Stood and Cheddar: AADL as a Pivot Language for Analysing Performances of Real Time Architectures

Pierre Dissaux[1], Frank Singhoff[2]

1: Ellidiss Technologies, 24 quai de la douane, 29200 Brest, France
2: University of Brest/LISyC, 20 avenue Le Gorgeu, 29238 Brest Cedex 3, France

Abstract: Coupling of modelling and analysis tools requires that both ends strictly comply with the same semantic definition of the exchanged model. This is particularly important for real-time systems and software architectures. Such a guaranty can be brought by the common use of the Architecture Analysis and Design Language (AADL) all along the tool-chain. This paper discusses modelling and analysis options of various real-time architectural patterns expressed in AADL though an experiment with Stood and Cheddar tools.

Keywords: AADL, Real-Time, Performance analysis, Stood, Cheddar, Design Patterns

1. Introduction

The SAE Architecture Analysis and Design Language (AADL) is a textual and graphical language support for model-based engineering of embedded real time systems that has been approved and published as SAE Standard AS-5506 [1]. AADL is used to design and analyse software and hardware architecture of embedded real-time systems. Many tools provide support for the modelling and the analysis of AADL models. Ocarina implements Ada and C code generators for distributed systems [4]. TOPCASED, OSATE and Stood provide AADL modelling features [11,7,8]. The Fremont toolset and Cheddar implement AADL performance analysis methods [9,10,23]. An updated list of supporting tools can be found on the official AADL web site: http://www.aadl.info.

This article deals with the interoperability between AADL tools: we show how AADL can be used as a pivot language between a modelling tool (Stood) and a performance analysis tool (Cheddar).
Stood is a software design tool that provides an extended support of the AADL modelling language in addition to its compliancy with the HOOD methodology. With Stood, it is possible to manage a complete software project by building libraries of reusable components, reversing legacy code and specifying the real time application as well as its execution platform. Most of the modelling activities can be performed graphically and the corresponding AADL code is automatically generated by the tool.

The Cheddar framework is a set of Ada packages which aims at performing performance analysis of real time architectures. It includes analytical scheduling methods and most of classical real time scheduling algorithms. The Cheddar framework also offers a domain specific language (and its interpreter, compiler, …) for the design and the analysis of schedulers which are not already implemented into the framework.

In this article, in order to illustrate the interoperability between Stood and Cheddar, we propose a set of AADL design patterns to model usual real time synchronization paradigms [12].
This paper is organized as follows: In section 2, we present performance analysis methods that are expected to be applied on AADL design patterns. These AADL design patterns are then described in section 3. Finally, we conclude and describe ongoing works in section 4.

2. Real-Time performance analysis with AADL

2.1 Real time scheduling analysis and Queueing system analysis

From an AADL model, we can perform performance analysis based on real time scheduling theory and queueing system theory.
Real time scheduling theory helps the system designer to analyse the timing behaviour of a set of tasks with scheduling algorithms or with algebraic methods usually called feasibility tests.
For example, with the well known Liu and Layland real time task model [2], each task periodically performs a treatment. This “periodic” task is defined by three parameters: its deadline (*Di*), its period (*Pi*) and its capacity (*Ci*). *Pi* is a fixed delay between two release times of the task *i*. Each time the task *i* is released, it has to do a job whose execution time is bounded by *Ci* units of time. This job has to be ended before *Di* units of time after the task wake up time.
Some algebraic methods can provide a proof that an architecture will meet its periodic task performance requirements. Scheduling algorithms allow the designer to compute scheduling simulations of the architecture to analyse. Usually, simulations can not

lead to a proof. However, in some cases (with deterministic schedulers and with periodic tasks for example), scheduling simulation may lead to a schedulability proof if the designer is able to compute a scheduling during the base period [3].
Different kinds of feasibility tests exist: tests based on processor utilization factor and tests based on worst case task response time which are designed to check task deadlines [13]; tests based on buffer utilization factor which are designed to check buffer overflow [10].
For example, the worst case response time feasibility test consist in comparing the worst case response time of each task with its deadline. Joseph, Pandia, Audsley *et al.* [13] have proposed a way to compute the worst case response time of a task with pre-emptive fixed priority scheduling by:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{P_j} \right\rceil . C_j \qquad [1]$$

Where R_i is the worst case response time of the task *i*. This feasibility test can be easily extended to take into account task waiting time on shared resources, jitter on task release time, task precedency relationship …

Queueing system theory may be used to perform analysis of real time architecture when waiting times exist. Queueing system theory allows to study performance of a system composed of servers, clients and storage places [14]: people waiting in a room for a doctor, network switch routing data, ...
If clients becomes active while a server is busy, their requests are stored in a queue. By defining the average rate of clients request arrivals and the average rate of requests that the server can handle, a queueing system model allows to predict the average system occupation factor *L*, the average customer waiting time *W*, and the probability P_n of having *n* clients in the queue.

2.2 Investigated performance criteria

An AADL model is a set of hardware and software components. An AADL operational system is a set of process components encompassing thread and data components that are bound to an execution platform composed of processor, memory and bus components. Component relationships are modelled by ports and data access connections. Three different kinds of ports exist: data ports, event ports and event data ports. Data ports represent connection points for transfer of data values. Event ports represent connection points for transfer of control through raised events that can trigger thread dispatch or mode transition. Event data ports represent connection points for transfer of events with data, i.e., messages that may be queued. Data access represent asynchronous read or write operations on a shared data component.

From an AADL model, several performance criteria can be computed with the algebraic methods proposed by the real time scheduling theory and by the queueing systems theory. Some examples of these performance criteria are:

A. The worst case task response times;
B. The bounds on the thread waiting time due to data access;
C. The deadlocks and priority inversions due to data access;
D. The numbers of messages in queued message communication links. This criterion allows memory footprint analysis;
E. The numbers of context switches, pre-emption;
F. The processor utilization factor;
G. …

In the sequel, we only focus on the 4th first criteria.

3. Examples of AADL design patterns: from modelling to analysis

In the next sections we present four design patterns that can be used to express usual inter-thread communications.

For each pattern, an applicative test case is described under the form of an AADL model which has been formatted in purpose to highlight some of the possible performance analysis presented in the previous section. These criteria can not be investigated independently: in the case of the Blackboard design pattern for instance, one must compute first B before computing A.

Modelling pattern	Analysis criteria
Synchronous data-flows	A
Mutex protected data	A, B and C
Blackboard	A, B and C
Queued buffer	A, B, C and D

Other combinations could of course be considered, however the ones that are detailed above correspond to typical patterns that need to be properly managed by any real-time software development environment.

3.1 Synchronous data-flows pattern

Description: This first design pattern is the simplest one. The data sharing is achieved by a clock synchronization of the threads as Meta-H [1] proposed it. In this synchronization schema, thread dispatch is not affected by the inter-thread communications that are expressed by pure data-flows. With this communication pattern, each thread reads its input data ports at dispatch time and writes

its output data ports at complete time. This design pattern does not require the use of a shared data component. In this simple case, the execution platform consists in one processor running a scheduler that is compliant with fixed priority scheduling such as Rate Monotonic [2].

Example: Corresponding modelling case study consists in three periodic threads linked together by two data port connections. The AADL graphical representation of this case study and fragments of its textual specification are given below:

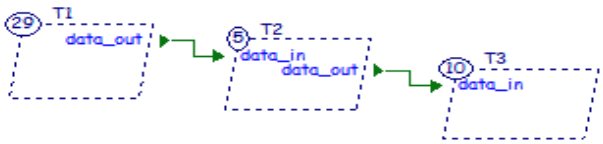

Figure 1: synchronous data-flows pattern

Real-time attributes of each thread can be described individually by a set of pre-defined properties:

```
THREAD IMPLEMENTATION T1.others
PROPERTIES
 Dispatch_Protocol => Periodic;
 Period => 29ms;
 Deadline => 29ms;
 Compute_Execution_Time => 7ms..7ms;
END T1.others;
```

The complete operational system must specify static instantiation of the threads within each process, as well as a description of the execution platform.

```
SYSTEM IMPLEMENTATION sched.others
SUBCOMPONENTS
 rma: PROCESS rma.others;
 cpu: PROCESSOR cpu.others;
PROPERTIES
 Actual_Processor_Binding =>
      REFERENCE cpu APPLIES TO rma;
END sched.others;

PROCESSOR IMPLEMENTATION cpu.others
PROPERTIES
 Scheduling_Protocol =>
      RATE_MONOTONIC_PROTOCOL;
END cpu.others;

PROCESS IMPLEMENTATION rma.others
SUBCOMPONENTS
 T1 : THREAD T1.others;
 T2 : THREAD T2.others;
 T3 : THREAD T3.others;
CONNECTIONS
  DATA PORT T1.data_out ->
T2.data_in;
  DATA PORT T2.data_out ->
T3.data_in;
END rma.others;
```

Analysis: This design pattern leads to a very static scheduling which is difficult to change but which is also very easy to analyse. Indeed, in this design pattern, threads are independent, which allows the use of numerous simple feasibility tests such as the processor utilization factor test or the worst case response time. These simple feasibility tests are available for most usual real time schedulers such a EDF, LLF and fixed priority schedulers. Verifying this design pattern only consists in checking that thread deadlines will be met. Since no shared data component is used in this design pattern to implement data exchange, no thread blocking time on shared resource has to be computed.

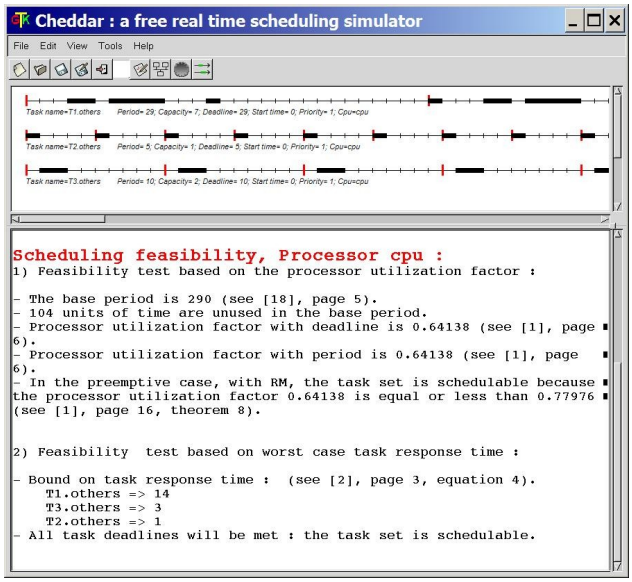

Figure 2: analysis performed by Cheddar

We can note however, that this kind of analysis still depends on the validity of execution time estimates. Figure 2 depicts an analysis of this test case by Cheddar. The top part of the window shows the thread scheduling. The worst case response time computed from this scheduling and with feasibility tests are shown in the bottom part of the window.

3.2 Mutex protected communication pattern.

Main drawback of the previous pattern is its lack of flexibility at run time: each thread will always execute, read and write data at pre-defined times, even if useless. In order to introduce more flexibility at that level, asynchronous inter-thread communications must be considered. An example of such a run-time environment is given by the Ravenscar profile.

Description: Ravenscar is a part of Ada 2005 [5,6]. Ravenscar is a set of Ada program restrictions usually enforced at compilation time, which allows a more flexible design of the architecture and which guaranties that this architecture remains compliant with real time scheduling theory/queueing systems theory analysis assumptions.

Ravenscar is an Ada subset from which one can write applications composed of a set of threads. In Ravenscar, threads access shared data components asynchronously according to priority inheritance protocols. Ravenscar assumes that threads are scheduled with a fixed priority scheduler and that data are accessed with the ICPP protocol [6]. However, it may be possible to apply Ravenscar to some dynamic schedulers such as EDF if a data access protocol exists for such schedulers (eg. PLCP protocol for EDF scheduler [5]). Ravenscar is then a subset of concurrency features which can be defined in many real time executive such as POSIX 1003.1b [21], ARINC 653 [20] and Java-RT [19].

Exemple: To illustrate asynchronous inter-thread communications, the chosen case study shows an implementation of the classical P and V procedures of a Mutex. A complete description of this protocol in AADL requires the use of the Behavior Annex [16] to provide a detailed specification of the internal realisation of the P and V subprograms.

```
SUBPROGRAM IMPLEMENTATION P.others
ANNEX Behavior_Specification {**
 states
  s0: initial state;
  s1: return state;
 transitions:
  busy: s0 -[on me.The_Value=0]->
s1{};
  free: s0-[on me.The_Value=1]-> s1
    { me.The_Value := 0; };
**};
END P.others;

SUBPROGRAM IMPLEMENTATION V.others
ANNEX Behavior_Specification {**
 states
  s: initial return state;
 transitions:
  s -[]-> s { me.The_Value  := 1; };
**};
END V.others;
```

A more complete modelling case consists in two threads sharing the same two mutex protected data components. The AADL graphical representation of this example is shown below:

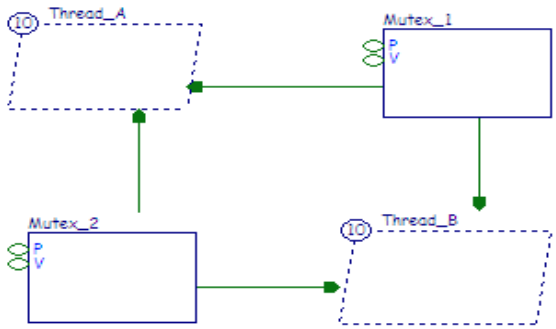

Figure 3: mutex protected communication pattern

The use of the AADL Behavior Annex is required again to express the internal functional structure of the applicative threads. Such a level of details is needed to describe the atomic actions and the critical sections.

```
THREAD IMPLEMENTATION Thread_A.others
PROPERTIES
 Dispatch_Protocol => Periodic;
 Period => 10 ms;
ANNEX Behavior_Specification {**
 states
  s0: initial state;
  s1, s2, s3, s4: state;
  s5: complete state;
 transitions
  acquire_M1: s0-[]->s1
{P!(Mutex_1);};
  acquire_M2: s1-[]->s2
{P!(Mutex_2);};
  critical_section: s2-[]->s3 {…};
  release_M1: s3-[]->s4
{V!(Mutex_1);};
  release_M2: s4-[]->s5
{V!(Mutex_2);};
**};
END Thread_A.others;

THREAD IMPLEMENTATION Thread_B.others
…
ANNEX Behavior_Specification {**
…
 transitions
  acquire_M2: s0-[]->s1
{P!(Mutex_2);};
  acquire_M1: s1-[]->s2
{P!(Mutex_1);};
…
**};
END Thread_B.others;
```

Analysis: With this design pattern, additional analysis such as deadlock detection, can be performed. As threads are not independent anymore, response time analysis becomes more complicated to

investigate. Verifying thread deadline with this design pattern requires to first compute the thread waiting time before computing the worst case thread response time. This waiting time is computed according to the duration of the critical sections and the shared resource access protocol (eg. Priority Ceiling Protocol [22]).

3.3 Blackboard communication pattern

Description: Ravenscar allows a thread to allocate/release several shared resources (eg. AADL data). Real time scheduling theory usually models such a shared resource as a semaphore, to represent, for example, a critical section. In classical operating system, it exists many synchronization design patterns such as critical section, barrier, Readers-Writer, private semaphore, and various Producer-Consummer synchronization design patterns [12]. Programming languages also propose specific synchronization design patterns such as Java synchronized methods [15] or Ada protected types [5]. The Blackboard communication is one of them. The Blackboard design pattern implements a Readers-Writer synchronization protocol. At a given time, only one writer can get the access to the Blackboard in order to update the stored data, as opposed to the readers which are allowed to read the data simultaneously. The usual implementation of this protocol implies that readers and writers do not perform the same semaphore access.

Example: The following case study shows the AADL implementation of a multi-threads application with asynchronous communications through a Blackboard using the Readers-Writer protocol.

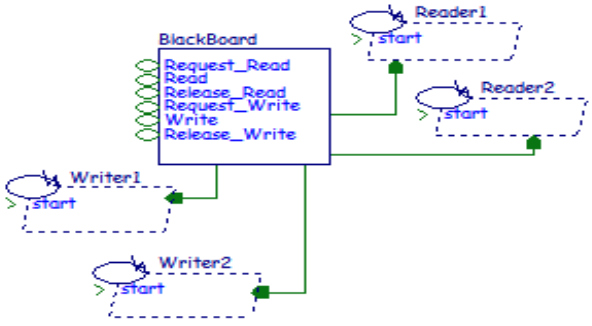

Figure 4: Blackboard communication pattern

The communication object is an instance of an AADL data component that can be described by its provided services, implemented by subprogram features, and its internal structure consisting in a set of state variables.

```
DATA T_BlackBoard
 FEATURES
  Request_Read: SUBPROGRAM Read0.o;
  Read: SUBPROGRAM Read1.o;
  Release_Read: SUBPROGRAM Read2.o;
  Request_Write: SUBPROGRAM Write0.o;
  Write: SUBPROGRAM Write1.o;
  Release_Write: SUBPROGRAM Write2.o;
END T_BlackBoard;

DATA IMPLEMENTATION T_BlackBoard.o
 SUBCOMPONENTS
  Contents: DATA T_Item;
  Readers: DATA Behavior::Integer;
  Is_Idle: DATA Behavior::Boolean;
  Is_Reading: DATA Behavior::Boolean;
  Is_Writing: DATA Behavior::Boolean;
END T_BlackBoard.o;
```

The precise implementation of the Readers-Writer protocol requires a detailed specification of the functional structure of each subprogram, using again the AADL Behavior Annex.

```
SUBPROGRAM IMPLEMENTATION Read0.o
 ANNEX Behavior_Specification {**
  states
   s: initial return state;
  transitions
   s -[on me.Is_Idle
       and me.Readers=0 ]-> s {
    me.Readers := me.Readers + 1;
    me.Is_Reading := true;
    me.Is_Idle := false;
   };
 **};
END Read0.o;
```

Analysis: With this kind of design pattern, we can expect to compute the same performance criteria that we have shown for the Mutex design pattern. However, with complex synchronization design patterns, before computing worst case thread response time, we have first to analyse shared resource blocking time. For example, with the Blackboard design pattern, a thread may get the access to a different number of semaphores depending on its type (eg. reader or writer thread). It means that we have to first evaluate the semaphore access of each thread in order to compute shared data waiting time, for a given critical section and for a given priority inheritance protocol.

3.4 Queued buffer communication pattern

Description: In the Blackboard design pattern, at any time, only the last written message is made available to the threads. Some real time executives (eg. ARINC 653) provide communication features which allow to store all written messages in a memory unit. AADL also propose such a feature with event data ports or shared data components. An event data port

may contain several messages. We assume that buffer messages are handled with a FIFO protocol.

Example: The case of the Producer-Consumer can be used to illustrate the queued buffer communication pattern. Like the Blackboard, the buffer component can be described in AADL by its provided services and its internal variables.

```
DATA IMPLEMENTATION T_Buffer.others
SUBCOMPONENTS
 Stack: DATA T_Item;
 Current: DATA Behavior::Integer;
 Max: DATA Behavior::Integer;
END T_Buffer.others;

SUBPROGRAM Push
FEATURES
 me: IN OUT PARAMETER
T_Buffer.others;
 Item: IN PARAMETER T_Item;
END Push;

SUBPROGRAM IMPLEMENTATION Push.others
ANNEX Behavior_Specification {**
 states
  s: initial return state;
 transitions
  s-[on me.Current < me.Max]->s {
      me.Stack(me.Current) := Item;
      me.Current := me.Current+1;};
**};
END Push.others;

SUBPROGRAM Pop
FEATURES
 me: IN OUT PARAMETER
T_Buffer.others;
 Item: OUT PARAMETER T_Item;
END Pop;

SUBPROGRAM IMPLEMENTATION Pop.others
ANNEX Behavior_Specification {**
 states
  s: initial return state;
 transitions
  s-[on me.Current > 1]->s {
      Item := me.Stack(me.Current);
      me.Current := me.Current-1;};
**};
END Pop.others;
```

The simplest Producer-Consumer test case can be represented in AADL by a sporadic producer thread and a periodic consumer thread connected to the same buffer.

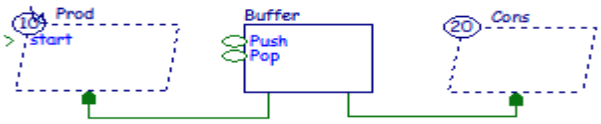

Figure 5: Queued buffer communication pattern

```
THREAD IMPLEMENTATION Prod.others
PROPERTIES
 Dispatch_Protocol => Sporadic;
 Period => 10 ms;
ANNEX Behavior_Specification {**
 state variables
  v: T_Item;
 states
  s: initial complete state;
 transitions
  s-[]->s {Push!(Buffer,v);};
**};
END Prod.others;

THREAD IMPLEMENTATION Cons.others
PROPERTIES
 Dispatch_Protocol => Periodic;
 Period => 20 ms;
ANNEX Behavior_Specification {**
 state variables
  v: T_Item;
 states
  s: initial complete state;
 transitions
  s-[]->s {Pop!(Buffer,v);};
**};
END Cons.others;
```

Analysis: This design pattern needs the same schedulability analysis as the previous one. An other typical analysis for such design pattern is also to estimate the memory footprint of the buffer in order to ensure no loss of data when the rate of the producer thread is temporarily greater than the one of the consumer thread. In [17,18], Legrand et al. have proposed a set of feasibility tests based on queueing system. These feasibility tests were adapted to AADL in [10]. It was shown how perform memory footprint analysis with AADL models containing event data ports. For example, if both producers and consumers are periodic thread, a worst case number of messages in a buffer can be computed with this feasibility test:

L=2.n if threads are harmonic [2]

or

L=2.n+1 otherwise

Where L is the maximum number of messages and n is the number of producers. In the case of sporadic threads, the same worst case analysis can be

performed, but [18] also have proposed feasibility tests for average analysis.

4. Conclusions and perspectives

This article deals with the interoperability between a modelling tool called Stood and an analysis tool called Cheddar. Coupling of modelling and analysis tools requires that both ends strictly comply with the same semantic definition of the exchanged model. For such a purpose, we have chosen AADL as a pivot language between Stood and Cheddar. AADL is a textual and graphical language support for model-based engineering of embedded real time systems. AADL is a flexible language which allows the modelling of both synchronous and asynchronous systems. Then, in order to illustrate interoperability between Stood and Cheddar, we have proposed a set of AADL design patterns. For each design pattern, we have listed a set of performance criteria that can be checked. Each design pattern is also illustrated by an example.

This article presents a first step to achieve interoperability between Stood and Cheddar. The next steps must refine the set of AADL design pattern presented above. Other synchronization patterns must also be investigated (eg. Private semaphore, Ada protected type, …).

5. Acknowledgement

Cheddar AADL analysis features rely on Ocarina. We would like to thank the ENST Ocarina's Team (B. Zalila, J. Hugues, L. Pautet and F. Kordon) [4].

6. References

|1] SAE, Architecture Analysis and Design Language (AADL) AS 5506 ; Technical report, The Engineering Society For Advancing Mobility Land Sea Air and Space, Aerospace Information Report, Version 1.0, November 2004.

[2] C. L. Liu et J. W. Layland : Scheduling algorithms for multiprogramming in a hard real-time environnment ; Journal of the Association for Computing Machinery, vol. 20, n° 1, pp. 46- 61, January, 1973.

[3] J.Y.T. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. Performance Evaluation 2, 237-250 (1982).

[4] J. Hugues, B. Zalila, and L. Pautet. Rapid Prototyping of Distributed Real-Time Embedded Systems Using the AADL and Ocarina. In 18th IEEE/IFIP International Workshop on Rapid System Prototyping (RSP'07), Porto Allegre, Brésil, June 2007.

[5] ISO, Ada reference manual ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1 (Draft 16).

[6] A. Burns and A. Wellings. *Concurrent and Real Time programming in Ada.* 2007. Cambridge University Press.

[7] P. Dissaux, Using AADL for mission critical software development. 2nd European Congress ERTS (Embedded Real Time Software), 21-23 january 2004.

[8] SEI. OSATE : an extensible Source AADL tool environment. SEI AADL team technical report. December 2004.

[9] O. Sokolsky, I. Lee, D. Clake. Schedulability analysis of AADL models. Parallel and Distributed Processing Symposium, IPDPS, 25-29 April 2006.

[10] F. Singhoff, J. Legrand, L. Nana, and L. Marcé.Scheduling and Memory requirement analysis with AADL. ACM Ada Letters journal, 25(4):1-10, ACM Press. Also published in the proceedings of the ACM SIGAda International Conference, Atlanta, 14-17 November, 2005.

[11] TOPCASED web site. http://www.topcased.org

[12] A. Tanenbaum. Modern Operating Systems. Prentice-Hall. 2001.

[13] L. George, N. Rivierre and M. Spuri. Preemptive and Non-Preemptive Real Time Uni-Processor Scheduling. INRIA Research Report number 2966. September 1996.

[14] L. Kleinrock. Computer Applications, Volume 2, Queueing Systems. New York, John Wiley and sons. Wiley-Interscience. 1976.

[15] S. Zakhour, S. Hommel, J. Royal, I. Rabinovitch, T. Risser, M. Hoeber. The Java Tutorial: A Short Course on the Basics, 4th Edition (The Java Series). Addison Wesley. 2001.

[16] AADL committee. AADL Behavior Annex, draft 2.0, September 2007.

[17] J. Legrand. Contribution à l'ordonnancement des systèmes temps réel comprenant des tampons Phd thesis, Université de Bretagne Occidentale. December 2004.

[18] J. Legrand, F. Singhoff, L. Nana, L. Marcé. Performance Analysis of Buffers Shared by Independent Periodic Tasks. LISyC Technical report number legrand-02-2004, January 2004.

[19] Time Sys Corp. Specification: JSR001 Real-time Specification for Java,Version: 1.0.2, June 2006

[20] Arinc Committee, Arinc 653 (Avionics Application Software Standard Interface): January 1997.

[21] B. O. Gallmeister. POSIX 4 : Programming for the Real World. O'Reilly and Associates, January 1995.

[22] Sha, R. Rajkumar and J.P. Lehoczky. Priority Inheritance Protocols : An Approach to Real Time Synchronization. IEEE Transactions on computers, 39(9):1175-1185. 1990.

[23] F. Singhoff, A. Plantec. AADL Modeling and Analysis of Hierarchical Schedulers. ACM SIGAda International Conference, Washington DC (USA)