Rapid Prototyping of Distributed Real-Time Embedded Systems Using the AADL and Ocarina

Jérôme HUGUES, Bechir ZALILA,* Laurent PAUTET
GET-Télécom Paris – LTCI-UMR 5141 CNRS
46, rue Barrault, F-75634 Paris CEDEX 13, France
jerome.hugues@enst.fr, bechir.zalila@enst.fr, laurent.pautet@enst.fr

Fabrice KORDON
Université Pierre & Marie Curie, Laboratoire d'Informatique de Paris 6/MoVe
4, place Jussieu, F-75252 Paris CEDEX 05, France
fabrice.kordon@lip6.fr

Abstract

Building Distributed Real-Time Embedded systems requires a stringent methodology, from early requirements capture to full implementation. However, there is a strong link between the requirements and the final implementation (e.g. scheduling, resource dimensioning). Therefore, a rapid prototyping process based on automation of tedious and error-prone tasks (analysis, code generation) is required to speed up the development cycle. In this article, we show how the AADL (Architecture, Analysis and Description Language), appeared late 2005, helps solving these issues thanks to a dedicated tool-suite. We then detail the prototyping process and its current implementation: Ocarina.

1 Introduction

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

However, the distance between requirements and implementation usually slows down this process: one has to carefully respect non-functional attributes when implementing tasks; any change in the specification has to be carefully propagated at the implementation level; interactions between entities have to be mapped onto run-time entities in a safe manner (deadlock-free, no starvation, no overrun, etc).

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

Hence, developers and system architects need common interchange models to dialog and exchange their requirements and concerns. The AADL (*Architecture Analysis and Design Language*) [9] recently appears as an architecture description language suitable to describe systems, from high-level concerns down to implementation.

"Evolutionary" prototyping is now becoming a well accepted development approach. It is based on a central model that is refined as long as it is not satisfactory. Programs can be generated from this model and constitute a version of the product. The last refined model corresponds to the final system. Also called "Model Driven Engineering" (MDE) it is promoted by OMG.

The goal of this paper is to propose a prototyping methodology based on AADL and dedicated to DRE. AADL is interesting compared to other modeling formalism as it is backed by several industrial from the space and avionics domain. Tools already exist to build and exploit AADL models, from early validation to full implementation.

In the following, we give a brief overview of the AADL, we then discuss how the AADL can serve as a vehicle for a rapid prototyping methodology for DRE systems. Finally, we present our current work on the Ocarina AADL tool suite and assess its use to build High-Integrity DRE.

2 An overview of the AADL

AADL (*Architecture Analysis and Design Language*) [9] aims at describing DRE systems by assembling blocks separately developed.

The AADL allows for the description of both software and hardware parts of a system. It focuses on the defini-

tion of clear block interfaces, and separates the implementations from these interfaces. It can be expressed using both a graphical or a textual syntax.

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

An AADL description is made of *components*. The AADL standard defines software components (data, threads, thread groups, subprograms, processes) and execution platform components (memory, buses, processors, devices) and hybrid components (systems).

Components describe well identified elements of the actual architecture. *Subprograms* model procedures like in C or Ada. *Threads* model the active part of an application (such as POSIX threads). *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, etc. *Buses* model all kinds of networks, wires, etc. *Devices* model sensors, etc. Unlike other components.

Systems do not represent anything concrete; they actually create building blocks to help structure the description.

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

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

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

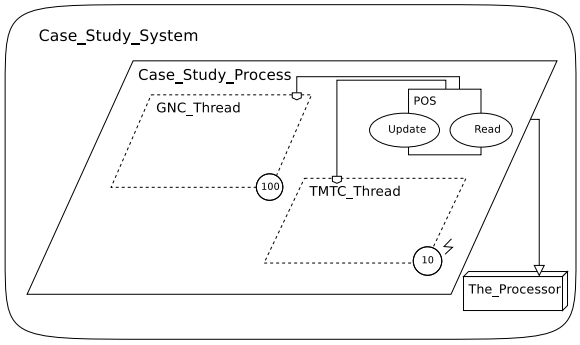

Figure 1. A simple AADL model

Figure 1 presents a simple AADL model that depicts two threads: one periodic (GNC, "guidance navigation control"; one sporadic (TMTC, "telemetry/telecommand") that interact to read and update a shared variable (POS, "position"). Such system models for instance a satellite guidance system controlled by a ground station.

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

Projects such as OSATE [11] define modeling environments to build AADL models, using the Eclipse platform.

We have developed the Ocarina tool-suite [16] to process AADL models and allow the developer to develop, configure and deploy distributed systems. Ocarina offers scheduling analysis capabilities, connection with formal verification tools, and more notably code generation to Ada 2005.

3 A Rapid Prototyping Process for DRE

A DRE is unique in that it should support two very opposite constraints: it should be compatible with needs for critical systems (life-, mission-, business-) and their normative process; but also embrace new standards or technologies [7]; for instance for distributed or embedded systems where new standards or products arise frequently. Therefore, a prototyping process is required to test as soon as possible the impact of deployment decisions, or the use of one software/hardware component in the system. Tools can support this process by providing quick feedback and executable tests to the developer.

3.1 Building prototypes

Two approaches in prototyping are usually distinguished [6]:

- "throw-away": prototypes are built to validate a concept, prior to implementing the real system. The throw-away approach is used to refine requirements.
- "evolutionary": prototypes tend to become the final product. Prototypes are refined to create more accurate ones. The last prototype actually corresponds to the final system (figure 2). Then, feed-back on the system may be provided at various levels and the model is the main reference for describing the system.

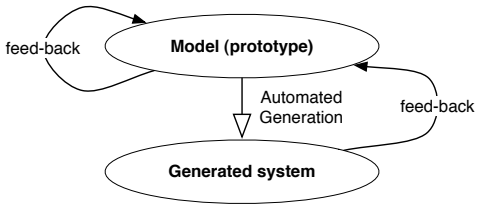

Figure 2. Evolutionary prototyping

Given testing and validation costs, we believe a "evolutionary" approach should be applied to DRE, where prototypes are discarded after some tests, if they are not viable. It is a way to preserve knowledge on the software and hardware that is precious and costly to rebuild. Therefore, one should leverage previous knowledge to build new systems, or refine existing ones.

3.2 Requirements for prototyping

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

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

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

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

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

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

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

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

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

Hence, built models should be exchangeable between tools, and eventually lead to code generation to ease the construction of prototypes. Such prototyping process should therefore be compatible with a MDA-like development cycle [8]. To reduce model discrepancy, a general modeling formalism should be used and conserved during the different steps of the process.

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

3.3 Related Work

Generating High-Integrity code from a model is not limited to AADL models. In [1], the authors state that generating code minimizes the risk of several semantic breaches when translating the model towards code. The manual coding exposes the developer to these breaches. They propose some guidelines to generate Ravenscar compliant Ada 95 code from HRT-UML. However, the excess of using generic instantiations introduces a considerable overhead in the executable code size (30%).

More closely to this paper's scope, the Annex D of the AADL language [10] describes some coding guidelines to translate the AADL software components into source code (Ada 95 and C). These rules are not complete mapping specifications, but they provide guidelines for those who want to generate code from AADL models. In our case, we took from these guidelines the rules that are compliant with the Ravenscar profile.

More concretely, STOOD is a tool developed by Ellidiss Software [13]. It allows users to model their real-time applications using the AADL or the notations proposed by the HOOD method. STOOD allows the code generation from AADL to Ada 95 by converting AADL models to HOOD models and then applying the HOOD to Ada 95 mapping rules. However, the generated code does not rely on a middleware layer and works only for local applications.

In the following, we illustrate how the AADL implements this process; and allows rapid prototyping of complete ready-to-run systems.

4 Rapid Prototyping Using the AADL

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

4.1 Capturing requirements in AADL

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

- data types and related functions to operate on them
- supporting runtime entities (`threads`) and interactions between them (through `connection` and `ports`)
- association of functions to threads
- mapping of threads onto `processes` and `processors` to form the deployed system.

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

Furthermore, AADL allows one to refine the description of each entity to detail more precisely its behavior or some non-functional attributes, allowing one to support design-by-refinement; or even to support inheritance to provide multiple implementations of one component (e.g. a periodic sensor implemented as either a thermal or speed sensor).

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

4.2 Assessing an AADL model

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

In this section, we show how such analysis can be conducted using our AADL model processing suite (figure 3).

Semantic analysis is performed using our AADL compiler Ocarina. Ocarina checks that the given AADL model is conforming to the AADL grammar and that some additional restrictions are respected:

- All event or data ports are connected,
- All threads are either periodic or sporadic,
- All shared data use a concurrent access protocol that bounds priority inversion (e.g. the Priority Ceiling Protocol mandated by the Ravenscar profile).

AADL defines one standard execution semantics, this allows us to go further and assess the system is meaningful, and can run prior to its generation and execution. We allow both schedulability analysis and model checking techniques to fully assess node liveness.

Schedulability analysis is performed using Cheddar [12]. Cheddar is an Ada 95 framework that provides tools and library to check whether AADL threads will meet their deadline at execution time. Cheddar uses the Ocarina [16] libraries to analyze the AADL models.

From an AADL model, a model of interacting tasks is computed. Tasks can interact either locally sharing data through protected objects (or mutexes), or remotely through a communication bus. The first allows for traditional Rate Monotonic Analysis, while the second requires advanced techniques such as Holistic analysis [14]. Cheddar supports both; this enables one to check whether one's architecture can run within expected bounds.

Verifying the behavior of the nodes is performed by transforming the AADL model into a Petri net and then by performing formal verification on the resulting Petri net. The transformation into Petri net is performed using a Petri net generator module of Ocarina. The formal verification (absence of deadlocks...) is performed using the CPN-AMI Petri Net modeling and model checking environment [3].

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

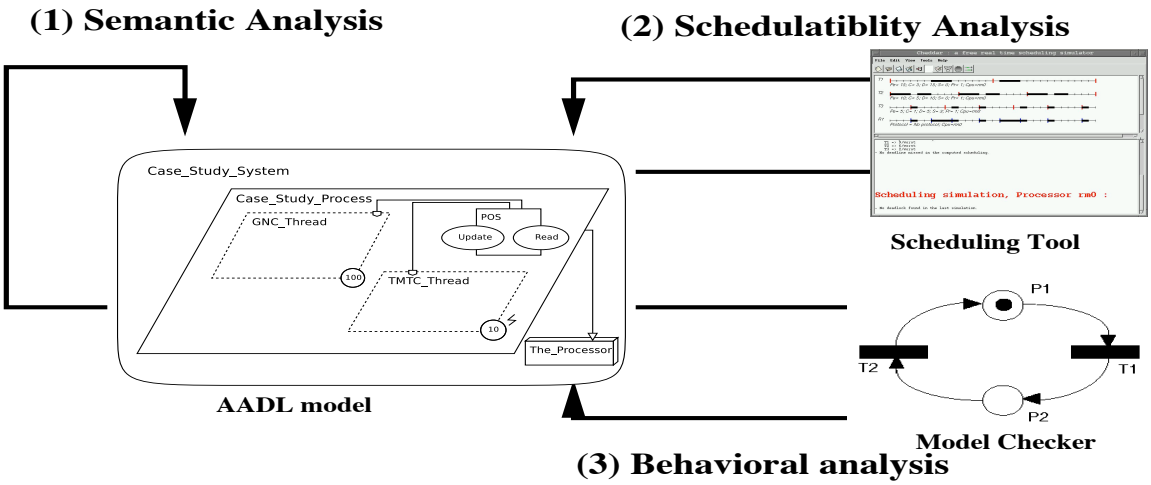

Figure 3. Exploiting AADL models

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

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

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

Listing 1. AADL data component

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

thread implementation TMTC_Thread.Impl
calls {
  Welcome  : subprogram TMTC_Identity;
  TMTC_Job : subprogram TMTC_Job;
  Update   : subprogram POS.Update;
  Bye      : subprogram TMTC_Identity;
};
connections
  Cnx_TMTC_1 : data access TMTC_POS
     -> Update.this;
properties
  Dispatch_Protocol       => Periodic;
  Period                  => 100 ms;
  Compute_Execution_time  => 0 ms .. 50 ms;
  Deadline                => 100 ms;
end TMTC_Thread.Impl;
```

Listing 2. AADL TMTC Thread

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

4.3 Generating executable code

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

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

Code generation relies on well-known patterns for High-Integrity systems, inherited from previous work on code generation from Ravenscar [1] and classical design patterns for distribution such as the Broker [2], constrained to remove all dynamic behavior supported by the minimal middleware for HI systems: PolyORB-HI.

We named this middleware "PolyORB-HI" (PolyORB High-Integrity)as a follow up to the PolyORB project we develop [15]. It shares many common architectural notions while using a different code base. As for PolyORB, this middleware is built on isolated elements that embody key steps in request processing, allowing for finer configuration

of each blocks.

PolyORB-HI [4] strictly follows restrictions set by High-Integrity applications on object orientation, scheduling, use of memory. It is developed in Ada 2005 [5]. It is compliant with both the Ravenscar profile and the High Integrity System restrictions (Annexes D and H of the Ada 2005 standard). Let us note that most restrictions are enforced at compile time (no dispatching, no floating point, no allocator, etc). This simply yet efficiently enforces no unwanted features are used by the middleware, increasing the confidence in the code generated while limiting its complexity. Code generated by Ocarina also follows the same compilation restrictions.

User code is also tested for consistency with the above restrictions. To ensure the user code does not impact scheduling (which might modify scheduling, and thus threatens asserted properties at the model-level), we ensure at compile-time it uses no tasking constructs (tasks and protected objects) by positioning the corresponding restrictions.

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

5 Metrics

Going back to the case study we presented in figure 1, we now detail how these analyses and code generation can be combined to build one ready-to-run system.

5.1 Schedulability analysis & Model checking

The case study we retained is simple enough to be analyzed by these tools. Let us note that intermediate models (Cheddar or Petri Nets) are of similar complexity than the initial AADL model, this implies these analyses are not impeded by the model transformation we propose, but by the initial AADL model and the analysis capability of the tools.

Cheddar can handle only limited time range for simulations (approx. 1500 time units, configurable). CPN-AMI can handle large state spaces but is limited by typical combinatorial explosion problems.

5.2 Code generation

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

Table 1 summarizes the code size for our case study, that combines GNC, TMTC and POS in one node, using local connection through mailbox. The display both the actual lines of code (SLOCs) and the size of the binary objects. All tests were done in local, using the `tsim` simulator for LEON, emulating a $50Mhz$ processor. This functional test allows the computation of minimal footprint of the system. Future work will use a `SpaceWire` interface[1].

The total size of the executable, combining real-time kernel, middleware and the application, is $310kB$, using the GNAT for LEON compiler. It fits the requirements from minimal embedded systems, and is clearly under the typical memory range for API-based middleware such as nORB or microORB, which are above $450kB$.

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

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

Component	SLOCs	.o size (bytes)
*A*pplication	87	4496
*M*iddleware	474	37267
*G*enerated code	436	133985
*A*da Run-Time	N/A	< 166308

Table 1. Generated code metrics

5.3 Assessment of the process

Hence, we demonstrate how to exploit one AADL model and user-provided code for some processing functions. AADL serves both as a documentation of the system and as a template to validate it and generate its implementation: it preserves system design.

Therefore, we have an immediate benefit from an engineering point of view: the developer can focus on its system architecture. The complete tool suite ensures it is correct,

[1] For completeness and as a proof of validity of the architecture, another test was built on GNU/Linux: it implements the low layer transport, using the `sockets` API without notable change to the build process.

and handles the configuration of all code-level entities. This suppresses many manual code writing, a tedious and error-prone process underlined by well-known software failures in the space domain like the Ariane V maiden flight. It also tremendously reduce the development cycle and allow one to go faster to testing.

6 Conclusion

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

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

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

Future work will consider the addition of the SpaceWire fieldbus, a better integration with the Ocarina tool suite, and the extension of the configuration of the middleware to a wider range of platforms and requirements, addressing fault tolerance, or other schedulers, but also other programming languages such as C or embedded Java variants.

Acknowledgement The authors thank F. Singhoff from the Cheddar project and the members of the IST-ASSERT project for their feedback on earlier version of this work. This work is partially funded by the IST-ASSERT project.

References

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

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, New York, 1996.

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

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

[5] ISO/IEC 8652:2007(E) Ed. 3. Annotated Ada 2005 Language Reference Manual. Technical report, 2006.

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

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

[8] OMG. Model Driven Architecture (MDA), Document number ormsc/2001-07-01. Technical report, OMG, 2001.

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

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

[11] SAE. Open Source AADL Tool Environment. Technical report, SAE, 2006.

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

[13] E. Software. STOOD. http://www.ellidiss.com/stood.shtml.

[14] K. Tindell. Holistic schedulability analysis for distributed hard real-time systems. Technical report, University of York, 1993.

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

[16] T. Vergnaud and B. Zalila. Ocarina: a Compiler for the AADL. Technical report, Télécom Paris, 2006. available at `http://ocarina.enst.fr`.