A Modeling Paradigm for Integrated Modular Avionics Design

Christian Brunette, Romain Delamare, Abdoulaye Gamatié, Thierry Gautier, Jean-Pierre Talpin

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Modeling Paradigm for Integrated Modular Avionics Design

Christian Brunette , Romain Delamare , Abdoulaye Gamatié , Thierry Gautier ,

Jean-Pierre Talpin

N°5715

Octobre 2005

Systèmes communicants

ISSN 0249-6399 ISRN INRIA/RR--5715--FR+ENG

A Modeling Paradigm for Integrated Modular Avionics Design

Christian Brunette* , Romain Delamare† , Abdoulaye Gamatié‡ , Thierry Gautier§ , Jean-Pierre Talpin¶

Systèmes communicants
Projet Espresso

Rapport de recherche n ° 5715 — Octobre 2005 — 38 pages

Abstract: The Modeling paradigm for Integrated Modular Avionics Design (Mimad) is an extensible component-oriented framework that enables high level models of systems designed on integrated modular avionics architectures. 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. Mimad is built upon component models dedicated to avionic applications design, which are defined within the Polychrony platform. Hence, its descriptions can be transformed into Polychrony's models in order to access the available formal tools and techniques for validation. Users do not need to be experts of formal methods (in particular, of the synchronous approach) to be able to manipulate the proposed concepts. This contributes to satisfying the present industrial demand on the use of general-purpose modeling formalisms for system design. This paper first presents the main features of Mimad V0. Then, it illustrates the use of the paradigm to design a simple application example within Gme.

Key-words: Avionics design, metamodeling, Gme, synchronous languages, Signal

(Résumé : tsvp)

* christian.brunette@irisa.fr
† romain.delamare@gmail.com
‡ abdoulaye.gamatie@irisa.fr
§ thierry.gautier@irisa.fr
¶ jean-pierre.talpin@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Un paradigme de modélisation pour la conception de l'avionique modulaire intégrée

Résumé : Le paradigme de modélisation pour la conception de l'avionique modulaire intégrée (MIMAD) offre un cadre extensible orienté composant, permettant des descriptions de haut niveau de systèmes conçus sur des architectures avioniques modulaires intégrées. Il est basé sur l'environnement générique de modélisation (GME) qui est orienté objet. GME fournit un ensemble configurable d'outils permettant la création d'environnements aussi bien de modélisation pour des domaines spécifiques que de synthèse de programmes. MIMAD est construit au-dessus de modèles de composants dédiés à la conception d'applications avioniques, définis dans la plate-forme POLYCHRONY. Ainsi, ses descriptions peuvent être transformées en modèles polychrones dans le but d'accéder aux outils et techniques formels disponibles pour la validation. Les utilisateurs n'ont pas besoin d'être experts en méthodes formelles (en particulier, en approche synchrone) pour être capable de manipuler les concepts proposés. Cela contribue à la satisfaction de la demande industrielle présente par rapport à l'utilisation de formalismes généraux de modélisation pour la conception de systèmes. Ce rapport présente d'abord les principales caractéristiques de MIMAD V0. Ensuite, il illustre l'utilisation de ce dernier pour concevoir un exemple simple d'application dans GME.

Mots-clé : Conception avionique, métamodélisation, GME, langages synchrones, SIGNAL

Contents

1 Introduction

Originally inspired by concepts and practices borrowed to digital circuit design and automatic control, the *synchronous hypothesis* has been proposed in the late '80s and extensively used for embedded software design ever since to facilitate the specification and analysis of control-dominated systems. Nowadays domain-specific programming environments based on that hypothesis are commonly used in the European industry, especially in avionics, to rapidly prototype, simulate, verify and synthesize embedded software for mission critical applications.

In this spirit, synchronous data-flow programming languages, such as Lustre [12] and Signal [17], implement a model of computation in which time is abstracted by symbolic synchronization and scheduling relations to facilitate behavioral reasoning and functional correctness verification. In the case of the Polychrony toolset, on which Signal is based, 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 relation to gradually refine its synchronization and scheduling structure to meet the target architecture's specified requirements. 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 of choice.

These design principles have been put to work in the context of the European IST projects Sacres and SafeAir with the definition of a methodology for the formal design of avionic applications based on the Integrated Modular Avionics (IMA) model [10]. In this approach, the synchronous paradigm is used to model components and describe the main features of IMA architectures by means of a library of APEX/ARINC-653 compliant generic RTOS component models. The IMA library is available together with the experimental Polychrony toolset [9], has been commercialized by TNI-Valiosys' toolset RT-Builder and successfully used at Hispano-Suiza, Airbus Industries and MBDA for rapid prototyping and simulation of avionics architectures.

In the context of the Airbus Industries TOPCASED initiative [27], our objective is to bring this technology in the context of model-driven engineering environments such as the Gme and of the Uml in order to provide engineers with better ergonomy and higher-level design abstraction facilities. To meet this objective, we aim at bringing the Polychrony's IMA library in the *General Modeling Environment* (Gme) [15], which is based on an object-oriented approach very close to the UML. In this paper, we propose a *modeling paradigm* called Mimad that allows engineers to design avionic applications on IMA architectures. The Mimad paradigm consists of a set of basic constructs together with their associated combination rules allowing to describe IMA concepts. In Gme, the specification of the modeling paradigms is achieved through *metamodels*, which are abstract, high-level descriptions

whose semantics depends on the context in which they are used.

The remainder of the paper is organized as follows: Section 2 first introduces the IMA architectural concepts; then, Section 3 briefly present the SIGNAL language and its environment POLYCHRONY. Next, Section 4 gives an overview of the generic modeling environment. Section 5 presents the main features of the MIMAD paradigm and the ongoing efforts about its implementation and Section 6 illustrates the use of MIMAD to model a simple avionic application [11]. The adopted approach is discussed in Section 7 and finally, conclusions are given in Section 8.

2 Integrated modular avionics

IMA [1] [2] is the recent architecture proposed for avionic systems in order to reduce the design cost inherent to traditional *federated architectures*, which are still widely adopted in modern aircrafts. The basic principle of IMA is that several functions (even of different criticality levels) can share common computing resources. This is not the case in federated architectures where each function executes exclusively on its dedicated computer system. While this favors fault containment, it is penalizing due to high price, maintenance costs, 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 a number of *modules* grouped in cabinets throughout the aircraft. A module can contain several partitions that possibly belong to applications of different criticality levels. 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*. In the former, no message queue is allowed. A message remains in the source port until it is transmitted via the channel or it is overwritten by a new occurrence of the message. A received message remains in the destination port until it is overwritten. A refresh period attribute is associated with each sampling port. When reading a port, a *validity* parameter indicates whether the age of the read message is consistent with the required refresh period attribute of the port. In the queuing mode, ports are allowed to store messages from a source partition in FIFO queues until their reception by the destination partition.

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 dead-

line 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. Synchronizations are achieved using a *semaphore*.

The APEX-ARINC 653 standard [2] defines an interface allowing IMA applications to access the 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, partition, processes, and time management services.

3 The synchronous language SIGNAL and POLYCHRONY

SIGNAL [18] is a declarative synchronous data-flow language dedicated to the design of embedded systems for critical application domains such as avionics and automotive. Its associated development environment POLYCHRONY [9] offers several tools composed of the SIGNAL batch compiler providing a set of functionalities, such as program transformations, optimizations, formal verification, and code generation, a graphical user interface, and the SIGALI model checker [21], which enables both verification and controller synthesis.

The SIGNAL language handles unbounded series of typed values $(x_t)_{t\in\mathbb{N}}$, called *signals*, denoted as `x` in the language, and implicitly indexed by discrete time (denoted by t in the semantic notation). At a given instant, a signal may be present, at which point it holds a value; or absent, at which point it is denoted by the special symbol $\perp$ in the semantic notation. The set of instants where a signal `x` is present is called its *clock*. It is noted as `^x`. Signals that have the same clock are said to be *synchronous*. A SIGNAL *process* is a system of equations over signals that specifies relations between values and clocks of the involved signals. A *program* is a process. SIGNAL relies on a handful of primitive constructs, which are combined using a composition operator. These core constructs are of sufficient expressive power to derive other constructs for comfort and structuring. In the following, we give a sketch of primitive constructs by mentioning the syntax and the corresponding definition:

Functions/Relations:
`y:= f(x1,...,xn)` $\stackrel{def}{=} y_t \neq\perp\Leftrightarrow x1_t \neq\perp\Leftrightarrow ... \Leftrightarrow xn_t \neq\perp$, $\forall t$: $y_t = f(x1_t, ..., xn_t)$.

Delay:
`y:= x $ 1 init c` $\stackrel{def}{=} x_t \neq \perp \Leftrightarrow y_t \neq \perp,\ \forall t > 0$: $y_t = x_{t-1}, y_0 = c$.

Down sampling:
`y:= x when b` $\stackrel{def}{=} y_t = x_t$ if $b_t = true$, else $y_t = \perp$.

Deterministic merging:
`z:= x default y` $\stackrel{def}{=} z_t = x_t$ if $x_t \neq \perp$, else $z_t = y_t$.

Parallel composition:
`(| P | Q |)` $\stackrel{def}{=}$ union of equations associated with `P` and `Q`.

Hiding:
`P where x` $\stackrel{def}{=}$ `x` is local to the process `P`.

SIGNAL provides a process frame (see FIG. 6) in which any process may be "encapsulated". This allows to abstract a process to an interface, so that the process can be used afterwards as a black box through its interface which describes the input-output signals and parameters. The frame also enables the definition of sub-processes. Sub-processes that are only specified by an interface without 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.

The mathematical foundations of SIGNAL enable formal verification and analysis techniques. We can distinguish two kinds of properties: *functional* and *non functional* properties. Functional properties consist of *invariant properties* on the one hand (e.g. determinism, absence of cyclic definitions, absence of empty clocks to ensure a consistent reactivity of the program), and *dynamic properties* on the other hand (e.g. reachability, liveness). The SIGNAL compiler addresses only invariant properties, while dynamic properties (those that can be expressed with a finite automaton) are checked with SIGALI. Non functional properties include *temporal properties* that are of high interest for real-time systems. A technique has been defined in order to allow timing analysis of SIGNAL programs [16]. Basically, it consists of formal transformations of a program initially describing an application, which yield another SIGNAL program that corresponds to a *temporal interpretation* of the initial one. The new program will serve as an *observer* [13] of the initial program.

4 The generic modeling environment

GME is a configurable UML-based toolkit that supports the creation of domain-specific modeling and program synthesis environments [19]. It is developed by the ISIS institute at Vanderbilt University, and is freely available at [15]. Metamodels are proposed in the envi-

ronment 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.

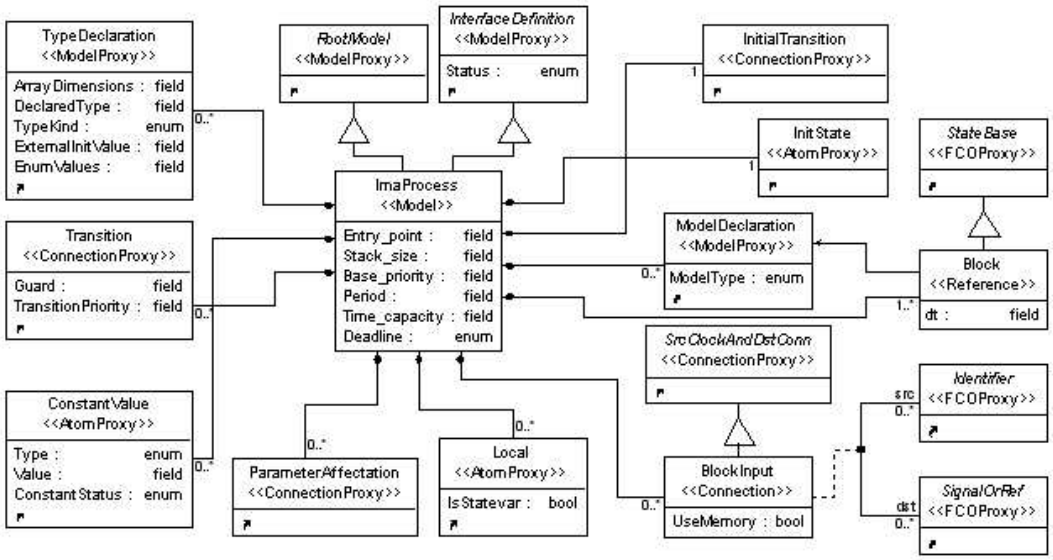

Figure 1: MIMAD metamodel: the ImaProcess class diagram.

Using GME is quite simple. 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 habitual UML class diagrams. To realize these class diagrams, MetaGME offers some predefined UML-stereotypes [14]: *First Class Object (FCO)*, *Atom*, *Model*, *Reference*, *Connection*, etc. *FCO* constitutes the basic stereotype in the sense that all the other stereotypes inherit from it. It is used to represent abstract concepts (represented by classes). *Atoms* are elementary objects in the sense that they cannot include any sub-part contrarily to *Models* that may be composed of various FCOs. This *containment* relation is characterized on the class diagram by a link ending with a diamond on the container side. Such a link is shown in FIG. 1 for example between the *Local* atom and the *ImaProcess* 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 *Block* reference points to a Model of type *ModelDeclaration*). Inheritance relations are represented as in UML. All the other types of relationship are specified through various kinds of *Connection*. In addition, each stereotype "X" is associated with another stereotype called "XProxy", which allows to reference any FCO within the metamodel. This allows to describe the metamodel in a modular way.

Thus, a user can specify several class diagrams and reuse a concept created in one of these diagrams in all others. For example, the *Local* AtomProxy (see FIG. 1) refers to the *Local* Atom defined in another class diagram.

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). Finally, some *OCL Constraints* can be added to class diagrams in order to check some dynamic properties on a model designed with this paradigm (e.g. the number of allowed connections associated with a component model). The whole above concepts constitute the basic building blocks that are used to define modeling paradigms in GME.

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 interface. The interface represents the link between the GME GUI and the programs executed by the component (e.g. executing a function in external libraries). It can be generated in C/C++ or JAVA. In C++, the interface can use the low-level COM language or the *Builder Object Network* (BON) API [19]. GME distinguishes three families of components that can be plugged to its environment: *Interpreter*, *Addon*, and *PlugIn*.

- The role of an Interpreter is to check information, such as the correctness of a model, and/or produce a result, such as a description file. It is the case for the MetaGME Interpreter. An interpreter is applied on user demand and has a punctual execution. Further details on the MIMAD Interpreter are given in Section 5.3.
- Contrarily to the Interpreter, an Addon is executed as soon as a project is opened, and it works throughout the graphical modeling. An Addon reacts to specific events sent by GME. The GME *Constraint checker* is an example of an Addon. During the description of models, it checks each OCL constraint specified in the used paradigm whenever events to which they relate are emitted by GME.
- 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. For example, the *Auto-Layout*

PlugIn provided with GME has to set the position of each graphical entity to minimize the number of link intersections and to improve the readability of the selected model.

In the next section, we describe the MIMAD modeling paradigm defined in GME and its interpreter.

5 A Modeling Paradigm for IMA Design

To present[1] the MIMAD paradigm, we first expose the overall approach (Section 5.1). We show how MIMAD can be combined with other design frameworks in general, and POLYCHRONY in particular. Then, we discuss the modeling of the basic components that enable to describe applications following the IMA architecture (Section 5.2). Finally, we focus on the generation of SIGNAL descriptions from GME models using MIMAD (Section 5.3).

5.1 Overall Approach

FIG. 2 illustrates our intended approach. 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 particularly consider the POLYCHRONY environment. However, one can observe that the approach is extensible to other technologies or models of computation (represented by the dots in the figure) 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. Moreover, this favors a high flexibility and interoperability of the approach.

GME also provides specific facilities that enable to connect new environments to its associated platform. This possibility permits to implement the code generation directly from GME models without exporting them in XML (see Section 5.3). It also facilitates the interactive dialog between GME and the connected environments.

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. 2) 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 of the initial application description. This is achieved by using the IMA-based components already defined in POLYCHRONY [10].

[1] More details can be found in the annex (see Sections A and B).

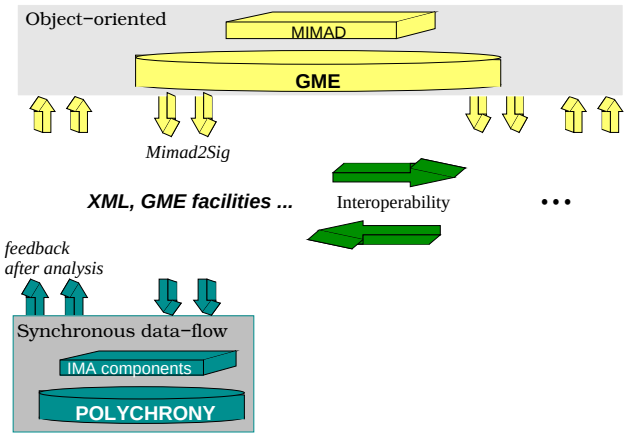

Figure 2: Our overall approach.

Thereon, the formal analysis and transformations techniques available in the platform can be applied to the generated SIGNAL specification. Finally, a feedback is sent to the object-oriented layer to notify the user about possible incoherences in initial descriptions.

We can observe that without being an expert of synchronous technologies, a user can design applications based on the IMA modeling approach proposed in POLYCHRONY. The next Section focuses on the definition of MIMAD.

5.2 Definition of Basic Components

Our reference model of a system description on an IMA architecture (see Section 2) is depicted in FIG. 3. Basically, a system is composed of several modules. Each module is itself formed of partitions whose execution is under the control of the module level OS (also part of a module). A partition contains processes associated with the partition level OS, which is responsible of the correct execution of processes within that partition. Finally, a process consists of its control part, which triggers blocks of actions (OS functionalities called via APEX-ARINC services or other functions) specified in the computation part of the process.

MIMAD is built as an extension of Signal-Meta [7], which is the metamodel designed for the SIGNAL language. Signal-Meta class diagrams describe all the syntactic elements defined in SIGNAL V4 [5]. Among the described concepts, Atoms are associated with SIGNAL operators (e.g. arithmetic operators, clock relations), Models specify SIGNAL *containers* (e.g. process frame, module), and Connections describe relations between SIGNAL operators (e.g. definition, dependency). Signal-Meta comprises three main Aspects: *Interface*, *Computation part* and *Clock and Dependence Relations*. The first Aspect manages all input/output

signals and static parameters. The second one shows all data-flow relations between Models. The last one reflects all clock relations between signals.

Concerning the part specific to the IMA architecture, the reference model conforms to the decomposition we adopted in [10]. Its description in GME is done in a modular way. Each level of the MIMAD paradigm is modeled by a class diagram and inherits from the *InterfaceDefinition* and *RootModel* Models of Signal-Meta. The first inheritance means that, *ImaSystem*, *ImaModule*, *ImaPartition*, and *ImaProcess* (see FIG. 1) Models can contain *Input*, *Output*, and *Parameter* Atoms. The second inheritance expresses that Models can be added as children of the *Root Folder* (the root of a project within GME).

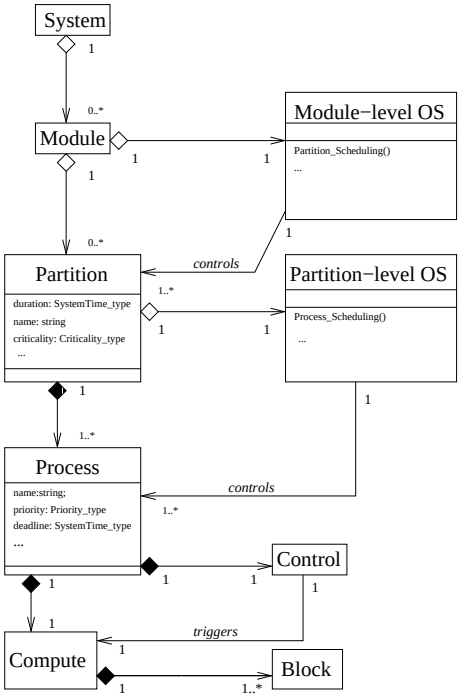

Figure 3: A reference model of IMA systems.

In addition, at each level, the corresponding Model contains the sub-level Model as illustrated in FIG. 3. *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. 1. It contains *Block* References, which refer to APEX-ARINC services or other functions defined by the user in a *ModelDeclaration* Model defined in Signal-Meta. The control and computation parts of an IMA process Model are separated into two Aspects.

In the computation part, Connections between inputs and outputs of *Block* are explicitly described. The control part is represented by a *mode automaton* [20]. Basically, such an automaton is characterized by a finite number of states, referred to as modes. A mode can be associated with one or more actions to be achieved (one can make an analogy between modes and tasks). Modes get activated on the occurrence of some events. At each moment, the automaton is in one (and only one) mode. Therefore, actions associated with this mode can be achieved. In the case of the modeling of basic components for MIMAD, each *Block* represents a state of the mode automaton while guarded *Transitions* realize the connections between *Blocks*. These aspects are illustrated in FIG. 9 and 10 for the ON_FLIGHT application example (Section 6).

To complete the metamodel, we have to define models associated with APEX-ARINC services, which are required to describe communications and synchronization between processes and partitions, time management, scheduling issues. For illustration, let us consider the *read_ blackboard* service as depicted in FIG. 4. The service is modeled, at this stage, by its interface: *blackboard_ ID* and *timeout* are Inputs while *message*, *length* and *return_ code* are Outputs; *process_ ID* is a Parameter of the service Model (required whenever the calling process gets suspended [10]); finally, the SPEC Model of the interface enables to specify properties between Inputs, Outputs and Parameters. Otherwise stated, APEX-ARINC services are represented as *black box* abstractions in the object-oriented layer. The implementation of a service is described when moving to lower layers (e.g. when generating the SIGNAL code associated with a Model - see Section 5.3).

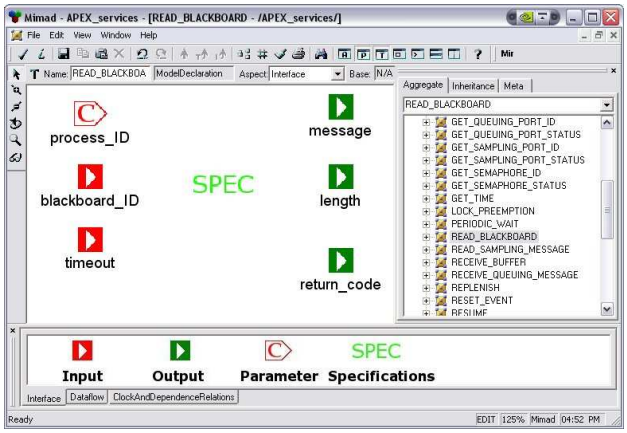

Figure 4: Interface of *read_ blackboard* in GME.

The overall metamodel results from the above component models: IMA architectural elements (process, partition, etc.) and APEX-ARINC services. In fact, the MIMAD meta-

model is built on a metamodel for the SIGNAL language. *ModelDeclaration* Model and *Local* Atom (see FIG. 1) are examples of concepts issued from this metamodel.

5.3 From GME to SIGNAL

As mentioned previously in Section 5.1, there are two possibilities for the transformation of models from the upper layer to lower layers. Here, we consider the second possibility, which consists in directly generating SIGNAL files using the specific facilities offered in GME. In the following, we expose the generation of SIGNAL code from GME Models, as implemented in the current version of MIMAD. We have developed an interpreter that generates a SIGNAL program corresponding to the Model described in the GME environment using the MIMAD paradigm. This interpreter uses the BON API (see Section 4). FIG. 5 summarizes the different steps performed by the MIMAD interpreter.

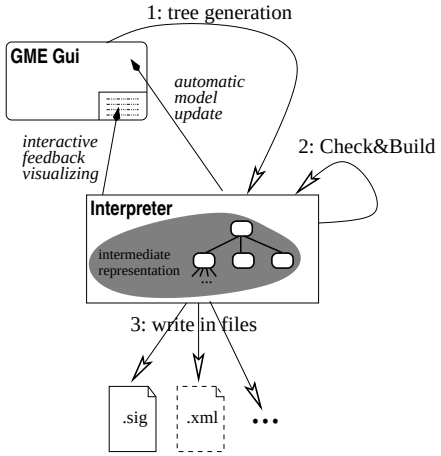

Figure 5: Generation of SIGNAL models from GME.

There are three main steps in the interpretation of a MIMAD Model.

1. **Tree generation.**

   Each FCO selected in the GME GUI is associated to 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 future generated program. The tree is built by recursive instantiations of each node into BON objects 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 same process is applied recursively on each sub-Models. For example, the instantiation of an *ImaPartition* Model results in the instantiation of its contained elements: *ImaProcess*, *PartitionLevelOS*, and intra-partition mechanisms, which are *BlackBoard*, *Buffer*, *Event*, and *Semaphore*. All these elements are represented by atoms except *ImaProcess*, which is a Model. Finally, in the same manner, each of these elements recursively initializes its own fields.

2. **Check&Build.**

   This step consists in building the inner SIGNAL equations of each node of the tree created at the previous step. Each Model (*ImaPartition, ImaProcess*, etc.) has to build the equations corresponding to each element it contains. We do not give details here. Instead, we illustrate the code generation on the example described in the next section.

   So, let us consider the code represented in FIG. 6. The interface of the corresponding SIGNAL process, named `COMPUTE`, is composed of an input signal `active_block` and two outputs `ret` and `dt`. This description is a partial view of the SIGNAL code corresponding to the computation part of an IMA process as defined in [10]. It has been generated automatically from the MIMAD description depicted in FIG. 9. The names of blocks present in the computation part (e.g. `Send_Buffer2`, `Compute_Position1`) are initially used to create an enumerated type for the input `active_block`, which is produced by the control part of an IMA process. The value of this signal corresponds to the current state of the mode automaton encoding the control part (see FIG. 10). The code associated with each state is the instantiation of the Model referred by the Block (APEX-ARINC services or user functions). For all interface signals (Input, Output, and Parameter) of a Block, an intermediate signal is created. The equations corresponding to these intermediate signals are not given in FIG. 6. They are defined using information obtained from the Connection attributes (e.g. the *UseMemory* attribute of the *BlockInput* Connection as illustrated in FIG. 1) and the FCOs on both sides of the Connection.

   The value of the Output `ret` indicates the returned code of the executed APEX-ARINC services. The other Output denoted by `dt` provides the execution duration of each activated Block, which is specified in an attribute of the Block.

   In the same step, some corrections could be applied to the graphical Model, for example, when a Reference points to an FCO, which 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 the FCOs concerned by the error. The concerned FCOs are displayed as HTML links. Whenever the user clicks on a link, the corresponding

```
process COMPUTE =
   ( ? block_enum_type active_block;
     ! ReturnCode_type ret;
       SystemTime_type dt; )
   (| case active_block in
      {#Send_Buffer2}: (| Id0 := SEND_BUFFER{Id1}(Id2,Id3,Id4,Id5) |)
      {#Compute_Position1}: (| (Id6,Id7) := COMPUTE_POS{}(Id8,Id9,Id10,Id11) |)
      {#Read_BlackBoard1}: (| (Id12,Id13,Id14) := READ_BLACKBOARD{Id15}(Id16,Id17)|)
      {#Wait_Event1}: (| Id18 := WAIT_EVENT{Id19}(Id20,Id21) |)
      {#Send_Buffer1}: (| Id22 := SEND_BUFFER{Id23}(Id24,Id25,Id26,Id27) |)
      {#Set_Date1}: (| (Id28,Id29) := SET_DATE{}() |)
      end
    | ret:= Id0 default Id4 default Id8 default Id12 default Id1
    | zblock := active_block$
    | dt :=    2 when zblock = #Send_Buffer2
       default 3 when zblock = #Compute_Position1
       default 2 when zblock = #Read_BlackBoard1
       default 2 when zblock = #Wait_Event1
       default 2 when zblock = #Send_Buffer1
       default 2 when zblock = #Set_Date1
    | ...
    |)
   where
      block_enum_type zblock;
   end; %process COMPUTE%
```

Figure 6: A SIGNAL code example generated from the MIMAD Model of the computation part of an IMA process.

graphical object is automatically displayed. This is very convenient to make rapid corrections.

3. **Write in output files.**

   The third and last step consists in visiting one more time each node of the tree and in writing the corresponding equations into destination files.

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 opened in GME).

Finally, we can notice that the second and the third steps can be specialized. The interpreter generates files using SIGNAL syntax. However, it is possible to specialize the interpreter to construct equations using, for example, XML syntax. This provides another way to use XML as intermediate representation. Further details about MIMAD definition are given in the annex (see Sections A and B).

6 An Example

The example considered here illustrates how to model a simple avionic application within GME using MIMAD. This application, called ON_FLIGHT, has been specified and already modeled in SIGNAL [11].

The entire ON_FLIGHT application is represented by a single partition. Its main function consists in computing information about the current position of an airplane and its fuel level. Then, a report message is produced, which contains the information. ON_FLIGHT is decomposed into three processes (see FIG. 7):

- The POSITION INDICATOR that first produces the report message, which is updated with the current position information (height, latitude, and longitude).
- The FUEL INDICATOR that updates the report message (produced by the POSITION INDICATOR) with the current fuel level information.
- The PARAMETER REFRESHER, which refreshes all the global parameters used by the other processes in the partition.

Sections 6.1 and 6.2 roughly describe how the main parts of the application are modeled using MIMAD.

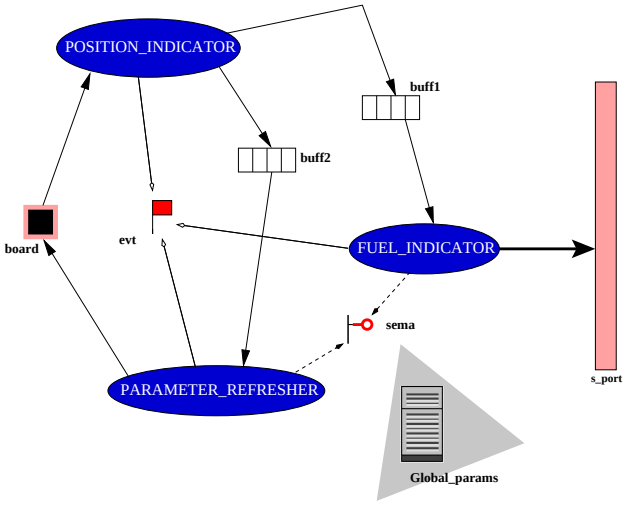

Figure 7: The ON_FLIGHT partition.

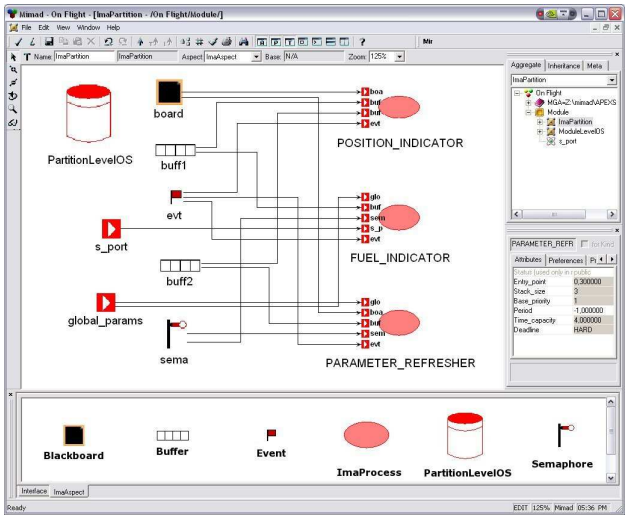

Figure 8: A Mimad Model of ON_FLIGHT.

6.1 Partition Level Design

The partition is composed of two views, also referred to as Aspects. The first one specifies the interface: Inputs, Outputs, Parameters and interface properties described in the SPEC Model. The second Aspect, termed ImaAspect, specifies the architecture of applications based on IMA concepts. Although the interface remains visible in the ImaAspect, it can only be modified in the interface aspect. The constituent elements of the ImaAspect are the following (see FIG. 8, bottom frame): the partition level OS, processes and mechanisms required for communication and synchronization between processes.

The partition level OS 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 that at this stage, 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 6.2 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 creation. Finally a type can be specified with an Atom *TypeDeclaration*, in case one needs to share a value of that type between the processes.

The Model of the partition ON_FLIGHT is shown in FIG. 8. The partition contains three processes (POSITION_INDICATOR, FUEL_INDICATOR and PARAMETER_REFRESHER), five mechanisms, and the partition level OS. Among the communication and synchronization mechanisms, there are a Blackboard `board`, two Buffers `buff1` and `buff2`, an Event `evt`, and a Semaphore `sema`. The global parameters required by the partition are represented by the Input `global_params`. Finally, the Input `s_port` identifies a sampling port via which communications are achieved between different partitions (this input is assumed to be created at the module level).

6.2 Process Level Design

To illustrate the process level design we focus on the process POSITION_INDICATOR. In [10], the SIGNAL model of an IMA process consists of two sub-parts: a *control* part that selects a sub-set of actions, called block, to be executed, and a *compute* part, which is composed of blocks. In MIMAD, IMA processes are designed in a slightly different way, using three aspects: the Interface as in the partition level, the ImaAspect, which includes the computation sub-part, and the ImaProcessControl containing the control flow of the process.

The computation part (see FIG. 9) is containing Blocks, which are References to *ModelDeclaration* specified in the same Aspect or in a library (e.g. APEX-ARINC services). It

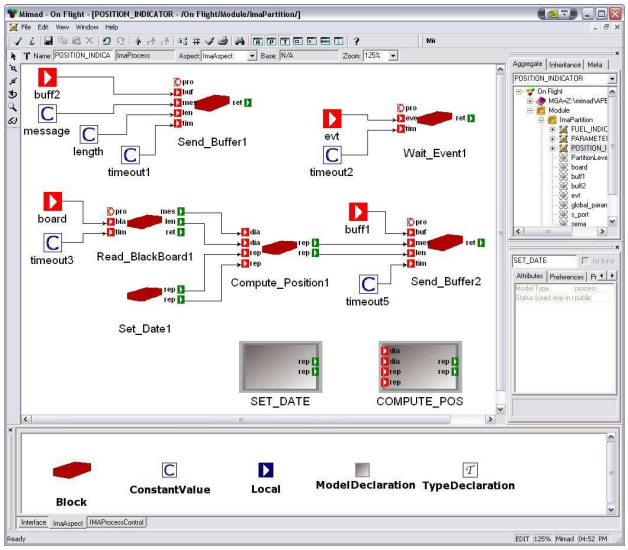

Figure 9: The computation sub-part of POSITION_INDICATOR (ImaAspect).

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.

Finally, the control part of the process is described by the mode automaton depicted by FIG. 10. At each activation, the current state of this automaton indicates which Block of actions must be executed in the process.

From the above MIMAD descriptions, a corresponding SIGNAL code can be automatically generated as illustrated in FIG. 6 (see Section 5.3). Then, the functionalities of POLYCHRONY can be used in order to formally analyze the application model.

7 Discussion

The central feature of the modeling paradigm introduced in this paper is to allow embedded system designers and engineers to describe both the system architecture and functionalities based on platform-independent models, within the component-oriented design framework MIMAD, dedicated to integrated modular avionics. Simulation code (C, C++, or JAVA) can directly be generated from these specifications (or translations) to specific formalisms such as the behavioral notations of the UML. MIMAD relies on the domain-specific language SIGNAL and its associated development environment POLYCHRONY for the description, re-

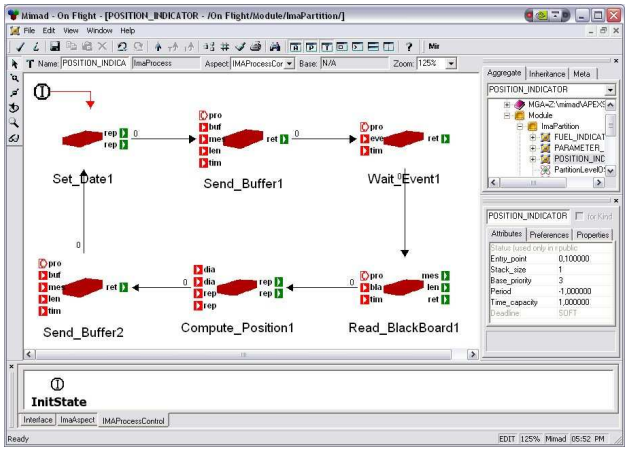

Figure 10: The control sub-part of POSITION_INDICATOR (ImaProcessControl).

finement and formal verification of the application.

MIMAD is an open modeling framework that ideally complements more general-purpose UML profiles such as the AADL [25] or MARTE [24] with an application-domain-specific model of computation suitable for trusted avionics architecture design. It is equally 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 the temporal property verifiers such as KRONOS [28] or UPPAAL [4]. In the context of GME, abstractions and refinements from and to the metamodel are best considered under the concepts and mechanisms of model transformation.

From the above observations, we believe that the approach promoted by MIMAD favors Model-Driven Engineering [6]. The assessment of MIMAD can be done with respect to the following criteria, which are highly desirable in an efficient design approach:

Usability. A great advantage of GME is that it is quite simple to be used for modeling. Its graphical user interface facilitates a component-based design by just dragging and dropping predefined components. This contributes to making MIMAD very intuitive to use. As a result, a user is not required to have strong skills in synchronous programming to be able to design models.

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

Portability. GME's descriptions are fully platform-independent. They can be automatically exported using an XML representation. This inherent feature of the GME environment enables, depending on the target platform, different implementations of the same application model defined using MIMAD.

Analyzability. The key properties of application models designed with MIMAD are those addressed by tools available in the lower layers (see FIG. 2): functional properties (e.g. safety, liveness) and non functional properties (e.g. response times). Currently, the SIGNAL code generated from GME Models can be analyzed using the tools provided in the POLYCHRONY platform. Static properties can be checked with the compiler while dynamic properties are addressed using the model-checker SIGALI [21].

Scalability. GME plays an important role in the scalability of MIMAD. And 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 of 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.

Finally, we can mention a few studies that are close to our work. Among these, the ATLAS Model Management Architecture (AMMA) [3], which has been defined on top of the Eclipse Modeling Framework (EMF) [8], another MDE platform. AMMA allows to interoperate with different environments by extending the facilities offered by EMF. Our approach also promotes interoperability by exploiting the possibility of generating, from GME descriptions, XML files as intermediate representation (see FIG. 2). There are also several studies that are specifically based on GME [23] [22] [26]. In [23], the domain-specific modeling facilities provided by GME are applied to define a visual language dedicated to the description of instruction set and generation decoders, while in [22], authors define a visual modeling environment (called EWD) where multiple models of computation together with their interaction can be captured for the design of complex embedded systems. In [26], GME is rather used to teach the design of domain-specific modeling environments. The MIMAD framework shares similar features with the last three studies: on the one hand, it proposes visual components allowing to describe both IMA and SIGNAL concepts, and on the other hand, it could be used to teach an IMA-based design as well as synchronous programming.

8 Conclusions

We have presented the definition of a modeling paradigm, called MIMAD, for the design of IMA systems. One major goal is to provide developers with a practical and component-based modeling framework that favors rapid prototyping for design exploration. This is to answer

to a growing industry demand for higher levels of abstraction in the system design process. MIMAD also aims at formal validation by reusing results from our previous studies on IMA design using the synchronous approach. We are still working on the MIMAD metamodel itself by testing some examples.

As for the POLYCHRONY environment, we plan to make the resulting modeling framework freely available to users. The inherent flexibility of the adopted approach makes the MIMAD framework extensible to other environments such as those based on timed automata to allow, for instance, temporal analysis. This represents an important perspective of the work exposed in this paper.

References

[1] Airlines Electronic Engineering Committee. ARINC report 651-1: Design guidance for integrated modular avionics. In *Aeronautical radio, Inc., Annapolis, Maryland*, November 1997.

[2] Airlines Electronic Engineering Committee. ARINC specification 653: Avionics application software standard interface. In *Aeronaut. radio, Inc., Annapolis, Maryland*, January 1997.

[3] ATLAS Group. Amma reference site. `http://www.sciences.univ-nantes.fr/lina/atl/AMMAROOT`.

[4] J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal - a tool suite for automatic verification of real-time systems. In *4th DIMACS Workshop on Verification and Control of Hybrid Systems, New Brunswick, New Jersey*, pages 22–24, October 1995.

[5] L. Besnard, T. Gautier, and P. Le Guernic. SIGNAL v4 - INRIA version: Reference manual. `http://www.irisa.fr/espresso/Polychrony`, May 2005.

[6] J. Bézivin. In search of a basic principle for model-driven engineering. *Novatica Journal*, V(2):21–24, April 2004.

[7] C. Brunette. Modeling SIGNAL programs using the Generic Modeling Environment. Technical report, IRISA/INRIA Rennes, 2005. (To appear soon).

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

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

[10] A. Gamatié and T. Gautier. Synchronous modeling of avionics applications using the SIGNAL language. In *Proc. of the 9th IEEE Real-time/Embedded technology and Applications symposium (RTAS'03)*. Washington D.C., USA, May 2003.

[11] A. Gamatié, T. Gautier, and L. Besnard. Modeling of avionics applications and performance evaluation techniques using the synchronous language SIGNAL. In *proceedings of Synchronous Languages, Applications, and Programming (SLAP'03)*. Portugal, July 2003.

[12] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language LUSTRE. In *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, Septembre 1992.

[13] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In *Algebraic Methodology and Software Technology*, pages 83–96, 1993.

[14] ISIS, Vanderbilt University. The GME user manual. `http://www.isis.vanderbilt.edu/Projects/gme/GMEUMan.pdf`.

[15] ISIS, Vanderbilt University. The GME website. `http://www.isis.vanderbilt.edu/Projects/gme`.

[16] A. Kountouris and P. Le Guernic. Profiling of SIGNAL Programs and its Application in the Timing Evaluation of Design Implementations. In *IEE Colloq. on HW-SW Cosynthesis for Reconfigurable Systems, IEE*, pages 6/1–6/9. HP Labs, Bristol, UK, Feb 1996.

[17] P. Le Guernic and T. Gautier. Data-flow to von Neumann: the signal approach. In *Advanced Topics in Data-Flow Computing, J.-L. Gaudiot and L. Bic, Eds, Prentice-Hall*, pages 413–438, 1991.

[18] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. In *IEEE, 79(9)*, pages 1321–1336, Sep 1991.

[19] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The Generic Modeling Environment. In *Proc. of the IEEE Workshop on Intelligent Signal Processing (WISP'01)*, May 2001.

[20] F. Maraninchi and Y. Rémond. Mode-automata: About modes and states for reactive systems. In *proceedings of European Symposium On Programming, Lisbon, Portugal, Springer-Verlag*, March 1998.

[21] H. Marchand, 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, 10(4)*, pages 325–346, October 2000.

[22] D. Mathaikutty, H. Patel, S. Shukla, and A. Jantsch. EWD: A Metamodeling Driven Customizable Multi-MoC System Modeling Environment. Technical Report 2004-20, FERMAT lab - Virginia Polytechnic Institute and State University, 2004.

[23] T. Meyerowitz, J. Sprinkle, and A. Sangiovanni-Vincentelli. A visual language for describing instruction sets and generating decoders. In *Proc. of the 4th ACM OOPSLA Workshop on Domain Specific Modeling, Vancouver, BC*, pages 23–32, October 2004.

[24] OMG. Uml profile for modeling and analysis of real-time and embedded systems (marte), omg document realtime/05-02-06.

[25] Society of Automotive Engineers Standard. Sae aadl information site. `http://www.aadl.info`.

[26] J. Sprinkle, J. Davis, and G. Nordstrom. A paradigm for teaching modeling environment design. In *Proc. of the OOPSLA'04 Educators Symposium (Poster Session), Vancouver, BC*, pages 24–28, October 2004.

[27] TOPCASED website. `http://www.topcased.org`.

[28] S. Yovine. KRONOS: A verification tool for real-time systems. In *Software Tools for Technology Transfer, 1(1+2)*, pages 123–133, December 1997.

Annexes

A Description of the MIMAD metamodel

In this Section, we describe the class diagrams added to Signal-Meta in order to define the MIMAD metamodel. For each class diagram, we briefly specify the corresponding OCL constraints and explain the Signal-Meta concepts that are reused in these classes. These OCL constraints are checked during the definition of any MIMAD-based application model by the *Constraint Checker* provided with GME.

Here, an OCL constraint is characterized by a textual description (allowing to precise more information about the constraint), an equation, an event, a priority and a depth. The equation corresponds to an invariant property of the Model, which must hold during the whole design phase. If one or more events are specified in a constraint, the associated equation is checked whenever the events are produced by the GME environment. Examples of events are *On create*, *On delete* and *On connect*. When no event is specified, the constraint is only checked on user demand. The priority specified on a constraint is an integer: from 1 (for the highest) to 10 (for the lowest). Finally, events can be produced by constrained FCOs or from their descendant FCOs (e.g. this is the case for Models, which may include other FCOs). The depth characterizes the sensitiveness of the constraint: 0 for events from the directly concerned FCO, 1 for events from the FCO or any of its direct descendant and *Any* for events from any of its descendant.

A.1 IMA model

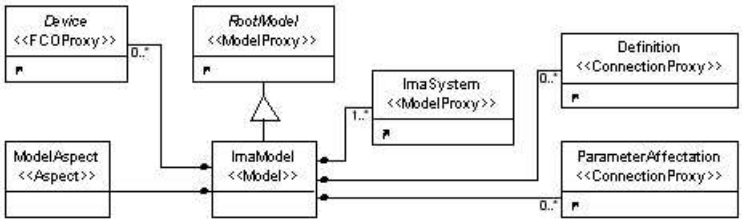

Figure 11: IMA model paradigm sheet.

The IMA model level allows to compose different IMA systems (see Section A.2). In addition, some devices (see Section A.6 in annex) can be specified at this level. It is also the case at the IMA system and module levels. While devices appear in MIMAD V0 basic concepts, they will be fully operational only in the next versions of MIMAD.

In FIG. 11, we can distinguish some FCOs from Signal-Meta, which are used at different levels in the MIMAD metamodel. On the one hand, *RootModel* is an abstract concept representing FCOs, which can be added as children of the *Root Folder* (the root of a project within GME). Note that the MIMAD interpreter can only be applied on FCOs, which inherit from *RootModel*. On the other hand, connections *Definition* and *ParameterAffectation* realize the links between respectively inputs/outputs, and parameters of different IMA design levels (e.g. at the IMA model level, they enable to connect input/output signals, and parameters of different IMA systems).

The IMA model has only one Aspect for visualization, called *ModelAspect*. This Aspect includes all the elements at this level.

A.2 IMA system

The IMA system level allows to compose different IMA modules (see Section A.3). Its structure is approximately the same as for the IMA model level, except that an *ImaSystem* also inherits from the Signal-Meta *InterfaceDefinition* concept. This concept is an abstract Model representing all Models that have an *Interface* Aspect, i.e. Models in which inputs/outputs, static parameters, and a *Specification* Model can be added.

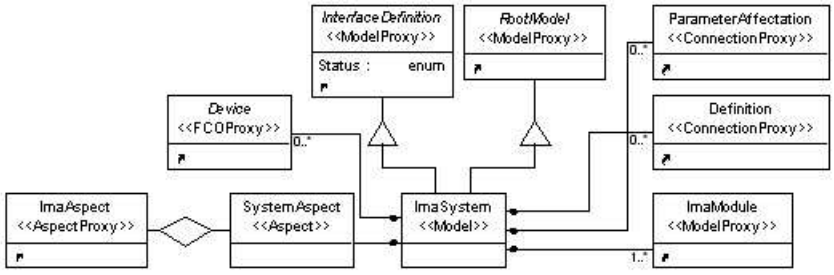

Figure 12: IMA system paradigm sheet.

Another Aspect, called *ImaAspect*, is associated with IMA systems. It allows to visualize all the elements specified in FIG. 12, including inputs/outputs, and static parameters added in the *Interface* Aspect. Thus, *ImaAspect* is used to connect the inputs/outputs and parameters of an IMA system and those of the contained IMA modules.

A.3 IMA module

The IMA module level allows to compose different IMA partitions (see Section A.4). Roughly, on the left half part of FIG. 13, one can observe that the structure of the diagram is close to the digram for IMA systems (see FIG. 12). The main difference is that IMA modules must contain a *ModuleLevelOS*, and they can also include type declarations.

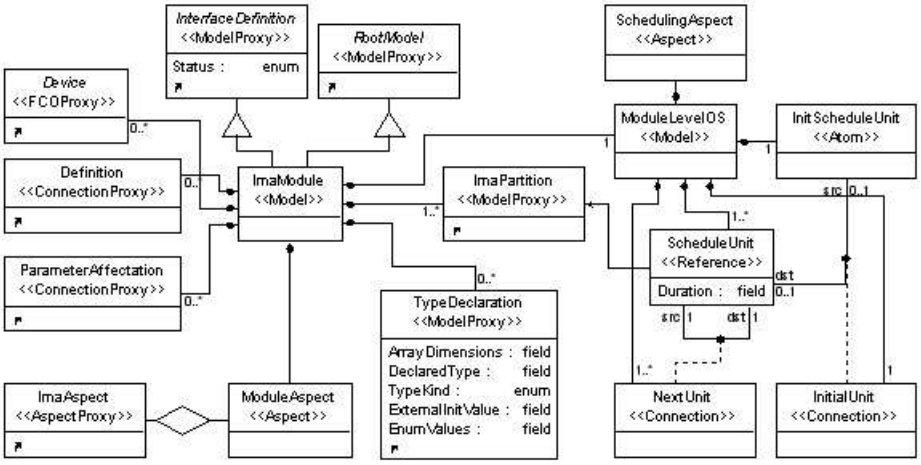

Figure 13: IMA module paradigm sheet.

The *ModuleLevelOS* is used to represent the scheduling policy of contained partitions as an automaton. Each state of this automaton is a Reference to an IMA partition (referred to as *ScheduleUnit*) of the IMA module. Each IMA partition must have at least one such Reference in the *ModuleLevelOS*. References haave an attribute *Duration* that indicates their allocated execution duration. In the automaton, transitions are represented as Connections (referred to as *NextUnit*) without any guard. Each Reference can only have one incoming transition and one outgoing transition. In fact, the automaton is a cycle. Finally, an Atom (*InitScheduleUnit*) is added to the *ModuleLevelOS* to indicate the initial state of the automaton through the use of a *InitialUnit* Connection. The right half part of the diagram shown in FIG. 13 describes the scheduling automaton.

Constraint: AllPartitionScheduled

Description: all partitions must be present in the scheduling cycle at least once.			
Attach to: *ModuleLevelOS*	Event: on Close Model	Priority : 2	Depth : 0

Constraint: ScheduleUnitCycle

Description: there must be a cycle starting with the initial ScheduleUnit.			
Attach to: *ModuleLevelOS*	Event: on Close Model	Priority : 1	Depth : 0

Constraint: SchedUnitReferenceValidity

Description: all ScheduleUnit must refer to a partition in the module.			
Attach to: *ModuleLevelOS*	Event: on Close Model	Priority : 1	Depth : 0

Constraint: SchedUnitConnectionValidity

Description: ScheduleUnits cannot have more than one NextUnit connection as source and one NextUnit connection as destination.			
Attach to: *NextUnit*	Event: on Connect	Priority : 1	Depth : 0

A.4 IMA partition

The IMA partition level shown in FIG. 14 allows to compose different IMA processes (see Section A.5). As for the IMA module level, a *PartitionLevelOS* is added to the IMA partition level. However, the scheduling policy is not described explicitly here. One must choose a scheduling policy (Priority, EDF, RM) through the *Scheduling* attribute of the *PartitionLevelOS*.

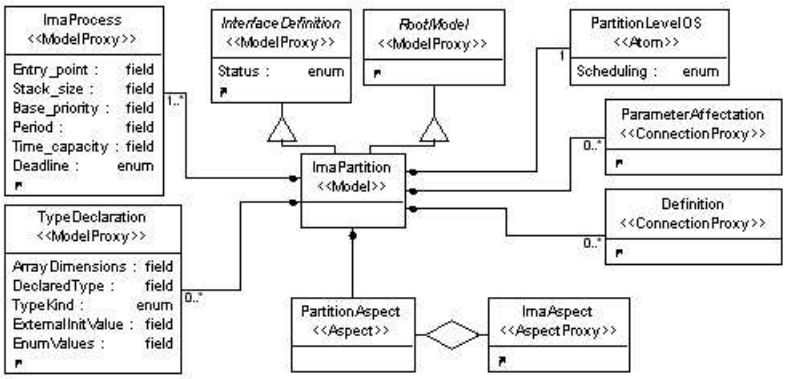

Figure 14: IMA partition paradigm sheet.

Similarly to the IMA system, the IMA partition level includes two Aspects: *ImaAspect* and *Interface* (inputs/outputs, static parameters, and types can be declared at this level).

Constraint: UniqueProcessName

Description: all processes in an ImaPartition must have different names.			
Attach to: *ImaPartition*	Event: none	Priority : 2	Depth : 1

A.5 IMA process

The IMA process level shown in FIG. 15 is the lowest level in our model hierarchy. It allows to compose different *Blocks*, which correspond to elementary computations (e.g. a user function, an APEX-ARINC service call). In MIMAD, a *Block* is described as a Reference to a *ModelDeclaration* concept, which can describe either an APEX service, or a user function. There is a library which contains all APEX services described as *ModelDeclaration* objects.

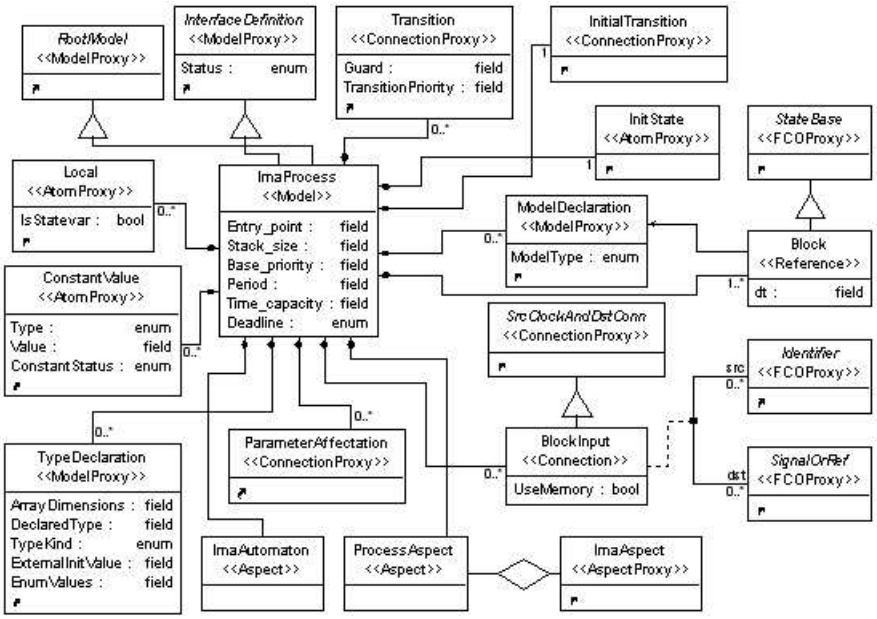

Figure 15: IMA process paradigm sheet.

As for the previous level, an IMA process has an *Interface* Aspect to describe input/output signals and static parameters. Two other Aspects are associated with IMA processes: *ImaAspect*, in which the computation part of the IMA process is specified, and *ImaAutomaton*, in which *Blocks* are scheduled.

In the *ImaAspect*, one can declare functions (as *ModelDeclaration*), types, constant and local signals. Then, one can also express all relations between these objects, the inputs/outputs of blocks and of the IMA Process. For this level, signals are connected through the *BlockInput* Connections, which correspond to *Definition* Connections to which an attribute (*UseMemory*) is added. Sometimes, the input signals of a Block have to use the output signals of other Blocks. Thus, because only one Block of an IMA process can be executed in an instant, inputs have to use the previous value of those outputs. This indication

is given by the *UseMemory* attribute.

In the *ImaAutomaton* Aspect, the scheduling algorithm is defined by a mode automaton, in which each state is associated with a block. Here, we consider an extension of SignalMeta to represent mode automata. As shown in FIG. 15, *Block* inherits from the *StateBase*. *InitState* and *InitialTransition* respectively indicate the initial state and the *Transition* Connection of the automaton. A Connection has two attributes: a *Guard* containing a boolean expression and a *TransitionPriority* that indicates the priority of a Transition. The value of the priority must be different for output Transitions of each block in order to guarantee the determinism of the automaton.

Constraint: NoNullReference

Description: all blocks must refer to a ModelDeclaration.			
Attach to: *ImaProcess*	Event: none	Priority : 1	Depth : 0

Constraint: UniqueBlockname

Description: all blocks must have different names.			
Attach to: *ImaProcess*	Event: on Close Model	Priority : 1	Depth : 0

Constraint: UniqueTransitionPriority

Description: all Transitions from a Block must have different priorities.			
Attach to: *ImaProcess*	Event: on Close Model	Priority : 2	Depth : 0

Constraint: BlockValidName

Description: the name of a Block must be only composed by alphanumeric characters.			
Attach to: *Block*	Event: on Change Property	Priority : 1	Depth : 0

A.6 Other class diagrams

FIG. 16 depicts some resources that can be used at the above description levels. So, *Blackboard*, *Buffer*, *Event*, and *Semaphore* can be declared at the IMA partition level, whereas *SamplingPort* and *QueuingPort* resources can only be declared at the IMA module and IMA system levels. These resources inherit from the abstract concept *Identifier*. As a result, they can be used as signals.

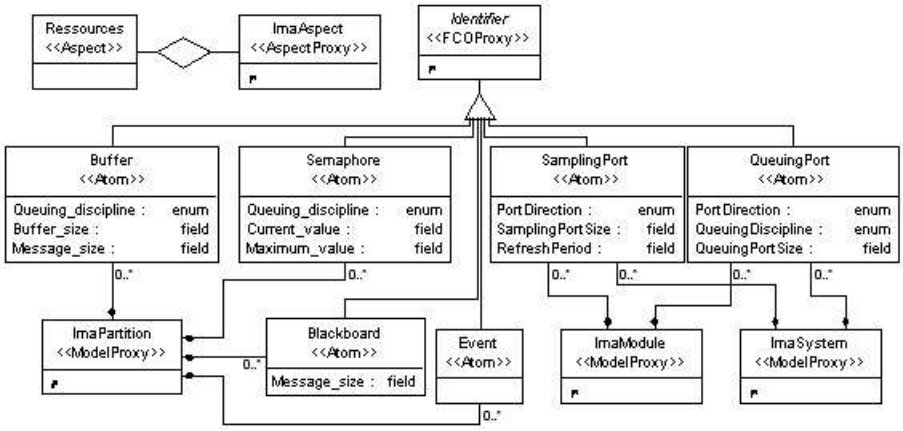

Figure 16: IMA Resource paradigm sheet.

Constraint: UniqueResourceName

Description: all resources must have different names.			
Attach to: *ImaPartition*, *ImaModule*, and *ImaSystem*	Event: on Close Model	Priority : 1	Depth : 0

As mentioned before, devices in FIG. 17 will be only used in future versions of MIMAD.

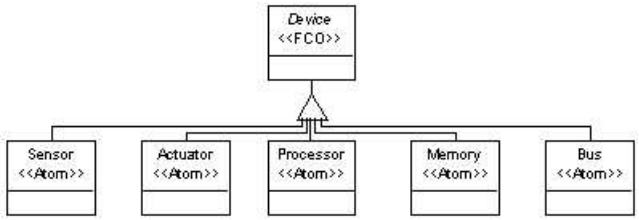

Figure 17: IMA Device paradigm sheet.

Note 1 *APEX types are added to the enumeration of signal and constant types of Signal-Meta. This enables a uniform way to design both Signal-Meta and* MIMAD *based models. The added types are the following:* `BlackboardStatus_type`, `BufferStatus_type`, `Deadline_type`, `EventStatus_type`, `Locklevel_type`, `PortDirection_type`, `ProcessAttribute_type`, `ProcessState_type`, `ProcessStatus_type`, `QueuingDiscipline_type`, `QueuingPortStatus_type`, `ReturnCode_type`, `SamplingPortStatus_type`, `SemaphoreStatus_type`, `SystemAddress_type`, *and* `SystemTime_type`.

B The MIMAD interpreter

We now present the model interpretation process for MIMAD. In the current version, there is no code generation for IMA models and IMA systems. The generation is limited to the IMA module level, which is actually enough to address our case studies. However, we plan to extend the interpretation process to the higher levels of the description hierarchy in the next versions of MIMAD.

As explained in Section 5.3, the interpretation is divided into three steps. GME itself is in charge of the first step. So, in the following, we will only precise, for each MIMAD Model what is checked in the second phase, and the skeleton of the code produced in the third step.

B.1 IMA module

For the IMA module, the interpreter first checks that one (and only one) ModuleLevelOS is declared so as to be able to produce the scheduling of contained IMA partitions. In addition, there must be at least one IMA partition in any IMA module. If there are several partitions, their names must be different. Finally, all resources (e.g. *SamplingPort* and *QueuingPort*) must have different names.

The following code represents a skeleton corresponding to an IMA module. In this code, inputs, outputs and parameters are listed according to their position in the *Interface* Aspect.

```
process <ImaModule name> =
   {< list of ImaModule parameters >}
   ( ? event TOPS, initialize;
       <list of ImaModule inputs>
     ! <list of ImaModule outputs> )
   (| active_partition_ID := ModuleLevelOS{}(TOPS,initialize)
    | (resource1,...,resourceN) := CREATE_RESOURCES{}(initialize)
    | < Instantiation of ImaPartitions >
    | < Equations corresponding to connections between partitions and/or signals >
    |)
   where
      process CREATE_RESOURCES =
         ( ? event initialize;
           ! integer resource1, ..., resourceN; )
         (| (resource1,_resource1_ret) := CREATE_<kind_of_resource1>{}(...)
          | ...
          | (resourceN,_resourceN_ret) := CREATE_<kind_of_resourceN>{}(...)
          |)
         where
            ReturnCode_type _resource1_ret, ..., _resourceN_ret;
         end; % process CREATE_RESOURCES %
      <Module Level OS declaration>
```

```
      <Declaration of ImaPartitions>
      <Local declarations of signals, types and constants>
   end; %process <ImaModule name>%
```

B.2 Module Level OS

Here, the interpreter checks that the specified automaton is "well-formed". That means first, one and only one initial state must be specified among all references on IMA partitions (represented by *ScheduleUnit*). Every Reference must refer to an IMA partition declared in the corresponding IMA module. Finally, the whole transitions describe a cycle, which begins from the unique initial state.

The following code represents the skeleton of a Module Level OS, which is described in the local declaration block of its containing IMA module (see Section B.1). In the code below, `active_partition_ID` indicates the current executed IMA partition while the duration allocated to an IMA partition is expressed as a number of `TOPS` event.

```
process ModuleLevelOS =
   ( ? event TOPS, initialize;
     ! PartitionID_type active_partition_ID; )
   (| active_partition_ID := next_unit when transition
                             default active_partition_ID$ init <Partition1>
    | cpt := <duration of Partition1> when initialize
             default <duration of Partition2> when transition
                           when (active_partition_ID = <Partition1>)
             default ...
             default <duration of Partition1> when transition
                           when (active_partition_ID = <PartitionN>)
             default (cpt$ -1)
    | transition := when (cpt$ = 1)
    | next_unit := <Partition2> when (active_partition_ID = <Partition1>)
           default ...
           default <Partition1> when (active_partition_ID = <PartitionN>)
    | active_partition_ID ^= cpt ^= TOPS
    |)
   where
      PartitionID_type next_unit;
      event transition;
      integer cpt;
   end; % process ModuleLevelOS
```

B.3 IMA partition

The properties checked by the interpreter on an IMA partition are similar to those for an IMA module. It means that there is one and only one *PartitionLevelOS*; IMA processes

and resources (e.g. *Buffer*, *Semaphore*) must have different names; and there is at least one IMA process in an IMA partition.

The following code represents the skeleton of an IMA partition. The PartitionLevelOS is in charge of instantiating every IMA process (`PROCESS_CREATION`) and defining the scheduling policy (through the `PROCESS_SCHEDULINGREQUEST` call).

```
process <ImaPartition name> =
   { PartitionID_type Partition_ID;
     <list of ImaPartition parameters> }
   ( ? PartitionID_type active_partition_ID;
       event initialize;
       <list of other ImaPartition inputs>
     ! <list of ImaPartition outputs> )
   (| (active_process_ID,timedout) := PARTITION_LEVEL_OS{Partition_ID}
                                          (active_partition_ID,initialize)
    | (resource1,...,resourceN) := CREATE_RESOURCES{}(initialize)
    | <Instantiation of ImaProcesses>
    | <Equations corresponding to GME connections between processes>
    |)
   where
      process PARTITION_LEVEL_OS =
         { PartitionID_type Partition_ID }
         ( ? PartitionID_type active_partition_ID;
             event initialize;
           ! ProcessID_type active_process_ID;
             [MAX_NUMBER_OF_PROCESSES]boolean timedout; )
         (| (_<Process1 name>_PID,...,_<ProcessN name>_PID) :=
                                              PROCESS_CREATION(initialize)
          | _<Process1 name>_ret := START{}(_<Process1 name>_PID)
          | ...
          | _<ProcessN name>_ret := START{}(_<ProcessN name>_PID)
          | partition_is_running := when active_partition_ID = Partition_ID
          | success :=
              PROCESS_SCHEDULINGREQUEST{<scheduling attribute>}(partition_is_running)
          | (active_process_ID,status) := PROCESS_GETACTIVE{}(when success)
          | timedout := UPDATE_COUNTERS{}()
          | timedout ^= partition_is_running
          |)
         where
            process PROCESS_CREATION =
               ( ? event initialize;
                 ! ProcessID_type _<Process1 name>_PID,...,_<ProcessN name>_PID;)
               (| _<Process1 name>_att.Name := "<Process1 name>"
                | _<Process1 name>_att.Entry_point := <its attribute value>
                | _<Process1 name>_att.Stack_Size := <its attribute value>
```

```
            | _<Process1 name>_att.Base_Priority := <its attribute value>
            | _<Process1 name>_att.Period := <its attribute value>
            | _<Process1 name>_att.Time_Capacity := <its attribute value>
            | _<Process1 name>_att.Deadline := <its attribute value>
            | _<Process1 name>_att ^= initialize
            | (_<Process1 name>_PID, _<Process1 name>_ret) :=
                                  CREATE_PROCESS{}(_<Process name>_att)
            | < the same equations for each ImaProcess of the Partition >
            |)
          where
             ProcessAttributes_type _<Process1 name>_att,
                                   ...,_<ProcessN name>_att;
             ReturnCode_type _<Process1 name>_ret,...,_<ProcessN name>_ret;
          end; % process PROCESS_CREATION %
      end; % process PARTITION_LEVEL_OS %

   process CREATE_RESOURCES =
      (? event initialize;
       ! integer resource1, ..., resourceN; )
      (| (resource1,_resource1_ret) := CREATE_<kind_of_resource1>{}(...)
       | ...
       | (resourceN,_resourceN_ret) := CREATE_<kind_of_resourceN>{}(...)
       |)
      where
         ReturnCode_type _resource1_ret, ..., _resourceN_ret;
      end; % process CREATE_RESOURCES %

   ProcessID_type active_process_ID;
   [MAX_NUMBER_OF_PROCESSES]boolean timedout;
   boolean success;
   event partition_is_running;
   ProcessStatus_type status;

   integer resource1, ..., resourceN;
   ProcessID_type _<Process1 name>_PID, ..., _<ProcessN name>_PID;
   ReturnCode_type _<Process1 name>_ret, ..., _<ProcessN name>_ret;
   real _<Process1 name>_dt, ..., _<ProcessN name>_dt;

   <Declarations of local ImaProcesses: <Process1 name>,...,<ProcessN name> >
   <Declaration of local signals and local types>
end; %process <ImaPartition name>%
```

B.4 IMA process

Similarly to the ModuleLevelOS, the interpreter checks, for each IMA process, that the associated automaton is "well-formed". Thus, there is one and only one initial transition and all

blocks refer to a *ModelDeclaration*. Moreover, Blocks must have different names, which are used to generate an enumerated type (`block_enum_type`). Finally, the interpreter checks that all output transitions of a block have different priorities so that they could be ordered correctly (for the `next_block` equation in the code skeleton below).

The computation and control parts of an IMA Process are respectively generated into `COMPUTE` and `CONTROL` processes illustrated by the following skeleton. The `CONTROL` process computes the current active block information, which is used by the *COMPUTE* process to compute the corresponding instruction set.

```
process <ImaProcess name> =
   { ProcessID_type Process_ID; }
   ( ? ProcessID_type active_process_ID;
       [MAX_NUMBER_OF_PROCESSES]boolean timedout;
       <list of other ImaProcess inputs>;
     ! real dt;
       <list of other ImaProcess outputs> )
   (| active_block := CONTROL{}(when (active_process_ID = Process_ID), ret)
    | (ret, dt) := COMPUTE{}(active_block)
    |)
   where
      type block_enum_type = enum(<Block1 name>,...,<BlockN name>);
      block_enum_type active_block;
      ReturnCode_type ret;

      process CONTROL =
         ( ? event trigger;
             ReturnCode_type ret;
           ! block_enum_type active_block; )
         (| blocked ^= active_block ^= trigger
          | blocked := (service_call ^- ^ret)
               default (false when service_call)
               default zblocked
          | zblocked := blocked$ init false
          | service_call:= when (active_block=#<Block, which calls an APEX service>)
          | next_block := #<Destination block of transition1>
                          when <guard of transition1>
                          when active_block = #<Source block of transition1>
                default ...
                default #<Destination block of transitionM>
                          when <guard of transitionM>
                          when active_block = #<Source block of transitionM>
                default #<Block1 name>
          | active_block := zactive_block when zblocked
                    default next_block$ init #<Block1 name>
          | zactive_block := active_block$ init #<Block1 name>
```

```
        |)
      where
         event service_call;
         boolean blocked, zblocked;
         block_enum_type zactive_block, next_block;
      end; % process CONTROL %

   process COMPUTE =
      ( ? block_enum_type active_block;
        ! ReturnCode_type ret;
          real dt; )
      (| case active_block in
         {#<Block1 name>} : (| < instantiation of the APEX services
                               (or user function) pointed by <Block1 name> |)
           ...
         {#<BlockN name>} : (| < instantiation of the APEX services
                               (or user function) pointed by <BlockN name> |)
         end
       | ret := < Return code of the current executed block, if it has any >
       | zactive_block := active_block$
       | dt := <duration of active_block>
       |)
      where
         block_enum_type zactive_block;
      end; % process COMPUTE %

   < Declaration of local user functions >
   < Declaration of local signals, constants, and types >
end; % process <ImaProcess name> %
```

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399