Leveraging Ada 2012 and SPARK 2014 For Assessing Generated Code From AADL Models

Jérôme Hugues
Université de Toulouse, ISAE
10 avenue E. Belin
31055 Toulouse, France
jerome.hugues@isae.fr

Christophe Garion
Université de Toulouse, ISAE
10 avenue E. Belin
31055 Toulouse, France
christophe.garion@isae.fr

ABSTRACT

Modeling of Distributed Real-time Embedded systems using Architecture Description Language provides the foundations for various levels of analysis: scheduling, reliability, consistency, etc.; but also allows for automatic code generation. A challenge is to demonstrate that generated code matches quality required for safety-critical systems. In the scope of the AADL, the Ocarina toolchain proposes code generation towards the Ada Ravenscar profile with restrictions for High-Integrity. It has been extensively used in the space domain as part of the TASTE project within the European Space Agency.

In this paper, we illustrate how the combined use of Ada 2012 and SPARK 2014 significantly increases code quality and exhibits absence of run-time errors at both run-time and generated code levels.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Correctness proofs, Formal methods, Programming by contract; I.6.5 [**Model Development**]: Modeling methodologies

General Terms

Design, Languages, Verification

Keywords

AADL; Ada 2012; SPARK 2014; Ocarina

1. INTRODUCTION

The Model-Based System Engineering (MBSE) paradigm allows for a high-level description of a system, its analysis and eventually its automatic generation. Significant efforts have been undertaken to design modeling frameworks that support this view with a sufficient level of expression and fidelity towards the system being built. Besides, MBSE already demonstrated capability for scheduling and reliability assessment in combination with analysis tools . This provides foundations for a wide support for the engineering of safety-critical systems.

The Architecture Analysis and Design Language, standardized by SAE International [10], provides such an integrated framework for the modeling of safety-critical systems. Combined with the Ocarina code generation toolchain, it allows for the automatic code generation towards Ada runtimes using a restricted middleware: PolyORB-HI/Ada. Both generated code and runtime were initially written in Ada95, with significant effort done to ensure compatibility with both the Ravenscar profile and High-Integrity restrictions such as absence of dynamic features (object-orientation, memory allocation, streams, etc.). This greatly reduces the benefit of code generation compared to traditional hand coding strategies in the context of safety-critical systems: the only strategy to ensure absence of run-time errors was through careful code review and testing of the generated code on the target.

The advent of Ada 2012 programming-by-contract approach, and the availability of SPARK 2014 language and toolset to assess a) that contracts are true, and b) the absence of runtime errors promise to increase confidence in source code, and stronger link with formalized specifications of programs. In the following, we illustrate how the use of these two technologies allows us to streamline code generation and analysis effort by providing easier access to code quality assessment.

In section 2 we present the AADL and some existing tools to support the model-based engineering of embedded systems. In section 3, we outline the basic principles of the PolyORB-HI/Ada runtime, inherited from the schizophrenic middleware architecture deployed in the PolyORB middleware,and the code generation strategies used in the Ocarina toolchain. In section 4, we discuss modernization of the initial code base to take advantage of Ada 2012's new constructs, and how we had to overcome some existing limitations in the current SPARK2014. In section 5 we show how SPARK 2014 allows us to demonstrate absence of run-time errors of the middleware components. Finally, we provide some elements for future works.

2. A BRIEF OVERVIEW OF AADLV2

The "Architecture Analysis and Design Language" (AADL) [10] is a textual and graphical language for model-based engineering of embedded real-time systems. AADL is used to design and analyze software and hardware architectures of embedded real-time systems.

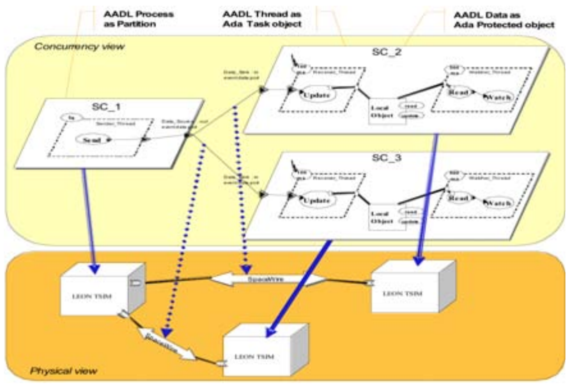

Figure 1: IST-ASSERT demonstrator

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. From the separate description of these blocks, one can build an assembly of blocks that represents the full system. To take into account the multiple ways to connect components, the AADL defines different connection patterns: subcomponent, connection, binding. For example, figure 1 provides the graphical descrition of a case study used in the scope of the ASSERT project.

An AADL model can incorporate non-architectural elements: non-functional properties (execution time, priority, scheduler, ...) and behavioral or fault descriptions. It is hence possible to use AADL as a backbone to describe all the aspects of a system. Let us review these elements in the following.

An AADL description is made of *components*. Each component category describes well-identified elements of the actual architecture, using the same vocabulary of system or software engineering. The AADL standard defines software components (`data`, `thread`, `thread group`, `subprogram`, `process`) and execution platform components (`memory`, `bus`, `processor`, `device`, `virtual processor`, `virtual bus`) and hybrid components (`system`) or imprecise (`abstract`).

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 certain kind of components (*processor*, *process*, *bus*, *device*, *abstract* and *memory*), thus providing the root of the architecture tree. The architecture in itself is the instantiation of this system: the *root 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*. A given component type correspond zero or several implementations. Each of them describes the internal structure of the components: subcomponents, connections between those subcomponents, and refine non-functional properties.

The AADL defines the notion of *properties*. They model non-functional properties that can be attached to model elements (components, connections, features, instances, etc.). Properties are typed attributes that specify constraints or characteristics that apply to the elements of the architecture such as clock frequency of a processor, execution time of a thread, bandwidth of a bus. Some standard properties are defined, e.g. for timing aspects; but it is possible to define new properties for different analysis (e.g. to define particular security policies). Besides, the language is defined by a companion standard document that defines legality rules for component assemblies, its static and execution semantics.

AADL's initial requirement document mentions analysis as the key objective. AADL is backed with a large set of analysis tools[1], covering many different domains: scheduling analysis like Cheddar [11] and MAST [5]; dependability assessment: AADL provides an annex for modeling propagation of error, like COMPASS project [3], or ADAPT [6]; behavioral analysis: mapping to formal methods and associated model checkers have been defined for Petri Nets [9], RT-Maude [8] and many others code generation: Ocarina implements Ada and C code generators for distributed real-time embedded systems [7].

3. CODE GENERATION FROM AADL

Automatic code generation from AADL models require first a comprehensive definition of a versatile middleware. Versatility is required so as to ensure a consistent mapping from AADL concerns (multiplicity of schedulers, transport protocols) to well-defined implementable services. Such versatility has been captured in the schizophrenic middleware architecture and later used in the Ocarina code generation toolset.

3.1 Schizophrenic middleware architecture

In previous projects, we defined the "schizophrenic middleware architecture" [13]. 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 3 layers: *application-level* and *protocol-level* personalities built around a *neutral core*. The user's application interacts with application personalities; protocol personalities operate with the network.

Application personalities constitute the adaptation layer between application components and the middleware through a dedicated API or code generator. They provide APIs to interface application components with the core middleware; they interact with the core layer in order to allow the exchange of requests between entities. Application personalities can either support specifications such as CORBA, JMS, etc. or dedicated API for specific needs.

Protocol personalities handle the mapping of personality-neutral requests (representing interactions between application entities) onto messages exchanged using a chosen communication network and protocol. Protocol personalities can instantiate middleware protocols such as IIOP (for CORBA), SOAP (for Web Services), etc.

The neutral core acts as an adaptation layer between application and protocol personalities. It manages execution resources and provides the necessary abstractions to transparently pass requests between protocol and application per-

[1]An updated list of supporting tools, projects and papers can be found on the official AADL web site `http://www.aadl.info`.

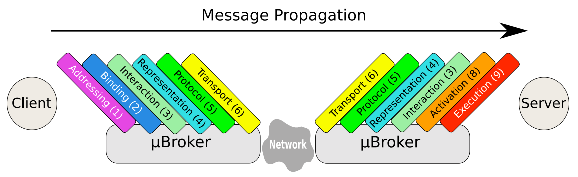

Figure 2: Services of a schizophrenic middleware

sonalities in a neutral way. It is completely independent from both application and protocol personalities.

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 middleware core provides neutral services that correspond to the identification of the key functions involved in request processing. In figure 2, we define the canonical operations performed by any middleware.

The *µbroker* is the core component that provides support for interaction between the canonical services:

- *addressing* manages references of entities connected to the middleware,
- *binding* handles a connection with the remote nodes,
- *representation* takes care of marshaling and unmarshaling when necessary,
- *interaction* manages the liaisons between connected entities in the application,
- *protocol* supports the transmission between two nodes thanks to the network link,
- *typing* manages the typing system in the application (sophisticated when it comes to CORBA *any* mechanism for instance),
- *transport* handles the physical line,
- *activation* ensures that a concrete entity is available to execute requests,
- *execution* assigns resources to process the requests.

In [13], we presented PolyORB, our implementation of a schizophrenic middleware. PolyORB is a free software middleware framework. We assessed its suitability to build middleware platforms to support multiple heterogeneous specifications (CORBA, Ada Distributed Systems Annex, Web Applications, Ada Messaging Service close to Sun's JMS, OMG DDS) and as a COTS for industry projects.

3.2 Code generation with Ocarina

Our code generation strategy consists in using AADL to describe the user requirements as well as the deployment information. We reuse the schizophrenic architecture and its canonical functions to automatically generate most of them in order to implement an AADL distribution model restricted to those features required by the application. We take advantage of the deployment information to statically instantiate the policies needed. To do so, we had to revisit PolyORB implementation as its initial version was based on design patterns. The new schizophrenic middleware, PolyORB-HI, is composed of a minimal middleware core and several automatically generated functions.

PolyORB-HI strictly follows restrictions set by High-Integrity applications on object orientation, scheduling, use of memory. It was developed in Ada95. It is compliant with both the Ravenscar profile and the High-Integrity system restrictions (Annexes D and H of the Ada standard). High-Integrity system restrictions are facilities provided by the Ada 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.

We defined our distribution model as a set of *sender/receiver* tuples that interact through asynchronous oneway messages. This allows for clean support of the Ravenscar model of computation. 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.

These elements will be later refined in section 5.

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 code attached to AADL threads.

4. LEVERAGING ADA 2012 AND SPARK 2014

Ada 2012 [12] is the latest revision of the Ada programming language. One of the most interesting features in the context of code generation is the capability to attach contracts to subprograms as pre and post-conditions, or invariants applied to types. This brings to Ada a feature that existed in other languages like Fortran, Eiffel, and SARK. Ada 2012 rationale [2] lists all details of this feature. Other new additions related to expressions, iterators or multi-core processing are noticeable, but not relevant to our experiments. These will not be discussed here

SPARK 2014 [1] is built on top of Ada 2012 well-defined semantics and definition of run-time errors, as well as programming-by-contract constructs to bring to the user evidence his code will under no circumstances raise a runtime error or that all contracts are true. SPARK 2014 relies on the same concepts as the initial SPARK language and toolset: definition of a sound subset of Ada, combined with a Ada processor that generates Verification Conditions (VCs). VCs denotes boolean predicates that should be proved correct (or discharged) based on the current context of a call, or on existing hypothesis such as assertions or pre and post-conditions.

4.1 Adapting for Ada 2012

Updating PolyORB-HI/Ada to Ada 2012 requires first capturing the relevant features that deserve attention. Being implemented in Ada95 and regular restrictions, compiling the existing code base under Ada 2012 language definition does not require any modification thanks to backwards compatibility. Yet, the current code base makes extensive usage of assertions in the form of pragma Assert in the implementation code to assess the validity of input parameters. These can be re-implemented as pre-conditions, attached to subprogram signatures as shown in this example:

```
package PolyORB_HI.Messages is
   --  ...
   function Sender (M : Message_Type)
      return Entity_Type
       with Pre => (Valid (M));
       --  Ensure M is valid

private
   subtype PDU_Index is Stream_Element_Count
      range 0 .. PDU_Size;
   subtype PDU is Stream_Element_Array
      (1 .. PDU_Index'Last);

   Empty_PDU : constant PDU := (others => 0);

  type Message_Type is record
      Content : PDU := Empty_PDU;
      First   : PDU_Index := 1;
      Last    : PDU_Index := 0;
   end record;

   function Valid (M : Message_Type)
      return Boolean is
     (M.First >= M.Content'First
         and then M.First < M.Last
         and then M.Last <= M.Content'Last);
```

Such adaptations are truly minor adaptations of the existing code base, and do not require excessive rewritings.

4.2 Introducing SPARK 2014

Adaptations towards SPARK2014 require a more complex reengineering of the existing code base. First, one needs to understand in-depth the additional requirements of SPARK2014, and adapt to these. Luckily enough, SPARK2014 is a much more extensive language subset than its ancestor SPARK2005. In the context of PolyORB-HI, this means

- *contracts:* No need for extensive additions of SPARK-specific contracts for global variables, package visibility, etc. All these are now deduced when need to be, and not necessary for checking for the absence of run-time errors, or validity of pre- and post-conditions;
- *generics:* Ada generic packages are allowed. They are used extensively in PolyORB-HI/Ada to support code-generation driven instantiations of message queues, message marshallers or task artifacts;
- *access types:* Use of access types is explicitly forbidden. The runtime does not use access types for dynamic memory allocation. Yet, a few occurrences of access type are present in the generated code for the implementation of routing matrix. For each thread, we need a structure that holds the destinations associated to each outgoing ports. In the initial design of the runtime, these were encoded as static arrays following this pattern: For each port, an array encoding all destinations is generated; its address is used in an array indexed by the port type. This was to circumvent the limitation in Ada in creating non-rectangular 2D arrays, so as to limit memory consumption, as shown in this example:

```
Foo_Destinations :
 constant Foo_Destinations_Array :=
 Foo_Destinations_Array'
 (1 => Foo_Signal_K);

Task_Foo_Destinations :
 constant Foo_Address_Array :=
 (Foo_Port => Foo_Destinations'Address);
```

  We changed this code pattern to implement a function that returns this array, avoiding the need for an Address attribute, at the expense of a slight memory consumption increase.

Let us note SPARK2014 has other restrictions we had to mitigate. We discuss them in section 5. These restrictions correspond to situation where we need bindings to C function or unchecked conversions. These functions are required for the proper operation of the middleware and have been kept as-is. We made usage of the `SPARK_Mode` aspect to hide some elements of the implementation, and kept those as minimal as possible.

4.3 Addressing concurrency

More problematic to the current definition of SPARK2014 and associated implementations is the lack of support for concurrency constructs. PolyORB-HI/Ada uses the Ravenscar model of computation, it thus needs tasks and protected objects. Yet, we may work-around this issue.

4.3.1 Revisiting the Ravenscar Profile

Let us recall that the intent when moving to SPARK2014 is to demonstrate the absence of run-time errors. Considering we are under the restrictions of the Ravenscar profile, we can make the following statements:

- *Use of tasks:* A task has no entry, it can either execute sequential code, or makes use of a protected object. To demonstrate that a task cannot cause a runtime-error, it is therefore sufficient to demonstrate that the sequential code is correct, and the pre-conditions for calling entries or procedures on protected object are met;

- *Use of protected object procedures* Protected object procedures cannot block as a consequence of the pragma Detect_Blocking. Furthermore, in PolyORB-HI/Ada they do not call any other protected objects. As a consequence, these are equivalent to sequential code. The conditions for a run-time error in such procedures are therefore reduced to runtime error in sequential code, or violation of the Ceiling policy. The latter can be assessed through external code reviews by reviewing accessors of the protecting object;

- *Use of protected object entries* By symmetry with the previous case, entries share the same considerations. Besides, they cannot use the requeue mechanism. Hence, these are limited to sequential code encapsulated in an entry block.

Hence, per construction of the Ravenscar profile, concurrency constructs can induce run-time errors only in a limited number of situations, namely violations of the ceiling protocol or blocking inside a protected object procedure or entry. We take advantage of this to revisit our code generation strategy.

4.3.2 Updating code generation strategies

From the previous considerations, we may now update code generation strategies implemented in Ocarina:

- *Management of tasks* Tasks are used for implementing one of the AADLv2 dispatching policy: periodic, sporadic, background, timed, hybrid or aperiodic. The generated code instantiates one generic package that implements the corresponding task skeleton.

  In the context of SPARK2014, we revisited this pattern, and decided to diverge from the Ravenscar profile and introduce a Round-Robin non-preemptive scheduler. Under this scheduling policy, we instantiate a reduced task skeleton that is limited to an infinite loop executing user code, without usage of Ada tasking. This skeleton is a simple function that represents an AADLv2 thread automata, made of initialization code and a call to the user code.

- *Management of queues* Protected objects are used to support message exchanges between tasks. We split the implementation of this package in two separate packages: one package implementing the management of the message queue; another being the implementation of the protected message queue. Under the Round-Robin non-preemptive scheduler, we use the non-protected variant; under the Ravenscar implementation, we use the protected variant.

As a consequence, we can now generate SPARK2014 compliant code from AADL models under the hypothesis of tasks being scheduled under a non-preemptive Round-Robin scheduler. Should we achieve formal proof of this code, then we have arguments to derive similar results in the case of a Ravenscar-compliant model through additional review or testing.

- Correct usage of Ceiling protocol is reduced to external review of the control flow of the program. In our setting, each protected object is associated with a ceiling priority set to the maximum value. Hence violations are not possible.

- Correct implementation of the task skeletons used in the Ravenscar-compilant case. We rely on existing well-known patterns from [4], these are validated from extensive usage.

Other potential errors due to the use of Ravenscar profile are yet to be determined, yet we are confident those will be limited to simple cases.

5. DEMONSTRATING ABSENCE OF RUN-TIME ERRORS

Having adapted code generation strategies for Ada2012 and SPARK2014, we now discuss in this section the application of the SPARK2014 toolset to demonstrate the absence of runtime errors in both the PolyORB-HI/Ada middleware and generated code.

For this study, we considered the usage of the GNATProve GPL 2014 edition. The input AADL model used is from [7], and is derived from the case study from [4]. This model exhibits a set of periodic threads interchanging data. We simply adapted it to use the non-preemptive Round Robin scheduler we introduced instead of a Ravenscar preemptive one. Although the functional code is relevant, we will focus only on the proof of absence of runtime errors of the generated code and the underlying middleware.

In our setting, GNATProve generates Verification Conditions (VCs) based on possibility of a runtime error (and an exception being raised), or due to pre-conditions. Thanks to the GNAT front-end technology, GNATProve generated VCs only for the non-trivial cases that cannot be eliminated by the front-end. Hence, several potential VCs, per strict compliance to the Ada Reference Manual, are discharged internally and not shown to the user.

5.1 PolyORB-HI/Ada code

We first review each package of the current distribution, and discuss how to prove absence of runtime errors.

- `PolyORB_HI`, `PolyORB_HI.Errors`, `PolyORB_HI.Streams`: these packages only define types, there is no code associated, hence no generation of Verification Conditions;

- `PolyORB_HI.Port_Kinds`: this package defines enumeration types for ports and basic test function. Although code exists, it is simple enough to not require any generation of Verification Conditions;

- `PolyORB_HI.Utils`: this package defines basic conversion function, bounded string manipulation API. All 21 functions are considered, 16 VCs are generated and fully discharged;

- `PolyORB_HI.Output`: this package defines basic output function that emulates Ada.Text_IO.Put_Line using a thin binding over the C function `write`, along with basic formatting.

  Let us note that we take advantage of the `SPARK_Mode` aspect to hide from the toolset code that make thread-safe calls to the print functions. Similarly, the binding to the C function `write()` is hidden from the toolset, as it uses C pointers.

  Only the functions in charge of formatting text are "seen" by the toolset. The 14 associated functions yields to 13 VCs fully discharged;

- `PolyORB_HI.Protocols`: this package provides a single function, which is a wrapper that simply calls generated corresponding function from generated code in `PolyORB_HI.Generated.Transport` to send requests. It does not have proper VCs generated

- `PolyORB_HI.Suspenders`: this package provides a mechanism to support a synchronized start of all tasks. It relies on two Ada runtime packages (`Ada.Real_Time` and `Ada.Synchronous.Task_Control`) to either suspend the environment task (using a delay until) or awake all tasks.

  Although SPARK2014 does not fully support these packages, the way primitives operations are used is compatible with the supported toolset. Since the code simply iterates through bounded arrays (denoting tasks to be awaken), it does not cause any VC to be generated.

- `PolyORB_HI.Messages`: this package is in charge of the request life cycle: building, marshalling elements, encapsulation of message destination, etc. This package makes heavy usage of arrays to represent the underlying message, and copying to perform marshalling and encapsulation of data.

  This package has 16 functions, generating 35 VCs. We note 5 VCs are not discharged. These are related to the usage of slicing operation, for which the toolset requires some enhancements.

- `PolyORB_HI.Port_Type_Marshallers` and `PolyORB_HI.Time_Marshallers`: these packages instantiate the generic package `PolyORB_HI.Marshallers_G`. This package provides generic marshalling functions based on performing an unchecked conversion from a base type to a corresponding array of Stream_Elements.

  To date, this package cannot be proved as the toolset does not support the `Size` attribute. More problematic, `SPARK_Mode`, as of GNATProve GPL2014, is inoperant on generic packages: these are systematically considered for proof. These instances have been isolated in a package, and hidden from the toolset.

- The generic packages `PolyORB_HI.Null_Periodic_Task` and `PolyORB_HI.Thread_Interrogators` support respectively the task skeleton and the message queue used for our Round-Robin scheduling strategy. Being generic packages, these can only be proved at instance-level. The generic package cannot be proved alone.

Let us note that the following packages were outside of this study: all packages implementing task skeletons and protected queues, `PolyORB_HI.Scheduler` (used for task migration) and `PolyORB_HI.Transport_Low_Level` (used in a distributed setting). This code requires support for tasking, or access to networking APIs that are currently not available.

From this first round of analysis, we conclude that the toolset is able to proof a significant portion of the code of the runtime. This can be explained by the extensive usage of simple patterns compatible with High-Integrity requirements: no dynamicity, basic transformations, etc. Handling slice operations is a known-issue, this is likely to be addressed by enhancing the GNATProve toolset.

5.2 Generated code

We now review code generated from the AADL model, and bound to the PolyORB-HI/Ada runtime.

- `PolyORB_HI.Generated.Types`: this package defines types mapped from AADL types. There is no code and no VC associated;

- `PolyORB_HI.Generated.Deployment`: this package defines types derived from the deployment information from the AADL model: name of threads, processes. There is no code, but 17 VCs associated. They correspond to elaboration code to build an id-to-string correspondance table.

- `PolyORB_HI.Generated.Marshallers`: this packages defines per-type marshaller/unmarshaller functions. All 32 functions, 23 VCs could be discharged completely;

- `PolyORB_HI.Generated.Transport`: this package implements the send/receive operation used by AADL threads to communicate. In a local setting, these functions simply dispatch the request to the local message queue. 8 functions are generated, 3 VCs out of 4 could not be proved. They relate to pre-conditions that cannot be fullfilled: similarly to the `PolyORB_HI.Messages` package, the corresponding code performs slicing operations, for which support at proof-level is incomplete;

- `PolyORB_HI.Generated.Activity`: this package instantiates task skeletons and message queues from the set of threads and ports of the AADL model. Proof of the task skeleton is trivial, and leads to no VC generation as we simply call in sequence two functions. Proof of the message queue in a non-protected case is a bit more complex, as it involved 50 VCs, of which 7 are not discharged. As stated previously, slicing is again the main issue.

5.3 Lessons learnt

In this experiment, we were mostly interested in demonstrating the absence of runtime errors. We used GNATProve GPL2014 as an oracle to report on potential issues. Surprisingly, the process went smoothly: most of the VCs could be proved at the first round. A few were discharged after rewriting part of the code to cover some corner cases (e.g. non-initialized variable, potential out-of-bound execution, etc.). These were corrected and integrated in PolyORB-HI/Ada.

Although GNATProve allows one to use an IDE to discharge proof manually, we note it was not necessary in our case. On the one hand, the code complexity is quite small, despite the services being made. The runtime mostly transforms data, and moves it to queues. On the other hand, some code patterns like slicing cause troubles. We decided not to change them with manual copies through loops: reports on SPARK2014 forums indicate this is being addressed. Hopefully, the full code base will soon be fully proved in the next iteration of the SPARK2014 toolset.

Finally, time to run the toolset in an automatic way to discharge VCs is reasonably low. The full analysis of code generated + runtime is less than 3 minutes to cover the corresponding 5 kSLOCs.

6. CONCLUSION

In this paper, we aimed at leveraging both Ada 2012 and SPARK 2014 to assess the absence of run-time errors in a restricted middleware that is used for code generation from AADL models using Ocarina and PolyORB-HI/Ada. The initial implementation already includes many restrictions for both the Ravenscar profile, and the High-Integrity domain.

We first introduced how we updated the existing code base to use Ada 2012 pre- and post-conditions, and then adaptation to work-around restrictions of SPARK 2014. We also discussed a few corner cases when this was not possible (bindings to C functions, unchecked conversion). Noting the strong restriction set by the absence of concurrency constructs, we introduced a restricted non-preemptive scheduler in the runtime to evaluate sequential code with a good level of coverage. We also introduced arguments to exploit results in a Ravenscar-context by translating some of the results, combined with careful code review.

Future work will consider two directions: impact of the code refactoring on time and memory performances; further definition of the contracts to demonstrate the code implemented matches the actual semantics as defined in the AADLv2 standard. The second item is likely to stretch SPARK2014 limits in terms of proof of complex contracts.

Acknowledgments

The authors wish to thank Yannick Moy from AdaCore for his insightful comments and suggestions when discussing adaptation of the initial code base to SPARK2014 and in our effort to prove it using the GNATProve GPL2014 toolset.

7. REFERENCES

[1] AdaCore and Altran. SPARK 2014 Reference Manual. Technical report, 2011-2014.

[2] J. Barnes. Ada 2012 Rationale, Chapter 1: Contracts and Aspects. Technical report, 2014.

[3] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri. The COMPASS Approach: Correctness, Modelling and Performability of Aerospace Systems. In *Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security*, SAFECOMP '09, pages 173–186, Berlin, Heidelberg, 2009. Springer-Verlag.

[4] B. Dobbing, A. Burns, and T. Vardanega. Guide for the use of the of the Ravenscar Profile in High Integrity Systems. Technical report, 2003.

[5] M. González Harbour, J. Gutiérrez García, J. Palencia Gutiérrez, and J. Drake Moyano. MAST: Modeling and Analysis Suite for Real Time Applications. In *13th Euromicro Conference on Real-Time Systems*, pages 125–134. IEEE, 2001.

[6] M. Hecht, A. Lam, and C. Vogl. A Tool Set for Integrated Software and Hardware Dependability Analysis Using the Architecture Analysis and Design Language (AADL) and Error Model Annex. In I. Perseil, K. Breitman, and R. Sterritt, editors, *ICECCS*, pages 361–366. IEEE Computer Society, 2011.

[7] G. Lasnier, B. Zalila, L. Pautet, and J. Hugues. OCARINA: An Environment for AADL Models Analysis and Automatic Code Generation for High Integrity Applications. In *Reliable Software Technologies'09 - Ada Europe*, volume LNCS, pages 237–250, Brest, France, June 2009.

[8] P. C. Ölveczky, A. Boronat, and J. Meseguer. Formal Semantics and Analysis of Behavioral AADL Models in Real-Time Maude. In J. Hatcliff and E. Zucca, editors, *FMOODS/FORTE*, volume 6117 of *Lecture Notes in Computer Science*, pages 47–62. Springer, 2010.

[9] X. Renault, F. Kordon, and J. Hugues. Adapting models to model checkers, a case study : Analysing AADL using Time or Colored Petri Nets. In *IEEE/IFIP 20th International Sypmosium on Rapid System Prototyping* , Paris, France, June 2009.

[10] SAE. Architecture Analysis and Design Language (AADL) AS-5506A. Technical report, The Engineering Society For Advancing Mobility Land Sea Air and Space, Aerospace Information Report, Version 2.0, January 2009.

[11] F. Singhoff, A. Plantec, P. Dissaux, and J. Legrand. Investigating the usability of real-time scheduling theory with the Cheddar project. *Journal of Real-Time Systems, Springer Verlag*, 43(3):259–295, November 2009.

[12] S. T. Taft, R. A. Duff, R. Brukardt, E. Ploedereder, P. Leroy, and E. Schonberg. *Ada 2012 Reference Manual. Language and Standard Libraries - International Standard ISO/IEC 8652/2012 (E)*, volume 8339 of *Lecture Notes in Computer Science*. Springer, 2013.

[13] T. Vergnaud, J. Hugues, L. Pautet, and F. Kordon. PolyORB: a Schizophrenic Middleware to Build Versatile Reliable Distributed Applications. In *Proceedings of the 9th International Conference on Reliable Software Techologies Ada-Europe 2004 (RST'04)*, volume NCS 3063, pages 106–119, Palma de Mallorca, Spain, June 2004. Springer Verlag.