

Code Generation Strategies from AADL Architectural Descriptions Targeting the High Integrity Domain

Julien Delange, Jérôme Hugues, Laurent Pautet, Bechir Zalila

▶ **To cite this version:**

Julien Delange, Jérôme Hugues, Laurent Pautet, Bechir Zalila. Code Generation Strategies from AADL Architectural Descriptions Targeting the High Integrity Domain. Embedded Real Time Software and Systems (ERTS2008), Jan 2008, toulouse, France. insu-02270097

HAL Id: insu-02270097

https://insu.hal.science/insu-02270097

Submitted on 23 Aug 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Code Generation Strategies from AADL Architectural Descriptions Targeting the High Integrity Domain

Julien Delange, Jérôme Hugues, Laurent Pautet, Bechir Zalila

GET-Télécom Paris – LTCI-UMR 5141 CNRS
46, rue Barrault, F-75634 Paris cedex 13, France
{name}@enst.fr

Abstract: Distributed Real-time Embedded (DRE) systems are increasingly used in critical domains such as avionics, vehicle and industrial control as well as in medical systems. They must be designed carefully and have to provide safety properties because a failure could mean loss of life. For these reasons, it is recommended to automatically generate a significant part of the code from the models describing the critical aspects. In our approach, we automatically generate two kinds of code from architectural models. The first one plugs the user functional code in the middleware, the second one provides a significant part of the middleware functions. Both rely on a hand coded written middleware that provides the minimal facilities to plug the generated code and to resolve portability issues. In this paper, we present our code generator and the middleware designed to generate High Integrity (HI) systems. We demonstrate via several use-cases how we succeeded in meeting the requirements of DRE systems (small memory footprint, no dead-code, etc...).

Keywords: AADL, code generation, high integrity

1. Introduction

Developing High Integrity (HI) systems is very difficult compared to classical ones as they are used in safety-critical domains such as space or avionics. These systems have to conform to strong requirements (small memory footprint, limited computation capacity) and standards.

Manually writing HI applications appears to be very tedious for the developer. It is also very difficult to comply with standards and requirements. It is preferable to automatically generate a large part of the code from analyzable models. Thus, the code generation process avoids all programming errors introduced by manual coding and ensures that it follows the properties included in the model. In addition, using model transformation, this process produces a verifiable model (e.g.: Petri Net) from the application model. It is simpler and safer than producing this model from source code.

AADL, which stands for “Architecture Analysis and Design Language” [1] is an architecture description language that allows the modelling of distributed, real-time embedded applications. AADL was first introduced to model the hardware and software architectures in the avionics domain, but was later extended to the general DRE (distributed real-time embedded) domain. This pedigree results in a language that is more amenable to static analysis and verification than other more general purpose modelling languages.

Our past work led us to the design of a Ada Ravenscar code generator [2] from AADL models. The Ravenscar Profile [3] is a subset of rules and coding guidelines for Ada that ensure certain properties including static schedulability analyzability and the absence of deadlock. Generation of Ravenscar compliant Ada code ensures that the generated code provides these properties. However, Ada needs a specific runtime and this may reduce the range of potential executive platforms. On the contrary, the C language can be used on more platforms than Ada. However, the C language does not provide any static analysis method at compile time such as Ravenscar, so we have to create it by hand.

The process of producing a working application from an AADL model is divided into two main parts: a code generator and a minimal middleware. The code generator automatically produces C code from AADL models. The generated code relies on a minimal middleware that provides all the services it needs by the generated code. The middleware is considered as minimal because it only contains the functions requested by the application. Moreover, a large part of the middleware is automatically generated from the model, whereas another part is statically written. In our work, we focus on embedded systems, so we designed the middleware to be compliant with the requirements of such high integrity and embedded systems (small memory footprint, etc...).

In the remainder of this paper, we present the AADL language, the Ravenscar profile and give an overview of the code generation process. We explain the internals of the code generation process. Then, we present the minimal middleware we created to use the generated code: PolyORB-HI-C [4]. We present its design and

architecture. We give three use cases and the platforms used to perform our tests. Finally, we conclude and give some ideas for future work.

2. Context

In this section, we give an overview of AADL [1, 5], the language used to model critical systems. Then, we present the Ravenscar profile for Ada and what it implies for the code we generate. Finally, we describe the code generation process.

2.1 Overview of AADL

AADL has been designed by the SAE (Society of Automotive Engineers). This language has been created to model the architecture of systems. It allows the description of both software and hardware parts of a system and focuses on the definition of clear block interfaces and separate implementations from these interfaces. Models can be expressed using a graphical or textual interface.

The standard property set of the language and the possibility to add new ones offers a way to model low-level aspects of the system and defines the specific behavior of each component. The fine-tuning of the model and the code-reuse ability constitute great advantages for choosing this modelling language.

AADL defines several components, each of them models a hardware or software entity. Hardware components are

- *Processor*: models the processor of the system and the underlying executive. In other words, this component represents the operating system and the architecture of the system. For a standard computer, we declare a processor component and add properties to tell that it represents an x86 platform with a Native operating system such as Linux or Windows.
- *Memory*: models any memory in the system, from RAM to ROM, or hard drives.
- *Bus*: models different kinds of networks, wires, etc...
- *Device*: models any other device in the system (sensor, motion detector, etc...).

Software components are :

- *Process*: models a single address space. The notion expressed in this component is similar the UNIX process. As in operating systems, a process component can contain *thread* components.
- *Thread*: models lightweight process. This component can be understood as a POSIX thread. It can contain subprogram calls.
- *Subprogram*: models a flow of instructions. This component typically represents a function or a procedure in a language. An execution unit (such as thread) can declare call section to invoke subprograms.
- *Data*: models a type manipulated by an entity of the system. For example, it can be used to describe the type of the arguments of a subprogram or any variable declared in an execution unit (such as thread).

A special component (*system*) aggregates all the components of the system.

Components can contain subcomponents, in order to describe the hierarchy of the system. For example, a process component can contain several thread components. AADL contains a set of properties, which can be extended by the user. Properties are applied to components to model their characteristics and behaviors (such as the period of a thread, the size of the stack for a thread, etc...). AADL components can declare features, which are their interface to communicate with other components (for example, the parameters of a function are considered as features for an AADL subprogram). In addition, the *connections* clause of each component declares its relation with other entities (for example, a connection between the features of a thread and the ones of a subprogram means that the variables of the thread are used as arguments in a function-call). The *features* and *connections* section are used to model the communications between all nodes of a distributed system.

2.2 The Ravenscar Profile

The runtime of the Ada programming language provides extensive tasking capabilities. However, some of them are not suitable for safety-critical real time systems. Rendez-vous, multiple entries in protected objects and *select* statements render execution time analysis very complex. The Ravenscar Profile [3] is an effort to define a subset of Ada for use in High Integrity systems; the major restrictions therein are detailed below.

- Static tasking stipulates that the task-set is statically defined, there is no task terminations or *abort* statements and no dynamic priorities
- Static synchronization model enforces non-hierarchical protected objects, a static set of protected objects, no select statements and no entries on tasks
- Deterministic memory usage states that there be no implicit heap allocation
- Deterministic execution stipulates that there is a maximum of one entry per protected object, the queue length of said entry be one, there be no *requeue* statements, no

asynchronous control of tasks and delays be absolute

- Runtime configuration states that task dispatching be *FIFO_Within_Priorities*, the priority ceiling protocol [7] be used for protected object access and that there be no suspendible code in a protected object procedure, function or entry

Only periodic and sporadic tasks are allowed. Periodic tasks are dispatched at regular time intervals (their period). Sporadic tasks are dispatched as a result of events, but with a specified minimal inter-arrival time between events. The behavior in case of violation of this rule is implementation-dependant.

When using the Ravenscar profile in Ada source code, the compiler checks each statement. With the C language, we cannot check the behavior of the code at compile time. On the contrary, the code generation patterns create code that provides all the requirements of the Ravencar profile. Therefore, the C code generation process checks the behavior only one time (code generation), whereas the Ada code generation checks it twice (code generation and compilation).

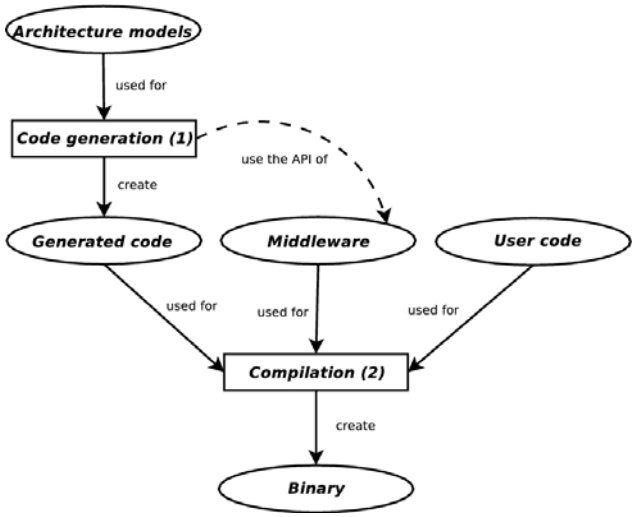


Figure 1 : Steps to create a distributed application from AADL model

2.3 Overview of the code generation process

The steps for building a distributed application from AADL models are illustrated in figure 1. The developer provides his architectural model and his own behavior code. Then, the code generator uses the model to automatically generate code (step 1 on the figure). The code generator knows the API of the minimal middleware so it uses it in the generated code. Consequently, it uses services provided by the middleware and statically allocates all the resources needed by the system. Then, the generated code is compiled with the minimal middleware and the code provided by the user (step 2). In this step, the middleware uses the resources allocated in the generated code to provide its services.

2.4 Advantages of code generation process

Writing a distributed application, even using a middleware, remains a tedious task for the developer, who always encounters some problems to design, deploy or configure it. Models provide the deployment and configuration information, so the generated application doesn't need to be configured by the developer. The deployment of the generated applications is also automatically performed. While automatic code generation avoids all programming errors in the generated code, our approach automatically generate Ravenscar-like C code.

A traditional application often embeds some dead-code and has a memory overhead, due to many reasons (use of externals libraries, etc...). If these properties are not significant for most applications, they are important for embedded platforms that have memory constraints. In the code generation process, the architectural model is used to allocate statically all the resources at the initialization (threads, buffers, etc...). We allocate only the resources needed by the system. Such a method ensures a minimal memory footprint. We use the information from the model to create all data structures used by our algorithms. In the middleware it is easy to bound the execution time of our algorithms. This helps us to compute the worst execution time.

2.5 Related work

Research on AADL is very intense, and many applications have been created to manipulate AADL models. The Topcased [15] project offers a way to model systems with AADL using a textual as well as graphical representation.

In addition, AADL models can be used to check properties of the system. The Cheddar tool [6] can prove the system's schedulability. To make this possible, one adds some properties on the components to describe their timing constraints (deadline, scheduling algorithm, mutexes ceilings, etc...). Then, Cheddar analyzes the schedulability and tells its feasibility.

Code generation process from models is a topic already developed. Many code generation techniques have been developed. Most of them use UML models [8]. However, these methods are used to help the user to write his code and do not fit with the requirements of HI systems. Other methods focus on the HI domain [9], but always use UML

models and may introduce a memory overhead.

On the contrary, the originality of the present approach resides in the possibility of automatically generating a Ravenscar-compliant application written in C with a reduced memory overhead.

3. Automatic code generation

This section details the code generation process. It presents the code generation patterns used to transform AADL models into C code. We explain what part of code of the middleware we generate and how we can generate Ravenscar-compliant code. The internal structure of the code generator is also described.

3.1 Internals of our code generator

For each component of the model, we apply patterns that translate the AADL declaration into C code. For example, a thread component is mapped into a thread in the system. Consequently, when we find this kind of component, we generate a function call to create the thread and define a function for its execution. If the mapping of the thread component is easy to understand, the mapping of other components is more complex. Moreover, we have to be compliant with the Ravenscar profile, so our patterns must create code that follows the Ravenscar restrictions. All the code generation patterns defined in our code generator are explained in this section.

We designed a tool for manipulating AADL models: Ocarina [10]. It is used to manipulate models and have many features: semantic analysis of AADL models, code generation for the PolyORB middleware [11], graphic model representation from a textual one, etc... It contains the entire infrastructure to manipulate AADL models and describe other languages (as target languages for C code generation). It was natural for us to extend this tool and implement our code generator as a new feature. We use the existing code to read and manipulate AADL models and implement our code-generation patterns and code writing method. In the following paragraphs, we explain the internal work of Ocarina and the steps of the code generation.

Our code generation process uses at least two syntaxic trees: one for the modelling language (in our case, AADL), and another for the target language (Ada, C, etc...). The structure of each tree is described and we automatically generate functions to create and manipulate the tree. This architecture eases the creation of new syntaxic trees and the extension of our tool to new languages.

The code generation process is divided into four steps:

1. The parser analyzes the model. This first pass on the model creates a syntaxic tree, which contains all the declarations of the model. When parsing the model, we check that it is well formed.
2. We create an instance-tree, based on the syntaxic tree. This tree has one root system component with all its subcomponents. It precisely describes the model with its component hierarchy. Creating this tree is very important, because the model is well organised and unused components are not included in the instance-tree.
3. We traverse the instance tree and apply patterns on components to create the syntaxic-tree of the target language. For the C code generation, the result of this step is a tree with all generated C declarations.
4. The generated files are printed and we generate `Makefiles` to compile the code with the minimal middleware. At this step, the code shall be fully functional and the user can directly compile it.

3.2 Code generation patterns

The code generation patterns are very important, they automatically generate the code in the target language from the components of the model. For each component in the model, a pattern is applied and creates code in the syntaxic tree of the target language. The code-generation patterns are summarized in table 1.

For a thread component we generate statements that create the thread at initialization time and define a macro that describes the number of threads in the local node. These declarations ensure that the threads are created at initialization time, and that we allocate a fixed amount of threads. No other resources are allocated.

For each subprogram component, we generate a function that calls user-defined routines or generated ones.

For data component, we map it to a predefined type (such as integer, float, etc) according to its properties. However, data components could have subprograms as features (as the data component in listing 1). These subprograms get access to the data to read or write it. For these reasons, such data must be protected from concurrent access. We map such component to a structure with all its subcomponents and we add a unique identifier to this structure (the `pos_impl` structure in listing 2). Then, this identifier is used to lock or unlock the data before we read or write it (see the implementation of the functions in listing 2). The name of the structure is derived from the name of the component.

```
data POS_Internal_Type
properties
  ARAO::Data_Type => Integer;
end POS_Internal_Type;

data POS
features
  Update : subprogram Update;
  Read   : subprogram Read;
end POS;

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

Listing 1: AADL model of a type that uses subprograms as features

AADL Component	Code-generation pattern
System	Create a directory which contains each process
Process	Create a directory which contains the source code.
Thread	• Instantiate the thread at initialization time • Create function executed by the thread after the initialization • Define a macro to tell how many threads are created on the local node • Create identifiers for its ports • Create an entity identifier to communicate with other threads in the whole distributed system
Subprogram	Create a function that call other mapped subprograms or user-defined functions.
Data	• Map the data to existing predefined types • Declare a protected identifier if the data must be protected from concurent access
Processor	Put the right execution platform (architecture and operating system) in the Makefile in order to choose the right cross-compilation tools.

Table 1: Code generation patterns

To generate a fully working distributed system, we had to consider the features of each thread and analyse communications between the entities. For each thread that may communicate, we create a unique identifier in order to identify threads on each node. Information about the location of each node (IP address, port) is added in the code so the system automatically initializes connections between nodes.

The processor component models the architecture and the operating system. So, we use it to solve the portability problem. In our case, we defined a property set for all supported platforms and operating systems. We add the property value of the processor component in the `Makefile` used to compile each node. This value is used to choose the cross-compilation tools.

```
typedef int pos_internal_type;
typedef struct
{
  __po_hi_protected_t protected_id;
  pos_internal_type field;
} pos_impl;

void pos_impl_update (pos_impl* v)
{
  lock(v->protected_id);
  update(&(v->field));
  unlock(v->protected_id);
}

void pos_impl_read  (pos_impl* v)
{
  lock(v->protected_id);
  read(&(v->field));
  unlock(v->protected_id);
}
```

Listing 2: Generated code from a data component with subprograms as features

4. Middleware designed for the generated code

The previous section presents the code generation internals and its patterns to translate AADL models into C code. The generated code contains the resources used by the application and defines the functions used by each thread. However, the code needs to rely on services to use the operating system primitives. A service provides functionalities to execute the program: communications with other nodes, facilities to create and schedule tasks, etc... This is the purpose of the middleware: it manages the resources allocated in the generated code and calls the operating system's services. In this section, we describe the services provided by a traditional middleware and explain why we need to define a new middleware that is more compliant with high integrity system. Then, we present the minimal middleware we design, detail its services, its architecture and its compliance with commonly used operating systems.

4.1 Canonical middleware architecture

Our past research led us to design PolyORB [16], a schizophrenic middleware. It separates concerns between distribution model, API, communication protocols, and their implementation by refining the definition and role of personalities. The schizophrenic architecture consists of three layers: *application-level* personalities (adaptation layer between application components and middleware through a dedicated API) and *protocol-level* personalities (handles the mapping of personality-neutral requests onto messages exchanged using a chosen communication network and protocol) built around a *neutral core*. The *neutral core* layer enables the selection of any combination of application and/or protocol personalities. Several personalities can be collocated and cooperate in a given middleware instance, leading to its *schizophrenic* nature. The neutral core defines generic services used by application and protocol-level personalities. We also focus on it because it contains fundamental services for a middleware. The *µBroker* is the core component that provides support for interaction between the other canonical services :

1. *Addressing service* looks up objects on servers. This service is used when a client ask the server for an access to an object
2. *Binding factory service* establishes communications with the server using one communication channel
3. *Representation service* gives a common representation for the data (e.g: CDR). This solves the problem of endianness.
4. *Protocol service* provides several protocols to communicate with other nodes.
5. *Transport service* provides several transport layer to send and receive the data
6. *Activation service* ensures an entity can execute the request.
7. *Execution service* gives the resources to execute the request.
8. *Typing service* manages the typing system in the application (sophisticated when it comes to the CORBA *any* mechanism for instance)
9. *Interaction service* manages the liaisons between connected entities in the application,

However, such middleware deals with object-oriented methods and is not compliant with HI systems. Their configuration is performed at run-time (creating and managing POA), their resources are dynamically allocated (buffers, threads, etc...) and it is difficult to evaluate the worst case execution time of some functionalities. Such a design introduces temporal indeterminism and it remains hard to analyze the behavior of such system.

The middleware we wrote (PolyORB-HI-C [4]) matches a small memory footprint and ensures predictable properties. The use of AADL models eases the configuration as it provides the user requirements and the deployment information. Consequently, the configuration of the middleware is made at compile time and resources are statically allocated. It ensures that no dynamic configuration or allocation is performed at runtime such as in traditional object-oriented middleware. Moreover, AADL models precisely describe the communications between the node of the distributed system. Using this information, we can automatically establish communications between the nodes that may communicate at initialization time. It prevents dynamic creation of connections to other nodes. According to this static configuration, some services included in PolyORB become useless. For example, the addressing service and the binding factory are no longer used because we automatically add code to locate nodes and establish communications.

A large part of the middleware is automatically generated whereas another is statically written. The generated part of the middleware contains functionalities that are specific to the model. The statically written part defines generic functionalities used by all applications Moreover, the model provides the architecture of the system so we can select the services needed by the generated code (e.g.: if the model defines a single process, we don't include services that enable communication). Such method reduces the introduction of memory overhead and dead-code.

Many research projects designed middleware [12] for real-time and high-integrity embedded software and use other approaches than the one described in this paper. While projects tend to provides more and more features in their middleware, it is important to keep in mind that we just try to provide few services, and include only those needed by the generated code.

In the next subsection, we present our middleware PolyORB-HI-C. We describe its architecture and services according to their equivalence with the PolyORB ones. In addition, we explain what part of the middleware is automatically generated and what part we wrote by hand. We also explain how it fits with the requirements of the Ravenscar profile. Finally, we describe how we design it to be compliant with many systems.

4.2 Architecture of PolyORB-HI-C

The services provided by the middleware are listed below:

1. *Execution service* creates threads and provides scheduling functionalities.
2. *Representation service* configures the types available on the target platform and provides

a common representation of the types in the distributed system
3. *Transport service* provides functionalities to send and receive data from other nodes
4. *Protocol service* provides protocols to exchange data between the nodes of the distributed system

The *execution service* is used to create and manage threads on a local node. This service is quite different from the one in PolyORB : threads do not execute requests on an object, they execute an infinite loop with the code provided by the user. The *execution service* handles periodic and sporadic tasks as defined in the Ravenscar profile. It provides all the functions to be compliant with the strong timing requirements of HI systems. The task dispatching policy by the middleware is *FIFO_Within_Priorities*. This policy helps us analyze schedulability.

The *representation service* configures the middleware to choose the types on the target platform (set data representation according to the types available on the target such as 8, 16 or 32 bits *integer*, *float*, etc…) and provides functionalities to marshal/unmarshal data into a message. This service is the same as the one defined in PolyORB, we just add some functionalities such as type configuration and data protection. This service is partly generated (the code to marshal requests is automatically generated), whereas another part is statically written.

Transport service enables communication channels between the nodes of the distributed system. This service is equivalent to the one in PolyORB. Several transport layers can be implemented in the middleware so the user can choose it. At this time, only BSD sockets are implemented. Most of the code of the transport service is statically written.

Protocol service provides functionalities to structure the messages exchanged in the distributed system. The service is equivalent to the one of PolyORB. As for transport service, several protocols can be implemented. At this time, we implement two protocols: IIOP (Internet Inter-ORB Protocol, the protocol of CORBA [13]) and another protocol that directly serialize the data. The resources used by this service are automatically generated whereas the code remains statically written.

Figure 2 shows the architecture of the middleware and the organisation of the source files of each service. The *execution service* uses the representation service to express a representation of the time. The representation service is used by the transport and protocol services to get the types available on the platform. Finally, the transport service uses the execution service to create a thread that listen for incoming messages from other nodes.

All the resources of the system are statically allocated at compile time, as is the deployment information. If the resources cannot be allocated at compile time (creation of threads, mutexes, etc...), we do it at initialization time. In each program, a function initializes all resources before the system starts. This ensures that we don't make any dynamic allocation or configuration at runtime.

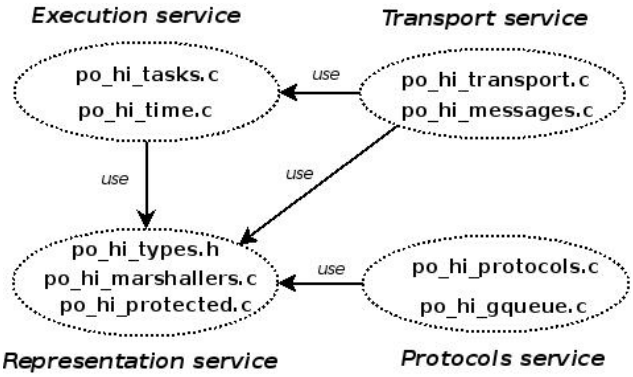


Figure 2: Architecture of the minimal middleware

4.3 Portability

All portability aspects are included in the middleware as well. This problem is solved in two steps. At configuration time, the middleware detects the endianness of the architecture and available types. It chooses its functions and the type management method according to the information it has on the platform. The second aspect is about compilation: choosing the right cross-compilation tools, using specific command-line options when compiling, etc... To solve this problem, we use the information about the execution platform provided in the generated `Makefile` for each node. The middleware then has a configuration of each compilation step for all execution platforms and adapts its behavior according to all these parameters. Moreover, we pay attention to the functions used in the middleware to get portable code. Most of the code uses POSIX standard, in order to be easily ported to other systems that support it. Also, we add an abstraction layer to the POSIX one in order to remain Ravenscar-compliant.

The code of the middleware must respect all constraints of the HI domain. The execution time of the algorithms used can be easily bounded and all resources are statically allocated at compile time or at initialization time. The size of each resource is defined with the information provided by the generated code. This ensures that we have a minimal footprint (no more memory is allocated), and no non-deterministic service is used at runtime (dynamic memory allocation, etc...).

5. Tests and benchmarks

This section gives an overview of the platforms we used and their specifications. Moreover, we present some benchmarks we carried out. They detail the memory footprint on each platform.

5.1 Platforms

We tested our examples on the following platforms:

1. Nokia N770: ARM architecture, 64 MB of RAM with a Linux-based operating system.
2. Nintendo DS: ARM architecture, 4 MB of RAM (2Mb for the operating system, 2MB for userland applications) with a port of Linux: DSLinux [14].
3. Native system: x86 architecture with Linux operating system.

The Nokia N770 platform proves that our approach could be used for personal systems such as smartphone, PDA and other personal embedded systems. Such devices have a limited capacity of computation, but do not have hard memory constraints. On the contrary, the Nintendo DS has strong memory constraints. Only 2 MB of RAM are available for the execution of third-party programs, so it shows that our generated applications have a small memory footprint. The last architecture, called native may not seem to be as interesting as the others. On the contrary, this platform shows that the generated code works on a standard personal computer and helps us for debugging purposes.

5.2 Tests

During the design of the middleware, we designed some tests that use the code generator and the minimal middleware. We focus on three of them: RMA, Sunseeker and Flight Management [17]. All these use-cases are available on our website dedicated to AADL [10].

The RMA test creates two periodic tasks (T1 and T2) in one process with the following properties: 2 x $Period_{(T1)}$ = $Period_{(T2)}$. Each task prints a message to show when it is being run. This test shows the correctness of the scheduling.

The Sunseeker test was written by the Software Engineering Institute (SEI) of the Carnegie Mellon University (CMU) in order to test some AADL features and code generation patterns. It models a missile guidance example. It defines two periodic tasks. Each task receives data from the other, computes a value and sends it to the other task. This test introduces data exchange between distributed nodes so that the code related to distributed application is added in each program of the system.

The last test (Flight Management [17]) creates five tasks: three periodic tasks and two sporadic tasks. The communication graph between tasks is quite complex. All kinds of ports are used (event port, data port and event data port). This test shows the correct mapping of each port type to C and that communications with several ports works well.

5.3 Metrics

From these tests, we carried out some metrics on memory footprints. Table 2 shows the memory footprint of each test on each platform. All the binaries are compiled with GCC and are statically linked. In all cases, the size of the binary is less than one megabyte. The Nintendo DS uses a port of Linux (DSLinux [14]), but uses a C-library designed for embedded system: µClibC. The binary created with this embedded-profiled C-library is smaller than the ones that use traditional C-library (such as the binaries compiled for the native platform).

	RMA	Sunseeker	Flight management
Nokia N770	589 KB	645 KB	638 KB
Nintendo DS	144 KB	204 KB	209 KB
Native	471 KB	527 KB	532 KB

Table 2: Memory footprint for each test

6. Conclusions and Future Work

Using architectural models to automatically generate code is a good way to create code for high-integrity systems. Such a method avoids all errors traditionally introduced by manually produced code, it can also create code that provides safety properties. In our approach, the code generator implements the Ravenscar profile for C.

The minimal middleware limits the memory overhead and is compiled only with the needed functions. A large part of the middleware (functions and resources) is automatically generated whereas a static part uses the resources and calls the services specific to the underling operating systems. The architecture of the middleware eases its port to other platforms.

If our code generator produces code that provides all the requirements of the Ravenscar profile, we can't ensure that the user-code does not break our safety rules. Besides, if the Ada compiler checks the compliance of the Ravenscar profile on the whole system, the C compiler doesn't perform any check. Consequently, even if the generated code is well formed, the user can break all the safety rules introduced in the system with his functions.

We can extend the current work in many ways. First, we can create other language generators in order to automatically produce a distributed system with several nodes that use

different languages. At this time, our code generator, Ocarina, only supports Ada and C languages. Another way consists of adding some security and safety information to the model. Such information is used to automatically generate a safe and secure distributed system (data-flow analysis between nodes, encryption in data transfer, etc...). Research on architecture models will probably remain very intense in the next years, due to the needs to check and validate the system before its implementation.

Acknowledgment: This work is partly funded by the RNTL Flex-eWare project.

References

[1] SAE Aerospace. Architecture Analysis & Design Language (AADL), September 2004

[2] B. Zalila, I. Hamid, J. Hugues, and L. Pautet. Generating Distributed High Integrity Applications from their Architectural Description. In AdaEurope'07, Geneva, Switzerland, jun 2007.

[3] Brian Dobbing, Tullio Vardenega and Alan Burns. Guide for the use of the Ada Ravenscar Profile in high integrity systems. January 2003.

[4] Julien Delange. PolyORB-HI-C User Guide, 2007.

[5] Peter H. Feiler, David P. Gluch and John J. Hudak. The Architecture Analysis & Design Language (AADL) : An Introduction. Technical report, February 2006.

[6] F. Singhoff, J. Legrand, L. Nana and L. Marcé. Cheddar : a flexible real time scheduling framework. ACM, 11 2004.

[7] Lui Sha, Ragunathan Rajkumar and John P. Lehoczky. In IEEE Transactions on Computers, pages 1175–1185, Washington DC, USA, 1990.

[8] Pornsiri Muenchaisri and Mathupayas Thongmak. Design of Rules for Transforming UML Sequence Diagrams into Java code. 2002.

[9] Matteo Bordin and Tullio Vardaneda. Automated Model-Based Generation of Ravenscar-Compliant Source Code. 2005.

[10] T. Vergnaud, B. Zalila and J. Hugues. Ocarina documentation, see http://aadl.enst.fr.

[11] Jérôme Hugues, Fabrice Kordon, Laurent Pautet, and Thomas Vergnaud. A Factory To Design and Build Tailorable and Verifiable Middleware. In Proceedings of the Monterey Workshop 2005 on Networked Systems: realization of reliable systems on top of unreliable networked platforms, volume LNCS 4322, pages 123–144, University of California. Irvine, CA, USA, Feb 2007. Springer Verlag.

[12] D. Schmidt, D. Levine, and C. Cleeland. Architectures and patterns for developing high-performance, real-time orb endsystems, 1999.

[13] Object Management Group. Common object request broker architecture : Core specification. Technical report, OMG, 2004.

[14] Dslinux. http://www.dslinux.org.

[15] Toolkit in Open Source for Critical Applications & Systems (TOPCASED). http://www.topcased.org.

[16] Jérôme Hugues, Laurent Pautet and Fabrice Kordon. A framework for DRE middleware, an application to DDS. In Proceedings of the 9th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC'06), April 2006

[17] Irfan Hamid, Elie Najm, Bechir Zalila and Jérôme Hugues. A Generative Approach to Building a Framework for Hard Real-Time Applications. In 31st IEEE Software Engineering Workshop (SEW 2007), 2007