbilt University, and is freely available at [11]. Metamodels are proposed in the environment to describe modeling paradigms for specific domains. Such a paradigm includes, for a given domain, the necessary basic concepts in order to represent models from a syntactical viewpoint to a semantical one. It also includes all relationships between those concepts, their organization, and all rules governing the construction of models.

Note 1 *In the rest of this paper, words beginning with a capital letter refer to* GME *concepts, and those in italics refer to Signal-Meta concepts. Mainly, be careful with the notion of model.*

To use GME, a user first needs to describe a modeling paradigm by defining a project using the MetaGME paradigm. This paradigm is distributed with GME. All modeling paradigm concepts must be specified as classes through usual UML class diagrams. To build these class diagrams, MetaGME offers some predefined UML-stereotypes [14], among which we use only the following in our metamodel: First Class Object (FCO), Model, Set, Atom, Reference, and Connection. FCO constitutes the basic stereotype in the sense that all the other stereotypes inherit from it. It is basically used to represent abstract concepts (represented by classes). Atoms are elementary objects in the sense that they cannot include any sub-part, while the Model is used for classes that may be composed of various FCOs. In a different way, a class with the Set stereotype can contain a sub-set of FCOs registered in the same Model. A Reference is a typed pointer (as in C++), which refers to another FCO. The type of the pointed FCO is indicated on the metamodel by an arrow (in FIG. 1, the *SignalRef* reference points to a *Signal*).

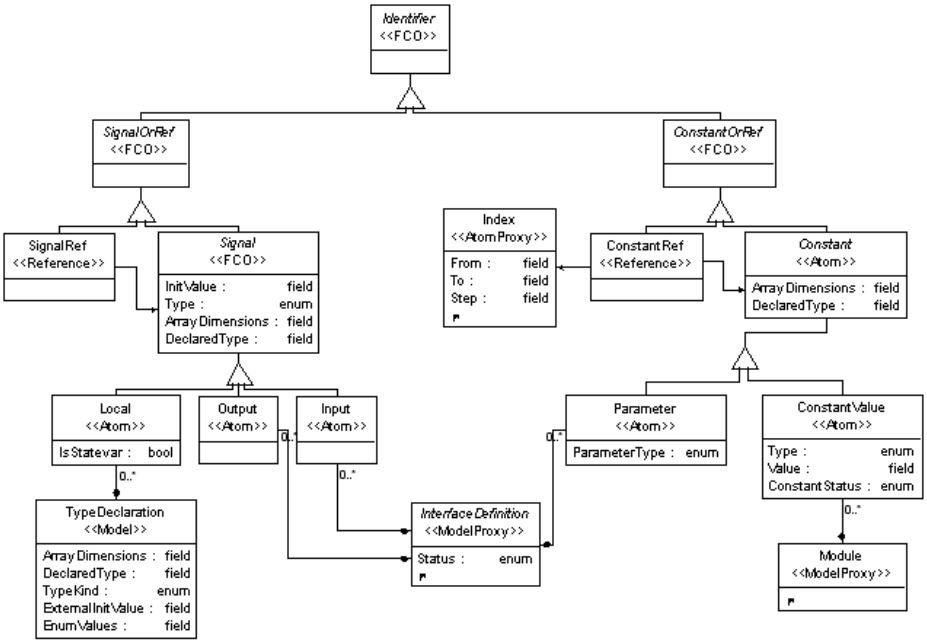


Figure 1. Signal-Meta's Identifier class diagram

There are different kinds of relations that can be expressed between classes, which use these stereotypes. First, the 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. 1 for example between the *Input* atom and the *InterfaceDefinition* Model. Inheritance relations can be represented as in UML. All the other types of relationship are specified by classes that use the Connection stereotype.

In FIG. 1, some FCOs use a stereotype suffixed by "Proxy", such as Module that uses "ModelProxy". Such stereotypes are references inside the metamodel

to a FCO declared in another paradigm sheet. To complete these class diagrams, attributes can be added to classes. These attributes are typed: BooleanAttribute, EnumAttribute that corresponds to a finite list of choices, and FieldAttribute that is a typed text field (string, integer or double).

In these class diagrams, 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). Moreover, it is possible to restrict the use of certain FCOs (add/remove in/from a Model) to a specific Aspect, even if these FCOs are visible in other Aspects.

Finally, OCL Constraints can be added to class diagrams in order to check some "parametric" properties on a model designed with this paradigm (e.g. the type of connections linked to an FCO according to the value of a FCO attribute). A parametric property depends on values given during the edition of a model. OCL constraints are checked when the events on which constraints are associated with are emitted. There are different kinds of events corresponding to the main action during the edition of a model, such as create, connect, and change an attribute.

The whole above concepts constitute the basic building blocks that are used to define modeling paradigms in GME. Such a modeling paradigm is always associated with a paradigm file that is produced automatically. GME uses this file to configure its environment for the creation of models using the newly defined paradigm. This is achieved by the **MetaGME Interpreter**, which is a plug-in accessible via the GME Graphical User Interface (GUI). This tool first checks the correctness of the metamodel, then generates the paradigm file, and finally registers it into GME.

Similarly to the MetaGME Interpreter, other components can be developed and plugged into the GME environment. The role of such a component consists of interacting with the graphical designs. To achieve the connection between the component and GME, an executable module is provided with the GME distribution, which enables the generation of the component 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 [14]. GME distinguishes three families of components that can be plugged to its environment: Interpreter, Addon, and PlugIn. The main difference between the two former components is that the Addon is executed as soon as a project is opened, and it works throughout the graphical edition of models, while the Interpreter has only a punctual execution on user demand. Finally, the PlugIn differs from the above two families of components in that it is paradigm-independent. This means that a PlugIn could apply generic operations on models independently of their modeling paradigm.

4 SIGNAL metamodel

The SIGNAL metamodel, called Signal-Meta, describes all the syntactic elements defined in SIGNAL v4 [4]. Signal-Meta is composed of several paradigm sheets that define all the relations between the different kinds of signals, SIGNAL operators, and SIGNAL process models. They define as Atom each SIGNAL operator presented in Section 2 and each other one derived from them described in [4]. They also define

as Model each SIGNAL container (e.g. process model, sub-process, module), and as Connection each kind of relation between operators and/or identifiers. Section 4 gives more details about the representation of Signal-Meta concepts. Moreover, to facilitate the edition of a model, we specify different Aspects presented in Section 4. The corresponding division separates mainly the data-flow part and the control part of SIGNAL specifications. To complete this metamodel, OCL constraints defined in Signal-Meta bring some interactivity during the edition of a model. The main goal was to keep as much expressiveness as possible in Signal-Meta than in SIGNAL, and to facilitate the edition of models with, for example, n-ary operators. More details on Signal-Meta can be found in [3].

Signal-Meta concepts

Among all paradigm sheets, the *Identifiers*' one represented in FIG. 1 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 is an enumeration of all intrinsic types of SIGNAL. The *DeclaredType* attribute is dedicated to a type imported from a SIGNAL library or to a type already declared in GME. The declaration of a new type is done via the *TypeDeclaration* Model. There are different kinds of types, such as enumeration type, structure type or process model type, which are chosen in the *TypeKind* attribute. The way to declare a new type is different according to the kind of the type: for example, a structure type is specified by adding *Local* Atoms in the *TypeDeclaration* Model and by ordering them, whereas all values of an enumerated type have to be specified in the *EnumValues* attribute.

To use in a Model some signal (resp. some constant or index of an iteration) declared in an upper-level Model, one can use a *SignalRef* (resp. a *ConstantRef*) with the same name. Another way for the 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.

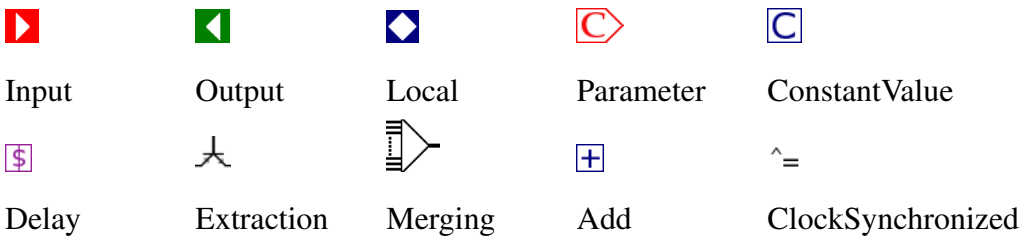


Figure 2. Some of Signal-Meta concepts and their icons

The second line of FIG. 2 shows the graphical forms of some SIGNAL operator FCOs during the edition of a model. These forms are images given in an FCO attribute in the metamodel. *Delay* corresponds to the delay operator, *Extraction* to the sampling operator, *Merging* to the merge operator, *Add* to the addition opera-

tor, and *ClockSynchronized* to the clock synchronization operator (all arithmetic, comparison and clock relation operators are represented as for the *Add* Atom with their corresponding symbol). To facilitate the specification of a model, we add an Atom for boolean expressions and another one for arithmetic expressions in which respectively the boolean expression and the arithmetic expression can be expressed as a textual formula in an attribute.

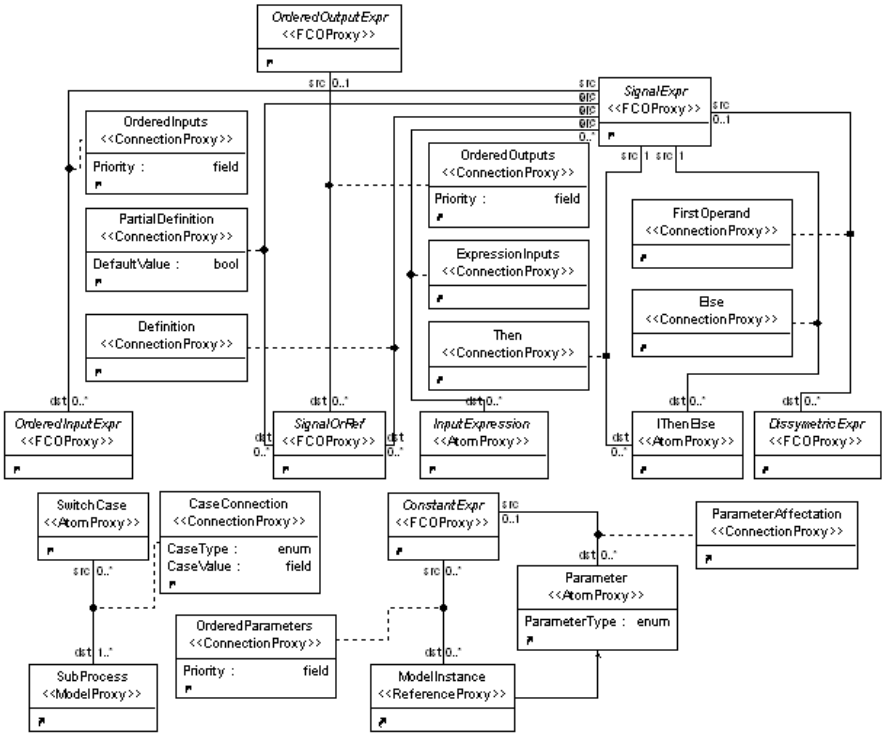


Figure 3. Signal-Meta's 'Expression Connection' paradigm sheet

The main Models of Signal-Meta 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 choice is done via a *ModelDeclaration* attribute. A *ModelDeclaration* consists of a container in which are declared *Input/Output/Local* signals, static *Parameters*, *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. Another interesting point is the way to represent SIGNAL process model instantiations. GME provides a means to express instance objects, thus it would be possible to create instances of *ModelDeclaration* Models. A GME instance of a Model is a deep copy of this Model in which no FCO can be added or removed, but in which attribute values can be modified. To guarantee the exact correspondence between the instance and the corresponding *ModelDeclaration*, we add a Reference, called *ModelInstance*, in Signal-Meta. Thus, *ModelDeclaration* objects are referenced without creating a deep copy of the Model and in a way that guarantees the exact correspondence between the instance and the declaration.

Concerning relations, FIG. 3 corresponds to one of these paradigm sheets and represents all relations between Signal-Meta concepts, except clock relations that are 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. 1), 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 rule applies. Similarly, Signal-Meta offers the *PartialDefinition* Connection to be able to define, in different Models, the different parts of the signal definition.

To simplify the specification of a model, we make several SIGNAL operators become n-ary operators in Signal-Meta. This is done in different ways according to the operator. For operators of type *OrderedInputExpr* (e.g. *Merging*), we use *OrderedInputs* Connections that have a *Priority* attribute whose value allows to order the incoming Connections. Operators of type *InputExpression* (e.g. arithmetic) are divided into two categories: associative and commutative operators, and the other ones (*DissymetricExpr*) for which the first element needs to be identified (e.g. the substraction operator). The first category uses only *ExpressionInputs* Connections, while the second one uses *FirstOperand* to identify the first element.

Aspects, OCL constraints, and extension

Signal-Meta organizes its concepts 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, a *Specifications* Model 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 design all computations of the process and its data flow, whereas the *ClockAndDependence* Aspect contains all clock and dependence relations between signals, instantiations, and sub-processes. Thus, the latter contains mainly clock constraint and relation operators (e.g. *ClockSynchronized*, *ClockUnion*), the *Dependence* Atom, *SignalOrRef*s, and all Connections to link them. The *Dataflow* Aspect can contain all other SIGNAL operators.

This separation of concerns, also recommended by Jackson in [12], makes the models more readable. Indeed, Connections in the *Dataflow* Aspect represent data flows, while they represent only relations in the *Clock Relation*. However, this separation between the data-flow and the control parts is not so obvious in SIGNAL. Actually, SIGNAL primitives implicitly express clock relations between their input/output signals, for example the delay and arithmetic operators synchronize automatically their inputs and their outputs. Thus, operators in the *Dataflow* Aspect also express the control part of the process.

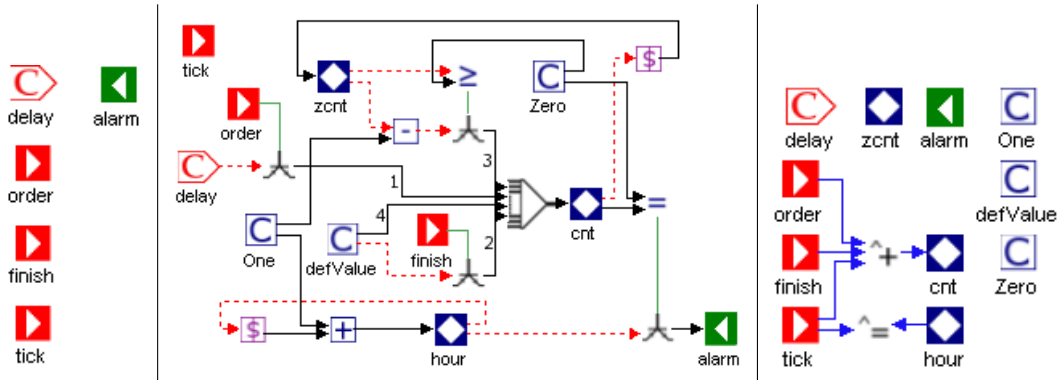
Finally, a *Library* Aspect is dedicated for the concept of *Module*. Indeed a *Module* does not express the data-flow or the control of a process. It only corresponds to a library of constant, type, and process model declarations.

To be able to give some specific information during the edition of a model, we specify a number of OCL constraints to Signal-Meta. They are mainly used to check the coherence of the values of FCO attributes. For example, one constraint checks that the values of the *Priority* attribute for all *OrderedInputs* Connections with the same destination FCO are different. This kind of constraints is checked on user demand or when the Model containing them is closed. Other constraints are checked immediately: for example, a constraint checks that, in a *ModelDeclaration*, the destination of a *Definition* Connection cannot be linked to an *Input* of this *ModelDeclaration*. To complete the constraints we defined, other constraints are automatically generated by the MetaGME Interpreter to check the cardinality affected to each relation. The precise list of OCL constraints is given in [3].

To facilitate the specification of models, we have defined all processes considered intrinsic by SIGNAL in a GME library. In fact, this library contains three *Modules* and one *ModelDeclaration* to represent all usual mathematical functions (e.g. cosine, sine), specific functions to manage complex number and to read (resp. write to) the standard input (resp. output). To represent these functions, we have only defined their interface, which is enough to create *ModelInstance* FCO that refers to them, and to connect their input/output signals and parameters.

5 Example

Here, we apply the metamodel presented in the previous Section to the design of a classical watchdog example. The goal of this watchdog process is to control that some action process is executed within some `delay`. At each time, the action process emits an `order` signal when it begins its execution, and a `finish` event when it finishes it. 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 when the previous one is not finished, the time counting restarts from zero. A `finish` signal out of delay, or not related to an `order`, will be ignored.



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

Figure 4. The Watchdog example

The `Watchdog` process can be specified in GME as shown in FIG. 4. FCOs used in these figures are listed in FIG. 2. FIG. 4(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 must be provided at regular interval, 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. 4(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) `defValue`. This order is fixed using the *Priority* attribute of all incoming Connections on the *Merging* Atom. In FIG. 4(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 Connection specific for each operators (cf. FIG. 3).

In the Clock Aspect (FIG. 4(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 affected to `cnt`.

6 Model Interpretation

Taking advantage of the ease of editing a model in GME, we make GME become a front-end for POLYCHRONY, the current development platform for SIGNAL, through the use of Signal-Meta. Then, we need to transform the graphical specifications using Signal-Meta to the corresponding SIGNAL programs. Therefore, we have implemented a GME Interpreter, which acts similarly for Signal-Meta models as the MetaGME Interpreter for MetaGME metamodels. Figure 5 represents the different steps during the "interpretation", which can be summarized by (i) creating the tree structure of the SIGNAL programs corresponding to the graphical representation, (ii) generating the SIGNAL equation for each level of this intermediary representation, (iii) and finally writing the SIGNAL program into a file. This is pretty simple, because there is a correspondence between graphical elements and the SIGNAL syntax. Here, we detail each of these three main steps.

Step 1: Tree generation. Each FCO selected in the GME GUI is associated with a tree (the intermediate representation in FIG. 5) whose root is the selected FCO. Each node of these trees corresponds to a SIGNAL process model, and each leaf to a symbol (e.g. signal, constant) in the generated program. The tree is built by recursive instantiations of each node into BON objects [14] according to their type in the metamodel. The root FCO is first instantiated. Then, all its contained Models and FCOs, which correspond to symbols (e.g. *Input*, *Output*), are instantiated. The

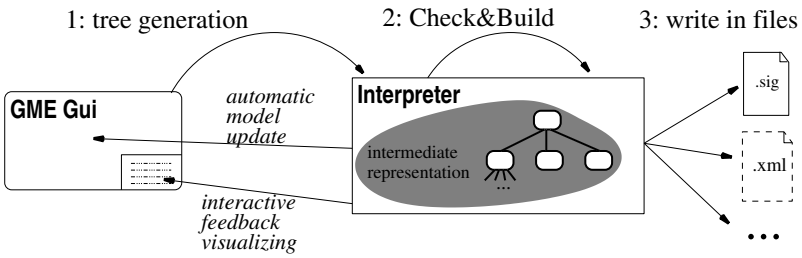


Figure 5. From GME to SIGNAL files.

same process is applied recursively on each sub-Model. For example, the instantiation of a *Module* Model results in the instantiation of its contained elements among which *ModelDeclaration*, *TypeDeclaration*, and *ConstantValue* FCOs. While *ConstantValue*s are only Atoms, and thus leafs of the tree, *ModelDeclaration*s and *TypeDeclaration*s contain subparts. In the same manner, each of these container elements recursively instantiates its own symbols and SIGNAL process models.

Step 2: Check&build. This step consists in building the SIGNAL equations for each node of the tree created at the previous step. Each Model (*ModelDeclaration*, *SubProcess*, etc.) has to build the equations corresponding to each element it contains.

To produce these equations, we need to analyze all concepts and all relations (i.e. Connections) between them. This analysis consists in visiting each node of a directed graph whose arcs are Connections. To analyze the graph, we need to select start/end points, which allows, as much as possible, to avoid visiting the same path twice. We call **end-statements** these start/end points. Basically, it corresponds to all named elements (e.g. signal, declaration of model, model instance). Actually, the analysis consists first in producing the SIGNAL code corresponding to the end-statement from which the analysis starts; then if there are specific Connections (for example *Definition*) whose source is the starting FCO, the analysis follows them in the backward direction and tries recursively to produce the SIGNAL code corresponding to the FCO(s) which is(are) the source of this(these) connection(s). The analysis is stopped when the source of a Connection is an end-statement or when an error, such as cycle or FCO without a needed Connection, is detected. More precisely, inside a Model, an equation is produced:

- for each Connection of type *Definition*, *PartialDefinition*, and *ConstraintInputs* whose destination is either a *Local* or an *Output* Atom (or *SignalRef* Reference which points to such an Atom). As shown in FIG. 4(b), taking the example of the `hour` local signal, there is a *Definition* Connection whose destination is `hour` and whose source is the *Add* Atom. Then, the analysis follows the two Connections whose destination is the *Add* Atom. The first one leads to the `One` *ConstantValue*, which is an end-statement, thus the analysis stops. The second one leads to a *Delay* Atom, thus the analysis needs to continue following the Connection linked to the *Delay* Atom. This leads finally to an end-statement, which is the `hour` local signal itself. So the produced equation is `hour := (One + (hour$(1) init (0))).`

- for each Atom representing a clock constraint or a dependence relation. For example, in FIG. 4(c), the *ClockSynchronized* Atom is the destination of two *ConstraintInputs* Connections: one from the `tick` signal and one from the `hour` signal. As result, the equation `tick ^= hour` is produced.
- for each *ModelInstance* Reference, which refers to a *ModelDeclaration* that indicates the model to instantiate. To produce an equation of a SIGNAL process instantiation, an intermediate signal is generated for each *Input/Output/Parameter* FCO of the referred Model. For each Connection to these FCOs, an equation is created using the intermediate signal as the Connection destinations.
- for each *TypeDeclaration* Model. According to the kind of type declaration, the analysis is different: for enumeration types, we use the *EnumValues* attribute in which there is one value per line; for structure types, all *Local* Atoms are listed; for model types, all *Input/Ouput/Parameter* Atoms are listed and the corresponding interface is generated; finally, for external types, the content of the *DeclaredType* attribute is used.

Input/output signals and parameters are ordered in the interface of a SIGNAL model according to the position of their corresponding Atoms in the *Interface* Aspect.

In the same step before the equation generation, some corrections could be applied to the graphical Model, for example, when a Reference points to an FCO that is not declared in the same scope as the Reference. In this situation, the properties of the corresponding graphical components are systematically updated.

As soon as an error is encountered during this second step, a message is displayed in the GME console, indicating FCOs concerned by the error as HTML links. Whenever the user clicks on a link, the corresponding graphical object is automatically displayed. This is very convenient to make rapid corrections.

```
process Watchdog =
   { integer delay; }
   ( ? integer order; event finish, tick; ! integer alarm; )
   (| alarm := (hour when (cnt = Zero))
    | hour := (One + (hour$(1) init (0) ))
    | cnt ^= (order ^+ tick ^+ finish)
    | cnt := ((delay when ^order) default (defValue when finish)
             default ((zcnt - One) when (zcnt >= Zero)) default defValue)
    | zcnt := (cnt$(1) init (-1) )
    | tick ^= hour
    |)
   where
      constant integer One = (1), defValue = (-1), Zero = (0);
      integer hour, cnt, zcnt;
   end; % process Watchdog
```

Figure 6. Code generated by the Interpreter on the watchdog example

Step 3: Dump in files. The third and last step consists in visiting one more time each node of the tree and writing the corresponding equations into destination files at the relevant place in the SIGNAL model. The declarations of signals, constants, and labels are built and added at the same time. The code of FIG. 6 corresponds to the application of our interpreter on the watchdog example described in FIG. 4.

As a global remark, we have to mention that the interpretation process can only be applied to higher-level Models. We impose this restriction in order to be sure that the selected Models do not use signals declared at an upper level in the hierarchy of a Model. So, the interpreter only generates a file for selected Models, which are immediate children of the Root Folder (i.e. the root of the current project).

Finally, we can notice that the second and the third steps can be specialized. The interpreter generates files using the SIGNAL syntax. However, it is possible to specialize the interpreter to construct equations using, for example, XML syntax.

7 Discussion

The modeling paradigm introduced in this paper constitutes the first work for generalizing the use of formal methods proposed by POLYCHRONY. This approach is developed using metamodels to achieve a relative independence from the modeling platform. The higher their abstraction expression level is, the more adaptable to various operational environments they will be. Indeed, Model Driven Software Development is based on a number of common standards such as XMI, OCL and UML, that can be mapped onto different environments. Thus, we have chosen GME to develop the metamodel, because it is indeed a good solution to create a metamodel quickly, and it offers automatically a customized modeling environment.

The metamodel combined with the Interpreter makes from Signal-Meta a new front-end for POLYCHRONY. Actually, Signal-Meta and its Interpreter check only structural information of the graphical specification, such as cyclic definitions or the well-formedness of a Model. Here, we consider a Model well-formed when the required attributes of all FCOs it contained are defined and when all required Connections are linked to these FCOs. Obviously, such a Model is only partially correct. To check the complete correctness of the Model, we have to apply the clock calculus and the type checking on the generated program. This task is devoted to the SIGNAL compiler: it would be too costly to use OCL constraints to express the clock calculus. Indeed, the clock calculus has to detect clock specification problems, and localize them. However, in GME, the check of an OCL constraint returns only true or false. Thus, to be able to localize the failure, we have to check specific FCO properties (i.e. inside a Model) one at a time. The clock calculus requires a global computation. One of our future goals is to obtain a graphical and fully interactive edition of a model under GME. Currently, the interaction is limited to structural error detections by the OCL constraints. We should extend the environment with other external components to be able to check deeper semantic errors, such as clock problems, and to display them graphically and dynamically during the edition of a model.

Anyway, to really generalize the use of formal methods, our metamodel must be accessible in more popular frameworks, such as Eclipse. The ATLAS Group from INRIA [2] has realized a bridge between GME and the Eclipse Modeling Framework (EMF) [8]. Some transformations [5] have been developed between MetaGME metamodels and EMF metamodels. However, these transformations keep only concepts and relations between them, they do not cover all features offered by GME: for

example, all informations concerning Aspects disappear. With this restriction, all metamodels realized with GME, and particularly for our interest Signal-Meta, can be transformed to metamodels under EMF. However, it is important to note that the current Interpreter uses the BON API and so is dedicated to GME. Thus, it must be specifically developed for Eclipse.

This metamodel can be considered as a first effort toward the development of a more general-purpose UML profile for modeling real-time and embedded systems, called MARTE [17]. Moreover, Signal-Meta constitutes a kernel to create environments for multi-clock systems. Signal-Meta has already been extended for different purpose. A first extension has been done to model multi-clock mode automata [7]. In this work, the automaton describes the control of the systems, and in each state of the automaton, SIGNAL equations are built in the Signal-Meta way.

Another extension, called MIMAD [6], concerns the design of avionics systems based on the Integrated Modular Avionics (IMA) architecture develop around the APEX-ARINC 653 standard. Initially, some predefined services have been implemented in a POLYCHRONY library [10]. All these services and all levels of the IMA Architecture extend Signal-Meta to build MIMAD. Each level of the IMA Architecture is represented and inherits from Signal-Meta FCOs. This inheritance allows to reuse easily features of the Interpreter. Signal-Meta is mainly used in MIMAD to represent specific user-designed functions and the data flows between the input/output signals of IMA levels. For these both extensions, the Interpreter was also extended to produce the corresponding SIGNAL program.

8 Conclusions

In this paper, we have presented Signal-Meta, the metamodel of the SIGNAL language developed in GME and its Interpreter to transform the graphical specifications into SIGNAL programs. Both tools make from GME a new front-end for the POLYCHRONY workbench. Moreover, Signal-Meta has already been used as foundations to build more specialized multi-clocked environment such as for mode automata and for avionics system design.

As discussed in Section 7, Signal-Meta and its interpreter check only structural information. There is no type checking or clock constraint verification. To complete the edition of model using Signal-Meta and to overcome these limitations, one of the possible way is to interface directly POLYCHRONY as a GME Addon. Thus, the internal representation of the SIGNAL compiler could be produced automatically during the modeling, and then give access to possible clock or type problems.

References

[1] Agrawal, A., G. Karsai and A. Ledeczi, *An end-to-end domain-driven software development framework*, in: *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, New York, NY, USA, 2003, pp. 8–15.

[2] ATLAS Group (INRIA & Lina, Université de Nantes), *ATL, ATLAS Transformation Language, Reference site*, http://www.sciences.univ-nantes.fr/lina/atl/.

[3] Besnard, L., C. Brunette, T. Gautier and J.-P. Talpin, *Modeling multi-clocked data-flow programs using the Generic Modeling Environment*, Technical Report RR-5775, INRIA (2005).

[4] Besnard, L., T. Gautier and P. Le Guernic, *SIGNAL V4-INRIA version: Reference Manual*, http://www.irisa.fr/espresso/Polychrony/doc/document/V4_def.pdf.

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

[6] Brunette, C., R. Delamare, A. Gamatié, T. Gautier and J.-P. Talpin, *A Modeling Paradigm for Integrated Modular Avionic Design*, Technical Report RR-5715, INRIA (2005).

[7] Brunette, C. and J.-P. Talpin, *Compositional modeling and transformation of multi-clocked mode automata*, Technical Report RR-5728, INRIA (2005).

[8] Eclipse Modeling Framework, *Reference site*, http://www.eclipse.org/emf/.

[9] ESPRESSO-IRISA, POLYCHRONY *website*, http://www.irisa.fr/espresso/Polychrony.

[10] Gamatié, A. and T. Gautier, *Synchronous Modeling of Modular Avionics Architectures using the* SIGNAL *Language*, Technical Report RR-4678, INRIA (2002).

[11] Institute for Software Integrated Systems (ISIS). Vanderbilt University, *The Generic Modeling Environment (GME)*, http://www.isis.vanderbilt.edu/Projects/gme/.

[12] Jackson, E. K. and J. Sztipanovits, *Using separation of concerns for embedded systems design*, in: *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software* (2005), pp. 25–34.

[13] Le Guernic, P., J.-P. Talpin and J.-C. Le Lann, POLYCHRONY *for system design*, Journal of Circuits, Systems, and Computers - Special Issue: Application Specific Hardware Design **12** (2003), pp. 261–303.

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

[15] Maraninchi, F. and Y. Rémond, *Mode-automata: a new domain-specific construct for the development of safe critical systems*, Sci. Comput. Program. **46** (2003), pp. 219–254.

[16] Marchand, H., P. Bournai, M. Le Borgne and P. Le Guernic, *Synthesis of Discrete-Event Controllers based on the* SIGNAL *Environment*, in: *Discrete Event Dynamic System: Theory and Applications*, 4 **10**, 2000, pp. 325–346.

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

[18] TOPCASED website, http://www.topcased.org.