From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite

JEROME HUGUES GET-Télécom Paris – LTCI-UMR 5141 CNRS
and
BECHIR ZALILA GET-Télécom Paris – LTCI-UMR 5141 CNRS
and
LAURENT PAUTET GET-Télécom Paris – LTCI-UMR 5141 CNRS
and
FABRICE KORDON Université Pierre & Marie Curie, Laboratoire d'Informatique de Paris 6/MoVe

Building distributed deal-time embedded systems requires a stringent methodology, from early requirement capture to full implementation. However, there is a strong link between the requirements and the final implementation (e.g. scheduling, resource dimensioning). Therefore, a rapid prototyping process based on automation of tedious and error-prone tasks (analysis, code generation) is required to speed up the development cycle. In this article, we show how the AADL (*Architecture Analysis and Design Language*), appeared late 2004, helps solve these issues thanks to a dedicated tool-suite. We then detail the prototyping process and its current implementation: Ocarina.

Categories and Subject Descriptors: D.2.2 [**Design Tools and Techniques**]: Evolutionary prototyping, Modules and interfaces; D.2.11 [**Software Architectures**]: Languages, Patterns; I.2.2 [**Automatic Programming**]: Program synthesis

General Terms: Design, Languages, Performance, Reliability, Verification

Additional Key Words and Phrases: AADL, Distributed, DRE, Embedded, Ocarina, PolyORB-HI, Real-Time

1. INTRODUCTION

Building Distributed Real-Time Embedded (DRE) systems involves many tightly coupled steps, from requirements capture (number of tasks and their interactions, non-functional attributes) to validation (feasibility of scheduling) down to implementation and testing.

However, the distance between requirements and implementation usually slows down this process: one has to carefully respect non-functional attributes when implementing tasks; any change in the specification has to be carefully propagated

This work has been funded in part by the IST Program of the European Commission under project IST-004033 (ASSERT).

at the implementation level; interactions between entities have to be mapped onto run-time entities in a safe manner (deadlock-free, no starvation, no overrun, etc...).

Hence, developers and system architects need common interchange models to dialog and exchange their requirements and concerns. The AADL (*Architecture Analysis and Design Language*) [SAE 2004] recently appeared as an architecture description language suitable to describe systems, from high-level concerns down to implementation.

"Evolutionary" prototyping is now becoming a well accepted development approach. It is based on a central model that is refined as long as it is not satisfactory. Programs can be generated from this model and constitute a version of the product. The last refined model corresponds to the final system. Also called "Model Driven Engineering" (MDE), it is promoted by OMG. The goal of this paper is to propose a prototyping methodology based on AADL and dedicated to DRE systems. AADL is interesting compared to other modeling formalisms as it is backed by several industrials from the space and avionics domain. Tools already exist to build and exploit AADL models, from early validation to full implementation.

In the following, we give a brief overview of the AADL, we then discuss how the AADL can serve as a vehicle for a rapid prototyping methodology for DRE systems. We show the resulting prototype is very close from the final product. Finally, we present our current work on the Ocarina AADL tool suite and assess its use to build High-Integrity DRE Systems.

2. AN OVERVIEW OF THE AADL

AADL (*Architecture Analysis and Design Language*) [SAE 2004] aims at describing DRE systems by assembling blocks separately developed.

The AADL allows for the description of both software and hardware parts of a system. It focuses on the definition of clear block interfaces, and separates the implementations from these interfaces. It can be expressed using both a graphical or a textual syntax.

An AADL model can incorporate non-architectural elements: embedded or real-time characteristics of the components (execution time, memory footprint, etc...), behavioral descriptions, etc... Hence it is possible to use AADL as a backbone to describe all the aspects of a system.

An AADL description is made of *components*. The AADL standard defines software components (`data`, `thread`, `thread group`, `subprogram`, `process`) and execution platform components (`memory`, `bus`, `processor`, `device`) and hybrid components (`system`).

Components describe well identified elements of the actual architecture. *Subprograms* model procedures like in C or Ada. *Threads* model the active part of an application (such as POSIX threads). AADL threads may have multiple operational modes. Each mode may describe a different behaviour and property values for the thread. *Processes* are memory spaces that contain the *threads*. *Thread groups* are used to create a hierarchy among threads.

Processors model micro-processors and a minimal operating system (mainly a scheduler). *Memories* model hard disks, RAMs, *buses* model all kinds of networks, wires, *devices* model sensors, etc...

Unlike other components, *Systems* do not represent anything concrete; they actually create building blocks to help structure the description.

Component declarations have to be instantiated into subcomponents of other components in order to model an architecture. At the top-level, a system contains all the component instances. Most components can have subcomponents, so that an AADL description is hierarchical. A complete AADL description must provide a top-most level system that will contain the other components, thus providing the root of the architecture tree. The architecture in itself is the instantiation of this system.

The interface of a component is called *component type*. It provides *features* (e.g. communication ports). Components communicate one with another by *connecting* their *features*. To a given component type correspond zero or several implementations. Each of them describe the internals of the components: subcomponents, connections between those subcomponents, etc... An implementation of a thread or a subprogram can specify *call sequences* to other subprograms, thus describing the execution flows in the architecture. Since there can be different implementations of a given component type, it is possible to select the actual components to put into the architecture, without having to change the other components, thus providing a convenient approach to configure applications.

The AADL defines the notion of *properties* that can be attached to most elements (components, connections, features, etc...). Properties are attributes that specify constraints or characteristics that apply to the elements of the architecture: clock frequency of a processor, execution time of a thread, bandwidth of a bus, etc... Some standard properties are defined; but it is possible to define one's own properties. A more detailed introduction to the AADL can be found in [Feiler et al. 2006].

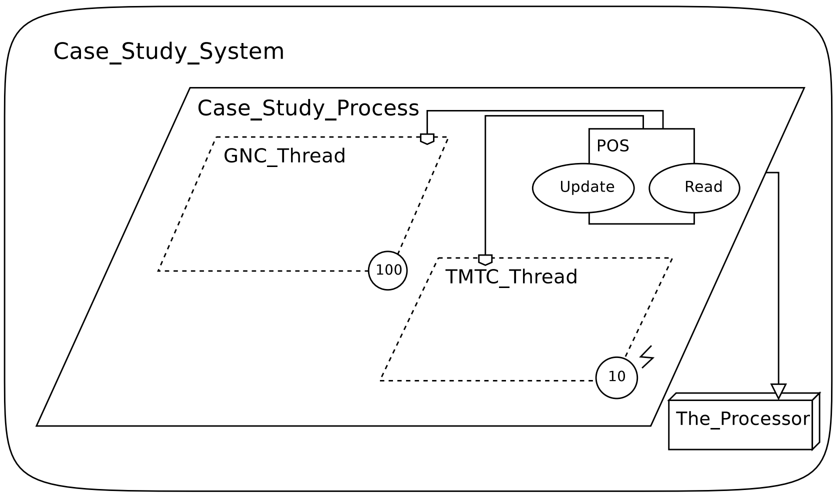

Fig. 1. A simple AADLmodel

Figure 1 presents a simple AADL model that depicts two threads: one periodic

(GNC, “guidance navigation control”); one sporadic (TMTC, “telemetry/telecommand”) that interact to read and update a shared variable (POS, “position”). This models a satellite guidance system.

Let us note the model depicted in figure 1 is only the high-level view of the system, additional elements have to be added to detail the signature of methods that apply on POS, the deployment of each element onto a physical architecture, worst case execution time (WCET) of each element, etc...

Projects such as OSATE [SAE 2006] define modeling environments to build AADL models, using the Eclipse platform.

We have developed the Ocarina standalone tool-suite [ENST 2006] to process AADL models and allow the programmer to develop, configure and deploy distributed systems. Ocarina offers scheduling analysis capabilities, connection with formal verification tools, and more notably code generation to Ada 2005.

AADL provides two major benefits for prototyping DRE systems. First, compared to other modeling languages, AADL defines low-level abstractions including hardware descriptions. These abstractions are more likely to help design a detailed prototype close to the final product. Second, the hybrid system components help refine the architecture as they can be detailed later on during the design process. For both reasons, we state that AADL is a good candidate for both prototyping and designing DRE systems.

3. A RAPID PROTOTYPING PROCESS FOR DRE SYSTEMS

A DRE system is unique in that it should support two contradictory constraints: it should be compatible with needs for critical systems (life-, mission-, business-) and their normative process; but also embrace rapidly new standards or technologies [Leveson 1997]. This is a hot problem in distributed or embedded systems where new standards or products arise frequently. It is difficult to handle both specific development techniques (for embedded systems) as well as the evolution of standards that might require big changes in a system’s architecture.

The contradiction is that both specific implementation techniques (i.e. to manage a small memory footprint) and software architecture flexibility (to integrate new techniques, for example when a product is upgraded) are simultaneously needed. Also, a fast way to handle these evolutions is needed for market purposes.

Therefore, a prototyping process is of interest to test as soon as possible the impact of deployment decisions, or the use of one software/hardware component in the system. Tools can support this process and provide quick feedback and executable programs to the developer for testing purposes.

3.1 Building prototypes

Two approaches in prototyping are usually distinguished [Kordon and Luqi 2002]:

(1) “throw-away”: prototypes are built to validate a concept, prior to implementing the real system. The throw-away approach is used to refine requirements.

(2) “evolutionary”: prototypes tend to become the final product. Prototypes are refined to create more accurate ones. The last prototype actually corresponds to the final system (figure 2). Then, feedback on the system may be provided at various levels and the model is the main reference for describing the system.

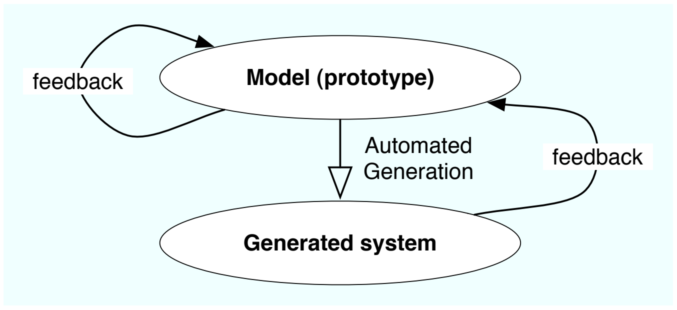

Fig. 2. Evolutionary prototyping

Given testing and validation costs, we believe an "evolutionary" approach should be applied to DRE. Development should be refined through successive prototypes that help designers to explore the design space of an application. It is a way to preserve knowledge on the software and hardware that is precious and costly to rebuild. Therefore, one should leverage previous knowledge to build new systems, or refine existing ones.

3.2 Requirements for prototyping

A distributed real-time and embedded system can be seen as a collection of many requirements covering many domains. System designers and developers need to describe both functional and non-functional requirements. These requirements must then be sorted and enforced at the deployment level (e.g. specific dispatching protocols, transport mechanisms), or flagged as wrong by tools (potential deadlocks, resource overrun).

Therefore, we list the following requirements for a prototyping process dedicated to DRE systems:

[R1] support design-by-refinement: allowing one to test for different scenarios from a common model; or to precise some elements later (promoting late binding decisions);

[R2] be extensible to support new policies (e.g. dispatching, QoS, security, etc...) via user-defined attributes;

[R3] support domain-specific analysis (e.g. model checking, schedulability analysis, safety analysis);

[R'1] support DRE domain entities: software (threads, shared data) and hardware (processors, buses, sensors);

[R'2] handle deployment of the system at both hardware and software levels in a consistent manner.

We note the first three requirements are general ones, while the latter two are specific to DRE systems. In this paper, we focus on DRE systems and address them as a whole.

These requirements call for modeling formalisms as media to support refinement, setting of attributes and analysis. Such modeling formalisms must support the complete cycle depicted in figure 2.

Hence, built models should be exchangeable between tools, and eventually lead to code generation to ease the construction of prototypes. Such a prototyping process should therefore be compatible with an MDA-like development cycle [OMG 2001]. To reduce model discrepancy, one common modeling notation should be used and conserved during the different steps of the process.

Without loss of generality, we chose the AADL as a core modeling language to support the different steps of system construction, from early prototypes to final implementation. Supported entities and extensible property sets allow one to build full models and adapt them to the application context. Furthermore, analysis tools can process the models to assess its viability, point out potential problems, and complete the specification when possible (full resource dimensioning, execution metrics).

3.3 Related Work

Generating High-Integrity code from a model is not limited to AADL models. In [Bordin and Vardanega 2005], the authors state that generating code minimizes the risk of several semantic breaches when translating the model towards code. The manual coding exposes the developer to these breaches. They propose some guidelines to generate Ravenscar compliant Ada code from HRT-UML (Hard Real-Time UML) which is a customized version of UML to model hard real-time systems. However the use of UML does not allow a low-level description of the system. In addition, the different views of the system (software view, concurrency view...) use different formalisms. Therefore, to have a coherent model, one must modify all views at each change of the system which does not help the rapid prototyping.

To have full control over middleware customization and to achieve minimal memory footprint, the graphical design tool Zen-Kit [Gorappa et al. 2005] allows middleware customization by controlling the actual components embedded by the application and minimizes the difficulty of this custom configuration. However, the focus of this approach on RT-CORBA does not allow the modification of the middleware architecture by adding new components that would diverge from the standard such as a lightweight invocation protocol. Besides, the ZEN middleware is based on many of the patterns and techniques from the ACE ORB (TAO [Schmidt et al. 1998]). These patterns are based on dynamic memory allocation and object oriented programming and cannot be used in High Integrity Real-time systems.

More closely to this paper's scope, the Annex D of the AADL [SAE 2005] describes some coding guidelines to translate the AADL software components into source code (Ada and C). These rules are not complete mapping specifications, but they provide guidelines for those who want to generate code from AADL models. The annex does not address issues such as configuring the middleware depending on the AADL model. In our knowledge, there is no implementation of the Annex D of the AADL standard.

More concretely, STOOD, which is a tool developed by Ellidiss Software [Ellidiss-Software 2007], allows users to model their real-time applications using the AADL or the notations proposed by the HOOD method. STOOD allows the code generation from AADL to Ada by converting AADL models to HOOD models and then applying the HOOD method to Ada mapping rules. However, the generated code does not rely on a middleware layer and works only for local applications. Also, some issues like automatic configuration and deployment and prior verification and analysis of the model are not addressed.

In the following, we illustrate how the AADL allows the rapid prototyping of complete ready-to-run systems.

4. RAPID PROTOTYPING USING THE AADL

In this section, we detail the use of AADL in a prototyping process, detailing our model processing chain, using Ocarina and companion tools. Then, we assess it.

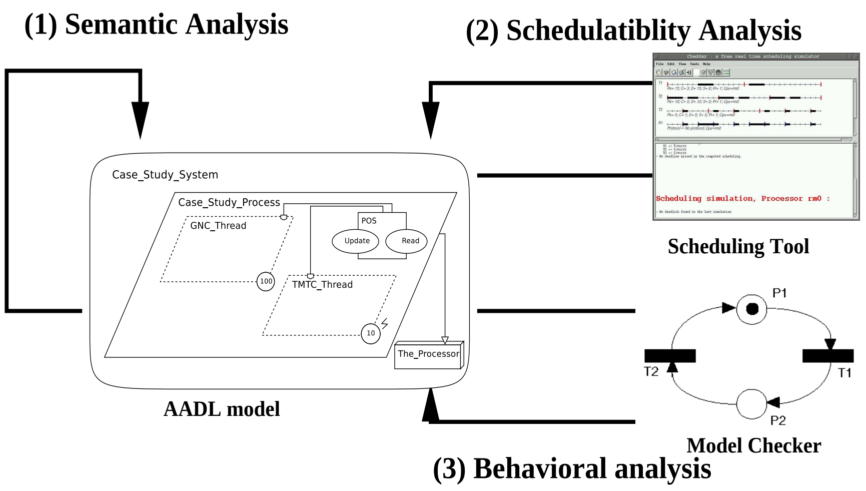

Fig. 3. Exploiting AADL models

4.1 Requirement Capture

AADL has been designed to build DRE systems. It is therefore no surprise it is well suited to express their requirements in an easy way. The process implements the following (possibly iterative) path to define and refine:

(1) data types and related functions to operate on them
(2) supporting runtime entities (*threads*) and interactions between them (through *ports* and *connections*)
(3) association of subprograms to threads
(4) mapping of threads onto *processes* and binding processes to hardware entities to form the deployed system.

For each AADL entity, `properties` can be attached to refine its non-functional attributes (e.g. its WCET, its priority or its transport mechanisms).

Furthermore, AADL allows one to refine the description of each entity to detail more precisely its behavior or some non-functional attributes, allowing one to support design-by-refinement; or even to support inheritance to provide multiple specializations of one component (e.g. a periodic sensor implemented as either a thermal or speed sensor). This allows us to have a library of reusable components and helps in rapid prototyping by refining and extending them.

We list some sample (reduced) AADL models in code snippets 1 and 2. These snippets correspond to an expanded description of the system represented graphically in figure 1.

```
data POS
features
  Update : subprogram Update;
  Read   : subprogram Read;
properties
   Concurrency_Control_Protocol => Priority_Ceiling;
end POS;

data implementation POS.Impl
subcomponents
  Field : data POS_Internal_Type;
end POS.Impl;

subprogram Update
-- Updates the internal value of POS
features
  this : requires data access POS.Impl;
properties
  source_language => Ada95;
  source_name     => "Repository.Update";
end Update;
```

Listing 1. AADL data component

```
thread TMTC_Thread
features
  TMTC_POS : requires data access POS.Impl;
end TMTC_Thread;

thread implementation TMTC_Thread.Impl
calls {
  Welcome  : subprogram TMTC_Identity; -- Display a begin-cycle message
  TMTC_Job : subprogram TMTC_Job;      -- Compute
  Update   : subprogram POS.Update;    -- Update the shared variable
  Bye      : subprogram TMTC_Identity; -- Display an end-cycle message
};
connections
  Cnx_TMTC_1 : data access TMTC_POS -> Update.this;
properties
  Dispatch_Protocol        => Periodic;
  Period                   => 100 ms;
  Compute_Execution_time => 0 ms .. 50 ms;
  Deadline                 => 100 ms;
end TMTC_Thread.Impl;
```

Listing 2. AADL TMTC Thread

4.2 AADL Model Assessments

AADL models support both code generation and model analysis. Analysis can range from simple semantic analysis to more sophisticated ones such as schedulability analysis, model checking of the behavior of the nodes, etc...

In this section, we show how such analysis can be conducted using Ocarina, our AADL model processing suite. The supported development process is sketched in figure 3 and conforms to the evolutionary prototyping approach. We emphasize three main steps to be supported by our processing suite: *semantic analysis*, *schedulability analysis*, *verification of node behavior*.

4.2.1 *Semantic analysis.* Our AADL compiler Ocarina checks that the given AADL model conforms to the AADL grammar and that some additional restrictions are respected:

—All event or data ports are connected, except those used to drive thread's operational mode switch,
—All threads are either periodic or sporadic to be compliant with the Ravenscar profile,
—All shared data use a concurrent access protocol that bounds priority inversion (e.g. the Priority Ceiling Protocol mandated by the Ravenscar profile).

If an error is detected, the analyzer displays a comprehensible message describing the error and its location to help the programmer in rapidly correcting his prototype.

AADL defines one standard execution semantics, this allows us to go further and assess the system is coherent and can run prior to its generation and execution.

4.2.2 *Schedulability analysis.* Cheddar [Singhoff et al. 2004], an Ada framework provides tools and libraries to check whether AADL threads will meet their deadline at execution time. Cheddar uses the Ocarina [ENST 2006] libraries to analyze the AADL models.

From an AADL model, a model of interacting tasks is computed. Tasks can interact either locally sharing data through protected objects (or mutexes), or remotely through a communication bus. The first allows for traditional Rate Monotonic Analysis, while the second requires advanced techniques such as Holistic analysis [Tindell 1993]. Cheddar supports both; this enables one to check whether one's architecture can run within expected bounds.

4.2.3 *Verification of node behavior.* To do so, we transform the AADL model into a Petri net to perform formal verification. The transformation into Petri net is performed using Ocarina's Petri net generator module. The formal verification (e.g. absence of deadlocks, etc...) is performed using CPN-AMI, a Petri Net modeling and verification environment [Hamez et al. 2006; MoVe-Team 2007].

For each interaction pattern expressed in the AADL model (interacting tasks, sent messages, etc...), we build the corresponding Petri Nets and assemble them to build one full model representing the system. From this model, we can either explore its state space and look for deadlock (state from which no progression is possible), look for inconsistent state or test for more complex timed logical formulae (such as if event $\mathcal{E}$ holds, then output $\mathcal{O}$ is always emitted).

These analyses allow one to fully assess system viability prior to its execution on the target. If required, the model can be refined to correct the behavior, adjust WCET; the model can also be updated after running some checks: e.g. priority or bounds on buffers can be computed by Cheddar.

4.3 Code Generation Strategies

We use code generation facilities in Ocarina to 1) analyze the AADL model, 2) expand it, compute required resources and 3) generate code conforming to High-Integrity (HI) restrictions.

First, the model is built by the application designer, he maps its application entities onto a hardware architecture (1). Then, this architecture is tested for completeness and soundness, any mismatch in the configuration is reported by the analysis tool (e.g. lack of connection from an object) (2). Consequently, model processing occurs, and code is generated from this AADL model, following the rules we presented in the previous section (3). Finally, middleware components are selected and compiled together with the generated code and the user implementations and run on the target (4).

Code generation relies on well-known patterns for High-Integrity systems, inherited from previous work on code generation from Ravenscar [Bordin and Vardanega 2005] and classical design patterns for distribution such as the Broker [Buschmann et al. 1996], constrained to remove all dynamic behavior to be supported by the minimal middleware for HI systems.

We named this middleware "PolyORB-HI" (PolyORB High-Integrity) as a follow up to the PolyORB project we develop [Vergnaud et al. 2004]. It shares many common architectural notions while using a different code base. As for PolyORB, this middleware is built on isolated elements that embody key steps in request processing, allowing for finer configuration of each blocks.

PolyORB-HI [Hugues et al. 2006] strictly follows restrictions set by High-Integrity applications on object orientation, scheduling, use of memory. It is developed in Ada 2005 [ISO/IEC 8652:2007(E) Ed. 3 2006]. It is compliant with both the Ravenscar profile and the High-Integrity system restrictions (Annexes D and H of the Ada 2005 standard). High-Integrity system restrictions are facilities provided by the Ada 2005 standard to help developers understanding their program, reviewing its code and restricting the language constructs that might compromise (or complicate) the demonstration of program correctness. Most of these restrictions are enforced at compile time (no dispatching, no floating point, no allocator, etc...). This simply yet efficiently enforces no unwanted features are used by the middleware, increasing the confidence in the code generated while limiting its complexity. Code generated by Ocarina also follows the same compilation restrictions.

User code is also tested for consistency with the above restrictions. To ensure the user code does not impact scheduling (which might modify scheduling, and thus threatens asserted properties at the model-level), we ensure at compile-time it uses no tasking constructs (tasks and protected objects) by positioning the corresponding restrictions on all user-provided code. Any violation of these restrictions will then be reported by the Ada compiler.

These steps allow the developer to go from the AADL model to executable code and forth, using one common model annotated with all required functional and non-functional elements, including its code base. Each tool works on the same model, allowing one to debug or enhance it at different steps, following an evolutionary prototyping approach. We discuss metrics of this process in section 6.

4.4 System Deployment Strategies

Since the deployment and configuration of a distributed application are complex tasks, they are usually performed automatically. There are two approaches to deploy and configure automatically a distributed application:

—*Dynamically*, in this case, the selected components (deployment) are instantiated dynamically using object factories and the interaction with these components generally uses object oriented programming patterns [Schmidt et al. 2000] and dynamic binding. The parametrization of the components (configuration) uses dynamic memory allocation.

—*Statically*, in this case, an analysis of the application is performed to determine, at compile time, the exact middleware components to be used by the application (deployment) and the exact properties of these components (configuration).

The advantage of the dynamic approach is that it makes the application code (provided by the user or generated automatically) simpler from a middleware designer point of view. However, it has two major drawbacks that impede its use for HI systems: (1) object oriented programming and dynamic binding have several safety problems such as the difficulty to guarantee correct initialization of dispatch tables which are the most classical way to implement dynamic binding in compiled object oriented languages. Some analyses (dead-code detecting) and testing (code coverage) become very hard to perform on object oriented programs [Gasperoni 2006]. (2) Dynamic memory allocation is not deterministic which may cause a problem during WCET analysis for real-time systems.

The static configuration is more suitable for HI distributed systems. It requires that an analysis phase computes statically all the resources (memory, bandwidth) and configuration parameters (buffer sizes...) needed by the application. It is obvious that the analysis phase is very tedious if performed by the programmer (it has to be redone after any modification of an application parameter). However, if the application is modeled using a well-chosen modeling language, this model is analyzed automatically to compute all resources and a (large) part of the application code can be generated automatically and contains all the statically computed resources, and their configuration. We selected AADL to model facets of real-time embedded systems. AADL is sufficient to detail the deployment view of the application: nodes, processors, network buses, tasks on each node; properties refine the type of tasks (periodicity, priority) and its associated implementation.

We defined our distribution model as a set of *sender/receiver* tuples that interact through asynchronous messages. It is supported by an AADL architectural model that defines the location of each node, and the payload of the message exchanged as a thread-port name plus possible additional data. From a system's AADL description, we compute required resources, then generate code for each logical node. We review the elements supporting this distribution model:

(1) Naming table lists one entry per remote node that can be reached, and one entry per opened communication channel on this node. We build one static table per node, computed from the topology of the interactions described in the AADL model. It is indexed by an enumeration affecting one tag per logical

node, resulting in $O(1)$ access time to communication handlers (e.g. sockets, SpaceWire).

(2) Marshallers handle type conversion between network streams and actual application data. They are derived from data components and thread interfaces, they describe the structure of data to be exchanged. This is computed beforehand from the AADL models, code has $O(payload)$ complexity.

(3) Stubs and skeletons handle the construction and analysis of network messages. Stubs transform a request for an interaction into a network stream, skeletons do the opposite operation. Both elements are built from AADL components interface and actual interaction between threads. We exploit this knowledge to have $O(payload)$ components.

(4) Protocol instances are asynchronous communication channels, set up at node initialization time. The complexity of the action performed by these instances depends on the underlying transport low-level layer (e.g. sockets, SpaceWire).

(5) Concurrent objects handle the execution logic of the node. We build one task per periodic or sporadic AADL thread. Subsequent tasks are built for the management of the transport low-level layer (at least one additional task to handle incoming network messages). Finally, we build one protected object (mutex-like entity) to allow for communication between tasks. Let us note all these objects strictly follow the Ravenscar Computation Model, ensuring code is analyzable using RMA (Rate Monotonic Analysis) and Hierarchic scheduling analysis[Davis and Burns 2005].

The generated code provides a framework that will call directly user code when necessary. This relieves the user from the necessity to know an extensive API, and allows a finer control of the behavior of the system that is under the sole responsibility of the code generation patterns.

The generated code can be interfaced with the user-implementations in several ways. A module of Ocarina, *Build_Utils*, handles the generation of makefiles with proper options to use these implementations:

(1) **Source code:** The user gives the source code of the implementation (in Ada or C). In this case, the generated code will include the calls to the corresponding unit name and the generated makefile will handle the compilation of the user source files automatically.

(2) **Object files or libraries:** The user indicates in the AADL model that the implementation of a given subprogram is provided by a set of object files and/or libraries. In this case, the generated code will include the calls to the compilation units and the generated makefile will automatically include the linking options against the given objects and/or libraries.

(3) **High level specialized language (LUSTRE, ...):** The generated code will include the calls to the compilation units that have to be generated from the high level language source. The makefile will handle the generation of sources from the high-level implementation and the compilation of these sources automatically.

This allows a very rapid and flexible design of the system and does not restrict the user implementations into a unique way.

Generating code to configure these entities reduces the need for a large middleware API. Hence, buffers, tasks, naming tables are allocated directly and statically from the application models. This enables a finer control on the code structure, reducing the need for complex structures to register application entities such as CORBA COS Naming, and the hand writing of error prone setup code (e.g. DDS [OMG 2004] policies).

5. THE MPC CASE STUDY

This case study has been provided by our partners from the IST-ASSERT project. It is an extended version of the model we presented in figure 1.

5.1 Scenario

The figure 4 shows the software view of our case study. This model holds three nodes, each is a spacecraft with different roles:

(1) SC_1 is a leader spacecraft that contains a periodic thread which sends its position to SC_2 and SC_3.

(2) SC_2 and SC_3 are follower spacecraft. They receive the position sent by SC_1 with a sporadic thread (`Receiver_Thread`), update their own position and store it in a local protected object. A second thread in these two spacecraft reads periodically the position value from the local protected object, and "watches and reports" all elements at that position (e.g. earth observation, etc...).

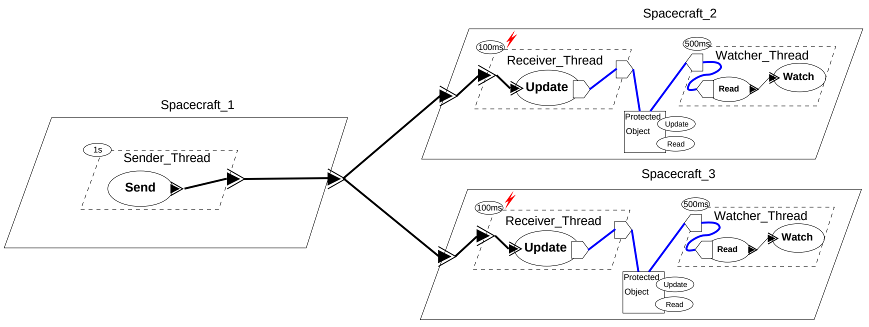

Fig. 4. Software view of the MPC case study

This model gathers typical elements from Distributed High Integrity (D-HI) systems, with a set of periodic tasks devoted to the processing of incoming orders (`Watcher_Thread`), buffers to store these orders (*Protected Object*) and sporadic threads to exchange data (`Receiver_Thread`). These entities work at different rates, and should all respect their deadlines so that the `Watcher_Thread` can process all observation orders in due time.

The software view only represents how the processing is distributed onto different entities (threads) and gathered as AADL processes to form partitions. The next step is to map this view onto a physical hardware view, so that CPU resources can be affected to each node.

The figure 5 is a graphical representation of the deployment view of the system. It only shows the global architecture of the application (number of nodes, their mapping to hardware components). It indicates that each partition is bound to a specific CPU, and how the communication between partitions occurs, using different buses. The details of each node will also be described using AADL.

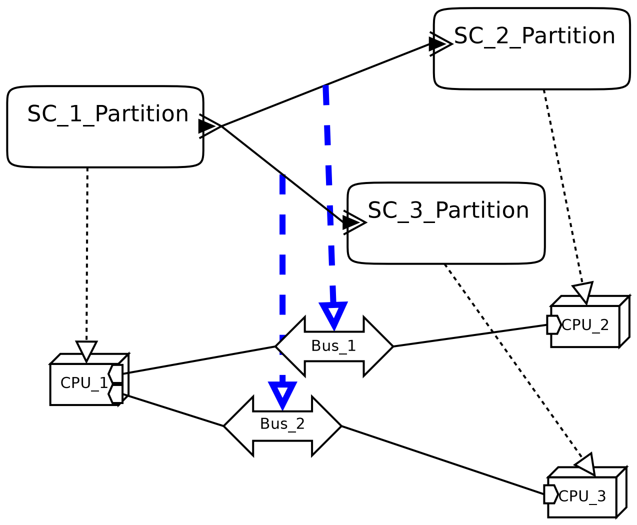

Fig. 5. Deployment view of the MPC case study

These two views are expressed using the same modeling notation, they can be merged to form the complete system: interacting entities in the software view represent the processing logic of the system, whereas the hardware view completes the system deployment information by allocating resources.

From this combined view, a set of analyses can be conducted, in conformance with the process we propose. In the following, we apply the methodology given in section 4 to analyze, generate code and deploy the MPC case study.

5.2 AADL Models

To build this system, we sorted out the different fundamental blocks, in an incremental fashion, from basic interaction between entities to their projection onto a hardware architecture built around multiple nodes.

The flexibility of AADL allows us to partially define components and use them in other components. This is very useful during the first steps of prototyping where every detail of the system is not clear yet. Details can be added to these components either by means of AADL properties or by component extension, without having to redefine all other components.

5.2.1 *Data types.* AADL data components model the messages that are exchanged among the nodes of a distributed application or inside one of these nodes.

To express the kind of a data type, we use custom AADL properties that are interpreted by our code generator to produce the proper type for the application (see listings 3 and 4).

```
-- The simple component type

data Component_Type
properties
  ARAO::Data_Type => Integer;
end Component_Type;

-- The record type used to vehicle information between nodes

data Record_Type
end Record_Type;

data implementation Record_Type.Impl
subcomponents
  X : data Component_Type;
  Y : data Component_Type;
  Z : data Component_Type;
end Record_Type.Impl;
```

Listing 3. MPC's classic data types

```
-- The protected type used to store and consult data in a node

data Protected_Type
  -- Update and Read modify the state of Protected_Type, protected
  -- against concurrent accesses using the Priority Ceiling Protocol.
features
  Update : subprogram Update;
  Read   : subprogram Read;
properties
  Concurrency_Control_Protocol => Priority_Ceiling;
end Protected_Type;

data implementation Protected_Type.Impl
subcomponents
  X : data Component_Type;
  Y : data Component_Type;
  Z : data Component_Type;
end Protected_Type.Impl;
```

Listing 4. MPC's protected data type

5.2.2 *Subprograms.* Subprograms encapsulate the behavioral aspects of a distributed application. They are modeled using the `subprogram` AADL component. Since AADL does not allow the description of the system behavior, the implementation of AADL subprograms must be provided by the user.

The implementation of a subprogram may be written entirely by the user. In this case he must indicate the source file or the pre-built libraries that contain the implementation. The listing 5 shows the `Update` subprogram. Since this is an accessor subprogram to a protected data type, it has a *required access* to this data component (the "`This`" feature). This subprogram is implemented in Ada and its implementation is given by the programmer in the package called `MPC`.

Subprograms can also be entirely specified with AADL if their behavior is limited to the call of other subprograms. The listing 6 shows the `Sender_Thread_Wrapper` subprogram which simply calls another AADL subprogram called `Observe_Object` (not showed in the listing). This subprogram models the "job" of a thread contained in SC_1 that sends periodically the spacecraft position to SC_2 and SC_3. The `connections` clause in the subprogram model describe the data flow between the caller subprogram and the callee.

```
subprogram Update
  -- Updates the protected local object
features
  Update_Value : in parameter Record_Type.Impl;
  This         : requires data access Protected_Type.Impl;
properties
  Source_Language => Ada;
  Source_Name     => "MPC.Update";
end Update;
```

Listing 5. MPC's opaque subprogram

```
subprogram Sender_Thread_Wrapper
features
  Data_Source : out parameter Record_Type.Impl;
end Sender_Thread_Wrapper;

subprogram implementation Sender_Thread_Wrapper.Impl
calls {
  Send : subprogram Observe_Object;};
connections
  parameter Send.Data_Source -> Data_Source;
end Sender_Thread_Wrapper.Impl;
```

Listing 6. MPC's wrapper subprogram

AADL subprograms can be modeled in several other ways. Ocarina allows even the direct reference to source written in synchronous languages such as LUSTRE or to combine the "opaque" way with the "pure call sequence way" to have hybrid subprograms that are more flexible. All this gives the programmer more flexibility when prototyping his system.

5.2.3 *Threads.* Threads are active parts of a distributed application. A node *must* contain at least one thread. The thread's interface is constituted by "ports". There are three kinds of ports:

(1) `event ports` which can be assimilated to signals and are used to trigger the threads and possibly change their operational mode.
(2) `data ports` which can be assimilated to messages between threads.
(3) `event data ports` which are like `data ports` but their reception triggers the receiver thread.

Threads may be periodic, i.e. triggered by a time event. In this case, and conforming to the *Ravenscar Profile* for HI systems [Dobbing et al. 2003], they must not be triggered by any other event. Therefore, they must not contain `in` `event [data] ports`. The listing 7 shows the AADL model of the periodic thread `Sender_Thread` that is located in the node SC_1 of our case study. This thread sends a data of type `Record_Type` shown in listing 3. The dispatch protocol of the thread and its period are specified using standard AADL properties. In the thread implementation, we describe the "job" of the thread by giving the subprogram that models its activity.

Threads may also be sporadic, in this case they are triggered by an "incoming event". The listing 8 shows the AADL model of the sporadic thread `Receiver_Thread` that is located in SC_2 and SC_3 and is triggered by the reception of a position sent from SC_1 by thread `Sender_Thread`. The modeling of sporadic threads is very similar to the modeling of periodic threads except that they *must* have at least one `in event [data] port` in order for them to be triggered. The `Period` given for `Receiver_Thread` indicates the minimal inter-arrival time between two successive events and is necessary for the schedulability analysis of HI systems. The reader should note that the receiver thread activity consists of updating the local protected object that denotes the spacecraft position, so it requires an access to this variable. Then, it connects this requirement to the subprogram that does the thread job which will call the `Update` subprogram.

```
thread Sender_Thread
features
  Data_Source : out event data port Record_Type.Impl;
properties
  Dispatch_Protocol => Periodic;
  Period            => 1 Sec;
end Sender_Thread;

thread implementation Sender_Thread.Impl
calls
   {Send : subprogram Observe_Object;};
connections
  parameter Send.Data_Source -> Data_Source;
end Sender_Thread.Impl;
```

Listing 7. MPC's periodic thread

```
thread Receiver_Thread
features
  Protected_Local : requires data access Protected_Type.Impl;
  Data_Sink       : in event data port Record_Type.Impl;
properties
  Dispatch_Protocol => Sporadic;
  Period            => 100 Ms;
end Receiver_Thread;

thread implementation Receiver_Thread.Impl
calls
   {Update : subprogram Protected_Type.Update;};
connections
  data access Protected_Local -> Update.Protected_Local;
  parameter Data_Sink         -> Update.Data_Sink;
end Receiver_Thread.Impl;
```

Listing 8. MPC's sporadic thread

5.2.4 *Processes.* Processes are the AADL components used to model the nodes of distributed applications. The listing 9 shows the AADL model of the process turning on SC_1.

```
process Sender_Process
features
  Data_Source : out event data port Record_Type.Impl;
end Sender_Process;

process implementation Sender_Process.Impl
subcomponents
  Sender : thread Sender_Thread.Impl;
connections
  event data port Sender.Data_Source -> Data_Source;
end Sender_Process.Impl;
```

Listing 9. MPC's SC_1 node

5.3 Code Generation

In this section, we show how code generation helped us to rapidly prototype the MPC case study and make it evolve from a very basic example to an elaborated distributed application in a Ravenscar-compliant manner.

The first models were relatively simple and resulted in a first implementation similar to the one depicted by figure 1. This implementation was evaluated to assert the feasibility of the system and the first set of requirements. Thanks to this first evaluation, both the requirements and the models were refined to integrate more complex constructions. For instance, compared to the current models (figure 4), the first version of the system was including only one node and discarding any remote communication. These models have been enriched to evaluate the appropriateness of introducing operational modes in threads. This last version is not presented here as it is not fully finalized.

This prototyping process helped us to analyze the case study on a native platform in order to easily debug and evaluate it before running it on an embedded platform. It allowed us to progressively integrate the automatically generated components with the functional user components and then to integrate the result with the predefined middleware components. In the case of MPC, the models were updated at several times to well separate the functional part from the non-functional part. The integration phase has been taken into account very soon in the prototyping process. Thanks to the automatic code generation, all these refinements were handled in a very short time.

As we wanted our system to be analyzable, it had to be compliant with the Ravenscar Concurrent Model and all the High-Integrity Systems restrictions. This compliance is ensured by the Ada compiler and are performed once all the code sources are available. Therefore, the code generation step is a core step in the prototyping process as it allows one to detect very soon any restriction violation. In the context of MPC, several restrictions violations were detected in the user components and have been fixed at compile time.

5.4 Deployment

The separation between software and hardware in AADL allows the programmer to model all the software part of his application, test it with a "native" platform (generally a PC). Then, if the tests are successful, he can reuse the same software part with the actual hardware AADL. In addition, going from one hardware architecture to another is reduced (in most of the times) to the modification of the values of a very few number of AADL properties.

5.4.1 *Buses.* A bus is the physical mean to perform the connection between the different components of an application. For example, SpaceWire buses [ECSS 2003] are used to connect distributed application nodes in the space domain.

The listing 10 shows a part of the AADL model for a SpaceWire bus. SpaceWire buses are the buses used to connect the different spacecraft of our example in figure 5. We can see that the main characteristics of the bus are expressed using AADL properties. Some of the properties are used without any prefix (`Allowed_Message_Size`). These are standard AADL properties for bus components. Some other properties are prefixed with "`ASSERT_Properties::`" (`ASSERT_Properties::Access_Bandwidth`). These are properties that are user-defined, they are gathered in a custom property set called `ASSERT_Properties`.

```
bus SpaceWire_Bus
properties
  ARAO::Transport_API => SpaceWire;

  -- Custom properties
  ASSERT_Properties::Access_Bandwidth => 100 Kbps;
  -- ...

  -- Standard properties
  Allowed_Message_Size => 1 Kb .. 10 Kb;
  -- ...
end SpaceWire_Bus;
```

Listing 10. MPC's SpaceWire bus

5.4.2 *Memories.* Memories are modeled using the `memory` AADL component. This component is used to model all kinds of memories (RAM, ROM, Disks, magnetic tapes, etc...). The AADL properties within the component model refine the component description and introduce more precisely its characteristics.

The listing 11 shows the AADL model for the random access memory (RAM) used in each node of our example. This model introduces a new feature of the AADL language, the *required access* to a bus. Since a memory has to be plugged in a bus in order for it to be accessed by the processor (section 5.4.3), the AADL model of a memory has to specify that this memory requires an access to a bus of a specific kind. In our example, the memory has to be plugged in a bus called `MemBus`. As for the bus component models, the memory model has several properties to introduce more precisely the characteristics of the memory.

5.4.3 *Processors.* To model the processor and the operating system (or the real-time kernel), we use the `processor` AADL component. Since processors are more

```
memory RAM
features
  Mem_Bus : requires bus access MemBus;
properties
  ASSERT_Properties::Memory_Size => 100 Kb;
  --  ...
end RAM;
```

Listing 11. AADL memory component

complicated to model than buses or memories, we will use AADL extension features to model them incrementally.

The first model in listing 12 shows a generic model for a LEON 2 processor[1]. This is a very simple model that shows that the processor needs to access to a memory bus (in order to read and write into the RAM) and specifies the processor frequency by mean of the custom AADL property `ASSERT_Properties::Processor_Speed`.

The second model in the same listing describes a LEON 2 having an access to a SpaceWire bus. The model `LEON_2_OneSpaceWire` enriches the `LEON_2` model by adding the corresponding bus access feature.

The last model in the listing enriches the definition of the `LEON_2_OneSpaceWire` component by extending it and adding some properties relative to the GNAT for LEON runtime [de la Puente et al. 2000]. The model does not show all the added properties because the list is too long. Of course this component inherits all the aspects of its parents (bus accesses and frequency).

```
-- MODEL 1: Generic LEON 2 processor

processor LEON_2
features
  Mem_Bus : requires bus access MemBus;
properties
  ASSERT_Properties::processor_speed => 150 MHz;
end LEON_2;

-- MODEL 2: LEON 2 processor with a SpaceWire bus

processor LEON_2_OneSpaceWire extends LEON_2
features
  SPW : requires bus access SpaceWire;
end LEON_2_OneSpaceWire;

-- MODEL 3: LEON 2 processor with a SpaceWire bus with GNAT 1.3 runtime

processor LEON_2_OneSpaceWire_GNAT_1_3 extends LEON_2_OneSpaceWire
properties
  ASSERT_Properties::min_priority  => 0;
  ASSERT_Properties::max_priority  => 240;

  Global_Scheduler_Policy => EDF;

  -- More properties...
end LEON_2_OneSpaceWire_GNAT_1_3;
```

Listing 12. AADL processor component

[1]LEON is the micro-processor used by the European Space Agency in the next-generation satellites

6. METRICS

In this section, we detail how analyses and code generation can be combined to build one ready-to-run system. Then we assess our process on the case study of figure 4.

6.1 Schedulability analysis & Model checking

The case study we retained is well-formed following the Ravenscar profile. This model focuses on a restricted set of the Ada concurrency model to ensure that all Ravenscar-compliant programs can be fully analyzed following the Rate Monotonic Analyses, and its extensions for distribution. Therefore, one can assess the schedulability of the model.

In addition, a Petri model can be deduced from the architecture, allowing one to check for deadlocks or do some time logic analysis. In the context of this case study, such analysis is very trivial.

Intermediate models (Cheddar or Petri Net models) are of similar complexity than the initial AADL model: they only reflect the number of communication channels and its topology. This implies these analyses are not impeded by the model transformation we propose, but by the initial AADL model and the analysis capability of the tools.

In this respect, Cheddar can handle only limited time range for simulations. CPN-AMI can handle large state spaces but is limited by typical combinatorial explosion problems. However, this was not a limit on this model. Besides, let us note related work on the PolyORB project shows it is possible to tackle combinatorial explosion issues by using optimized model checker [Hugues et al. 2004].

The three steps of our prototyping methodology are automated. So, if Cheddar detects schedulability problems or CPN-AMI outlines an erroneous execution on the associated Petri Net, the user just has to modify the AADL model and then re-run the analysis. In many cases, Cheddar proposes to the user the corrections that should be performed. CPN-AMI also provides relevant data (e.g an execution path) when detecting problems such as a deadlock. Such information is relevant to perform a new refinement step on the model.

6.2 Code generation

A prototype of PolyORB-HI, running on both ERC32 and LEON2 targets has been built. These processors are used by the European Space Agency for its next generation of embedded systems (satellites, vehicles, etc...). Thanks to Ada portability, the same code can also be tested on native targets, or on other boards, such as PowerPC-based. This makes the prototyping of embedded system easier since we can test them on native platform before embedding them on their corresponding hardware. In this section, we study the footprint of the code generated on LEON2 targets.

Table I summarizes the code size for the node SC_2 of our case study. It that combines periodic and sporadic threads, data components and a SpaceWire interface to receive inbound messages. We display both the actual lines of code (SLOCs) and the size of the binary objects. The used compiler is the GNAT for LEON 1.3 Ada compiler. All tests were done in local, using the `tsim` simulator for LEON, emulating a $50Mhz$ processor. The `SpaceWire` interface is simulated in `tsim` as an I/O module bound to the LEON processor.

The code generation strategy we retained maps AADL constructs onto Ada equivalent ones so that there exists traceability between the AADL model and the corresponding Ada source code: e.g. between AADL threads and Ada tasks, AADL data component and Ada records or protected objects. Such strategy reduces the need for a large API, and eases code review after generation.

The total size of the executable, combining real-time kernel, middleware and the application, is $576kB$, using the GNAT for LEON 1.3 compiler. It fits the requirements from minimal embedded systems, and is clearly under the typical memory range for API-based middleware such as nORB or microORB, which are above $1MB$ for a complete system, including full OS support.

Given the development process we retained, most code is automatically generated from the AADL model. The code in the middleware handles simple and low-level actions: messages, protocol, transport. Generated code adds tasking constructs required to execute the application and enables interaction between entities: transport handler, application activities, advanced marshallers, naming tables, etc...

The code generation strategy we chose accounts for a large part of the distribution aspects of the application: it includes the elaboration of tasks and protected objects, the allocation of dedicated memory area, stubs and skeletons, marshallers and naming table. Finally, the runtime accounts for another large part of the size of the application.

Component	SLOCs	.o size (bytes)
Application	89	8852
Generated code	961	66804
Middleware	1068	32957
Ada Run-Time + drivers	N/A	$\approx 541Kb$

Table I. Footprint for the SC_2 node

6.3 Assessment of the process

From the AADL model, we are capable of generating both, information that the model is sound and the corresponding executable, ready to run on LEON2 boards.

We demonstrate how to exploit one AADL model and user-provided code for some processing functions. AADL serves both as a documentation of the system (requirements expression, functional and non functional properties, topology of the system can be expressed in one model) and as a template to validate it and generate its implementation: it preserves system design.

Therefore, we have an immediate benefit from an engineering point of view: the developer can focus on its system architecture. The complete tool suite ensures it is correct, and handles the configuration of all code-level entities. This suppresses many manual code writing, a tedious and error-prone process underlined by well-known software failures in the space domain like the Ariane V maiden flight. It also tremendously reduce the development cycle and allow one to go faster from the prototyping phase to the design of the final system.

7. CONCLUSION

In this paper, we proposed a rapid prototyping process to develop Distributed Real-Time and Embedded (DRE) systems around the *Architecture Analysis and Design Language* (AADL). We first motivated the need for such process, focusing on two hard constraints: constraints to build and qualify such systems in a timely manner, difficulty to master implementation and dimensioning issues.

We selected the AADL to implement an efficient rapid prototyping process for DRE systems, focusing on its design-by-refinement approach, and its extensibility through user-defined properties. We illustrated this approach by presenting our current tool chain built around the Ocarina tool suite and the PolyORB High-Integrity middleware. We assessed the process on complete examples to evaluate each step.

We showed that an integrated set of tools enables the user to focus directly on the overall system, and leverage his architecture to directly generate code for High-Integrity systems without any user intervention. Besides, analysis tools have been proposed to check model consistency and viability prior to generation. This increases confidence in the model while being fully automated.

A system designed through our prototyping process is very close to the final product. The user functional components have to be completed but the overall design and the integration are mostly set up. A large part of the system has already been validated. The user may have to check the schedulability of the final system accounting the new WCET of new functional components. We claim our approach significantly reduces the time needed to specify, prototype and produce a distributed real-time embedded system.

Future work will consider a better integration of third party tools (stack bound computation, precise WCET measurement tools, simulation...) with the Ocarina tool suite, and the extension of the configuration of the middleware to a wider range of platforms and requirements, addressing fault tolerance, or other schedulers, but also other programming languages such as C or embedded Java variants.

ACKNOWLEDGMENTS

The authors thank F. SINGHOFF from the Cheddar project as well as M. BLANCQUART, D. THOMAS, P. DISSAUX, J. S. CRUZ, M. PEROTTIN and the other members of the IST-ASSERT project for their provided case studies and their feedback on earlier version of this work.

REFERENCES

BORDIN, M. AND VARDANEGA, T. 2005. Automated Model-Based Generation of Ravenscar-Compliant Source Code. In *ECRTS '05: Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*. IEEE Computer Society, Washington, DC, USA, 59–67.

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. 1996. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, New York.

DAVIS, R. I. AND BURNS, A. 2005. Hierarchical Fixed Priority Pre-Emptive Scheduling. In *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*. IEEE Computer Society, Washington, DC, USA, 389–398.

DE LA PUENTE, J. A., RUIZ, J. F., AND ZAMORANO, J. 2000. An open ravenscar real-time kernel for gnat. In *Ada-Europe'00: Proceedings of the 5th Ada-Europe International Conference on Reliable Software Technologies*. Springer-Verlag, London, UK, 5–15.

Dobbing, B., Burns, A., and Vardanega, T. 2003. Guide for the use of the of the Ravenscar Profile in High Integrity Systems. Tech. rep.

ECSS. 2003. *Space Engineering. SpaceWire - Links, nodes, routers and networks.*

Ellidiss-Software. 2007. STOOD. http://www.ellidiss.com/stood.shtml.

ENST. 2006. Ocarina: An AADL model processing suite. http://ocarina.enst.fr.

Feiler, P. H., Gluch, D. P., and Hudak, J. J. 2006. The Architecture Analysis & Design Language (AADL): An Introduction. Tech. rep. CMU/SEI-2006-TN-011.

Gasperoni, F. 2006. Safety, security, and object-oriented programming. *SIGBED Rev. 3,* 4, 15–26.

Gorappa, S., Colmenares, J. A., Jafarpour, H., and Klefstad, R. 2005. Tool-based Configuration of Real-time CORBA Middleware for Embedded Systems. In *International Symposium on Object-oriented Real-time distributed Computing (ISORC'05).* Seattle, USA.

Hamez, A., Hillah, L., Kordon, F., Linard, A., Paviot-Adet, E., Renault, X., and Thierry-Mieg, Y. 2006. New features in CPN-AMI 3 : focusing on the analysis of complex distributed systems. In *6^{th} International Conference on Application of Concurrency to System Design (ACSD'06).* IEEE Computer Society, Turku, Finland, 273–275.

Hugues, J., Thierry-Mieg, Y., Kordon, F., Pautet, L., Baarir, S., and Vergnaud, T. 2004. On the Formal Verification of Middleware Behavioral Properties. In *Proceedings of the 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'04).* Vol. ENTCS 133. Elsevier, Linz, Austria, 139 – 157.

Hugues, J., Zalila, B., and Pautet, L. 2006. Middleware and Tool suite for High Integrity Systems. In *Proceedings of RTSS-WiP'06.* IEEE, Rio de Janeiro, Brazil.

ISO/IEC 8652:2007(E) Ed. 3. 2006. Annotated Ada 2005 Language Reference Manual. Tech. rep.

Kordon, F. and Luqi. 2002. An introduction to Rapid System Prototyping. *IEEE Transaction on Software Engineering 28,* 9 (September), 817–821.

Leveson, N. 1997. Software engineering: Stretching the limits of complexity. *Communications of the ACM 40(2)*, 129–131.

MoVe-Team. 2007. The CPN-AMI Home page, url: `http://www.lip6.fr/cpn-ami`.

OMG. 2001. Model Driven Architecture (MDA), Document number ormsc/2001-07-01. Tech. rep., OMG.

OMG. 2004. *Data Distribution Service for Real-time Systems Specification version 1.0.* OMG. OMG Technical Document.

SAE. 2004. Architecture Analysis & Design Language (AS5506). available at `http://www.sae.org`.

SAE. 2005. *Language Compliance and Application Program Interface.* SAE. The AADL Specification Document Annex D.

SAE. 2006. Open Source AADL Tool Environment. Tech. rep., SAE.

Schmidt, D., Stal, M., Rohnert, H., and Buschmann, F. 2000. *Pattern-Oriented Software Architecture – Volume 2 – Patterns for Concurrent and Networked Objects.* Wiley & Sons, New York, NY, USA.

Schmidt, D. C., Levine, D. L., and Mungee, S. 1998. The design of the TAO real-time object request broker. *Computer Communications 21,* 4 (Apr.), 294–324.

Singhoff, F., Legrand, J., Tchamnda, L. N., and Marcé, L. 2004. Cheddar : a Flexible Real Time Scheduling Framework. *ACM Ada Letters journal, 24(4):1-8, ACM Press. Also published in the proceedings of the ACM SIGADA International Conference, Atlanta, 15-18 November, 2004*.

Tindell, K. 1993. Holistic schedulability analysis for distributed hard real-time systems. Tech. rep., University of York.

Vergnaud, T., Hugues, J., Pautet, L., and Kordon, F. 2004. PolyORB: a schizophrenic middleware to build versatile reliable distributed applications. *LNCS 3063*, 106 – 119.