A Modeling Paradigm for Integrated Modular Avionics Design

Abdoulaye Gamatié[1] Christian Brunette[2] Romain Delamare[2] Thierry Gautier[2] Jean-Pierre Talpin[2]
[1] INRIA Futurs - rue Pierre et Marie Curie - 59062 Lezennes, France
[2] IRISA/INRIA - Campus de Beaulieu - 35042 Rennes cedex, France

Abstract

This paper presents the modeling paradigm for Integrated Modular Avionics Design MIMAD *V0, which is an extensible component-oriented framework that enables high level models of systems designed on integrated modular avionics architectures.* MIMAD *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 a library of components within the* POLYCHRONY *platform, dedicated to the design of avionic applications. Its descriptions can be therefore 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.*

1. Introduction

Originally inspired by concepts and practices borrowed from digital circuit design and automatic control, the *synchronous hypothesis* [3] 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 [4], such as LUSTRE and SIGNAL, 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 [10], 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 [12].

In contrast with related approaches, SIGNAL favors the progressive design of systems that are correct by construction, 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. 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 [8].

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 [10], 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 [21], our objective is to bring this technology in the context of model-driven engineering environments such as the **General Modeling Environment** (GME) [11] and of the UML in order to provide engineers with better ergonomy and higher-level design abstraction facilities.

Along the path to this objective, the present article introduces an important milestone: the definition of a **modeling paradigm** called MIMAD to allow for a conceptually unified engineering of avionic applications based on IMA architectures and using Polychrony. This paradigm is based on the Polychrony toolset and consists of a meta-model that describes Polychrony and its IMA library in GME.

In the remainder, Section 2 first introduces the IMA architectural concepts; then, Section 3 briefly presents the SIGNAL language and its environment POLYCHRONY. Next, Section 4 gives an overview of GME. Section 5 presents the main features of the MIMAD paradigm and

Section 6 illustrates its use to model a simple avionic application. The adopted approach is discussed in Section 7. Finally, conclusions are given in Section 8.

2. Integrated modular avionics

IMA [1] 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 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 **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 **ports** and **channels**.

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

The APEX-ARINC 653 standard [1] 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

SIGNAL [12] is a declarative synchronous data-flow language dedicated to the design of embedded systems for critical application domains such as avionics and automotive. Its development environment POLYCHRONY offers several tools composed of the SIGNAL 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 [15] for verification and controller synthesis.

SIGNAL 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, then it holds a value; or absent, denoted by $\perp$ 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 *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 six primitive constructs, which are of sufficient expressive power to derive other constructs for comfort and structuring. In the following, we enumerate the primitive constructs by giving their corresponding syntax and semantics:

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

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

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

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

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

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

SIGNAL provides a process frame (see FIG. 4) in which any process may be encapsulated. This frame also enables the definition of sub-processes. SIGNAL also allows one to import external modules (e.g. C++ functions). 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 (e.g. determinism, absence of cyclic definitions, absence of empty clocks to ensure a consistent reactivity of the program), and dynamic properties (e.g. reachability, liveness). The SIGNAL compiler addresses invariant properties, while dynamic properties are checked with SIGALI [15]. Non functional properties include temporal properties that are of high interest for real-time systems. A specific technique has been defined in order to allow timing analysis of SIGNAL programs [9].

4. The Generic Modeling Environment

GME is a freely available [11] configurable UML-based toolkit that supports the creation of domain-specific modeling and program synthesis environments [13]. Metamodeling concepts are proposed in GME to describe *modeling paradigms* for specific domains. Such a paradigm includes, for a given domain, the basic concepts required to represent models from both syntactical and semantical viewpoints.

To describe a new modeling paradigm, one uses the built-in **MetaGME** paradigm. All modeling paradigm concepts have to be specified as classes through usual UML class diagrams. To construct these class diagrams, MetaGME offers predefined UML-stereotypes: First Class Object (FCO), Atom, Model, Reference, Connection. FCO is the basic stereotype in the sense that all other stereotypes inherit from it. It is used to represent abstract concepts (represented by classes). Atoms are elementary objects: they cannot include any sub-part. On the contrary, Models may be composed of several 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* [5]). Inheritance relations are represented as in UML. All the other types of relationships are specified through different kinds of Connection.

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 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 properties on a model designed with this paradigm (e.g. the number of allowed connections associated with a component model).

The above concepts form the basic building blocks 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 Interpreters can be developed and plugged into the GME environment. The role of such an Interpreter consists of interacting with the graphical designs. To realize the connection between the Interpreter and GME, an executable module is provided with the GME distribution, which enables the generation of the Interpreter skeleton. It can be generated in C/C++ or JAVA. In C++, the skeleton is written using the low-level COM language or the **Builder Object Network** (BON) API [13].

5. A modeling paradigm for IMA design

Our intended approach is illustrated in FIG. 2. 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 that offer specific functionalities to the UML layer. As GME allows to import and export XML files, information exchange between the layers can rely on this intermediate format. This favors a high flexibility and interoperability of the approach. GME also proposes specific facilities that enable to connect new environments to its associated platform. This latter possibility permits to implement the code generation directly from GME models without exporting them in XML (see Section 5.2). 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 [8]. 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

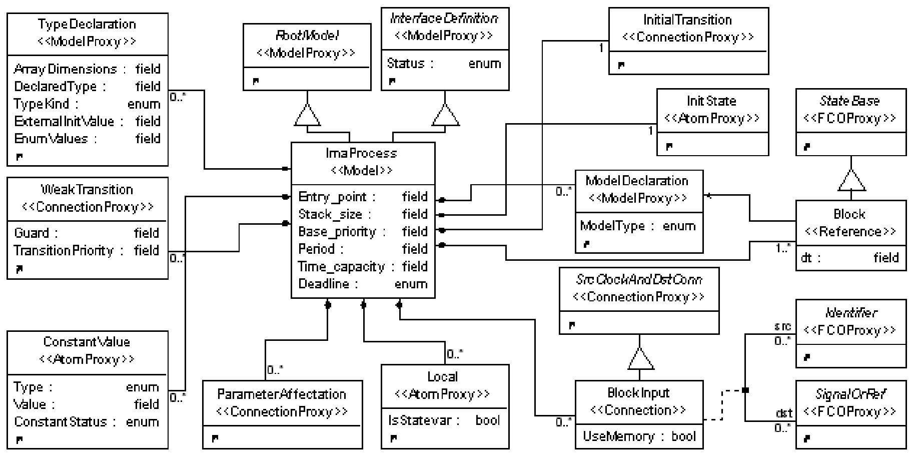


Figure 1. A part of the MIMAD metamodel.

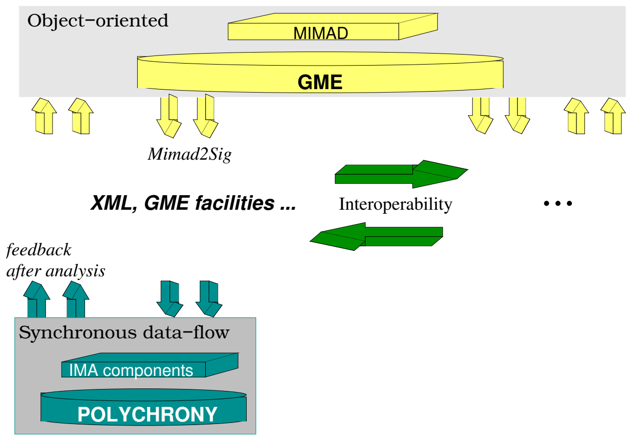


Figure 2. The overall approach.

descriptions. In this way, a user can design quite easily applications based on the IMA modeling approach proposed in POLYCHRONY.

5.1. Definition of basic IMA components

As presented in Section 2, 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 for the correct execution of processes within that partition. Finally, a process consists of a computation part and 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 [8].

The description of the MIMAD paradigm in GME is done in a modular way. It also uses a few concepts defined in the SIGNAL metamodel [5]. The presentation of these concepts is not addressed in this paper since it is not essential to understand our overall approach. Each level of the MIMAD paradigm is modeled by a class diagram and inherits from the *InterfaceDefinition* Model. It means that, as in the *ImaProcess* Model depicted in FIG. 1, *ImaSystem*, *ImaModule*, and *ImaPartition* Models can contain *Input*, *Output*, and *Parameter* Atoms. *ImaPartition* also includes a *PartitionLevelOS* Atom, which allows to specify the scheduling policy. The most complex class diagram is the *ImaProcess* Model shown in FIG. 1. It contains *Block* References, which refer to APEX-ARINC services or other functions defined by the user in a *ModelDeclaration* Model [5]. The control and computation parts of an IMA process Model are separated into two Aspects. The computation part is represented as a block-diagram in which Connections between *Input*s and *Output*s of *Block*s are explicitly described. The control part is represented by a **mode automaton** [14]. 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 instant, the automaton is in one (and only one) mode. Therefore, actions associated with this mode can be performed. In the case of MIMAD, each *Block* represents a state of the mode automaton while guarded *WeakTransition*s realize the connections between *Block*s. These aspects are illustrated in FIG. 6(a) and 6(b) for the `On_flight` application example described below.

APEX-ARINC services, which are required to describe e.g. communications and synchronization between processes and partitions, and time management, are represented as black box abstractions in the object-oriented layer [6].

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

5.2. From GME to SIGNAL

As already mentioned, there are two possible ways to transform models from the upper layer to lower layers. Here, we consider one possibility, which consists in directly generating SIGNAL files using the specific facilities offered in GME. So, we have developed a GMEInterpreter (see Section 4) that generates SIGNAL programs which correspond to models described with MIMAD. This interpreter uses the BON API (see Section 4). FIG. 3 summarizes the different steps performed by the MIMAD Interpreter. There are three main steps in the interpretation of a MIMAD Model:

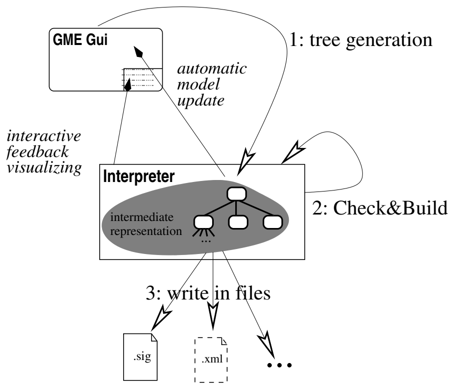


Figure 3. Generation of SIGNAL models.

1. Tree generation. Every FCO selected in the GME GUI is associated with a tree (the intermediate representation in FIG. 3) 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 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. *Local*, *ConstantValue*), are instantiated. The same process is applied recursively on each sub-Model. For example, the instantiation of an *ImaPartition* results in the instantiation of its contained elements, among which *ImaProcess*, *PartitionLevelOS*, and intra-partition mechanisms, which are *BlackBoard*, *Buffer*, *Event*, and *Semaphore*. While intra-partition mechanisms and *PartitionLevelOS* are just Atoms, and thus leafs of the tree, *ImaProcess*es contain subparts. In the same way, each *ImaProcess* recursively instantiates its own symbols and SIGNAL process models.

2. Check&Build. This step consists in building the 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 on how each equation is produced here. Instead, we illustrate the code generation on an example in Section 6. So, let us consider the code represented in FIG. 4. The interface of the corresponding SIGNAL process, named `COMPUTE`, is composed of an *Input* signal `active_block` and two *Output*s `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 [8]. It has been generated automatically from the MIMAD description depicted in FIG. 6(a). The names of blocks present in the computation part (e.g. `Send2`, `ComputePos`) are initially used to create an enumerated type for the input `active_block`, which is produced as output of the control part of IMA processes. The value of this signal corresponds to the current state of the mode automaton encoding the control part (see FIG. 6(b)). 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. 4. They are defined using information obtained from the Connection attributes (e.g. the *UseMemory* attribute of the *BlockInput* Connection in FIG. 1) and the FCO on the other side 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*.

During the same step before the equation generation, some corrections could be applied to the graphical Model, e.g. when a Reference points to an FCO which is not declared in the same scope as the Reference. In such 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 represented 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.

3. Write in output files. The last step consists in visiting one other time each model of the tree and in writing the corresponding equations into destination files.

We can notice that the second and third steps can be specialized. The Interpreter generates files using SIGNAL syntax. It could be modified so as to generate the equations using, for example, XML syntax. This is another way to use XML as an intermediate representation.

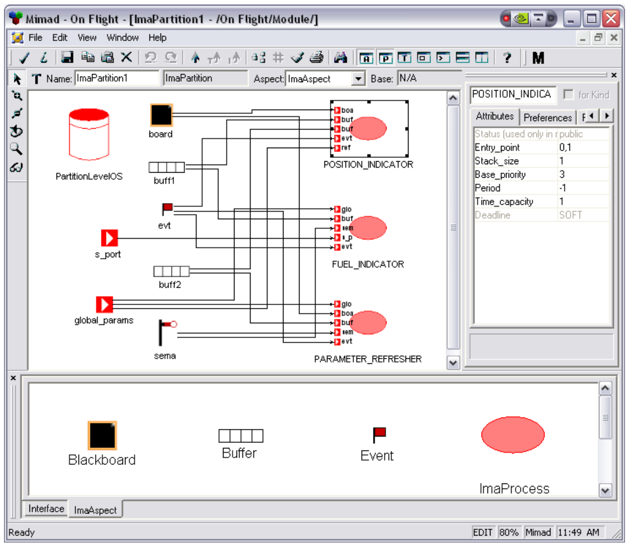
```
process COMPUTE =
( ? block_enum_type active_block;
  ! ReturnCode_type ret; SystemTime_type dt; )
(| case active_block in
   {#Send2}: (|Id0:= SEND_BUFFER{Id1}(Id2,Id3,Id4,Id5)|)
   {#ComputePos}: (|(Id6,Id7):=
                     COMPUTE_POS{}(Id8,Id9,Id10,Id11)|)
   {#ReadBB}: (| (Id12,Id13,Id14):=
                     READ_BLACKBOARD{Id15}(Id16,Id17)|)
   {#WaitEvt}: (| Id18:= WAIT_EVENT{Id19}(Id20,Id21) |)
   {#Send1}: (| Id22:= SEND_BUFFER{Id23}
                                (Id24,Id25,Id26,Id27) |)
   {#SetDate}: (| (Id28,Id29):= SET_DATE{}() |)
   end
 | ret:= Id0 default Id4 default Id8 default Id12
         default Id1
 | zblock:= active_block $
 | dt:= 2 when zblock = #Send2
         default 3 when zblock = #ComputePos
         default 2 when zblock = #ReadBB
         default 2 when zblock = #WaitEvt
         default 2 when zblock = #Send1
         default 2 when zblock = #SetDate
 | ...
 |) where block_enum_type zblock;
end; %process COMPUTE%
```

Figure 4. A SIGNAL code example generated from the MIMAD Model of the computation part of an IMA process.

6. Design of an application

The On_flight example considered here illustrates how to model a simple avionic application with MIMAD. It is represented by a single partition [9]. The main function of On_flight consists in computing information about the current position of an airplane and its fuel level. A report message is then produced, which records all information. On_flight is composed of three processes (see FIG. 5): (a) Position_indicator that first produces the report message, which is updated with the current position information (height, latitude, and longitude); (b) Fuel_indicator that updates the report message (produced by Position_indicator) with the current fuel level information; and (c) Parameter_refresher that refreshes all the global parameters used by the other processes in the partition.

Partition Level Design. The partition is composed of two views, also referred to as Aspects. The first one specifies the interface: *Input*s, *Output*s, *Parameter*s and interface properties. The second Aspect, termed *ImaAspect*, specifies the architecture of applications based on IMA concepts. Although the interface signals and parameters remain visible in the *ImaAspect*, they can only be modified in the *Interface* Aspect. The constituent elements of the *ImaAspect* are the following (see FIG. 5, bottom frame): the partition level OS, processes, and their communication and synchronization mechanisms.



Name: ImaPartition1 ImaPartition Aspect: ImaAspect Base: N/A

POSITION_INDICA

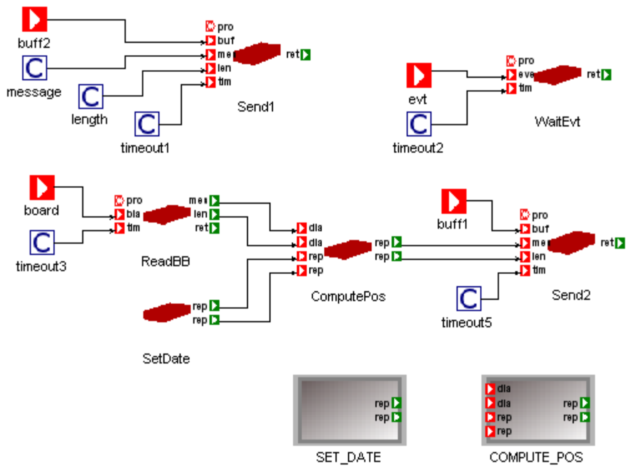
Status (used only in r	public
Entry_point	0,1
Stack_size	1
Base_priority	3
Period	-1
Time_capacity	1
Deadline	SOFT

Figure 5. A MIMAD Model of On_Flight.

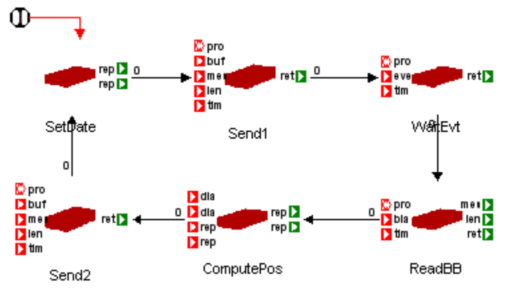
The partition level OS is represented by an Atom whose attribute specifies the scheduling policy adopted by the partition (e.g. Rate Monotonic Scheduling, Earliest Deadline First). We observe that at this stage, the presence of this element is more for structural and visual convenience. However, the 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. There are four kinds of inter-process communication and synchronization mechanisms: *Blackboard*s, *Buffer*s, *Event*s, and *Semaphore*s. The attributes of their associated Atoms are those needed by the partition level OS for creation.

In the On_flight Model shown in FIG. 5, the partition contains three processes, five mechanisms and the partition level OS. The communication and synchronization mechanisms contain a *Blackboard* board, two *Buffer*s 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 port used for communications between different partitions (this input is assumed to be created at the module level).

Process Level Design. To illustrate the process level design we focus on the process Position_indicator. In [8], 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 computation part, which is composed of blocks. In MIMAD, IMA processes are designed using three aspects: the *Interface* as in the partition level, the *ImaAspect*, which includes the computation subpart, and the *ImaProcessControl* containing the control flow of the process.



(a) Computation subpart (*ImaAspect*)



(b) Control subpart (*ImaProcessControl*)

Figure 6. The Position_indicator process.

The computation part (see FIG. 6(a)) is containing *Blocks*, which are References to *ModelDeclarations* specified in the same Aspect or in some library (e.g. APEX-ARINC services). It also contains constant values, local signals, and type declarations. The connections between the *Interface* and the *Blocks* on the one hand, and between *Blocks* on the other hand are also specified in this Aspect.

Finally, the control part of the process is described by the mode automaton depicted by FIG. 6(b). 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 is automatically generated (e.g. see FIG. 4 in Section 5.2). Then, starting from this generated code that forms the SIGNAL model of the application, the analysis and verification functionalities of POLYCHRONY are used in order to formally validate specified functional or performance requirements.

7. Discussion and related work

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. MIMAD relies on the domain-specific language SIGNAL and its associated development environment POLYCHRONY for the description, refinement and formal verification of the application. Simulation code (C, C++, or JAVA) can be directly generated from specifications. MIMAD is an open modeling framework that ideally complements general-purpose UML profiles such as the AADL [19] or MARTE [18] 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 temporal property verifiers such as UPPAAL or Kronos. In the context of GME, abstractions and refinements from and to the meta-model 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. The assessment of MIMAD can be done with respect to the usability and the portability offered by GME, and the following highly desirable criteria:

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 costs in time.

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). The SIGNAL code generated from MIMAD can be analyzed using the tools provided in the POLYCHRONY platform. Static properties are checked with the compiler and dynamic properties with the SIGALI [15]. In the current version of MIMAD, the analyzability of designed models is enabled only at the synchronous data-flow layer. Future works will extend this analyzability to the MIMAD layer.

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 in a modular generation approach. The current version of

GME enables to select and generate sub-parts of a model. Afterward, they can be stored in repositories without regenerating them when reused.

Now, we mention a few studies that are close to our work. Among these, the ATLAS Model Management Architecture (AMMA) [2], which has been defined on top of the Eclipse Modeling Framework (EMF) [7], another MDE platform as GME. 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). In [17], 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 [16], 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 [20], GME is merely 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 learn IMA 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 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 is an important perspective of the present work.

References

[1] Airlines Electronic Engineering Committee. ARINC specification 653: Avionics application software standard interface. `Aeronaut.radio,Inc`, January 1997.

[2] ATLAS Group. AMMA. `www.sciences.univ-nantes.fr/lina/atl/AMMAROOT`.

[3] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. In *IEEE, vol. 79, No. 9*, pages 1270–1282, Avril 1991.

[4] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The Synchronous Languages Twelve Years Later. In *Special issue on Embedded Systems, IEEE*, 2003.

[5] L. Besnard, C. Brunette, T. Gautier, and J.-P. Talpin. Modeling multi-clocked data-flow programs in GME. Technical Report 5775, INRIA, December 2005.

[6] C. Brunette, R. Delamare, A. Gamatié, T. Gautier, and J.-P. Talpin. A Modeling Paradigm for IMA Design. Technical Report 5715, INRIA, October 2005.

[7] Eclipse Modeling Framework. Reference site. `www.eclipse.org/emf`.

[8] 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.

[9] A. Gamatié, T. Gautier, and L. Besnard. Modeling of avionics applications and performance evaluation techniques using the synchronous language SIGNAL. In *Synchronous Languages, Applications, and Programming*. Portugal, 2003.

[10] INRIA project Espresso. POLYCHRONY. `www.irisa.fr/espresso/Polychrony`.

[11] ISIS, Vanderbilt University. GME. `www.isis.vanderbilt.edu/Projects/gme`.

[12] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for System Design. In *Journal for Circuits, Systems and Computers. Special Issue on Application Specific Hardware Design. (c) World Scientific*, April 2002.

[13] 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*, May 2001.

[14] 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.

[15] 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.

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

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

[18] OMG. Uml profile for modeling and analysis of real-time and embedded systems (MARTE).

[19] Society of Automotive Engineers Standard. AADL. `www.aadl.info`.

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

[21] TOPCASED website. `www.topcased.org`.