UML&AADL ’2007 Grand challenges

Sébastien Gérard [1] Peter Feiler [2] Jean-François Rolland [3*] Mamoun Filali [3]
Mark-Oliver Reiser [4] Didier Delanote [5] Yolande Berbers [5] Laurent Pautet [6]
Isabelle Perseil [6]

[1] CEA-LIST
[2] Carnegie Mellon University
[3] IRIT-UMR 5505 CNRS
[4] Technical University of Berlin
[5] Katholieke Universiteit Leuven, Departement Computerwetenschappen
[6] GET-Télécom Paris – LTCI-UMR 5141 CNRS

ABSTRACT

On today’s sharply competitive industrial market, engineers must focus on their core competencies to produce ever more innovative products, while also reducing development times and costs. This has further heightened the complexity of the development process. At the same time, industrial systems, and specifically real-time embedded systems, have become increasingly software-intensive. New software development approaches and methods must therefore be found to free engineers from the even more complex technical constraints of development and to enable them to concentrate on their core business specialties. One emerging solution is to foster model-based development by defining modeling artifacts well-suited to their domain concerns instead of asking them to write code. However, model-driven approaches will be solutions to the previous issues only if models evolves from a contemplative role to a productive role within the development processes. In this context, model transformation is a key design paradigm that will foster this revolution. This paper is the result of discussions and exchanges that took place within the second edition of the workshop “UML&AADL” (http://www.artist-embedded.org/artist/Topics.html) that-was hold in 2007 in Auckland, New Zealand, in conjunction with the ICECCS07 conference. The purpose of this workshop was to gather people of both communities from UML (including its domain specific extensions, with a focus on MARTE) and AADL (including its annexes) in order to foster sharing of results and experiments. More specially this year, the focus was on how both standards do subscribe to the model driven engineering paradigm, or to be more precise, how MDE may ease and foster the usage of both sets of standards for developing real-time embedded systems. This paper will show that, even if the work is not yet finished, the current results seems to be already very promising.

*Work founded by CNES and EADS Astrium Satellites

Keywords

Real-Time, Embedded, MDA, MDD, MDE, ADL, UML, MARTE, AADL, xUML, TLA+

1. INTRODUCTION

On today’s sharply competitive industrial market, engineers must focus on their core competencies to produce ever more innovative products, while also reducing development times and costs. This has further heightened the complexity of the development process. At the same time, industrial systems, and specifically real-time embedded systems, have become increasingly software-intensive. New software development approaches and methods must therefore be found to free engineers from the even more complex technical constraints of development and to enable them to concentrate on their core business specialties. One emerging solution is to foster model-based development by defining modeling artifacts well-suited to their domain concerns instead of asking them to write code. However, model-driven approaches will be solutions to the previous issues only if models evolves from a contemplative role to a productive role within the development processes. Model transformation is the key design paradigm that will enables this revolution. Indeed, models will be productive either if one may refine them iteratively (e.g., going seamlessly from the requirements to the code), or if one may abstract models from other models in order to ease one problem or to focus on a specific concern. Abstraction and refinement are then the two essential principles of model-driven engineering.

The concept of abstraction is intrinsic to that of modeling, which by definition consists of representing a real world object, called the system, in a simplified form. This involves two possible types of abstraction – vertical and horizontal.

• *Vertical abstraction* follows development process flow, producing models that focus on the pertinent level of detail. There is a recurrent need, in system development, for models of standardized software (RTOS and/or middleware) and of hardware implementation platforms (e.g. POSIX, OSEK) that identify dependencies between application models and implementation choices/constraints.

•*Horizontal abstraction* – takes place at a same level of definition and emphasizes certain system facets or complementary viewpoints. Examples include task models for RT analysis, architectural models centering on system functions and scenarios models for system testing.

Abstraction relies also on suitable concepts available within the modeling language used to denote the models depending on their granularity level and their focused concern.

For refinement purposes, the goal is to master and automate the process of building one specialized model from another. This typically means producing executable applications by for instance model compilation, formalization, use of design patterns for the domain, etc.

Finally, fast prototyping, evaluation and validation are vital to the development of real time embedded systems. To this effect, engineers need models with well-defined execution semantics, i.e. models whose behavior (i.e. dynamics) can be analyzed, executed, or simulated. More specifically, they must also be:

•*Predictable behavior models* – An RT/E application must always behave in the same way in a given, same initial context. Typically this behavior is expected to be deterministic. In some application contexts stochastic event occurrences are expected to be processed in a predictable manner. The semantics of the models used, and therefore, of the underlying modeling language, must be precisely defined and unambiguous.

•*Complete models* – A complete model contains all the data required to analyze its behavior or to generate an executable view of this data.

The model of a real-time embedded system then has predictable execution semantics if it exhibits both the above properties – predictability and completeness. It can be executed in different ways, e.g. via automatic code generation on a given computing platform, either software- or hardware-based, or by simulating of the generated code, or through formal model analysis tools that trace system operation and verify behavior, like symbolic execution engine.

This paper will then present some current works that address these different issues. The next section presents the new OMG standard dedicated to complement the UML2 with the required extensions for supporting modelling and analysis of real-time embedded systems, MARTE [39]. Section 3 and 4 are then dedicated to an other modelling language AADL language. Section 3 is going about firstly the usage of xUML to denote executable models independently of any computing platforms, and secondly how it is possible to map such models onto a specific AADL-like computing platform. The third section focusses then on the AADL standard and specially how a model-driven engineering may ease its usage. The fourth section introduces the reader to an automotive emerging architecture description language, EAST-ADL. This paper is finally concluded within the last section.

2. MARTE, THE UML PROFILE FOR MODELLING AND ANALYSIS OF RTES

The Object Management Group (OMG[1]) is one of the key international organization promoting standards fostering the usage of model-based paradigm. The Unified Modelling language (UML) [40] standard is maybe one of its most representative success within the software industry but also in other kinds of domain such as IT and Banking systems. UML is now the most widespread language used for modelling among industrials, and academics as well. But, because UML has been designed to be a general purpose modelling language, specializations need, and hence are expected, to be defined in order to suit better to specific domains or applications. The real-time embedded (RTE) domain is one of this specific domain for which extensions to UML are required to provide more suitable concepts related to the domain area. Previously, the OMG has defined the UML profile for Schedulability, Performance and Time (SPT, [35]). This latter UML extension was providing mainly in the one hand concepts for dealing with model-based schedulability analysis focussed on Rate Monotonic Analysis and model-based performance analysis focussed on the queuing theory. In other hand, SPT was also proposing a rich framework for time and time related mechanisms. However, experiments on SPT revealed shortcomings within the profile as for example lacks in terms of concepts to better suit to model-based development of RTE systems. There was also a strong need for modifications to comply with the evolution of other OMG standards specially UML2, and to have a profile with a broader scope. As for example, a support for both hardware and software aspects design of embedded systems and a richer support for schedulability and performance analysis encompassing additional techniques such as hierarchical scheduling. This has resulted in a Request For Proposals (RFP) for a new UML profile named MARTE (Modeling and Analysis of Real-Time and Embedded systems, [34]), which should address issues such as compliance with the UML profile for Quality of Service and Fault Tolerance (QoS & FT, [37]), specification of not only real-time constraints but also other embedded non-functional characteristics such as memory and power consumption, modelling and analysis of component-based architectures, and the capability to model systems in different modelling paradigms (asynchronous, synchronous, and timed).

To cope with this new RFP, OMG members decided to gather to build a common answer to this new standard request. This consortium was known as the ProMARTE consortium and was consisting of following companies: Alcatel, ARTISAN Software Tools, Carleton University, CEA LIST, ESEO, ENSIETA, France Telecom, International Business Machines, INRIA, INSA from Lyon, Lockheed Martin, MathWorks, Mentor Graphics Corporation, No Magic, Software Engineering Institute (Carnegie Mellon University), Softeam, Telelogic AB, Thales, Tri-Pacific Software and Universidad de Cantabria. This common work results in a proposal to OMG standardization that was accepted and voted in June 2007 [39].

The purpose of this section is then to give the reader a brief overview of the MARTE standard and we invite the reader who wants to know more about this new specification to refer to the OMG web site dedicated to MARTE: www.omgmarte.org.

2.1 MARTE in a nutshell

The design principles adopted to design MARTE was a two step processes inspired from usual software engineering process. This consisted in one hand in specifying the language by describing the MARTE domain model, and in other hand in mapping the domain model towards UML2.

[1]www.omg.org

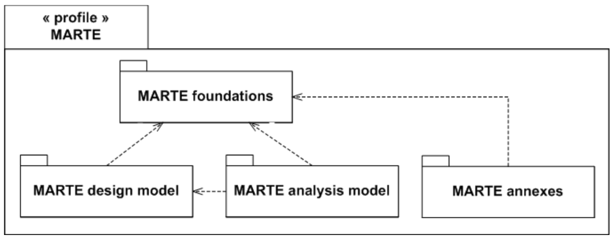


Figure 1: Overall architecture of MARTE

The purpose of the first stage is then to formally define the language constructs by describing a clean and well documented meta-model for the language specification. The second stage aims then in designing the language constructs described in the meta-model in terms of UML2 extensions, mainly stereotypes with properties and possibly OCL constraints. These latter are introduced within the profile to guide their usages in the context of a more global UML2 model. This process has been a key point to ensure a well-formed definition of the standard.

The profile has then been structured around two main concerns, one to model the features of real-time and embedded systems and the other to annotate application models so as to support analysis of system properties. These are shown in the following figure through both packages respectively named MARTE design model and MARTE analysis model. These two major parts share common concerns, as for example time and the use of concurrent resources, which are contained in the upper shared package named MARTE foundations. In addition, MARTE defined complementary transversal modeling constructs that was decided to gather in the MARTE annexes package.

The rest of this paper is then dedicated to tackle three important concerns when modeling real-time and embedded systems. The next section will introduce how MARTE support modeling of non-functional properties, time and which are the concepts dedicated to improve architecture description. After, one outlines respectively how to deal with platform modeling, and model-based analysis. Finally, one mentions some of the ongoing experiments on MARTE and sketches what are the main remaining challenges for MARTE.

2.2 Non-functional properties, time and architecture description

One of the main features that make the development of real-time systems different from general purpose systems development is that non functional properties, specially time aspects, are as much important as functional aspects. To deal with this concern, MARTE has defined two frameworks, the non-functional properties framework (NFPs) and the time framework (Time). The former package consists of a set of modeling constructs to declare, qualify and apply semantically well-formed non-functional properties into UML models. The NFPs framework is complemented with the Value Specification Language (VSL) that defines a textual syntax to formulate algebraic and time expressions conforming to an extended system of data types. NFP supports the declaration of the non-functional properties, whereas VSL is used to describe the values of those properties. The time package contains both basic concepts to define what is time and which time model are supported within MARTE, and also time-related mechanisms such as times event or clocks.

Considering the time aspect, real-time systems are specifically concerned with two important features: the cardinality of time (e.g., delay, duration or clock time) and the temporal ordering of behavior activities (e.g. event1 happens before event2). MARTE proposed concepts for supporting mainly three different time models: the chronometric time model, the logical time model and the synchronous logical time model. This latter is a refinement of the logical time model with the additional assumption of possible simultaneity. This property denotes the possibility for several events to exist at the same time. Let's note that the three models relies on partial ordering of instants.

Within MARTE, two packages are more focused on modeling of system architectures. The former is part of the foundation layer and defines a general component model (GCM). GCM relies on a refinement of the UML structured classes and provides a common denominator among various component models, which in principle do not target exclusively the real-time and embedded domain. The purpose was to provide in MARTE a model as general as possible, that is not tied to specific execution semantics, on which real-time characteristics can be applied later on. The MARTE GCM relies mainly on UML structured classes, on top of which a support to SysML blocks has been added. Aligning GCM with Lightweight-CCM, AADL and EAST-ADL2 [39] has also influenced the definition of some refinements of the MARTE General Component Model.

2.3 Platform modeling

Platform modeling is of course a key aspect in MARTE which fully subscribes to the MDA [2]. The platform concern is handled at the several places within MARTE depending firstly the purpose of the platform model and secondly the granularity level of the platform description. As depicted in the following figure, MARTE deals with platform modeling concern in its three main parts, foundations, design and analysis. The foundations part provides a basic framework for platform-based modeling, the General Resource Modeling (GRM). It intends to factorize general concepts required to platform modeling at a very high-level abstraction, at system level. It relies on a clear design pattern considering platforms as a set of resources containing possible sub resources in hierarchical manner and offering at least one service. Resources may be instantiated and the resulting instance may execute resource services. As denoted in the following figure, GRM is then refined for detailed modeling and analysis purposes. The Software Resource Model (SRM, [20]) and the Hardware Resource Model (HRM, [45] and [46]) are respectively dedicated to describing of software and hardware computing platforms. SRM consists of a set of concepts focussed on describing model-based API of Real-Time Operating Systems such as Semaphores and tasks. The MARTE annexes parts contains some examples of such API describing, as for example OSEK and Arinc. HRM provides concepts needed for describing hardware computing platforms at three levels of abstraction serving mainly the three following use cases:

- Software design and allocation using a high level hardware description model of the targeted platform architecture. This abstraction level is intended to be a formal alternative to block diagrams usual used for describing hardware platforms of embedded systems.

- Analysis of real-time and embedded properties of software-intensive systems on a specialized hardware description model.
- Simulation of detailed hardware models, the required level of detail depends on the simulation accuracy.

Platform describing is important for embedded systems design and analysis, but needs also the ability to denote the relationships between the platforms model and the functional application model. For that purpose, MARTE propose the allocation package that defined the allocation and refinement relationships. Allocation concepts of MARTE is aligned with the one defined in SysML but whereas SysML allocation enables to allocate any kind of elements set on any kind of element sets, MARTE restricts allocation to describe how application model elements are allocated on platform model elements. In addition, the allocation specification may further be refined by defining non-functional properties.

2.4 Model-based analysis

Finally, model-based analysis is the last main key point of MARTE. Considering this last concern, the intend of MARTE was not defined new analysis techniques but to enable apply the most popular real-time and embedded analysis results, mainly coming from both schedulability and performance analysis domains. But in order to provide a flexible and reusable model-based analysis framework, both schedulability and performance analysis packages relies on a generic one. This latter, called the Generic Quantitative Analysis Model (GQAM), defines the basic UML extensions needed to annotate UML model in order to perform any kind of analysis. It is then expected that GQAM will be further specialized or refined in order to cope with analysis techniques that may be not supported by MARTE at the moment.

The annotation mechanism used in MARTE to support model-based analysis uses UML stereotypes. These latter enable to map model elements of application description into the semantics of an analysis domain such as schedulability, and to give values for properties which are needed in order to carry out the analysis. We may distinguish "input" properties which are needed to carry out the analysis, and "output" properties which are results provided by the analysis. However the modeler may also input required values of output properties, which can be used to determine how well a system meets its requirements. Analysis is not always indeed simply "pass/fail", and the particular goals of analysis are specific to their domain. Output properties to be reported may include details of how and where time and resources are consumed in order to diagnose problems, and may include sensitivity studies to explore the importance of parameters whose values are uncertain.

The profile is intended to provide a foundation for applying transformations from UML models into a wide variety of analysis models. The environment for exploiting the profile would consist of a set of tools, including model transformers to gap the model technological space (the UML + MARTE design space in this case) to different possible analysis technological spaces (e.g., Rate Monotonic Analysis tools). The forward path of this process intend to denote the way the model is expected to be transformed (e.g., via the XMI output of the UML model) to the input format required by an analysis tool. The reverse path will then refer to a potential feedback path to re-import the analysis results into the previous UML models.

2.5 MARTE next steps

MARTE is now voted and accepted as a new OMG standard since end of June 2007. The actual version of the standard is a Beta1 version. The next step for MARTE w.r.t. the OMG standardization process is the finalization of the standard in order to define its version 1.0. This job has to be done by an OMG Finalization Task Force (FTF) which for MARTE has been launched in July 2007. Their work is split mainly into two periods. Firstly, MARTE experimenters are invited to provide their feedback on the standard by logging issues where appropriate using the OMG Issue Procedure (defined in the specification itself). The final deadline for sending issues to be resolved within this FTF is December 22^{nd}, 2007. After this date, the FTF is not obliged to take into new issues. Secondly, the FTF members have to solve all the issues received within the right time frame and will provide an enhancement of the standard that once voted and accepted becomes the version 1.0 of the standard.

3. FROM PIMS TO PSMS

The development of embedded systems through models requires the creation of both a platform independent model (PIM) and a platform specific model (PSM). During the platform independent modeling phase the focus is on capturing the application domain, the functionality of the system to be developed and the environment in which it is to be deployed. UML is a modeling notation of choice for platform independent modeling (PIM). The behavior of the system is captured in state charts, activity charts, and sequence charts. xUML is an extension to UML that adds precise execution semantics to models enabling a full description of platform independent models and the generation of code from them. xUML [42] [44] has been successfully used for the development of multiple systems in the context of Model-Driven Architecture. Of particular interest to us has been its use in avionics systems as discussed in [38]. Platform specific models are critical to gaining insight on runtime properties of a system, such as timing, fault tolerance, safety-criticality, and security. For example, control system applications are sensitive to the age and integrity of data. Latency jitter due to execution time variation and phase-delay variation due to non-deterministic sampling are affected by the choice of scheduling policy, communication mechanism, or system partitioning. In other words, a control algorithm that is stable on one hardware platform, such as a federated architecture, may become instable when moved to a different platform, e.g., a partitioned architecture of an Integrated Modular Avionics (IMA) system. Architecture Analysis and Design Language (AADL) [6] [5] is an SAE standard specifically designed to model runtime architectures and analyze their key runtime properties. It enables the creation and analysis of PSMs by modeling the task and communication architecture of embedded applications, their interaction with the physical environment they control, and the execution platform to which they are bound. In this section we present the integration of xUML and AADL into a model-driven development process. The combination of these two modeling languages provides a powerful synthesis for modeling, analyzing, and implementing embedded system. We

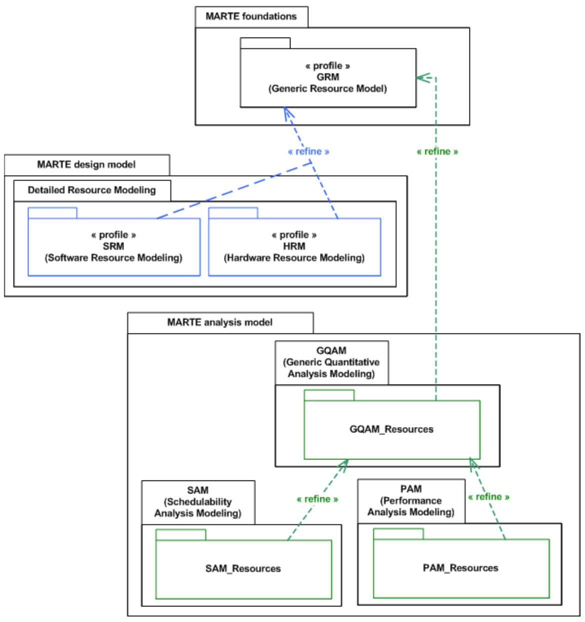


Figure 2: The different MARTE supports for platform modeling

do so by discussing the translation of the xUML concurrency model into AADL, the Behavior Annex extension of AADL as a capability to specify task interaction behavior, and an algorithmic language extension to describe atomic component behavior.

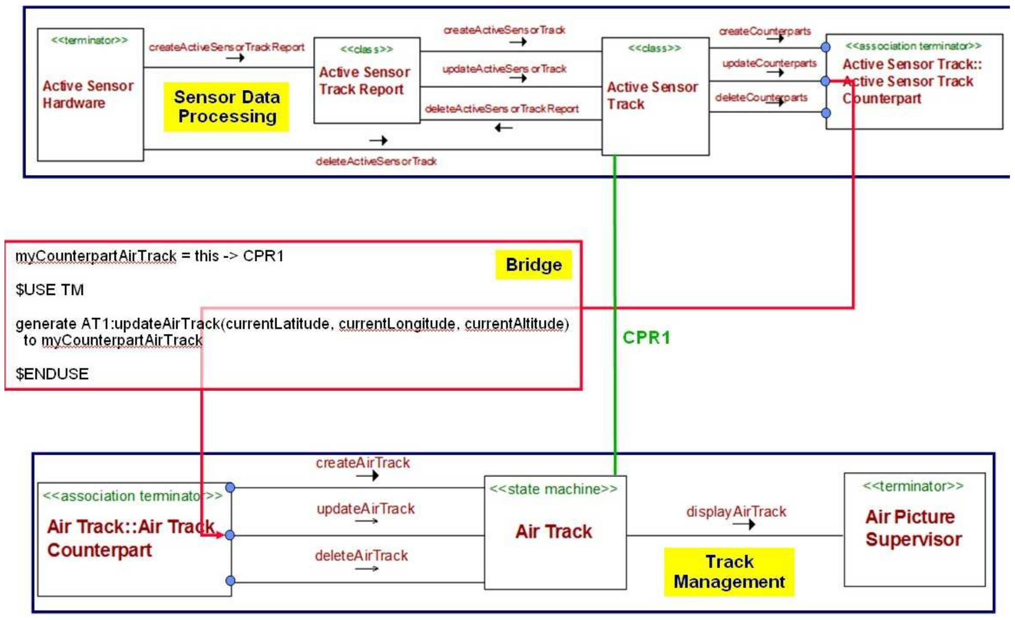
3.1 Extracting the concurrency requirements from xUML into an AADL model

The translation of a PIM expressed in xUML to a PSM expressed in AADL requires the interpretation of xUML models written in a certain style. This component concurrency and interaction model is described in [44]. First the system is captured in Domains in a Domain Chart, i.e., sets of classes with well-defined interactions between them but with a loose definition of interactions to the outside world. Inside the domain three aspects need to be defined: data, the lifecycles of the data and the actions that occur on entry to each lifecycle state. The data aspect of the domains is modeled in class diagrams where classes are defined along with associations between classes. For the lifetime of data, state machines are used to describe the states of classes and the transitions between states. To describe the actions that happen on entry to each state, an action language is used. Finally to complete the model, domain collaboration diagrams or domain sequence diagrams are used to capture the interaction between domains. Even though the xUML model does not define a final concurrency structure, it embeds some concurrency in its execution semantics. In particular two sources of concurrency need to be honored in an xUML model: state machines and object interactions. Both sources of concurrency are described in the domain collaboration diagram along with the reference to the classes used in them. In these models, special entities called Bridges are used to add the detail to how the autonomous domains need to interact. Although the interface from a component to other components is expressed through a set of functions, the code patterns used in the bridge determines the actual message synchronization semantics. xUML has three different message communication contract types: ”

- Closed Blocking. This represents a function call where the caller sends data and waits for an answer from the callee before continuing its execution.
- Closed Non-Blocking. In this case the caller also expects an answer but it will not wait to get it before continuing its execution. Instead it queries for the answer at a later time, or receives a closure notification from the callee.
- Open. This involves a transfer of data from the caller to the callee. The caller does not wait for the completion of the callee and does not expect any answer from it.

The semantics of these messages are properly translated into the following AADL patterns:

- Closed Blocking. In this case the callee is an AADL subprogram and the message from the caller to the callee a subprogram call. The callee and the caller may reside in the same AADL thread (sequential call) or in different threads (remote call). In the latter case the caller thread blocks until the call completes.
- Closed Non-Blocking. In this case the callee and caller are in different threads. The message is an AADL event data (message) port connection from caller to callee and an event data port connection from the callee to the caller to notify the completion of the execution and return result data.
- Open. In this case the caller and the callee are both AADL threads and the message is only an event data port connection from the caller to the callee.



```
myCounterpartAirTrack = this -> CPR1

$USE TM

generate AT1:updateAirTrack(currentLatitude, currentLongitude, currentAltitude)
  to myCounterpartAirTrack

$ENDUSE
```

Figure 3: Domain Collaboration Diagram

Note that AADL supports message communication through event data ports, as well as communication of continuous data streams, as found in control systems, through AADL data ports. In the latter case only the most recent data value is available to the recipient. These mappings are the basis to extract the concurrency patterns from the communication semantics in a xUML model. Figure 3 presents the collaboration diagrams of the xUML system. In this Figure you can see two domains (top and bottom rectangles) called Sensor Data Processing and Track Management. These domains are linked by a Bridge that maps calls to the updateCounterparts required operation of the Sensor Data Processing domain to the updateAirTrack provided operation of the Track Management domain. It can be seen that this bridge makes use of a domain-spanning association that links each object of the "Active Sensor Track" class in Sensor Data Processing to an object of the "Air Track" class in Track Management. Such an association is known as a "counterpart association", and is uniquely labeled using the form "CPR$\langle n \rangle$". Domains in xUML are used to organize the design space. They are mapped to AADL packages that play the same role. Packages define a container for the AADL types. Packages define public and private sections, for this translation we put every definition in the public section. Our strategy is to map the concurrency represented in an xUML model directly into an AADL model by representing the active objects as logical threads that then get mapped into operating system (OS) threads through a thread optimization step. An alterative strategy is to let the user describe the OS thread architecture in AADL and specify a mapping of active objects of xUML through a set of binding properties.

Any complete xUML model at least includes a thread where all the code runs, and stimuli from external devices need to be captured in another thread. In our sample an external device the Active Sensor Hardware is interfaced through the Active Sensor Thread, and a state machine in the Air Track Thread captures the behavior of the stimuli. The communication between the threads and between the sensor thread and the hardware is modeled with event data ports to represent the queuing of events and data that matches the xUML semantics of the communication with a state machine. The threads for our model are presented in the listing of Figure 4.

The call sequence is encoded inside the thread implementation and the connections to parameters and ports (subprograms can have both) are encoded as well. Note that in the ActiveSensorThread.Impl implementation we have three subcomponents: the reportCollection and the activeSensorTrackCollection and airTrackCollection. These subcomponents represent the reports, the active sensor tracks, and the collection of air tracks (the individual airTracks would be in independent threads) inside this thread and we will be sending calls to them. The level of abstraction of the calling sequence in AADL does not support expression of conditionals. In this case the action language part of the xUML model specifies that to create an AirTrackReport we first look to see if it already exists and we created if not or update if it exists. However, in AADL we encode this as just two possible calling paths, i.e., find+create and find+update. This modal view of a thread allows us to associate mode-specific runtime properties without having to explicitly model the logical behavior that drives these logical modes and perform

```
thread ActiveSensorThread
 features
  createActiveSensorTrackReport: in event data port
    CreateActiveSensorTrackEvent;
  initializeAirTrack: out event data port
    UpdateActiveSensorTrackEvent;
  updateAirTrack: out event data port
    TrackManagementDomain::InitializeAirTrackEvent;
  updateAirTrack: out event data port
    TrackManagementDomain::UpdateAirTrackEvent;
  deleteAirTrack: out event data port
    TrackManagementDomain::DeleteAirTrackEvent;
end ActiveSensorThread;

thread AirTrackThread
 features
  initializeAirTrack: in event data port
    InitializeAirTrackEvent;
  updateAirTrack: in event data port
    UpdateAirTrackEvent;
  deleteAirTrack: in event data port
    DeleteAirTrackEvent;
end AirTrackThread;
```

Figure 4: Thread Declarations

mode-sensitive timing analysis, for example.. If the conditions need to be explicitly modeled we can utilize the AADL Behavior Annex which will go into ballot in early 2008. Note that some of the calls embedded in the bridge component require special interpretation. Instead of becoming calls in a call sequence, they get mapped into the appropriate message communication mechanism in AADL. The logical thread architecture expressed in AADL can then be optimized in a number of ways. For example, we can take advantage of the fact that multiple logical threads operate at the same rate and can therefore be executed by a single OS on the same processor. Another interesting optimization is the use of the Bin Packing algorithms [14] to assign threads to processors. This algorithm minimizes the number of processors needed to run the system while reducing the amount of bandwidth needed to communicate these processors. A number of run-time properties may not be part of a PIM. They must be added by the designer to the resulting AADL model. These include:

- End-to-end latency requirements (response times)
- Periodicity of events, both external (e.g. sensor interrupts) and internal (timers)
- Period, deadline, worst-case execution time for threads
- Execution time of subprograms
- Processor Speed
- Network Bandwidth

Other properties include security related properties [25] or fault related properties [23]. This allows us to analyze multiple perspectives of the PSM from a single architecture model. Examples of time related analyses include scheduling analysis to determine whether all threads complete by their deadline for given deployment configuration [30], and validation that a required end-to-end latency can be achieved [22].

3.2 From the AADL Behavioral annex to Formal semantics of the AADL execution model in TLA+

3.2.1 *Introduction to the behavioral annex*

AADL incorporates very restricted form of behaviors: it is possible to specify mode changes in response to events and mode dependant sequences of subprogram calls. Actual behaviors are supposed to be described using the implementation language. However, AADL can be extended in two ways: by defining new sets of properties that can be attached to model elements and by defining annex languages allowing the inclusion of annex specific code to the AADL model. These two features have been used to attach abstract behaviors to AADL models. The proposed behavioral annex allows the expression of data dependant behaviors so that more precise behavioral analysis remain possible. A behavior can be attached to a subprogram and to a thread. It is described using an extension of AADL mode automata. For example, let us consider a sporadic thread sending an event if its two input data ports have different values:

```
thread check
features
  i1: in data port D;
  i2: in data port D;
  c: in event port;
  o: out event port;
properties
  Dispatch_Protocol => Sporadic;
  Period => 1 ms;
end check;

thread implementation check.i
annex behavior_specification {**
states
  s: initial complete state;
transitions
  s -[c?]-> s { if (i1 != i2) o!; };
**};
end check.i;
```

3.2.2 *Main annex features*

The main characteristics of the behavioral annex and of its link with the AADL execution model are the following:

- On the first dispatch, the execution of the component starts on an initial state. Several initial states are allowed, as well as several next states through identical guards: the annex can describe non-deterministic abstract behaviors even if the actual behavior will be deterministic. This is particulary usefull to specify the behavior of the environment of a component.

- A thread completes its execution after reaching a complete state and executing the action part of the transition. Subsequent dispatches will start from that state.

- A state can be declared composite. A subautomaton is attached to such states. A transition can exit a substate and have as target a descendant of any ancestor state of the composite state containing it.

- The annex has access to the content of input ports as it was at dispatch time. Subsequent received data are not accessible. The contents of an event data port is accessible as a queue data structure whose value is read from the port at dispatch time.

- Data written to output ports is transmitted at completion.
- Events and event data are explicitely sent by specific actions.
- Asynchrony is supported through events, access to shared data via AADL data access declarations and remote procedure calls.
- Real-time aspects are supported by three primitives: `delay(min,max)` specifies suspension during a non--deterministic amount of time; `computation(min,max)` abstracts a computation by its non-deterministic CPU consumption. A timeout construct can be used within guards.

The activity of the threads relies on the AADL execution model. When the thread is dispatched, the automaton starts on its initial state and does not receive events or data until completion and next dispatch. Before dispatch, data present on input ports are copied to internal port variables. Port variables associated to event or event data ports are seen as queue data structures.

3.2.3 HRT-HOOD-like synchronization declarations

The behavioral annex also defines properties extending AADL synchronization protocols. The interface of an AADL component is defined by a set of ports and by a set of subprograms. These subprograms can be called remotely in a synchronous way: the caller waits for the completion of call. This protocol corresponds to HRT-HOOD [11] `HSER` protocol and to the default AADL remote procedure call protocol. Other protocols can be defined: in the `LSER` (loosely synchronous) protocol, the caller waits for the call to be accepted; in the `ASER` (asynchronous) protocol, the caller continue its execution immediately. The later protocol is equivalent to sending a message to an event data port. However, attaching one of these protocols to a subprogram makes easier future changes in the protocol.

As in HRT-HOOD, it is possible to attach activation conditions to entry points. These dispatch conditions apply to subprogram entries as well as to input event (data) ports. The annex can be used to specify a dispatch condition: the thread is dispatched if an event or a subprogram call is received and if the automaton is in a state where this receipt is waited for. This property supposes an extension of AADL execution model where activation conditions are associated to input ports.

3.2.4 Timed primitives

Most timing features are declared in pure AADL (period, WCET, . . .). The annex makes these declarations more precise using three constructs taking as argument timed values whose syntax is that of AADL property constants expressed with a time unit.

- the `computation(min,max)` call abstracts an action to its bcet/wcet which can be compared to that of the subprogram or thread to which the behavior is attached.
- the `timeout(delay)` predicate can be used in a guard. It becomes true if the given duration has elapsed since the last completion. Implementing this primitive also needs an extension to the AADL run time support: a thread is dispatched when a given set of events occurs or at a given date.
- the `delay(min,max)` call suspends the current thread during the given (non deterministic) amount of time. The implementation of this primitive needs specific support of the AADL run time support: the thread must be descheduled as if it were waiting for the completion of a remote procedure call. This primitive comes from the Cotre project [21]. Its final integration is still under study because it may require to alter the AADL execution model.

3.2.5 High level constructs

The features presented in this section have been introduced in the annex in order to represent hierarchical automata. An alternative would be to use AADL hierarchical components (systems and thread groups) and modes to represent them. However, such a solution would identify states of the hierarchical automata to threads or systems, but their semantics are different. Thus, the annex allows the declaration of hierarchical states.

Such a refinement of annex states is extended to AADL modes, which comes to defining mode dependent behaviors. For this purpose, the annex declares AADL modes as composite and associates sub-automata to AADL modes as well as extended transitions between modes. Thus, behaviors, as well as instances, connections and property values can be mode dependant.

As in statecharts, annex states as well as AADL modes can be declared concurrent and refined as sets of regions. Each region contains a possibly hierarchical automaton. The current state of the concurrent automaton is defined to be the set of current states of each sub-automata. Sub-automata become active if a transition targets the concurrent state or if transitions from a fork state target states within each region. The concurrent machines terminate if a transition starting from the concurrent state can be fired, or if all transitions starting from states of each region and targeting a join state can be fired. Region automatons are composed in parallel but, to conform with AADL specification, region automatons are not associated with threads. They allow a more compact specification of a non-deterministic sequential behavior.

3.2.6 Interface with the AADL execution model

The semantic of the behavioral annex is based on the semantics of the AADL execution model. This has an impact on control and data flows. In this section we describe how the AADL execution model and the behavioral annex interact.

3.2.6.1 Thread synchronization.

The execution environement gives the control to the annex when a thread starts its execution. The automaton starts its execution either in the initial state, or in the last visited complete state. Completion occurs when the automaton reaches the complete state. At this time the next dispatch condition is computed. It depends on the guards and events of the exiting transitions.

Synchronization on a conjunction of events is supported by several real time operating systems. For example RTEMS [43] offers a primitive which suspends a thread until all or some of a set of conditions are satisfied. Such a call could be added to the AADL API. Such an extension is already needed to support the dispatch of a thread waiting for one of the events in guards of transitions starting from the current state. This new feature would allow to express transitions with guards containing several receipts. The combination of the two needs (or-wait, and-wait) leads us to the proposal of an advanced dispatch condition expressed in AADL using a boolean condition over ports. It would be possible to go one step further by specifying a minimal number of messages on a port.

3.2.6.2 Accessing data from the annex.

Messages are received through event, data or event data ports. Event and event data ports are associated to queues. On dispatch, zero, one, all, or a specified number of elements of the port queue are transferred to the thread, depending on the value of the `Dequeue_Protocol` property. If it has the AllItems or MultipleItems value, then all or a specified number of the queued messages are stored in an internal queue and can be accessed by the behavioral annex using the dequeue operator on the port name. The old contents of the queue is lost. Otherwise, it has the value OneItem and one message is popped from the queue and transferred to the thread. For data ports, zero or one message is transferred. If no new messages are transferred, the old contents is seen by the annex and the fresh attribute associated to the port is set to false. Single data is read using the port name. Each access to internally queued event/data dequeues one message from the internal queue. Messages are sent through output event, data or event data ports. A data can be stored in data and event data ports using the port name as a variable. It is implicitly transferred after deadline or completion. An event can be sent immediately to an event or event data port.

3.2.7 Formal semantics of the AADL execution model in TLA+

In this section, we are concerned by setting a formal semantics for the AADL execution model. The goal of such a semantics can be twofold:

- first it can be used to reason about an AADL design formally. Actually, since our semantics is stated in the TLA+ formalism [31], it will be possible to verify properties through model checking.
- second it can be used as a formal specification for the development of an AADL execution platform. One can imagine that an actual implementation would be certified with respect to the proposed model.

This study follows a first experiment of a partial translation of AADL and the behavioral annex using UPPAAL as target language [10], which could not be extended because of limitations of timed automata.

We are concerned by a subset of the execution model. We try to select a small enough subset so that it can be formalized easily, but expressive enough to allow performing small tests. The only components used in our model are threads and data. The communication between threads can be done through ports or shared variables.

We have developed a generic TLA+ architecture easily customizable (see fig. 5). The `Threads` and ports modules specify the execution model described in the AADL standard. The `application_thread` module contains the behavior of each thread. The module `AADL model` is a configuration file which describes the actual AADL model, e.g., threads, ports, connections, according to the AADL source text. The mapping between an AADL model and the configuration file is really easy and can be done automatically.

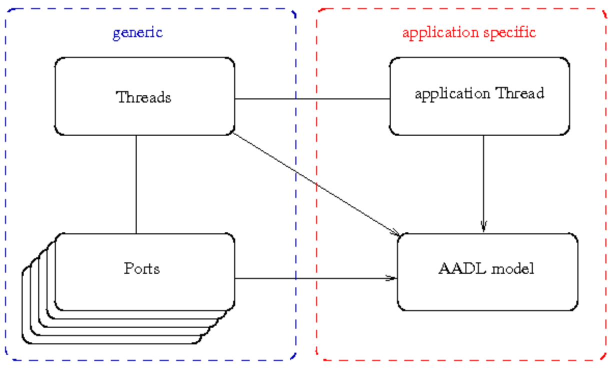


Figure 5: ***Architecture of the TLA modules***

The behavior of a thread is represented be a relation between its input and its output: the thread activity updates private port variables and shared data. Shared data is supposed to be locked while a thread requiring access to it is dispatched. Thus, we can consider that the thread activity between dispatch and completion is atomic, even if the thread can be preempted. Premption acts only on timing aspects. The transition associated to the thread activity updates the computation time of the thread as well as the global clock of the system.

As there is no primitive in TLA+ to handle time we have to use an explicit time approach, i.e. the evolution of time is modeled by the evolution of a specific variable. The execution model of AADL is expresed in TLA+ by a stopwatch automaton [4]. This automaton is a simplification of the description provided by AADL the standard which does not take into account initialization's mechanisms and remote procedure calls.

With this TLA+ model we are able to perform some schedulability analysis, and to study the dimensions of buffers.

3.2.8 Perspectives

First of all, we plan to extend the coverage of the AADL language. Then, we want to use the behavioral annex to produce a TLA+ description of the thread behavior. In TLA an automaton expressed in the annex formalism would be translated into a set of transition, each transition of this set would be a possible path in the automaton. The choice of the transition would depend on the internal state of the automaton and on the content of its ports. We could use the +cal [32] language, an algorithmic language based on TLA+, as an intermediate language between the behavioral annex and TLA+. In the presented framework we can verify standard properties like scheduling or buffer overflow. A possible evolution of this work is to permit the verification of application specific properties. For example we could use AADL flows to specify some timing properties and verify them using the TLA+ model. To go further, a property language and the corresponding annex could be defined as an extension of AADL.

3.3 An algorithm language to describe atomic components behavior

A state machine is a set of states and a set of commands, and each command generates a deterministic state transition. This formalism is interesting for the understanding, but not enough detailed for a formal specification and verification of non functional properties. In the embedded real-time critical domain, architectural configurations have to be formally verified. The construction of these systems is mostly based on the integration techniques of multiple domain-specific languages and tools. At the lowest design level, which is in fact the most detailed, we need dynamic, functional, set elements and operators to properly design the atomic components behavior. Leslie Lamport [31] wrote that "the most important aspect of the level of abstraction is the grain of atomicity, the choice of what system changes are represented as a single step of behavior.

Therefore, we present some of the languages that may answer to the need of atomic component behavior formal describing.

• **Natural language and Controlled natural languages:**

Natural language does not give an entry point to describe the algorithm complexity we have to handle. Of course, most of the requirements are still using the Natural language, even it is a bit formalized by the way of use cases. From uses cases, we can easily define scenarios in another languages (MSC, SDL, UML sequence diagrams, etc). It would be then a waste of time to go through an intermediate language, when you can directly express a behavior into a simple and understanding language, and even more, another opportunity to loose the requirements accuracy, with the language mappings.

Controlled Languages (RDL, Requirements Description Language as well) do provide a sufficient formalism to follow the requirements and ensure traceability, but are not effective considering the need of expressing complex algorithms. They may replace first order logic languages in very particular situations, with some restrictions.

• **Pseudo code :**

The simple and usual pseudo code is only interesting to bring the processing at an abstract level (genericity) with a minimum effort on the syntax. Usual pseudo code, however, is not linked to any logic language, and not rich enough to be able to follow state changes.

• **Formal languages, process algebras, for specifying and verifying concurrent aspects :**

CSP is a very rich language and perhaps the most adapted language to describe process behavior. CSP has nondeterministic choice operators, allows model-checking, and facilitate parallel composition of numerous primitive processes. In contrast, CSP is not easy to manipulate from the start of a project, when defining the requirements with average engineers. It needs complex transformations and mappings from the AADL specifications and then dramatically reduces the traceability.

• **The +CAL algorithm language :**

The problem to solve is we have to consider the right level of formalization. Meanwhile we choose to work with an semi formal ADL, at this step of the architecture design, we are constrained to insert an semi formal algorithm language, if we want to have an homogeneous level of formalism.

```
--algorithm bakery
variables Extraction = [k \in 1..N |-> FALSE],
Rank= [m \in 1..N|-> 0];

process a_process \in 1..N
variable q;
 begin
 Extraction[a_process]:= TRUE;
 Rank[a_process]:= 1 + max(Rank[1]..Rank[N]);
 Extraction[a_process]:= FALSE;
 q:=1;
 while q /= N+1 do
   while (Extraction[q])
     do skip;
   end while;
   while ((Rank[q]/= 0) /\ ((Rank[q], q) <
           (Rank[a_process],a_process)))
     do skip;
   end while;
 q:=q+1;
 end while;

      \*The critical section
      Rank [a_process]:=0;
      \* non-critical section...
end process
end algorithm
```

Listing 1: Example of a Mutex Algorithm in the +CAL algorithm language

Here, we present an example of a mutual exclusion algorithm to show how conflicting access to shared resources by concurrent processes can be solved. From the usual pseudo-code, we would not have enough constraint formalism to generate a formal language. Choosing the +CAL language brings numerous advantages. First, as all algorithm languages, it allows to describe algorithms, at the high level of description they have to be. Second, from the two versions: p and c, we can easily generate a p-like code (Pascal, Ada) or a c-like language. Third, this algorithm language is made to be translated to a formal language TLA+, with a simple command of the TLC translator (java pcal.trans bakery). The +CAL language provides the advantages of high-level code and the precision of a formal language that can be mechanically checked [32]. At last but not the least, from a TLA+, we can also translate a PVS specification, when the problem is not decidable.

If we aim to specify the behavior of each component in a continuous way, from the smallest granularity component (thread) behavior specification to the highest (modes configuration), we have to integrate the description of the atomic component behavior with the *AADL* global system implementation. The main argument that leads us to expand the language lies in the fact the real-time distributed systems we are studying are using complex algorithms to specify their atomic component behavior which have a huge influence on the whole of the resulting architecture. Not including construction of algorithms in architecture design represents a high risk that they will never be totally taken into account when choosing the final architecture configuration. To ensure the requirements traceability in the analysis and design of the architecture , we consider therefore, that algorithms must appear as a significant element of the design.

Therefore, we propose to expand the AADL language by the "annex mechanism" in order to integrate the main algorithmic specifications that play a role in configurations. Including an algorithm language also provides an opportunity for automatic proof and clean code generation. Considering the previous target of encompassing critical system requirements, it is necessary to retrieve proof at each level of architecture design. It is not enough to claim that proofs must occur during the earliest steps of the design. The final mode configurations must be chosen using proof argumentations. This leads us to integrate a formal behavior specification language right inside the architectural specification.

Although ADLs are more focused on *programming in the large*, the behavior of a component is correlated with the behavior of its subcomponents. Consequently, we have to build bridges between a strict and static design process that only focuses on the topology, and a dynamic process that raises all the design levels to bring the parameters of an optimal configuration to the upper levels, particularly, when we choose to implement an algorithm that involves atomic components.

3.4 Summary

The combination of xUML and AADL enables the creation of both an executable PIM and its corresponding analyzable PSM. Modelers can therefore carry out functional testing on the relatively simple PIM, using the increasingly sophisticated xUML testing environments, and then move on to assess performance, safety and other non-functional characteristics using an automatically generated AADL model. In this section we discussed how to extract the concurrency requirements from the xUML model into an AADL model to enable the analysis of its runtime architecture in various deployment configurations. We discussed an extension to AADL, the Behavior Annex, whose objective is to allow for annotating AADL models with behavioral characteristics, in particular interaction behavior between concurrent components. We also discussed an approach for capturing atomic component behavior through an algorithmic language extension.

4. A GENERAL APPROACH TO IMPROVE USABILITY OF AADL USING MDD

In recent trends, the Architecture Analysis and Design Language (AADL) has proven a good candidate as a modeling language for software-intensive systems. At the same time, Model Driven Development (MDD) is gaining popularity as a development process. This section provides an assessment of AADL for users with limited experience with the language. In this assessment, we concentrate on modeling the software aspect of real-time embedded systems, towards implementation in a procedural or object-oriented language. Strong component orientation, complex component composition and property ambiguity are identified as three issues in the usability of AADL as a modeling language for software-intensive systems. For resolving these issues, an approach is presented through integration of AADL models in a model driven development process with specifically designed model transformations. This approach enhances the usability of AADL for software developers.

The structure of this section is as follows. In 4.1 we assess the use of AADL, and present the three issues we identified. In 4.2 we propose an approach to ease these issues. Work related to this approach is stated in 4.3. Finally, in 4.4 we draw conclusions. [1]

[1]The described work is part of the EUREKA-ITEA SPICES project, and partly funded by the Flemish government institution IWT (Institute for the Promotion of Innovation by Science and Technology in Flanders).

4.1 Usability assessment of AADL

Over the last decade, a number of ADLs have been proposed [33]. From these proposals, the Architecture Analysis and Design Language (AADL) [1] has received increasing interest from mission-critical development industries. To assess the strengths and weaknesses of AADL, we have considered a representative modeling example [26] and identified a number of issues in the usability of AADL for system developers.

4.1.1 Strong component orientation

AADL uses the component-based paradigm, in which components are the primary modeling concepts. Szyperski [12] defines a component as follows: a software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. Bachman et al. [17] extend this definition by stating in a component-based system (i) components and frameworks should have certified properties; and (ii) these certified properties should provide the basis for predicting properties relative to the whole system built out of those components. From both these definitions, we can see that AADL has succeeded in building the component-based paradigm into a modeling language. AADL forces developers to describe a system in terms of hardware and software components, annotating each component with specific properties. Analyzing functional properties of the system and non-functional properties of components in the system, enables the prediction of non-functional properties of the system.

At the same time, AADL implicitly captures a modeling methodology, by dividing components into ten categories. Categories represent functionality of components rather than concepts in which they are implemented. Hardware-only systems tend to have a limited number of components with clearly defined and strictly separated functionality. For the description of hardware systems, AADL is very appropriate. AADL allows modeling processor components for processing instructions, memory components for storing data etc. Software components on the other hand, may have functionality which is obfuscated or very application-specific. What is missing, is the possibility to model components that do not serve functionality belonging to one of ten categories supported by AADL. Moreover, components can be implemented in different paradigms using specific concepts. As an example, modeling a software system implemented in an object-oriented language comes down to making a translation from components towards classes. Because components are at a higher level of abstraction and have coarser-grained granularity, part of the implementation of the software system cannot be modeled in AADL. For the description of software systems, developers using AADL may face a gap between modeling concepts and implementation concepts. In these cases, strong component orientation makes for additional complexity, which increases the entry level to the use of AADL.

4.1.2 Complex component composition

AADL allows composing components according to legality rules mentioned in the SAE AADL standard, which adds hierarchical structure to the description of systems. Legality rules are specific to every component category. Because these legality rules are different for each component category, component composition is quite complex. Moreover, components can be extended, which adds complexity to composition. It may be unclear to AADL users with limited experience what component can be composed with other components.

4.1.3 Property ambiguity

Components in AADL can be annotated with properties, expressing non-functional properties of the component. Legality rules in the AADL standard define which component categories are allowed to have a certain property, and which properties apply to each component category. A complete AADL model annotated with a specific set of properties makes the analysis of a non-functional property possible on the system model.

Unfortunately, there is no clearly defined relation between property sets and model analysis. Tools such as OSATE [47] provide guidance in user manuals, but are not always complete. Neither does the SAE AADL standard clearly state what properties need to be annotated in an AADL model, to analyze the model for a given non-functional property. Unclear relations between component categories and properties on the one hand, and between analyses and properties on the other hand, make for two causes of property ambiguity.

4.2 Improving usability of AADL using model driven development

In recent years, a lot of research has been done in the field of model driven development (MDD). In an MDD approach, models at different levels of abstraction are used as primary artifacts throughout the software development process. In a typical MDD approach, three levels of abstraction can be identified. At the highest level, a platform independent model (PIM) of the application is built. Through model transformation, this PIM is transformed into a platform specific model (PSM). The PSM contains platform-specific details of the application towards the targeted execution platform. Finally, at the lowest level of abstraction is code in a platform specific implementation language. In this manner, platform specific knowledge is shifted from the model to the model transformations.

4.2.1 AADL as a platform

Issues in the usability of AADL as a modeling language, identified in section 4.1, increase the entry level for the use of AADL, and decrease feasibility of the use of AADL in a development process. If we consider AADL and its runtime environment as being a platform, this means the use of AADL requires a considerable amount of platform specific knowledge. In the OMG MDA guide [29], a platform is defined as a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented. We propose a model driven development approach that reduces the requirement of this knowledge, and increases feasibility of the use of AADL by considering AADL as a target platform in an MDD approach.

At the highest level of abstraction in our approach is an incomplete AADL model of the software application, which we will refer to as the AADL PIM. The AADL PIM may or may not comply with AADL structural and semantic requirements. One level down is an AADL model of the software application which does fully comply to AADL semantic requirements. We will refer to this model as the AADL PSM, which is obtained from the AADL PIM through structural model transformations. Finally, at the lowest level of abstraction is an AADL model annotated with all necessary properties to analyze the application for a specific non-functional property. We will refer to this model as the AADL analysis model, which is acquired from the AADL PSM through non-functional model transformations. Our approach is semi-automatic, in the sense that properties added through non-functional transformations are given a proper value by the developer himself.

In the following, obtaining the AADL PIM is explained first. Next, structural and non-functional model transformations are explained. Finally, a note is given on how to integrate these steps.

4.2.2 Obtaining an AADL PIM

The UML has become a widespread modeling language in software development. Although the UML may not be specific enough for some purposes, it offers the advantage of having a lower entry level. To use AADL with UML, the SAE is currently working on an UML 2.0 profile for AADL as an annex to the SAE AADL standard.

We propose the usage of an AADL PIM as a UML model annotated with AADL stereotypes according to the UML profile for AADL. The obtainment of this model is split into two phases. First, a UML model is obtained that can be the result of an analysis or design phase early in the development process of the application, or it can be the result of reverse engineering an existing application. Second, this model is annotated with UML stereotypes identifying component categories for the application, to make a complete AADL PIM of the application.

4.2.3 Structural transformations

To comply with legality rules for component composition, the AADL PIM should be transformed. Component composition rules are related to structural properties of the software application. In this section, we use the term structural transformations for model transformations modifying structural properties of the application. The result of structural transformations is the AADL PSM of the application, which does comply with all SAE AADL standard legality rules. Platform specific knowledge, in this case enforcing SAE AADL standard legality rules, is built into structural transformations. As an example, consider a system with a network device. The driver for this device will be represented as a thread component. This thread component can not be a direct subcomponent of the system, since this is not in compliance with AADL legality rules. In this case a structural transformation will add a process subcomponent to the system, and make the device driver thread a subcomponent of this process. As an AADL PSM is more platform specific, instead of UML we use a domain-specific language to denote the PSM. Domain-specific languages allow adding more detail to modeling concepts. In our approach, we implemented this domain specific language using Ecore [18]. Models complying with an Ecore metamodel are supported by a XMI-file format representation, which has the advantage that these models are interoperable with tools such as

OSATE [47]. OSATE has a built-in semantic check, which allows checking compliance of the AADL PSM with AADL legality rules. Structural transformations consist of a number of transformation rules. These transformation rules apply the following:

- Transform UML classes with a component category stereotype applied, to a component type of the same category in a domain-specific modeling concept.
- Provide a component implementation for the component type according to AADL legality rules, if necessary.
- Transform UML composition links between UML classes into subcomponent relations between component implementations.
- Transform directed associations between UML classes into `in`, `out` or `in out` features, and provide connections between these features.
- Transform dependency relations between UML classes into `required bus access` features in the component type, and `bus access` connections in the component implementation.

Structural transformation rules as stated above could be modified or extended to enrich the transformation process. We consider extensions of these transformation rules as a future direction of research.

4.2.4 Non-functional transformations

The AADL PSM obtained through structural transformations is used as an input for one or several non-functional transformation steps. Non-functional transformations add properties to components of the AADL PSM for analysis of the model towards exactly one non-functional property of the system. Properties added to component type or implementation through non-functional transformations, have the right name and type but do not possess a value. Giving the newly added properties in this model a value, is left to the developer. The result of a non-functional transformation is an AADL analysis model.

As a proof of concept, we designed a non-functional transformation for schedulability analysis. To allow for schedulability analysis of an AADL system model, a number of properties need to be present on certain components in the system. These are the following:

- There needs to be at least one thread and at least one processor component in the system.
- Processors need to specify a scheduling policy.
- Threads need to specify a dispatch policy, period, deadline and binding to a processor. If a thread is aperiodic, sporadic or background, `in event ports` and `in event data ports` need to have an incoming connection.

Therefore, the transformation rules which make up the non-functional transformation for schedulability analysis apply the following:

- If no processor component type is present in the system model, add a processor component type.
- If a processor component has no `Scheduling_Protocol` property, add this property to the processor type or implementation.
- If no thread component type is present in the system, add a thread component type.
- If a thread component misses one of `Dispatch_Protocol`, `Period`, `Deadline` or `Actual_Processor_Binding` properties, add this property to the thread component type or implementation.
- If a thread component has the value of its property `Dispatch_Protocol` set to `aperiodic, sporadic or background`, add a connection for `in event ports` or `in event data ports` which do not have an incoming connection.

Since not all property sets are standardized and the relation between properties and analyses can vary amongst analysis tools, non-functional transformations are tool specific.

4.2.5 Integration of the transformation process

Structural and non-functional transformations make up a logical sequence of model transformations, which can be integrated and chained. Transformation chain modeling languages have been proposed as in [9]. A chain of structural and non-functional transformations makes up a transformation process targeted specifically towards model-based analysis of functional and non-functional properties, allowing an eased analysis process of both existing and newly designed software systems.

After a chain of model transformations has been applied to an AADL PIM, the AADL analysis model can be used with a tool that implements the AADL runtime environment. An example of such a tool is OSATE [47], which comes with a number of built-in analysis plug-ins. A schedulability analysis plug-in is one of the OSATE built-in plug-ins, as are semantic check, safety level and other analysis plug-ins.

4.3 Related work

A number of related approaches have been proposed. Dissaux [41] presents an approach to model transformation for AADL in combination with UML. In contrast to the approach we propose, Dissaux concentrates on the analysis of components from legacy code aimed specifically towards use with the HOOD Stood tool [41]. Whereas Dissauxs approach is comparable to the structural transformations we propose, our approach offers non-functional transformations as well. Bertolino and Mirandola [3] propose an approach for the specification and analysis of performance related properties of components using the RT-UML profile. Although the approach also uses a UML profile, it is not targeted towards model driven development like the approach presented in this section.

Finally, a number of tools are available that address the issues discussed in this section. Ocarina [28] allows model manipulation, generation of formal models, to perform scheduling analysis and generate distributed applications. Cheddar [19] is a free real-time scheduling tool, providing a simulation engine and feasibility tests. Cheddar provides a number of features to ease the development of specific schedulers and task models.

4.4 Conclusion

Although the use of AADL as a modeling language offers a number of interesting advantages such as the prediction of non-functional properties, we have identified a number of usability issues for introducing AADL. First, software developers may face a gap between modeling concepts and implementation concepts. Second, component composition rules in AADL are rather complex and hard to adopt. Third, the relation between the property mechanism used in AADL and prediction mechanisms used in analysis tools is unclear. These issues entail an increased entry level and decreased feasibility for using AADL in software-intensive embedded system development.

To ease these issues and facilitate the use of AADL in embedded system development, we proposed a model driven development process using AADL models on three levels of abstraction. The AADL PIM of an application is at the highest level of abstraction, and does not necessarily fully comply with AADL semantic requirements. Through structural transformations, this model is transformed into an AADL PSM. Structural transformations modify structural properties on the model, so that the AADL PSM does comply with all semantic AADL rules. Finally, using one or several non-functional transformations the AADL PSM is transformed into a full AADL analysis model. Non-functional transformations add non-functional properties to the AADL PSM, specific for analysis of the model towards one non-functional property. Using transformation chains, structural and non-functional transformations can be chained and integrated into a model driven development process aimed specifically towards the early verification of non-functional properties of the system.

5. DEFINITION OF A STANDARDIZED ADL FOR THE AUTOMOTIVE DOMAIN

The EAST-ADL is an architecture description language, dedicated to automotive embedded electronic systems, developed in the context of the ITEA cooperative project EAST-EEA [15] finished in 2004. This language is intended to support the development of automotive embedded software, by capturing all the related engineering information. The scope is the embedded system (hardware and software) and its environment. On top of the formal description of the elements, the language defines several abstraction levels that reflect different views and details of the architecture. They implicitly reflect different stages of an engineering process, but the detailed process definition is company specific. The EAST-ADL language constructs support:

- vehicle feature modeling including variability concepts to support product families,
- vehicle environment modeling to define context and perform validation,
- structural and behavioral modeling of software and hardware entities supporting refinement to code and binaries in the context of distributed systems,
- requirements modeling and tracing with all modeling entities,
- other information part of the system description, such as a definition of component timing and failure modes, necessary for design space exploration and system verification purposes.

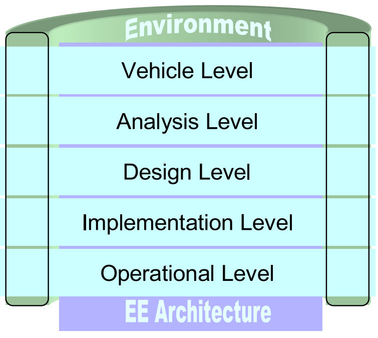


Figure 6: The structure of EAST-ADL system models.

The language is structured in five abstraction layers (see Figure 6), each with corresponding system representation (in parenthesis):

- Operational Level supporting final binary software deployment (*operational architecture*),
- Implementation Level describing reusable, platform independent code and AUTOSAR-compliant software and system configuration for hardware deployment (*implementation architecture*),
- Design Level for detailed functional definition of software including elementary decomposition (*design architecture*),
- Analysis Level for abstract functional definition of features in system context (*analysis architecture*),
- Vehicle Level for elaboration of electronic features (*vehicle feature model*).

Note that the environment model spans all abstraction levels, and that requirements and variability constructs apply to modeling elements regardless of abstraction level.

The European project ATESST [7] is aimed at refining the EAST-ADL language in the context of dependability concerns, supporting OMG standard alignment and the new automotive domain standardization effort AUTOSAR [8]. To harmonize the EAST-ADL language with the AUTOSAR modeling approach, the lower levels of the language are being reworked to support software and hardware model entities standardized in the AUTOSAR templates.

In order to better support the development of complex automotive systems, the EAST-ADL does not only include means to create analysis and design models of the system to be developed (at varying abstraction levels), but also language means to

- specify required properties of the system at varying degrees of abstraction,
- trace requirements between system refinement and system decomposition levels,
- refine the specification of requirements by behavioral models,

- manage information related to verification and validation activities.

Methodically, EAST-ADL differentiates between functional requirements, which typically focus on some part of the "normal" functionality that the system has to provide (e.g. "ABS shall control brake force via wheel slip control"), quality requirements, which typically focus on some external property of the system seen as a whole (e.g. "ABS shall have an MTTF of 10.000 hours"), and safety requirements. Safety requirement attributes, for example, include safety integrity level (SIL), operation state, fault time span, emergency operation times, safety state, and functional redundancy to record dependability characteristics [27]. A requirement can be traced from the abstract vehicle model all the way to its derived requirements allocated to the final hardware and software components. Depending on abstraction level, some or all of a requirement's attributes are applicable.

Furthermore, EAST-ADL offers detailed means to explicitly model central artifacts of verification and validation activities (V&V) and to relate these artifacts to requirements. This allows for explicitly and continuously planning, tracking, updating and managing important V&V-activities and their impact on the system in parallel to the development of the system. The combination of a V&V-case, its environment and its target object is described as a V&V context.

Another important focus of the ATESST project is to augment the EAST-ADL with means to support variability management and product-line oriented development. Variability has a growing significance in the automotive domain. Its complexity arises out of variation resulting from inevitable product differentiation as well as technically motivated variation due to the distribution of a variety of interacting functions over a number of hardware components from different suppliers. To rigidly analyze, define and evolve an automotive system's variability is the purpose of the variability management concepts provided by the EAST-ADL. On top of that, the notion of product-line oriented development means that the products offered by a certain manufacturer are not developed independently from one another but instead a single, variable product is developed from which the individual members of the product line are derived, thus shifting the focus of development from the individual product to the product line as a whole. In this sense, a product line can be defined as follows: "A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [13].

The industrial applicability of a modeling technique relies heavily on tool support and mature concepts. Tool availability and validation of concepts is fostered by standardization or alignment with existing standards. Standardization of the EAST-ADL is ensured in mainly two contexts: AUTOSAR and the OMG.

The AUTOSAR platform is a future de-facto standard for automotive embedded systems. It defines a set of middleware components that provides a standardized platform for application software. The modeling approach for application software components and hardware architecture contains the details necessary for correct integration. In ATESST, the entities corresponding to software and hardware components are taken from the AUTOSAR standard, but put in a context where the EAST-ADL system modeling concepts can be used. AUTOSAR-compliant software architecture can thus be modeled with support for e.g. variability, requirements, traceability and verification and validation. In this sense the EAST-ADL language complements the AUTOSAR initiative by providing higher-level abstraction means of modeling and analysis.

The OMG standards in the scope of EAST-ADL include UML2, SysML [36] and Marte [16, 24]. The refined EAST-ADL will be aligned with these approaches. Alignment with UML2 is done by construction because EAST-ADL is designed as a UML2 profile. SysML concepts are being reused wherever applicable, for example regarding requirements and plant modeling constructs. Many of the EAST-ADL concepts are thus reused SysML concepts, or specializations of these. Marte, on the other hand, is an ongoing effort to define a standard UML profile for Modeling and Analysis of Real-Time and Embedded systems. Harmonization with Marte is done by integrating Marte concepts in the EAST-ADL where real-time and embedded system properties are modeled. Furthermore, as members of the ATESST project are taking part in the elaboration of the Marte profile, it is planned that the EAST-ADL profile could be issued as an annex to the subsequent version of Marte.

Another important objective of the ATESST project is the development of a prototypical modeling environment. Based on Eclipse, it features a UML modeling tool with support for profile editing and application as well as various customizations to meet the need of a particular modeling approach. Ongoing development includes several additional plugins to be connected to this core platform so as to provide additional means of analysis, model checks and editing. This is also intended as being a validation platform for the EAST-ADL language as a whole.

6. CONCLUSIONS

The previous example on the use of MDE, and specially its two key principles abstraction and refinement, shows that it may already provides adapted and satisfying solutions to some of the issues the engineers have to face today to develop their real-time embedded systems. We show in this paper that dedicated languages were existing, some of them being standard, for instance MARTE, AADL or also EAST-ADL. These domain specific languages are good examples of how providing the right abstraction level within one language may be so helpful to ease system development. But it has been also shown that suitable abstractions are necessary but not enough. Refinement, and moreover assisted or automatic refinement, is also required to fully enable models to hold its active role in the new emerging model-driven development processes.

But do not forget that the transition from code-oriented to model-oriented development will not alleviate the need for answers to the usual problems of traceability, configuration and version management, etc. Therefore, if MDE is intended to be successful in industry, solutions to these issues will have to be available for all tools claiming to be MDE-compliant.

Finally, let's notice that definitions and developments of a set of MDE artifacts for development, validation, exploitation and maintenance of real-time embedded systems lies at the core of new large-scale joint research programs in the one hand, such as for example the *Software Factory* project of the System@tic French competitiveness cluster (http://www.systematic-paris-region.org) that federates more than 40 partners coming from both research and industry domains. In other hand, a lot of powerful open-source project dedicated to MDE are available, a lot of them being developed in the context of the Eclipse project, as the OSATE/Topcased toolkit (http://www.topcased.org), the Papyrus tool for UML2 (http://www.papyrus.org), the ATL language for model transformations, or also the Ocarina tool suit which is an AADL model processing suite written in Ada (http://ocarina.enst.fr).

7. REFERENCES

[1] Architecture analysis and design language (aadl). Technical Note AS5506, SAE, November 2004.

[2] *Architecturing Dependable Systems IV*, chapter Towards Improving Dependability of Automotive Systems by Using the EAST-ADL Architecture Description Language. LNCS 4615. Springer-Verlag, 2007.

[3] Bertolino A. and Mirandola R. Modeling and analysis of non-functional properties in component-based systems. In *International Workshop on Test and Analysis of Component Based Systems (TACoS 2003)*, number 82, Warsaw, Poland, November 2003. Electronic Notes in Theoretical Computer Science.

[4] Yasmina Abdeddaim and Oded Maler. Preemptive job-shop scheduling using stopwatch automata. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 113–126, 2002.

[5] SAE AS-2C. www.aadl.info.

[6] SAE AS-2C. *SAE Architecture Analysis and Design Language (AADL)*. SAE International, Nov 2004. SAE AS5506.

[7] ATESST. Atesst project web-site.

[8] AUTOSAR. Autosar project web-site.

[9] Vanhooff B., Ayed D., Van Baelen S., Joosen W., and Berbers Y. Uniti: A unified transformation infrastructure. In *ACM/IEEE 10th International Conference on Model-Driven Engineering Languages and Systems (MoDELS 2007)*, Nashville, USA, 2007.

[10] Jean-Paul Bodeveix, Mamoun Filali, Miloud Rached, David Chemouil, and Pierre Gaufillet. Experimenting an aadl behavioural annex and a verification method. In *Data Systems In Aerospace (DASIA), Berlin-Germany, 22/05/06-25/05/06*, http://www.esa.int/publications, 2006. European Space Agency (ESA Publications).

[11] A. Burns and A. Wellings. *A Structured Design Method for Hard Real-time Systems*. Elsevier, 1995.

[12] Szyperski C. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley, 1998.

[13] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[14] D. de Niz and R. Rajkumar. Partitioning bin-packing algorithms for distributed real-time systems. *International Journal of Embedded Systems*, 2(3/4):196–208, 2006.

[15] East-eea project web-site.

[16] H. Espinoza, J. Medina, H. Dubois, S. Grard, and F. Terrier. Towards a uml-based modelling standard for schedulability analysis of real-time systems. In *Proceedings of MARTES Workshop at MODELS Conference*, 2006.

[17] Bachmann F., Bass L., Buhman C., Comella-Dorda S., Long F., Robert J., Seacord R., and Wallnau K. Technical concepts of component-based engineering. Technical Report CMU/SEI-2000-TR-008, SAE, 2000.

[18] Budinsky F., Steinberg D., Merks E., Ellersick R., and Grose T. *Eclipse Modeling Framework*. Addison Wesley, 2003.

[19] Singhoff F., Legrand J., Nana L., and Marc L. Cheddar: a flexible real time scheduling framework. In *ACM SIGAda International Conference*, Atlanta, US, 2004.

[20] J. Delatour F. Thomas, S. Gérard and Francois Terrier. Software real-time resource modeling. In *Proceedings of the International Conference Forum on Specification and Design Languages (FDL) 2007*, Barcelona, Spain, September 2007.

[21] J.-M. Farines, B. Berthomieu, Jean-Paul Bodeveix, Pierre Dissaux, Patrick Farail, Mamoun Filali, Pierre Gaufillet, Hicham Hafidi, Jean-Luc Lambert, P. Michel, and F. Vernadat. *The Cotre Project: Rigorous Software Development for Real Time Systems in Avionics*. COLNARIC, ADAMSKI, WEGRZYN, novembre 2003.

[22] P. H. Feiler and J. Hansson. Flow latency analysis with the architecture analysis and design language (aadl). Technical Note CMU/SEI-2007-TN-010, Software Engineering Institute, 2007.

[23] P. H. Feiler and A. Rugina. Dependability modeling with the architecture analysis and design language (aadl). Technical Note CMU/SEI-2007-TN-043, Software Engineering Institute, 2007.

[24] S. Grard and H. Espinoza. *Rationale of the UML profile for Marte*. 2006.

[25] J. Hansson and A. Greenhouse. Dependability modeling with the architecture analysis and design language (aadl). Technical Note CMU/SEI-2007-TN-005, Software Engineering Institute, 2007.

[26] Hamid I. Flight control system, 2005.

[27] ISO. Iso cd 26262. Planned for end of 2007.

[28] Hugues J., Zalila B., and Pautet L. Rapid prototyping of distributed real-time embedded systems using the aadl and ocarina. In *18th IEEE/IFIP International Workshop on Rapid System Prototyping (RSP'07)*, Brazil, 2007.

[29] Miller J. and Mukerji J. *MDA Guide Version 1.0.1*. Object Management Group, 2003.

[30] M. Klein, T. Ralya, B. Pollak, R. Obenza, and M. Gonzalez Harbour. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Springer, 1993.

[31] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, Jul 2002.

[32] Leslie Lamport. The +cal algorithm language. In *+CAL*, Jul 2006.

[33] Medvidovic N. and Taylor R.N. A classification and comparison framework for software architecture

description languages. *IEEE Transactions on Software Engineering*, (26):70–93, 2000.

[34] OMG. Uml profile for modeling and analysis of real-time and embedded systems (marte) rfp, 2005.

[35] OMG. Uml profile for schedulability, performance, and time, v1.1, formal/05-01-02. http://www.omg.org/cgi-bin/doc?formal/2005-01-02, 2005.

[36] OMG. *Systems Modeling Language (SysML) Specification*, 2006. ptc/06-05-04.

[37] OMG. Uml profile for modeling qos and ft characteristics and mechanisms, v1.0, formal/06-05-02. http://www.omg.org/cgi-bin/doc?formal/06-05-02, 2006.

[38] OMG. Lockheed martin (mda success story). http://www.omg.org/mda/mda_files/LockheedMartin.pdf, 2007.

[39] OMG. Uml profile for marte, beta 1, ptc/07-08-04. http://www.omg.org/cgi-bin/doc?ptc/2007-08-04, 2007.

[40] OMG. Uml superstructure, v2.1.1, formal/07-02-05. http://www.omg.org/cgi-bin/doc?formal/07-02-05, 2007.

[41] Dissaux P. Aadl model transformations. In *DASIA 2005 Conference*, Edinburgh, UK, 2005.

[42] C. Raistrick, P. Francis, J. Wright, C. Carter, and I. Wilkie. *Model Driven Architecture with Executable UML*. Cambridge University Press, 2004.

[43] RTEMS. Rtems c user's guide. http://www.rtems.com/onlinedocs/releases/rtemsdocs-4.6.99.3/share/rtems/pdf/c_user.pdf. 117-118.

[44] M. J. Balcer S. J. Mellor. *Executable UML: A Foundation for Model-Driven Architecture*. Addison Wesley, 2002.

[45] S. Gerard S. Taha, A. Radermacher and J.-L. Dekeyzer. An open framework for hardware detailed modeling. In *IEEE Proceedings of SIES'2007*, Lisbon, Portugal, July 2007.

[46] S.Gerard S. Taha, A. Radermacher and J.-L. Dekeyzer. Uml-based hardware design from modeling to simulation. In *Proceedings of the International Conference Forum on Specification and Design Languages (FDL) 2007*, Barcelona, Spain, September 2007.

[47] SEI AADL Team. Osate: An extensible source aadl tool environment. Technical report, SEI, 2004.