Open Archive Toulouse Archive Ouverte (OATAO)

OATAO is an open access repository that collects the work of Toulouse researchers and makes it freely available over the web where possible.

This is an author-deposited version published in: http://oatao.univ-toulouse.fr/
Eprints ID: 13515


Any correspondence concerning this service should be sent to the repository administrator: staff-oatao@inp-toulouse.fr
Tighter integration of drivers and protocols in a AADL-based code generation process

Jérôme Hugues

1Université de Toulouse, ISAE
10, Avenue E. Belin 31055 Toulouse Cedex 4, France
jerome.hugues@isae.fr

Abstract

Model-based engineering provides an appealing framework for the precise modeling and analysis of embedded systems. Architecture Description Languages provide a clear and precise semantics to address multiple analysis dimensions: scheduling, fault, resource accounting, etc. This is completed by code generation tools that generate all required glue code to enable intercommunication between components and associated configuration mechanisms. The diversity of embedded targets requires extended configuration to support multiple devices, operating systems but also compilation toolchains. Yet, those are usually hard-wired in the code generation process.

In this paper, we propose several patterns to support model-level configuration of the target, but also increased analysis capabilities in the context of the AADLv2.

Keywords AADL, REAL, device driver and RTOS integration

1. Introduction

Model-based engineering provides an appealing framework for the precise modeling and analysis of embedded systems. Architecture Description Languages provide a clear and precise semantics to address multiple analysis dimensions: scheduling, fault, resource accounting, etc. This is completed by code generation tools that generate all required glue code to enable intercommunication between components and associated configuration mechanisms.

Even-though there are modeling patterns for decoupling platform specific concerns from the logic of the system (PIM/PSM decoupling), there is still a strong need for patterns to integrate configuration parameters of the target environment, but also implicit execution resources – tasks, buffers, etc. – used by the execution runtime. Higher precision in modeling patterns and associated information would bring more confidence in analysis results.

Besides, the diversity of embedded targets requires extended configuration to support multiple devices, OS but also compilation toolchains. Still, we note those are usually hard-wired in the code generation process.

In this paper, we consider the AADLv2 language [9]. This architecture description language promoted by SAE aims at the precise description of embedded systems for analysis and generation purposes. In [6], we underlined the fact that the AADL ecosystem is rich of many diverse analysis tools, covering most steps in a typical engineering cycle. We also underlined the fact that precise modeling is a key asset to be further addressed.

In the following, we consider precise modeling from the perspective of code generation. We present contributions to model precisely runtime elements such as interrupts, device drivers. Those are central for embedded systems, but are seldom contemplated in an inclusive and extensive code generation strategies. We introduce both modeling patterns and code generation artifacts to support them.

In section 2, we briefly introduce AADLv2; in section 3 we introduce AADLib, a library of reusable building blocks for AADLv2; section 4 introduces modeling patterns for platform elements: interrupt handlers and drivers. Then, we tackle the issue of code integration in section 5 and conclude.

2. An overview of AADLv2

The “Architecture Analysis and Design Language” AADL is a textual and graphical language for model-based engineering of embedded real-time systems. It has been published as an SAE Standard AS-5506B [9]. AADL is used to design and analyze software and hardware architectures of embedded real-time systems.

The AADL allows for the description of both software and hardware parts of a system. It focuses on the definition of clear block interfaces, and separates the implementations from these interfaces. It can be expressed using both a graphical and a textual syntax. From the description of these blocks, one can build an assembly of blocks that represent the full system. To take into account the multiple way to connect components, the AADL defines different
connection patterns: between subcomponents, across components and binding of software blocks to hardware.

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

An AADL description is a set of components. 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).

Each Component category describes well identified elements of the actual architecture, using the same vocabulary of system or software engineering:

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

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

- **Virtual bus** and **Virtual processor** models “virtual” hardware components. A virtual bus is a communication channel on top of a physical bus (e.g. TCP/IP over Ethernet); a virtual processor denotes a dedicated scheduling domain inside a processor (e.g. an ARINC653 partition running on a processor).

- Unlike other components, **Systems** do not represent anything concrete; they combine building blocks to help structure the description as a set of nested components. Packages add the notion of namespaces to help structuring the models. Abstracts model partially defined components, to be refined during the modeling process.

Component declarations have to be instantiated into subcomponents of other components in order to model 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 contains 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, which is called 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. Each component type can receive zero or several implementations. Each of them describes the internals of the components: subcomponents, connections between those subcomponents, ...

An implementation of a thread or a subprogram can specify call sequences to other subprograms, thus describing the execution flows in the architecture. Since there can be different implementations of a given component type, it is possible to select the actual components to put into the architecture, without having to change the other components, thus providing a convenient approach to configure applications.

The AADL defines the notion of properties that can be attached to most elements (components, connections, features, ...). Properties are typed attributes that specify constraints or characteristics that apply to the elements of the architecture: clock frequency of a processor, execution time of a thread, bandwidth of a bus, ... 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).

AADL is a language, with different representations. A textual representation provides a comprehensive view of all details of a system, and graphical provide a higher level of abstraction, and allow for a quick navigation in multiple dimensions. In the following, we illustrate both notations. Let us note that AADL can also be expressed as a UML model following the MARTE profile [4].

The concepts behind AADL are those typical to the construction of embedded systems, following a component-based approach: blocks with clear interfaces and properties are defined, and compose to form the complete system. Besides, the language is defined by a companion standard document that documents legality rules for component assemblies, with its static and execution semantics.

The figure 1 is derived from [3] is a case study for illustrating the concepts of the Ravenscar computational model, applied in AADL. It illustrates on an instance model how a set of tasks can be connected, packaged in a process and finally bound to a processor that abstracts away the system’s execution resources.

AADL is rich of many projects that address analysis dimensions: scheduling, fault, resource accounting, etc. This is completed by code generation tools that generate all required glue code to enable intercommunication between components and associated configuration mechanisms.

![Figure 1. Ravenscar case study](image)
OADLib [8] is a model processing framework, supporting code generation towards C and Ada. It acts as a compiler, generating code in one pass. Configuration parameters are limited to one parameter defining the target RTOS, and configuration of transport endpoints for communication layers.

• RAMSES [1] is a code generation framework based on ATL, and an extension of OSA TE2. It operates through successive refinement of the initial AADL models, making explicit all system calls (buffer and queue management, task creation, etc.). It targets two operating systems: OSEK and ARINC653/POK.

These two projects share common patterns for modeling and then generating code. Yet, the support of the target platform is imprecise, and reduced to the configuration of the scheduling parameters (scheduler, priority only). Communication mechanisms are hard-coded in the model transformation, relying on a restricted set of libraries.

As we mentioned earlier, AADL, or others like MARTE and EAST-ADL, provide similar constructs, and are conceptually really closed as underlined in [7]. A natural question is thus to review missing blocks for precise system modeling. In particular, how to define a library of reusable blocks? How one would model interrupts, bus and associated protocol stacks? how to support seamless integration of associated code blocks in the generated code? All those particular concerns are important to propose a complete view of the system, and to provide accurate analysis. In the following, we review each concern separately, and discuss solutions implemented in the Ocarina project1.

3. AADLib: Extended property sets and reusable models

Like most MDE notations, AADL has a rich ecosystem of tools supporting a wide range of concerns (safety, scheduling, budget analysis, etc.). This may be overwhelming for newcomers.

The general objective of this library is to provide a central repository of AADL models geared towards the community. To be effective, this library should be easily integrated with existing AADL modeling environment, but also provide a large variety of examples.

To support these objectives, we initiated a project on the GitHub forge codenamed “AADLib” for AADL Library. This project provides AADL models freely reusable, under a Free/Libre Software license.

3.1 Extended property sets

AADLv2 supports a wide set of non-functional properties. Yet, to our surprise, some key properties are not present in the current standard, and could be of great help to provide a clear description of blocks. AADLib provides additional properties. We list here the additional concerns modeled:

• processor_properties.aadl: this property set complements the properties applicable to processors with endianness, frequency, MIPS, FPU or multi-core concerns, see listing 1 for an example,
• bus_properties.aadl: adds bandwidth and channel type (duplex, half-duplex) considerations,
• data_sheet.aadl: connects AADL model entities to data sheets or bill of materials for physical implementation,
• electricity_properties.aadl: covers energy converters and electric units. This is useful for characterizing devices or processor consumptions,
• physical_properties.aadl: adds other units for power, mass, etc.,
• memory_segments.aadl: extends the description of memory components with fine-grained definition of segment or page descriptors.

These properties help providing a full description of a system, and it is used intensively to model the blocks forming the library of reusable AADL elements provided by AADLib.

property set Processor_Properties is
Processor_Family : enumeration (ARM, AVR, SPARC, PowerPC, x86, x86_64) applies to (processor);
Frequency : type aadlinteger 0 Hz .. 2#1#e32 Hz
type=aadlinteger; applies to (processor);
— Frequency of a processor
Endianness : enumeration (Little_Endian, Big_Endian, Bi_Endian) applies to (processor)
Word_Length : Size applies to (processor)
— Length of a word for this processor architecture
FPU_Present : aadiboolean applies to (processor);
MIPS : aadlinteger 0 .. Max_Aadlinteger applies to (processor);
Core_Id : aadlinteger 0 .. Max_Aadlinteger applies to (virtual processor);
end Processor_Properties;

Listing 1. Property set for processor

3.2 Reusable building blocks

In addition to extended property sets, AADLib proposes a set of building blocks. These blocks provide a valuable asset to start new models. The library is built following AADL model hierarchy of elements:

• Processors: various ARM, AVR, PowerPC, SPARC, x86 processors are available, with endianness, frequency, ports modeled;
• Buses: typical network interfaces are modeled, covering AFDX, ARINC429, CAN, Ethernet, I2C, MIL-STD 1553, PCI, SpaceWire, UART, USB, with known limits in bandwidth, packet size, etc.,
• Miscellaneous devices: battery, GPS, accelerometers, inertial measurement units, etc. Those are modeled after

1All models presented in this paper are available on-line. See http://www.openaadl.org for more details.
4. Modeling device drivers

An important aspect of embedded systems is their capability to associate physical events to software reactions. Such functions have a significant impact on software performance: bus usage, associated CPU overhead for copying data, specific memory mappings, etc. Thus, one needs to model the software blocks in charge of processing input/outputs.

AADL provides some concepts for attaching subprograms to devices, thus modeling associated device drivers. Yet, they are not precise enough to lead to code generation. In this section, we review additions supported by Ocarina to attach code representing an Interrupt Service Routine (or ISR) and drivers to AADL models.

4.1 Modeling Interrupts

An interrupt service routine can be seen as a particular kind of thread, attached to one interrupt line in a system. Its modeling is thus reduced to an extension of existing dispatch protocols supported by AADLv2.

We took advantage of some liberty provided by the language to extend the list of supported dispatch protocol, specified in the AADL_Project property set. This set defines the list of available enumerators for some properties, like the concurrency control protocol, queuing discipline, etc. Let us note similar allowance exists for Ada, it is thus typical.

The list of supported dispatch protocol has been extended with the “Interrupt” enumerator for specifying a new dispatch protocol. It is associated with an extended property definition Deployment::Configuration that represents the name of the associated interrupt. Depending on the target operating system or language, a tool generator like Ocarina will map this string to the corresponding type definition. In the context of Ada, it has to conform to one of the names defined in the Ada.Interrupts.Names package.

Several restrictions are put on this category of threads:

- To respect constraints on ISR (short time, no blocking, etc), ISR threads cannot have ports for communicating. This would require complex support from the underlying AADL runtime;
- in ports are also forbidden, which have no sense: ISR is triggered by an interrupt, external to the thread interface;
- out ports would require protected object for communicating, and thus could incur blocking.

Let us note that, should an ISR need to communicate and store information, it has to use global variables with associated concurrency protocol. This is supported through “required data access” mechanism in AADL.

Supporting this modeling pattern for code generation is straightforward in Ada: we take advantage of existing language features to bind the ISR subprograms to an interrupt handler represented as an Ada protected object. This is being defined as a particular task archetype in our Ada AADL runtime “PolyORB-HI/Ada”: PolyORB_HI.ISR_Task. This archetype follows typical pattern documented in [3].
4.2 Modeling drivers

In addition to modeling drivers for interrupt-driven devices, we need to express the relationship between a logical connection between two processes, and the associated runtime support through an actual communication stack.

From a modeling perspective (see figure 4), only AADL processes can interact with remote processes through logical connections. Thread would ultimately send an event on their outgoing ports to one of the outbound port of a process.

Supporting hardware devices (marked as (1) in the figure) are attached to AADL processor components, modeling the fact that the device is known by the support operating system (3). The device has also access to a bus (2), representing the physical connection. AADL “virtual bus” elements, subcomponents of the bus can be added to model actual communication protocols.

Finally, the logical connection is bound to the physical one to indicate which resources can be used to supporting the interaction between the two processes.

This modeling pattern is actually a faithful interpretation of AADL concepts; it provides all information required to map logical interactions to actual support resources (bus and devices). In order to complete this models, one needs additional patterns for modeling resources used by the devices, namely internal buffers, threads for processing incoming requests, links to actual protocol routines, etc.

To achieve separation of concerns, we take advantage of the Device_Driver property to model all associated resources, see figure 5 for all details.

In this model, we indicate the device is accessing an Ethernet bus, the abstract entity Driver_TCP_IP_Protocol provides two resources to send and receive packets. We use a dedicated subprogram for the emission of messages, and a thread for processing incoming requests. We need a separate thread to wait, due to the semantics of TCP/IP protocol, while we can use the user thread to perform the actual sending as part of its execution. The initialization is performed by the subprogram attach to the Initialize_Entrypoint property.

Actual configuration of the device is done when instantiating one component of this type, through the use of the Deployment::Location property.

Similarly to the interrupt-modeling pattern, several restrictions must be enforced:

- The receiving thread must use a background or time-dependent dispatch protocol, and cannot be dependent on a model-level event; its dispatch is triggered form the arrival of a message (e.g. TCP/IP) and/or specific time;
- Priority of receiving thread must be compatible with the overall schedulability objective of the system, e.g. to avoid risk of priority inversion in case a sender thread blocks a receiver ones.

This list is to be completed by the user with all platform-specific considerations, like level of priorities, restrictions for concurrent accesses to the bus, etc.

Ocarina code generation strategies, detailed in [8] have been enriched to support this new modeling pattern. Code generation takes advantage of the enriched model to

- add to the task set defined by the user the additional threads required by the device drivers;
- connect send/receive functions provided by the driver to the minimalist middleware generated from the architectural model;
- configuration parameters are passed to the initialization function of the device, and enforced during the partition elaboration.

The user has to respect a minimal set of conventions for the driver code: the signature of the Send function is derived from the AADL model, and holds the message and destination. It has all relevant information for sending the message.

On the receiving side, the user code has to unmarshall the request, and then make usage of one internal API to route the message to the receiving thread.

This modeling pattern has been implemented in Ocarina, and declined for various protocols, namely: UART based on GNAT.Serial, TCP/IP based on GNAT.Sockets, SpaceWire and UART based on ORK+ runtime [2].
5. From model patterns to correct integration of code

In the previous sections, we introduced modeling patterns for supporting interrupts and communication protocols through AADL devices. We also listed several restrictions to be respected.

In this section, we detail how these restrictions are checked at architecture level using the REAL language.

5.1 Validation of architectural constraints

An AADL architectural model is a combination of blocks. Its correctness is asserted in multiple dimensions: through the type systems, external tools for specific analysis. Yet, there is a gap in-between, e.g. assessing a device driver is compatible with a given processor/OS couple, or that a models fulfill a given set of patterns (e.g. synchronous, Ravenscar, . . . ).

These considerations lead us to define an AADL language annex: REAL. REAL (Requirement Enforcement Language) aims at checking constraints enforcement on architectural descriptions at the specification step, saving significant time over verification at execution time.

In this section, we describe the main features of this language. REAL pursues multiple design goals:

• Enabling easy navigation through AADL meta-model elements, yet being at a high-level abstraction. To do so, we discarded the use of the UML Object Constraint Language (OCL) and decided to define a specific DSL based on AADL meta-model concepts to ease writing of constraints.

• Allowing to define generic rules. We note that mathematics universal quantifiers ( ∀, ∃) notation is interesting to define metrics that can apply to a wide range of models, not just specific instances.

• Allowing for modularity through definition of separate constraints that can be later combined.

• Being integrated to the AADL as an annex language, so that constraints are coupled to models.

From these goals, we defined REAL with the following design decisions: REAL is based on set theory and associated mathematical notations. The basic unit of REAL is a theorem. A theorem verifies an expression over all the elements of a set that is called the range set. It allows one to build sets whose elements are AADL entities (connections, components or subprogram calls). Verification or computation can then be performed on either a set or its elements by stating Boolean expressions.

In order to write complex expressions, one can use predefined sets, which contain the instances of the AADL model of a given type, or build intermediary sets, using relations between elements of sets (e.g. returns the elements of the set A which are subcomponents of any elements of the set B). Listing 2 shows how to assess all threads are cyclic.

Listing 2. REAL example

```
theorem all_tasks_cyclic
  foreach t in Thread_Set do
```

Figure 6. Applying REAL constraints at model level

```
-- This system drags advanced AADL legality rules for drivers, protocols, etc.

system AADL_System
  annex real_specification [**
    theorem check_all
      foreach s in local_set do
        requires (check_aadl);
    **]
  meta-theorem, checking all rules
  end check_all;

end AADL_System;
```

REAL [5] has been integrated as an annex language in Ocarina, our AADL toolsuite. We present full examples of REAL in the next sections and show how it can help computing metrics of AADL models to drive an optimization process. It has been successfully applied to assess a model conforms to the Ravenscar, MILS or ARINC653 architectural profiles.

As part of the modeling of drivers and interrupts handlers, we defined in the previous sections a set of additional constraints to be met. These were encoded as a set of REAL predicates that are then bound to a model using AADL annex clauses and checked on the model during model analysis, and code generation.

We then apply one theorem at the top node of the hierarchy of components. This theorem has two objectives:

1. calls all subtheorems provided as external library;
2. apply recursively to all its subcomponents (process, bus, device, . . . )

As defined, this theorem serves as an architectural contract the subsequent implementation has to fulfill. Subsequently, we can check all constraints to be met by a set of blocks.

5.2 Integrating last bits: inclusion of user code

The last step towards full inclusion of model patterns and code is to instruct code generator, but also model builders how to link code to models, and ensure the code is valid for the model assembly.

We defined two additional sets of enrichments for the AADL library of models:

1. Constraints a model entity (e.g. a device) must met towards integration. For instance, a given driver can work only for a given operating system/runtime
2. Additional properties for configuring the build system

These two elements rely on a specific property set geared towards the Ada compilation system we use (GNAT in our case), and additional REAL constraints.

In the following example, we use a GNAT-specific project file for setting name of the compiler (following
Finally, we introduced patterns to capture compilation-specific concerns: compilation chain, configuration, link to user code.

By combining all those elements, we provide all building blocks to prepare for library of reusable model assets that match platform needs and associated code. We also, as part of the AadLib project, provide a ready-made set of such blocks.

Future direction will consider 1) the extension of this work to support more operating systems, but also languages targeting C, 2) moving from code generation towards more precise resource analysis (e.g., memory or scheduling).

References


