OCARINA: An Environment for AADL Models Analysis and Automatic Code Generation for High Integrity Applications

Gilles LASNIER, Bechir ZALILA, Laurent PAUTET, and Jérome HUGUES

Institut TELECOM – TELECOM ParisTech – LTCI
46, rue Barrault, F-75634 Paris CEDEX 13, France
{gilles.lasnier,bechir.zalila,laurent.pautet,jerome.hugues}@telecom-paristech.fr

Abstract. Developing safety-critical distributed applications is a difficult challenge. A failure may cause important damages as loss of human life or mission's failure. Such distributed applications must be designed and built with rigor. Reducing the tedious and error-prone development steps is required; we claim that automatic code generation is a natural solution. In order to ease the process of verification and certification, the user can use modeling languages to describe application critical aspects. In this paper we introduce the use of AADL as a modeling language for Distributed Real-time Embedded (DRE) systems. Then we present our tool-suite OCARINA which allows automatic code generation from AADL models. Finally, we present a comparison between OCARINA and traditional approaches.

1 Introduction

Distributed Real-time Embedded (DRE) systems are used in a variety of safety-critical domains such as avionics systems, medical devices, control systems, etc. Traditionally, their development process is based on manual work for requirement analysis, software design, verification and certification. Defining an automatic development framework for DRE is a challenge, potential building blocks are AADL and the Ravenscar Profile.

The Architecture Analysis and Design Language (AADL) [SAE04] is an architecture description language that defines system constructs such as threads, processes or processors to model real-time, safety-critical embedded systems. AADL provides a potential backbone to model DRE systems, to analyse their schedulability, safety and security properties and automatically produce the executable code.

The Ravenscar Profile [WG05] defines a subset of Ada to guarantee schedulability and safety properties by restricting the features of the language. A similar approach can be applied to restrict the features of a modeling language. Such an approach is promising as long as automatic code generation can be performed.

In this paper, we present our work on a development process for building DRE systems from architecture descriptions. The tool-suite OCARINA [TEL08] allows syntactic and semantic analysis from AADL models. An AADL subset helps us applying the Ravenscar Profile restrictions in order to perform real-time analysis through both OCARINA and CHEDDAR[SLNM04]. Then we combine both AADL, OCARINA and

POLYORB-HI [ZPH08], a light distribution middleware designed and targeted to high-integrity systems, to produce C and Ada code compliant with the Ravenscar Profile.

The contents of this paper is structured as follows: Section 2 presents related works aimed at building DRE systems from their models. Section 3 points out the rationale for our approach. Section 4 gives an overview of the AADL. Section 5 introduces our proposed development process and describes the architecture of our tool-suite OCARINA. In Section 6, we present a case study we carried using and validating our approach. Section 7 concludes this presentation and gives some future works.

2 Related Work

In this section, we present a couple of projects that aim at specifying, analyzing and producing DRE applications from their models.

2.1 COSMIC

COSMIC: *Component Syntheses using Model Integrated Computing* [LTGS03] is a tool suite to build DRE applications based on the OMG CCM [OMG06a] specification. It uses the MIC (*Model Integrated Computing*) [SK97] paradigm and conforms to the OMG D&C [OMG06b] specification.

Applications in COSMIC are modeled using a set of description languages: PICML to describe the components of the application, their interfaces and their QoS parameters, CADML to describe how components are deployed and configured and OCML to describe the middleware configuration options. COSMIC allows the expression of constraints on the application components and supports the integration of analysis modules. The applications are built on top of the component middleware CIAO. In [SSK+07], the benefits of this middleware are presented. CIAO offers capabilities to separate the development of the application from its deployment and configuration.

COSMIC uses RACE (Resource Allocation and Engine Control), a framework placed at the topmost layer of the middleware that controls and refines dynamically the use of resources by the application. The fact that COSMIC is based on several different languages to specify an application means that each representation must be consolidated after any change on the application model. This increases the development time.

The topmost-layer used to refine dynamically the properties of components is problematic for critical systems where all resources must be allocated statically. In addition, the use of UML and XML introduces a scalability issues when it comes to model applications with a large number of components [SBG08]. All these drawbacks restrict the use of COSMIC for DRE systems where correctness by construction is required.

2.2 AUTOSAR

AUTOSAR (AUTomotive Open System ARchitecture) is the result of a partnership between several automotive constructors. It allows developing DRE applications for the automotive domain while optimizing their maintainability and development cost [SnG07]. It allows the standardization of the exchange format of specifications, the abstraction

of the micro-controller to avoid redeveloping a system when only the platform changes and the standardization of the interfaces of different components.

A component in AUTOSAR is an atomic unit that cannot be distributed. It contains operations and data that are required and provided by the component, the needs of the component (access to sensors, bus...) and the resources the component uses (memory, CPU...). The main components categories are: software components and sensors. Several instances of the same component may exist on the system.

In [SVN07], the authors provide a set of limitations identified during their use of AUTOSAR. These problems are present in many modeling languages: (1) lack of separation between the functional part and the architecture, (2) lack of support for task definition and resources, (3) lack of support for analyzability and for retro-annotations through an iterative process of design. Moreover, to incorporate the user code, AUTOSAR generates skeletons that must be completed by the user. However, the possibility of automatic edition of the skeletons modified by the user is left to the implementations [aG06]. Therefore, after each update, the new skeletons must be filled or at least edited, (4) lack of preservation of semantics after code generation. It should be noted here that the code is generated in the C language on which most compilers do not provide semantic and consistency analysis.

All this makes questionable the use of AUTOSAR in the context of increasingly complex HI systems. Indeed, the static analyzability of these systems is crucial and the absence, of capabilities such as the definition of properties of tasks (ie. period, priority, etc.) makes difficult this kind of analysis.

3 Motivations

In Section 2, we presented several approaches for designing DRE systems with their limitations. In this section, we expose the rationale for a new approach.

We claim that it is critical to take the architecture into account very soon in the design process. Knowing the hardware specifications the designer can deduce that some redundancy requirements cannot be achieved. Changing the architecture and in particular the deployment of some software pieces has a great impact on the system schedulability.

We also want to follow guidelines that ensure system analysis. With this respect, system schedulabibility is a major issue for DRE system. Therefore, we want the user to follow a concurrency model that eases the applicability of real-time scheduling theory in particular the Rate Monotonic Analysis.

Finally, we want to be able to perform a seamless integration of user components into the global framework. To do so, we claim that massive code generation is a reachable goal and preserves the user from manual and error-prone coding. It also helps him in following the design guidelines. For instance, we may prevent him from messing around with mutexes and therefore violating the concurrency model. Code generation also gives the opportunity to dramatically optimize code.

In Section 3.1 we present AADL as description language to model DRE systems. Section 3.2 points out the motivations for the use of Ada and the Ravenscar Profile. Finally, Section 3.3 gives the motivations for the use of massive code generation.

3.1 AADL for a model-based design

Model-based approaches have radically improved the system design. Reuse of existing components, refining and extension techniques, rapid system prototyping, separation of functional and non functional aspects, allow designers to explore efficiently design alternatives and early analyze system properties during the development cycle.

Section 2 describes a couple of projects that aim at specifying, analyzing and producing DRE applications from their models. We have also seen that work needed several description languages, specific frameworks and middlewares which increased the development time and were not convenient for DRE applications.

We propose the use of AADL as **unique** description language to model DRE applications. AADL was designed and targeted to DRE systems and has all features mentioned above concerning model-based development. In addition, it allows the designer to specify all aspects required by DRE systems and integrates both functional and non functional aspects of applications using an efficient property mechanism. AADL also allows us to configure and deploy the components in the context of the target DRE system.

3.2 Ravenscar Profile for concurrency model

For our approach we chose to design our own tools suite OCARINA in Ada. Firstly, the Ada programming language owned historically a legacy about the theory of compilation (tree manipulation, etc.) and efficient compiler (as GNAT). Secondly, contrary to heavy Java frameworks, Ada is more maintainable.

As a first step, we also chose to generate Ada code. It provides a rich set of tasking constructs and is well-suited to real-time system development. In addition, the Ravenscar Profile [WG05] defines a subset of Ada that provides schedulability and safety guarantees by restricting the features of the language. These restrictions make Ada even more amenable to the development of DRE systems.

3.3 Code generation for an optimized and seamless integration

In our approach we focus on massive code generation because it allows us to reduce system development (time and cost) and break down the complexity of DRE system. Code generation provides facilities to integrate and deploy components automatically. It is also an efficient mechanism to analyze and verify consistency between initial models and the final code. In addition, it automatically enforces (coding) guidelines to preserve system analysis.

The development of our own environment allowed us to have a fine control over the production framework and take advantages from it (see Section 5). For analysis concerns, the code generation avoids object-oriented constructs, indirections and dynamic allocations. We chose to design our own specific execution platform POLYORB-HI as these complex constructs. It may also be introduced by a COTS middleware (executing the generated code). In our approach, the code generation aims at producing the glue code for wrapping application components but also a large part of the execution platform ; the other part being independant from the application. These facilities also help us optimizing the system integration and the code generated.

4 An Overview of AADL

AADL [SAE08] is an architecture description language standardized by SAE (Society of Automotive Engineers) that was first developed in the field of avionics. AADL is used to model safety-critical and real-time embedded systems. It uses a component-centric model and defines the system architecture as a set of interconnected components. Modeling components consists of describing their interfaces, their implementations and their properties. The standard defines the textual, graphical, and XMI representation of the language to facilitate model interchange between tools. Furthermore, AADL is an extensible language: external languages can be used to define annexes. Besides, the standard proposes annexes to specify the detailed behavior of applications, data representation and error modelling as well as code generation directives.

4.1 Components

Components in AADL represent elements providing and requiring services according to their specification. AADL gives support only for the specification of the non-functional aspects of the component. Behavioral or functional aspects of components must be given separately. For example, users can specify the source code of software components in a programming language such as Ada by means of the use of properties.

The specification of a component consists in the declaration of the component *type* and if necessary one or several component *implementations*. Component *types* define the interface of the component through the declaration of *features*. Component *implementations* define the internal aspect of a component, the subcomponents it contains and the connections between *features* and those subcomponents. AADL gives the possibility to specify properties for all components.

AADL allows the modelling of software components (*process*, *thread*, *thread group*, *data*, *subprogram* and *subprogram group*) , execution platform components (*processor*, *virtual processor*, *memory*, *bus*, *virtual bus* and *device*) and an hybrid component (*system*). AADL entities allow us the description of the software and the hardware architecture of the system:

1. Software components
   - *Data* represents a data type (if it is in the declarative part) or a data instance (if it is a subcomponent of another component). Data components may contain other data, subprograms and subprogram-groups.
   - *Subprogram* is an abstraction of the procedure from imperative languages.
   - *Thread* represents the basic unit of execution and schedulability in AADL. Threads may contain data and subprogram subcomponents, as well as *subprogram call sequences*.
   - *Process* represents a virtual address space. Process components may contain threads, data and thread-groups.
2. Execution platform components
   - *Processor* represents a microprocessor together with a scheduler.
   - *Virtual Processor* represents logical resource as virtual machine or scheduler able to schedule and execute threads.

- *Memory* represents a storage space (RAM/ROM or disk).
- *Bus* represents a hardware communication channel that links execution platform components.
- *Virtual Bus* represents communication protocol.
- *Device* represents a hardware with a known interface that can interact with the environment and/or the system under consideration (e.g. sensors/actuators).

3. System component
   - *System* is a hybrid component with no semantics that is used to group hierarchically software and hardware components.

In addition, AADL defines a generic component (*the abstract component*) which allow us the description of high-level abstract components of the architecture at early stages of the system modeling.

```
thread Slow
properties
  Dispatch_Protocol => Periodic;
  Period            => 1 sec;
end Slow;

thread Fast extends Slow
properties
  -- The value of Dispatch_Protocol
  -- is inherited from thread Slow.
  Period            => 100 ms;
end Fast;

process Slow_Fast end Slow_Fast;

process implementation Slow_Fast.Impl
subcomponents
  S : thread Slow;
  F : thread Fast;
end Slow_Fast.Impl;
```

Listing 1.1. AADL description

```
process Slower_Faster
end Slower_Faster;

process implementation Slower_Faster.Impl
subcomponents
  S : thread Slow
    {Period => 2 sec;};
    -- New value of Period
  F : thread Fast
    {Period => 10 ms;};
    -- New value of Period
end Slower_Faster.Impl;

system Root end Root;

system implementation Root.Impl
subcomponents
  P1 : process Slow_Fast.Impl;
  P2 : process Slower_Faster.Impl;
end Root.Impl;
```

Listing 1.2. AADL description (2)

Listings 1.1 and 1.2 show the use of AADL to model a system. In this example, we describe two threads *Slow*, *Fast* and their properties. *Root* and *Root.impl* represent respectively the interface and the implementation of the system. This system is composed of two processes that have two threads.

4.2 Component interfaces

Component *types* in AADL may declare *features* to define communication points exposed by the component and give it the possibility to communicate with its environment or other components. We describe here the different *features*:

Ports are data or control transfer-points. A `data port` is used to transfer only data. An `event port` is used to transfer only control; they send events corresponding to signals with no associated data type. `Event data port`s transfer control with accompanying data; they correspond to signals with an associated data type. Furthermore, ports may have Ada-like directional qualifiers for the flow (`in`, `out` or `in out`).

Parameters can be declared as features of subprograms.

Data accesses represent access to a data subcomponent by an external component. This feature is used to model shared/protected data among remote components. Data accesses may be provided or required.

Subprograms accesses declared as features of data components represent access procedures (similar to methods of a class). When they are declared as features of threads they represent RPCs.

4.3 Properties

AADL allows the system designer to attach properties to any component: *type*, *implementation*, subcomponent, connection or port [SAE08]. Properties are used to define functional aspects of the system, for instance `Dispatch_Protocol` and `Period` for threads; `Data_Type` for data; as well as `Compute_Entrypoint` for specifying subprograms attached to ports.

AADL has predefined standard property sets which express real-time, communication protocol, program, memory, time, thread or deployment aspects [SAE08]. The AADL property mechanism also allows the description of new functionnal aspect or overriding properties applied to components.

4.4 Modes

AADL provides *modes* that allow dynamic system reconfiguration. Modes represent the active states of the software and the execution platform components of a system. An initial mode may be specified for a component and *mode transitions* give the possibility to change the current mode. *Mode transitions* may be triggered by the reception of an event or data in ports of the component features.

5 The tool suite OCARINA

OCARINA [TEL08] is a tool suite designed in Ada by the S3 team at TELECOM ParisTech. It aims at providing model manipulation, syntactic/semantic analysis, scheduling analysis (CHEDDAR), verification and code generation from AADL models.

OCARINA is designed as a traditional compiler with two parts: a frontend and a backend. A central library has been developed to provide builder and finder routines that manipulate entities used in the compiler. Figure 1 represents the architecture of the tools suite OCARINA.

5.1 Frontends

The frontend of OCARINA is conceived around a modular architecture and give the possibility to use different modeling language. Currently, we have developed four modules corresponding to the AADL language. Those modules have been developed in accordance with the AADL 1.0 [SAE04] and the AADLv2 [SAE08] standards.

The lexical analyzer (*lexer*) recognizes a sequence of lexical elements as reserved keywords, identifier, separator and operator, which are specified by the AADL 1.0 and the AADLv2 standards.

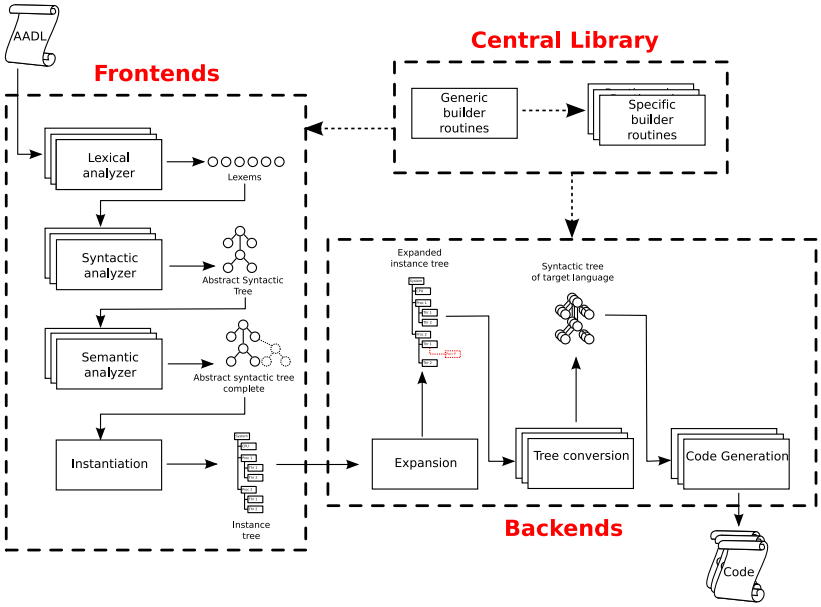

Fig. 1. Architecture of the compiler OCARINA

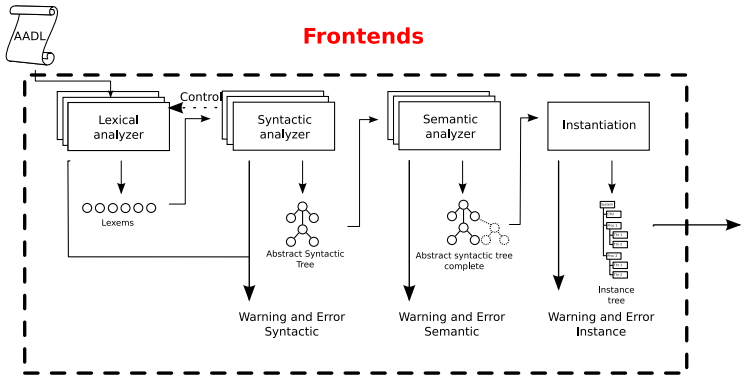

Fig. 2. OCARINA frontends architecture

The syntactic analyzer (*parser*) uses the lexer and calls the corresponding functions relative to the analysis of the sequence of lexical elements in accordance with the AADL grammar. Warning and errors relatives to the AADL syntax are raised and pointed our to the user. The result of this analysis leads to the construction of an Abstract Syntax Tree (*AST*) which is a representation of the AADL model.

The semantic analyzer scans the AST and checks the semantics of the AADL model. First, it proceeds to a resolution phase which adds different information to the AST and make easier it use (example, the resolution of property constant with the correct value). Secondly, the standard AADL defines a set of semantic rules which allow to check the semantic of the AADL model. The result of this analyze leads to an AST that conforms to the AADL semantics.

Figure 2 represents the architecture of the frontends and shows when syntactic, semantic and instance warnings and errors are detected and pointed out to the user.

The AADL standard [SAE08] specifies a set of rules which describe how to instantiate an AADL model. In AADL, an instance model defines an arborescent hierarchic representation of the system, its components and its subcomponents. The **instantiation** step computes the definitive values of properties referring to an instance component and detect some incoherence of the system (for example, a process must contain at least one thread). The result of this step leads to the construction of AADL instance tree corresponding to 'legal' AADL models that are used later for code generation purposes.

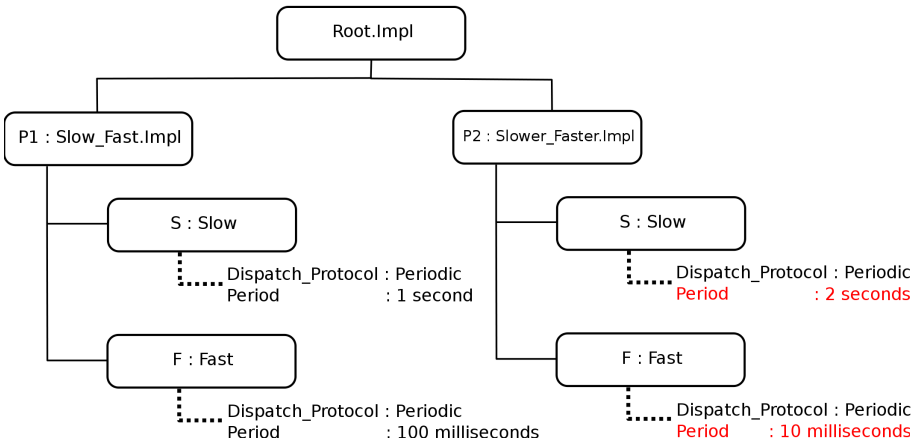

Fig. 3. Example of an AADL instance tree

Figure 3 represents the AADL instance tree relative to the AADL model described in the listing 1.1.

5.2 Backends

OCARINA has a modular architecture that allows users the use of several backends for different target languages. Three backends have been developed to support Ada, C and AADL code generation. We describe here the different steps needed for code generation.

The first stage of the backend, called *expansion*, simplifies complex structures of the instance model and decorates the *AST* with information associated with our code generation mapping rules. For example, one of our mapping-rules adds an *event port* in hybrid thread for facilitate their communication protocol implementation. The result of this step leads to an expanded AADL instance tree which allows us further analysis and make code generation easier.

The second step transforms an expanded AADL instance tree in the syntax tree of a target language called **intermediate tree**. Then, we scan this syntax tree in order to produce the code. This architecture is different than traditional compilers which produce code directly from the model. It allows us to make the maintainability of the compiler easier and give us more flexibility for code generation.

Finally, for code generation purposes we use both the intermediate tree and the PolyORB-HI middleware. PolyORB-HI is a minimal middleware targetted to DRE systems (designed by TELECOM ParisTech). It provides only those services required by the DRE system. It is consituted of two parts: the first part corresponds to services lowly customizable and used for each applications; the second part corresponds to services highly customizable automatically generated by the backend. PolyORB-HI supports construction for AADL entities describe in the AADL model and produce code for those entities.

6 Case study : Ravenscar example

Several case studies have been carried out to prove the correctness of our tool suite and validate our development process. Some of have been presented in previous publications [ZPH08,HZPK08] or elaborated in the context of the ASSERT European project. This section presents a new case study based on the classical example provided by the Ravenscar Profile guide [BDV04]. It illustrates the expression capacities of the profiles as well as some aspects covered by our production process.

6.1 Presentation

The Ravenscar Profile guide example illustrates a workload management system. The system has four processes: a light sporadic process **External Event Server** receives external interrupts and records them in a specific buffer; a light periodic process **Regular Producer** performs the regular workload (under specific conditions, the process delegates the additional workload and the treatment of external interrupts to other light processes); a light sporadic process **On Call Producer** performs the additional workload; a light sporadic process **Activation Log Reader** is the interrupt handler.

Three shared and protected data were defined: a **Request Buffer** filled by **Regular Producer** and used by **On Call Producer**; an **Event Queue** for external interrupts

used by **External Event Server**; an **Activation Log Reader** for interrupts treatment, filled by **External Event Server** and used by **Activation Log Reader**. Finally, the systems contains one passive entity, **Production Workload** which performs the *Small Whetstone* operation which treats the workload asked by a light process.

Figure 4 illustrates the workload management system in an AADL graphical representation.

6.2 Adaptation for AADL and distributed application

The Ravenscar Profile guide example describes a local application. In our context, we model it using the AADL language and we also adapt it for DRE system.

Active components of the system are modelled by thread components in AADL. The Ravenscar Profile example gives us all the information needed for specifying the properties of the different threads (ie. priority, period, etc). OCARINA's compiler and the POLYORB-HI middleware support the two kinds of thread components specified in the example. The behavior of threads is modelled by the properties `Compute_Entrypoint` associated directly with the thread or with the event port of the thread component *type*.

We could gain advantages of the use of AADL concerning shared and protected data model. Indeed, in AADL it was not necessary to specify explicitly the components **Request Buffer**, **Event Queue** and **Activation Log**. We could use the *features* of thread component *type* for it.

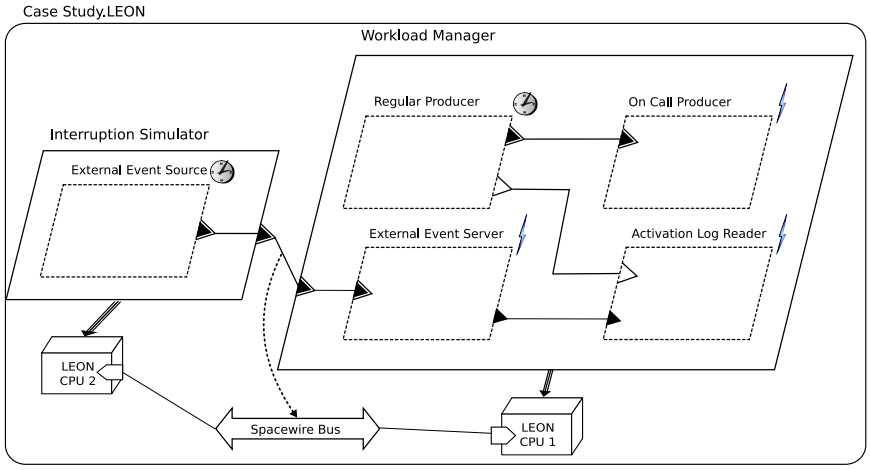

Fig. 4. AADL Adaptation of Ravenscar Profile guide example

Figure 4 gives an AADL graphical representation of the Ravenscar Profile guide example adapted for DRE systems. This example has been adapted for distributed application. To simulate external interrupts we added process **Interruption_Simulator**

which generates and sends random messages to the **Workload_Manager** process. The two processes ran on a LEON2 processor and communicate through a SPACEWIRE bus.

6.3 Results and metrics

To analyze the AADL model, we used the tools suite OCARINA and the CHEDDAR tool[1]. The analysis with OCARINA guaranteed the coherence of the model and the compliance to the restrictions of Ravenscar Profile. CHEDDAR uses the OCARINA libraries to process the model, then performs a response time analysis. Both analyses have been carried out successfully.

After that, we proceeded to the automatic code generation and the deployment and configuration of the application with OCARINA and POLYORB-HI. We ran with success the code generated using the TSIM [Aer08] a LEON2 simulator processor. To simulate the distributed application, we used an I/O module developed by SCISYS [2]. This module simulates the SPACEWIRE bus communication that connects the application nodes.

	Workload_Manager	**Interruption_Simulator**
User code	4	1
Generated code	482	123
Minimal middleware	245	143
Compiler runtime	475	467
Total object files	**1215**	**734**
Executable footprint	**2217**	**1745**

Table 1. Binary footprint (in Kbytes) for LEON2

The table 1 gives the memory footprint for the nodes of the application. A large part of this footprint is due to the threads stack sizes. The compiler we use, GNAT *for* LEON is a development version that requires 100 *KB* as a minimal task stack size. The ORK [dlPRZ00] kernel and the SPACEWIRE driver contribute also in increasing the footprint of the executables.

	Workload Manager	**Interruption Simulator**
POLYORB	2026	1994
POLYORB-HI	579	527

Table 2. Memory footprint (in Kbytes) GNU/LINUX

	AADL example	Ravenscar example
SLOCs	3352	488
Executable footprint	1535	1441

Table 3. Line number and binary footprint (in Kbytes) for the **Workload_Manager**

[1] OCARINA invokes CHEDDAR

[2] SCISYS is one of the industrial partners of the ASSERT project. `http://www.scisys.co.uk`

We wanted to compare the results obtained using PolyORB-HI with those we got using PolyORB [Qui03]. We recompiled our case study to a native platform. We then used an existing backend in Ocarina to produce Ada code for PolyORB. The table 2 gives the executable footprints. We see that using PolyORB-HI reduces considerably the memory footprint of the executables compared to PolyORB (26%-28%). This can be explained by the dynamic nature of PolyORB deployment and configuration.

We finally compared our case study with the original Ravenscar example (all the code is provided in the Ravenscar profile guide). To do this, we removed the **Interruption_Simulator** to get a local application. Table 3 gives the number of lines and the executable footprint for the **Workload_Manager**. We see that the Ravenscar example is 6% smaller than the example built automatically using Ocarina and PolyORB-HI. This difference of size is acceptable since almost all the code of the application is produced automatically thanks to our production process, and the model is also analysable; whereas the Ravenscar example was fully handwritten.

7 Conclusion and Future Work

In this paper we introduced Ocarina[3], a tool developped in Ada that gives support to a new approach for building DRE systems from AADL descriptions.

We proposed the use of the Architecture Analysis and Design Language (AADL) to model and validate efficiently DRE systems. AADL also allows to include functional and non-functional aspects of DRE systems in the description architecture. Thanks to automatic code generation, the application components are easily integrated with the execution platform.

We also designed PolyORB-HI, a highly-configurable middleware for the High-Integrity domain. Some middleware components are automatically generated by Ocarina from the system AADL models when the other components are selected from a minimal PolyORB-HI library. This approach allows us to produce deterministic executable with a very small footprint.

We showed how Ocarina has been designed for allowing external tool integration. Besides, some tools had been integrated to our tools suite as Cheddar to validate specific properties concerning real-time systems. We defined an AADL subset to ensure that systems are Ravenscar compliant by construction in order to analyse their schedulability.

A case study has been presented to illustrate the approach and validate our tool-suite Ocarina. This concrete example showed that our approach can be used to generate code for DRE systems compliant with the Ravenscar Profile. Finally, we assessed our work on a complete example to evaluate each step of our approach.

Future directions for our work include updating the AADLv2 support in Ocarina. We also intend to extend Ocarina and PolyORB-HI to address security and safety issues in DRE partitioned systems. These systems aim at being compliant with Arinc 653 and Mils standards.

[3] available at `http://ocarina.enst.fr`

References

[Aer08] Aeroflex Gaisler AB. TSIM ERC32/LEON Simulator. `http://www.gaisler.com`, 2008.

[aG06] AUTOSAR Gbr. Technical Overview. Technical report, 2006.

[BDV04] A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the Ada Ravenscar Profile in High Integrity Systems. *Ada Lett.*, XXIV(2):1–74, 2004.

[dlPRZ00] J. A. de la Puente, J. F. Ruiz, and J. Zamorano. An Open Ravenscar Real-Time Kernel for GNAT. In *Ada-Europe '00: Proceedings of the 5th Ada-Europe International Conference on Reliable Software Technologies*, pages 5–15, London, UK, 2000. Springer-Verlag.

[HZPK08] J. Hugues, B. Zalila, L. Pautet, and F. Kordon. From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite. *ACM Transactions in Embedded Computing Systems (TECS)*, 7(4):1–25, July 2008.

[LTGS03] T. Lu, E. Turkay, A. Gokhale, and D. C. Schmidt. CoSMIC: An MDA Tool suite for Application Deployment and Configuration,. In *Proceedings of the OOPSLA 2003 Workshop on Generative Techniques in the Context of Model Driven Architecture*, Anaheim, CA, October 2003.

[OMG06a] OMG. *CORBA Component Model Specification Version 4.0*. OMG, April 2006. OMG Technical Document formal/06-04-01.

[OMG06b] OMG. *Deployment and Configuration of Component-based Distributed Applications Specification, Version 4.0*. OMG, April 2006. OMG Technical Document formal/06-04-02.

[Qui03] T. Quinot. *Conception et Réalisation d'un intergiciel schizophrène pour la mise en oeuvre de systèmes répartis interopérables*. PhD thesis, École Nationale Supérieure des Télécommunications, March 2003.

[SAE04] SAE. *Architecture Analysis & Design Language (AS5506)*, September 2004.

[SAE08] SAE. *Architecture Analysis & Design Language v2.0 (AS5506)*, September 2008.

[SBG08] P. Sriplakich, X. Blanc, and M.-.P. Gervals. Collaborative Software Engineering on large-scale models: requirements and experience in ModelBus. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 674–681. ACM, 2008.

[SK97] Janos Sztipanovits and Gabor Karsai. Model-Integrated Computing. *Computer*, 30(4):110–111, 1997.

[SLNM04] F. Singhoff, J. Legrand, L. Nana, and L. Marc. Cheddar : a flexible real time scheduling framework. In *ACM SIGAda Ada Letters*. ACM Press, December 2004.

[SnG07] D. Schreiner and K. M. Goschka. A Component Model for the AUTOSAR Virtual Function Bus. In *COMPSAC'07: Proceedings of the 31st Annual International Computer Software and Applications Conference - Vol. 2- (COMPSAC 2007)*, pages 635–641, Washington, DC, USA, 2007. IEEE Computer Society.

[SSK+07] N. Shankaran, C. Schmidt, X. D. Koutsoukos, Y. Chen, and C. Lu. Design and performance evaluation of configurable component middleware for end-to-end adaptation of distributed real-time embedded systems. In *ISORC '07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 291–298. IEEE Computer Society, 2007.

[SVN07] A. Sangiovanni-Vincentelli and M. Di Natale. Embedded System Design for Automotive Applications. *Computer*, 40(10):42–51, 2007.

[TEL08] TELECOM ParisTech. Ocarina : An AADL model processing suite. `http://aadl.enst.fr`, 2008.

[WG05] Ada Working Group. *Ada Reference Manual*. ISO/IEC, 2005. `http://www.adaic.com/standards/05rm/RM-Final.pdf`.

[ZPH08] B. Zalila, L. Pautet, and J. Hugues. Towards Automatic Middleware Generation. In *11th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC'08)*, pages 221–228, Orlando, Florida, USA, May 2008.