Investigating the usability of real-time scheduling theory with the Cheddar project : The Cheddar project

Frank Singhoff, Alain Plantec, Pierre Dissaux, Jérôme Legrand

December 18, 2012

Abstract

This article deals with real-time critical systems modelling and verification. Real-time scheduling theory provides algebraic methods and algorithms in order to make timing constraints verifications of these systems. Nevertheless, many industrial projects do not perform analysis with real-time scheduling theory even if demand for use of this theory is large and the industrial application field is wide (avionics, aerospace, automotive, autonomous systems, ...). The *Cheddar* project investigates why real-time scheduling theory is not used and how its usability can be increased. The project was launched at the University of Brest in 2002. In [76], we have presented a short overview of this project. This article is an extended presentation of the *Cheddar* project, its contributions and also its ongoing works.

1 keywords

Scheduling theory, architecture modelling and verification, AADL, software engineering tools

2 Introduction

This article deals with real-time critical systems modelling and verification. Real-time systems have to meet hard timing constraints implied by the environment; safety critical systems failures, including violation of timing constraints, could lead to life losses or environmental damages [55].

Real-time scheduling theory provides algebraic methods and algorithms in order to make timing constraints verifications. Real-time scheduling theory foundations were proposed in 1970 [51] and have led to extensive researches. Since 1990, it allows the analysis of systems composed of periodic tasks sharing resources and running on a single processor [69]. Numerous operating systems

provide features allowing the implementation of such applications. Some standards and compilers also provide tools to enforce compliancy of applications with real-time scheduling theory assumptions (e.g. the Ravenscar profile defined in the Ada 2005 standard [79]).

Real-time scheduling theory has been successfully used in many projects [68]. However, it appears that in many practical cases no such analysis is performed although experience shows that it could be profitable.

We think that several reasons can explain why real-time scheduling analysis is not applied as much as it could be. First, real-time scheduling analysis may be difficult to apply on some kind of architectures. For example, few analytical methods were proposed for distributed systems analysis [82]. The scope of existing analytical method is usually restricted to specific architectures composed of simple schedulers and task models. In the other cases, practical real-time scheduling analysis solutions should at least provide means to properly model the system and run simulations.

Furthermore, we argue that this theory is not so easy to understand and to apply for many engineers. Most of the known analytical methods and algorithms have been elaborated during the last 30 years. These analytical methods provide a way to compute different performance criteria. Each criterion requires that the target system fulfills a set of specific assumptions. Thus, it may be difficult for a designer to choose the relevant analytical method. Unfortunately, there is currently a too limited support provided by design languages and software engineering tools which can help the designer to automatically apply real-time scheduling theory.

This article presents three possible directions, that we have investigated in the context of the *Cheddar* project, in order to increase the usability of real-time scheduling theory. In section 2, we present a set of tools that we have developed. These tools aim at helping the designer to apply real-time scheduling theory on an architectural model. Section 3 depicts how we can use an architecture description language with the *Cheddar* tools, in order to apply the real-time scheduling theory more automatically. In section 4, we propose a domain specific language and a set of tools that can be used on specific real-time architectures when no existing analytical method can be directly applied. Finally, section 5 is devoted to conclusions and presentation of *Cheddar* project future works.

3 Increasing the usability of real-time scheduling theory: easing analysis with flexible tools

Real-time scheduling theory provides scheduling algorithms and algebraic methods usually called feasibility tests. These methods help the system designer to analyze the timing behavior of his architecture. With the Liu and Layland real-time task model [51], each task periodically performs a treatment. This "periodic" task is defined by three parameters: its deadline (D_i), its period (P_i) and its capacity (C_i). P_i 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 C_i units of time. This job has to be ended before D_i units of time after the task release time. From this simple task model, some feasibility tests 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. Usually, simulations can not lead to a proof. However, with deterministic schedulers and periodic tasks, scheduling simulation may be considered as schedulability proof if the designer is able to compute the scheduling during the hyper-period [50].

Different kinds of feasibility tests exist such as tests based on processor utilization factor or tests based on worst case task response time. The worst case response time feasibility test consists in comparing the worst case response time of each task with its deadline. Joseph, Pandia, Audsley et al. [32] have proposed a way to compute the worst case response time of a task with preemptive fixed priority scheduler by:

$$r_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{r_i}{P_j} \right\rceil \cdot C_j \tag{1}$$

Where r_i is the worst case response time of the task i and $hp(i)$ is the set of tasks which have a higher priority level than i. This simple feasibility test must be extended to take into account task waiting time on shared resources, jitter on task release time or task precedence relationships.

In order to apply a feasibility test, the designer must check that his design and his execution platform fulfill all feasibility test assumptions. For example, in the feasibility test based on equation (1), the deadline must be less or equal than the period and all tasks must have the same first release time. Thus, we think that for a designer with a poor knowledge of real-time scheduling theory, verifying an architecture with feasibility tests becomes a difficult task. Indeed, for each part of an architecture, he must (see figure 1):

1. Choose the performance criterion that he would like to check.
2. Find the right model for each entity of his architecture. For example, should he model a function of his architecture as a set of periodic tasks or as a set of sporadic tasks? The designer must select the right abstraction level which decreases the model complexity but takes into account properties required for analysis.
3. Select a feasibility test which is able to compute the criterion chosen in (1) and compliant with the models chosen in (2). For such a purpose, model and feasibility test assumptions have to be compliant.

Of course, in some cases, this work can be quite simple since the studied architecture is simple too. In all other cases, the toolset must provide advanced support to guide the designer according to the level of complexity of the model to be processed.

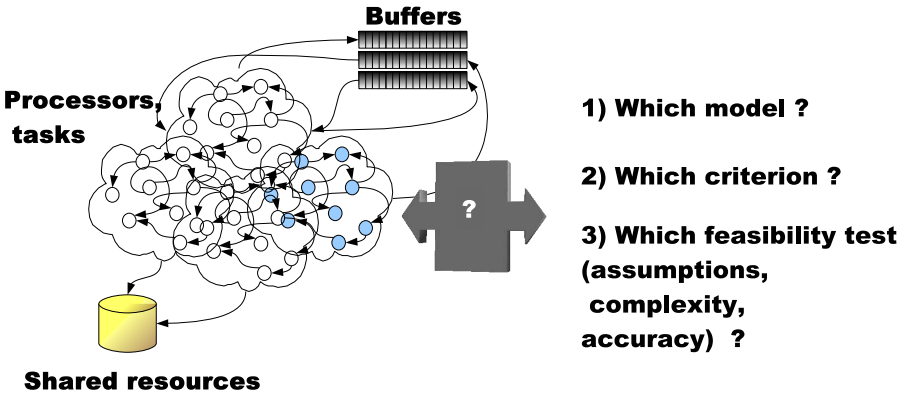

Figure 1: From the architecture modelling to the analysis

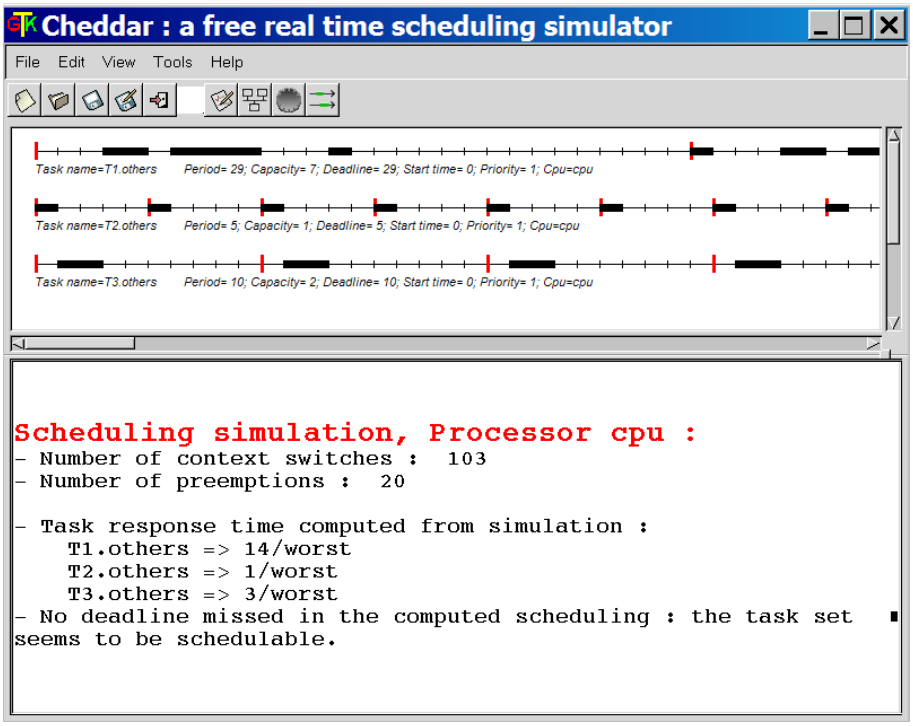

Figure 2: *Cheddar*'s GUI

For such a purpose, we propose a set of tools called *Cheddar*. *Cheddar* is a GPL open-source toolset composed of a graphical editor and a library of processing modules (see figure 2). The toolset is written in Ada. The designer can express his real-time architecture thanks to the *Cheddar* editor. However,

it is also expected that designers perform the modelling activity with separate systems or software engineering tools. The *Cheddar* library implements most of current feasibility tests and classical real-time scheduling algorithms. This library also offers a domain specific language together with an interpreter and a compiler. This specific language is used for the design and the analysis of schedulers that are not implemented into the library.

Of course, numerous other projects have investigated the use of real-time scheduling theory and have developed tools.

First, formal methods have proposed different approaches to perform scheduling analysis of real-time architectures. An architecture composed of a set of tasks sharing resources can be modelled by Petri Nets, timed automata or process algebra. These formal languages lead to the development of various simulators or model-checkers. For example, TIMES [25] is a simulation and a model-checker tool based on time automata while VESTA [77] makes use of process algebra. Several Petri-Net tools can also be used for schedulability analysis: CPN Tools [85], CPN-AMI [35], Tina [12] or PeNSMARTS [34].

The tools described above do not directly implement real-time scheduling theory. In fact, few tools actually implement real-time scheduling theory feasibility tests such as the one shown by equation (1). This is the case of Rapid-RMA [83], Timewiz [81] or MAST [36]. Rapid-RMA or Timewiz are commercial tools which only focus on some specific executive environments. These tools are not freely available to the community. The MAST and *Cheddar* projects have several similarities. Despite MAST does not focus on scheduling simulation with specific scheduler models, as *Cheddar*, this project implements several feasibility tests for different real-time schedulers. The MAST toolset is also freely available. MAST has investigated how real-time scheduling theory can be applied with UML models, and especially with the MARTE-UML profile and the RCM model. MARTE is a UML profile standardized by the OMG which allows timing analysis [28]. RCM is a metamodel based on Ravenscar [79] proposed by [57].

Finally, some projects such as Shark [30], BOSSA [8] or Giotto [38] have investigated how real-time schedulers can be modelled. These projects have proposed a language for the modelling of real-time architectures or schedulers and also some means to implement them. For example, BOSSA has provided a framework which allows to easily plug a scheduler model into an operating system. Shark and Giotto proposed some ways to generate the real-time system from the architecture model. In the *Cheddar* project, we do not intend to generate the source code implementing a scheduler model or an architecture model. However, the Stood [20] and the Ocarina/Poly-Orb teams [39] have proved that architectural models handled by *Cheddar* can be used for such a purpose.

Cheddar offers different use levels depending on the architecture to analyze, depending on the separate modelling tool *Cheddar* is supposed to be connected to or depending on the designer knowledge. We have the following typical use cases:

- First, an architecture model has just to be loaded into the *Cheddar* editor and a button has simply to be pushed to perform proper analysis. In this case, *Cheddar* chooses by itself the best feasibility test to apply, checks if the feasibility test assumptions are met and displays the result. It is assumed that the designer makes use of a design-pattern handled by *Cheddar*. This is the case for instance when the Ravenscar design-pattern is used [79]. This first way to use *Cheddar* is also best suited for students who have to learn about real-time scheduling foundations.
- A second way is to let the designer choose which performance criteria must be computed. This choice can be performed thanks to a customizable menu of the *Cheddar* user interface. Feasibility test assumptions are still automatically checked by *Cheddar*.
- Finally, if scheduling algorithms or feasibility tests implemented into *Cheddar* can not be applied for a given architecture, then the designer must extend the *Cheddar* library. Two ways exist for such a purpose. The library can be extended by the *Cheddar* domain specific language with the process explained in section 5. Otherwise, the designer manually implements the performance analysis tools in Ada. In this case, an advanced knowledge about the *Cheddar* library is required.

Many other ways to use a toolset such as *Cheddar* exist. For example, *Cheddar* can be associated with system or software engineering tools such as Stood [20] or Ocarina [39] in order to increase its usability. In this case, the designer does not use the *Cheddar* graphical editor anymore and the *Cheddar* library is directly called by connected tools. In this case, *Cheddar* exports analysis results as an XML data stream which can be processed back into the modelling tools display.

The next section presents how the use of an architecture description language can facilitate modelling and analysis tool interoperability.

4 Increasing the usability of real-time scheduling theory: from the engineering process to the performance analysis

A possible way to help the designer to apply real-time scheduling theory, is to embed knowledge on this theory into the engineering process through design languages and design-patterns.

Several design languages exist for real-time systems modelling: MARTE-UML [28], EAST-ADL [18] and AADL [66]. The SAE Architecture Analysis and Design Language (AADL) is a textual and a graphical language support for model-based engineering of embedded real-time systems. AADL has been approved and published as SAE Standard AS-5506. AADL is used to design and analyze software and hardware architecture of embedded real-time systems.

In the *Cheddar* project context, AADL was chosen to investigate how real-time scheduling theory can be automatically applied. As *Cheddar* provides the

most known real-time scheduling feasibility tests and scheduling algorithms, it was primarily used to check AADL standard real-time scheduling theory compliance. In a second step, we have investigated how memory footprint analysis can be conducted with AADL [73]. Finally, some design-patterns expressed in AADL were proposed. For each design-pattern, we are investigating the performance criteria that *Cheddar* is able to automatically compute. The use of these design-patterns enforces scheduling analysis compliance. When a designer builds an architecture model with these design-patterns, he can be sure that his model will be analyzed by *Cheddar*. The use of design-patterns also helps interoperability between AADL editor tools and AADL analysis tools [20].

In the sequel, we present the contributions of the *Cheddar* project to the performance analysis of AADL models.

4.1 Investigating AADL suitability for real-time scheduling theory

An AADL model is a set of hardware and software components such as data, threads, processes (architecture software side), processors, devices and busses (architecture hardware side). A data component may represent a data structure in the program source. An AADL data component can be implemented by an Ada tagged record or by a C++ class. A thread is a sequential flow of control that executes a program. A thread can sequentially call subprograms. An AADL thread can be implemented by an Ada task or by a POSIX thread. AADL threads can be released according to several policies: a thread can be periodic, sporadic or aperiodic. An AADL process models an address space. An AADL operational system instantiates a set of process components encompassing thread and data components that are bound to an execution platform composed of processor, memory and bus components.

An AADL model can also contain properties and component connections. Component connections allow the components to share data or exchange messages. Properties are assigned to components. Information provided by component properties can be related to the component behavior, its state, the way it will be implemented or anything else that allows analysis. A property is defined by a name, a value and a type.

```
DATA shared_resource_type
END shared_resource_type;
DATA IMPLEMENTATION shared_resource_type.Impl
   PROPERTIES
      Concurrency_Control_Protocol => Priority_Ceiling_Protocol;
END shared_resource_type.Impl;

THREAD task_type
   FEATURES
      get_access : REQUIRES DATA ACCESS shared_resource_type;
END task_type;
THREAD IMPLEMENTATION task_type.Impl
   PROPERTIES
      Dispatch_Protocol => Periodic;
      Period => 50;
      Compute_Execution_Time => 2 MS .. 2 MS;
      Cheddar_Properties::POSIX_Scheduling_Policy => SCHED_FIFO;
      Cheddar_Properties::Fixed_Priority => 5;
      Cheddar_Properties::Dispatch_Jitter => 10 MS;
END task_type.Impl;

PROCESSOR a_cpu
END a_cpu;
PROCESSOR IMPLEMENTATION a_cpu.Impl
   PROPERTIES
      Scheduling_Protocol => Rate_Monotonic;
      Cheddar_Properties::Scheduler_Quantum => 1 MS;
      Cheddar_Properties::Preemptive_Scheduler => True;
END a_cpu.Impl;

PROCESS a_proc
END a_proc;
PROCESS IMPLEMENTATION a_proc.Impl
   SUBCOMPONENTS
      th1 : THREAD task_type.Impl;
      th2 : THREAD task_type.Impl;
      r1  : DATA shared_resource_type.Impl;
   CONNECTIONS
      DATA ACCESS r1 -> th1.get_access;
      DATA ACCESS r1 -> th2.get_access;
END a_proc.Impl;

SYSTEM a_system
END a_system;
SYSTEM IMPLEMENTATION a_system.Impl
   SUBCOMPONENT
      cpu0  : PROCESSOR a_cpu.Impl;
      proc0 : PROCESS a_proc.Impl;
   PROPERTIES
      Actual_Processor_Binding => REFERENCE cpu0 APPLIES TO proc0;
      Compute_Execution_Time => 3 MS .. 3 MS APPLIES TO proc0.th1, proc0.th2;
END a_system.Impl;
```

Figure 3: Example of an AADL model

Figure 3 shows an AADL model. This model contains a shared resource (called $r1$) accessed by two threads (threads $th1$ and $th2$). The threads and the shared resource are defined into an address space (process $proc0$). The process $proc0$ is bound to a processor called $cpu0$. This figure presents several property examples. These properties allow the designer to express simple component behaviors that can be refined at various levels of description. For instance, the default value of the execution time of a thread (see $Compute\_Execution\_Time$ property) expressed into a *thread* component may be refined when the target processor has been defined in a system instance (in a *system* component definition).

When properties can not be used to express complex component information, designers can extend AADL models with the Behavior Annex. The AADL Behavior Annex is a language which can be used to specify the detailed behavior of AADL components such as threads, in order to perform model checking and advanced code generation [26].

Figure 3 shows an execution time of a thread defined by a property called $Compute\_Execution\_Time$. This execution time is given independently from the target hardware. This is not an elegant way to express this timing property if the description of the thread implementation must be reused for another project with a different hardware. For such a purpose, designers can express a more precise and flexible execution time with the Behavior Annex as we show it in section 4.3.2.

The first release of the AADL standard provides component properties required in order to allow the use of the simplest real-time scheduling analysis methods. Nevertheless, some properties were missing to apply several usual real-time scheduling theory analysis methods. AADL provides a way to extend the AADL standard property sets. Then, we have proposed a set of property extensions [72] to model:

- Usual properties of real-time schedulers in order to define if the scheduler works preemptively or not ($Cheddar\_Properties$::$Preemptive\_Scheduler$ property), if the scheduler makes use of a quantum or runs a POSIX 1003.1b scheduler ($Cheddar\_Properties$::$Scheduler\_Quantum$ and $Cheddar\_Properties$::$POSIX\_Scheduling\_Policy$ properties).

- Usual thread properties such as fixed priority, jitter, offset, first release time, shared resource blocking time, context switch overhead, criticality (e.g. $Cheddar\_Properties$::$Fixed\_Priority$, $Cheddar\_Properties$::$Dispatch\_Jitter$, $Cheddar\_Properties$::$Dispatch\_Offset$ properties).

- Properties to define when shared resources are accessed by threads ($Cheddar\_Properties$::$Critical\_Section$ property).

- And finally, AADL version 1 leading to some ambiguities, we have proposed some properties to express thread precedence relationships which can not be computed from standard AADL connections.

Some of the lacks presented above will be fixed in the next AADL standard release with the Behavior Annex [67, 13] and with some of *Cheddar* properties which was included in the standard AADL version 2 property set.

4.2 Memory footprint analysis with AADL

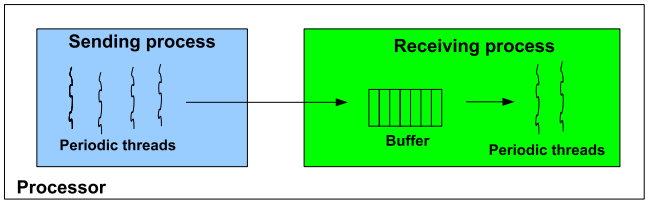

Figure 4: An architecture with threads exchanging messages.

One of the most interesting features of an architecture design language such as AADL is that it allows performance analysis on multiple resources. Figure 4 shows an architecture composed of two processes hosted on a processor. Threads of these processes exchange messages. With such a system, performance analysis on processor and memory unit can not be performed independently:

- On the one hand, if the periodic receiving/sending threads have a high priority level, then they have constant response times. With constant response times, the amount of memory required in the buffer to store messages may be low.
- On the other hand, when sending or/and receiving threads have a low priority level, their response times may vary at execution time. With varying response times, the amount of memory required in the buffer to store messages may be higher if no message has to be lost.

With AADL, thread synchronization and communication are expressed through port abstraction. Ports are logical connection points between components that can be used for control and data transfer between threads. Three kinds of ports exist: *data ports*, *event ports* and *event data ports*. Data ports represent connection points for transfer of state data such as sensor data. Event ports represent connection points for transfer of control through raised events. Events can trigger thread dispatch or mode transition. Event data ports represent connection points for transfer of events with data. With event data ports, messages may be queued before being read by consumer threads.

In the sequel, we present *Cheddar* memory footprint analysis tools which can be applied to AADL threads connected by event data ports. We assume that AADL consumer threads are periodic and AADL producer threads may be aperiodic or periodic. Consumers and producers are both scheduled with real-time schedulers such as Rate Monotonic.

4.2.1 Queueing system theory

The queueing system theory allows the performance analysis of a system composed of servers, customers and storage places [46]: people waiting in a room for a medical doctor, network switch routing data, ... A queueing system models a system composed of a server and a queue. When a customer arrives in the system and if the server is idle, the server immediately performs the customer request. When the server is busy, the customer request is stored in the queue.

By defining average customer arrival rate and average request rate that the server can handle, queueing system theory allows the designer to compute many performance criteria.

Different customer inter-arrival time distributions and service time distributions exist. The most frequently used are Deterministic, Markovian and General distributions:

- A D distribution (D stands for Deterministic) means constant delay between two customer arrivals and constant customer service times.
- A M distribution (M stands for Markovian) describes a customer arrival rate or a service time with an exponential probability distribution.
- Finally, if no assumption on probability distribution is given, the distribution is called G (G stands for General). A General distribution is only defined by an average rate and its variance.

Following Kendall notation, a queueing system is described by at least 3 parameters: $a|b|c$. The a parameter is the customer arrival rate. b describes the service time rate. Finally, c is the number of servers. For instance, a system with one server, a constant service time and an exponential client arrival is a M/D/1 queueing system.

Queue	L	W	P_n
M/M/1	$\frac{\lambda \cdot W_s}{1-\rho}$	$\frac{W_s}{1-\rho}$	$(1-\rho) \cdot \rho^n$
M/G/1	$\lambda \cdot W_s + \frac{\lambda^2 \cdot (W_s^2 + \sigma_s^2)}{2 \cdot (1-\rho)}$	$\frac{\lambda \cdot (W_s^2 + \sigma_s^2)}{2 \cdot (1-\rho)}$	-
M/D/1	$\lambda \cdot W_s + \frac{\lambda^2 \cdot W_s^2}{2 \cdot (1-\rho)}$	$W_s + \frac{\lambda \cdot W_s^2}{2 \cdot (1-\rho)}$	-

Table 1: Main performance criteria

Table 1 gives the main performance criteria that can be computed with most usual queueing systems. In this table, λ represents the customer arrival rate, ρ the queueing system utilization factor, W_s and σ_s^2, respectively, the average customer service time and its variance. With these parameters, one can compute average customer number in the queueing system (L), average customer waiting time in the queueing system (W), and probability of having n customers in the queue (P_n).

4.2.2 Memory requirements with AADL threads connected by event data port

To study event data port memory requirements when connected AADL threads are scheduled according to a real-time scheduler like Rate Monotonic, we propose a new service time distribution: the P distribution. Port buffers are modelled using queueing systems. The P distribution models the fact that periodic consumer/producer threads are scheduled with a real-time scheduler. The P distribution assumes that thread release times are not synchronized with message arrivals. An AADL consumer thread is periodically released even if no message has arrived in the buffer.

Thanks to the P distribution, two new queueing systems are defined: P/P/1 and M/P/1. An exact resolution of P/P/1 is given and we provide an approximation of M/P/1 [48]. This approximation is based on a M/G/1 queueing model.

AADL models with event data ports can then be studied by both worst case and average case analysis. Worst case analysis can be performed if assumptions are made on message arrival rate. In this case, the system is checked with P/P/1 assuming that the highest message arrival rate is known. Otherwise, if no worst case assumption is made, an average analysis can be realized with M/P/1.

4.2.3 Average buffer performance analysis: producer threads are aperiodic

In this section, we assume that messages arrive in the event data port buffer at a random rate. This case occurs when AADL producer threads are aperiodic.

In the sequel, according to the Kendall notation, a buffer receiving random rate messages and accessed by an AADL periodic consumer thread will be modelled with a M/P/1 queueing system [46, 64]. Messages are served in a FIFO manner. The earlier a message arrives, the earlier it is served.

We propose an approximation of the M/P/1 queueing system. This M/P/1 approximation consists in evaluating its average service time W_s and its variance σ_s^2. With W_s and σ_s^2, a M/P/1 queueing system can be modelled with a M/G/1 queueing system. Then, M/P/1 message waiting time and buffer message number can be computed with the following M/G/1 equations [46]:

$$W = W_s + \frac{\lambda \cdot (W_s^2 + \sigma_s^2)}{2 \cdot (1 - \rho)}$$

$$L = \lambda \cdot W_s + \frac{\lambda^2 \cdot (W_s^2 + \sigma_s^2)}{2 \cdot (1 - \rho)}$$

where ρ is the queueing system utilization factor and λ, the message arrival rate.

M/P/1 mean service time and its variance are [48]:

Theorem 1 *The M/P/1 average service time is equal to:*

$$W_s = \frac{P_{cons}}{2} \cdot (1 + \rho) = \frac{P_{cons}}{2 \cdot (1 - \lambda \cdot \frac{P_{cons}}{2})}$$

And average service time variance is:

$$\sigma_s^2 = \rho \cdot \left(\frac{1}{n} \sum_{i=1}^{n} S_i - W_s^2 \right) + (1 - \rho) \cdot \frac{P_{cons}^2}{12}$$

Where $\rho = \lambda \cdot W_s$, S_i is the consumer service time when ρ tends to 1. P_{cons} is the consumer thread period. Due to the fact that the server models a periodic consumer thread, the number of service time S_i is bounded by n.

4.2.4 Worst case buffer analysis: producer threads are periodic

We now study a system where event data port buffer production and consumption are assumed to be periodic. This case occurs when AADL producer threads are periodic.

According to Kendall notation, a buffer shared by n periodic producer AADL threads and 1 periodic consumer AADL thread scheduled with a real-time scheduler can be modelled with a P/P/1 queueing system. Messages are served in a FIFO manner.

Some similarities exist between this system and voice transmission service provided by ATM networks AAL1 layer [29]. In order to solve our P/P/1 queueing system, we apply results from this ATM layer.

For a buffer shared by n periodic producer threads and one periodic consumer thread, buffer bound is given by [49]:

Theorem 2 *For a P/P/1 buffer shared by an harmonic threads set* [1] *and $\forall i$: $D_i \leq P_i$, maximum buffer size and maximum waiting time are respectively:*

$$L = 2 \cdot n$$
$$W = 2 \cdot n \cdot P_{cons}$$

For a non harmonic threads set, maximum buffer size and maximum waiting time are respectively:

$$L = 2 \cdot n + 1$$
$$W = (2 \cdot n + 1) \cdot P_{cons}$$

4.2.5 Ada implementation of AADL memory footprint analysis tools

In order to implement the AADL event data port memory requirement analysis tools, we wrote a set of object oriented Ada packages within *Cheddar* library. These packages allow the designer to compute classical queueing system theory

[1] A thread set is said to be harmonic if and only if each thread period is a positive integer multiple of all smaller thread periods.

criteria (see table 1) for usual queueing systems and also for M/P/1 and P/P/1 (see theorem 1 and 2). Besides of these analytical tools, *Cheddar* library also provides a set of Ada packages in order to design, write and run simulations when no analytical criterion can be computed for a given queueing system.

4.3 Towards interoperability between AADL tools

In the previous sections, we have presented several analytical methods which can be applied on AADL models. These analytical methods are not easy to apply alone and we believe that engineers must be helped by modelling tools. 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 use of AADL standard all along the tool-chain. As an example, *Cheddar* has been well coupled with the modelling tool Stood [20] using AADL as a pivot language.

Stood was chosen because it provides an extended support for AADL in addition to its compliance with the HOOD methodology [14]. Stood allows the designer 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.

4.3.1 A set of AADL design-patterns

To ease the interoperability between Stood and *Cheddar*, we have proposed a set of AADL design-patterns. For each pattern, we are investigating the performance criteria that *Cheddar* is able to automatically compute. This set of design-patterns models usual real-time synchronization/threads-communication paradigms:

1. **Synchronous data-flows design-pattern:** this first design-pattern is the simplest one. The data sharing is achieved by a clock synchronization of the threads as Meta-H [66] proposed it. In this synchronization schema, thread dispatch is not affected by the inter-thread communications that are expressed by pure data-flows. Each thread reads its input data ports at dispatch time and writes its output data ports at completion 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 such as Rate Monotonic [51].

2. **Ravenscar design-pattern:** the main drawback of the previous design-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, asynchronous inter-thread communications can be proposed. An example of such a run-time environment is given by

the Ravenscar profile. Ravenscar is a part of the Ada 2005 standard [79]. It is a set of Ada program restrictions usually enforced at compilation time, which guarantee that the software architecture is real-time scheduling theory compliant. Ravenscar is an Ada subset with which real-time applications are composed of a set of tasks and shared data. Ravenscar assumes that tasks are scheduled with a fixed priority scheduler and that shared data are accessed with ICPP (ICPP stands for Inheritance Ceiling Priority Protocol).

3. **Blackboard design-pattern:** Ravenscar allows a thread to allocate/release several shared resources (e.g. AADL data component). Real-time scheduling theory usually models such a shared resource as a semaphore to represent a critical section for example. In classical operating systems, many synchronization design-patterns exist such as critical section, barrier, readers-writers, private semaphore and various producers-consumers synchronization [80]. The blackboard design-pattern implements a readers-writers 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, then, it requires extra analysis.

4. **Queued buffer design-pattern:** in the blackboard design-pattern, at any time, only the last written message is made available to the threads. Some real-time execution platforms provide communication features which allow all written messages to be store in a memory unit. AADL also proposes such a feature with event data ports or shared data components.

4.3.2 The AADL Behavior Annex

For each pattern, an applicative test case was described under the form of an AADL model. These models have been formatted in purpose to highlight some of the possible performance analysis that *Cheddar* is able to automatically compute (thread worst case response time, bound on shared resource blocking time, memory footprint analysis, ...) [20].

Indeed, performing proper analysis on real-time architectures making use of communication paradigms such as those expressed in the previous section requires that enough details are given about the awaited scheduling conditions and internal behavior of each thread and subprogram.

A first level of details about the real-time behavior of AADL threads is provided by the run-time semantics specified by the core AADL standard. When required, a deeper level of details can be provided by completing the description with the AADL Behavior Annex.

Such a behavior may be implicit. This is the case when the default run-time semantics of the core definition of the AADL is used. For instance, thread scheduling may be controlled at an architectural level by setting the appropriate

value to the predefined AADL property *Scheduling_Protocol*. AADL version 1 run-time handles periodic, sporadic and aperiodic threads. Similarly, interaction with shared data components can be specified thanks to data access features within thread or subprogram declarations. This is the case of the AADL model example of figure 3. The AADL standard defines the effect of such declarations in terms of calls to a default run-time execution service (*Raise_Event*, *Await_Dispatch*, *Get_Resource*, *Release_Resource*, ...). Most of the time, the end user will thus be able to avoid such a low level description of the behavior of the application and rely on the higher level architectural constructs provided by the core of the standard. However, a more explicit description of this behavior may be sometimes required. AADL Behavior Annex is being defined to cover this requirement.

As mentioned previously, the AADL language may be extended by specific property sets (as opposed to the predefined property set specified by the core of the language). This is the simplest extension mechanism which enables the addition of new properties to all the existing constructs of the language. When this is not sufficient, the solution consists in defining an annex to the language.

Annexes to the AADL language can be seen as sublanguages that can be used to provide more detailed descriptions on specialized topics, or to experiment possible future extensions to be included in a later release of the standard. They may define their own syntax that can be embedded inside core AADL statements thanks to an appropriate escape sequence.

In order to cope with fine real-time analysis of architectural models, definition of a Behavior Annex for the AADL is in progress and will be submitted to the standardization process. The origin of the AADL Behavior Annex is the COTRE project [22, 11, 23]. The AADL Behavior Annex provides a sublanguage extension to allow behavior specifications to be attached to AADL components that are expressed as state transition systems with guards and actions [13, 19]. This can be seen as a refinement to call sequences defined by the core of the language. Typical improvements brought by the use of the Behavior Annex are:

- A more precise value for threads capacities.
- A more precise value for critical section durations.
- A more precise definition of thread dispatch conditions.

One first usage of the Behavior Annex extension is to enable expression of conditional actions depending on the actual value of data received in a thread port or a subprogram parameter. This feature is useful to provide a finer expression of the thread capacity, also called worst case execution time and which value is handled by the *Compute_Execution_Time* predefined AADL property (see figure 3).

In figure 5, the example describes a subprogram called *switch* with two different implementations of remote subprogram calls. With the first implementation

```
SUBPROGRAM Switch
   FEATURES
      quick : IN PARAMETER Base_Types::Boolean;
END Switch;

SUBPROGRAM Short_Action
   PROPERTIES
      Compute_Execution_Time => 1 MS .. 2 MS;
END Short_Action;

SUBPROGRAM Long_Action
   PROPERTIES
      Compute_Execution_Time => 100 MS ..150 MS;
END Long_Action;

SUBPROGRAM IMPLEMENTATION Switch.NoBA
   CALLS {
      c1 : SUBPROGRAM Short_Action;
      c2 : SUBPROGRAM Long_Action;
   };
-- Without a Behavior Annex clause, an analysis tool will set the
-- capacity (also called Worst Case Execution Time, or WCET)
-- of Switch to at least 150 ms whereas it only occurs
-- when the parameter value is False.
END Switch.NoBA;


SUBPROGRAM IMPLEMENTATION Switch.BA
ANNEX behaviour_specification {**
   STATES
      s0 : INITIAL STATE;
      S1 : RETURN STATE;
   TRANSITIONS
      s0 -[ ON quick ]-> s1 { Short_Action!; };
      s0 -[ ON NOT quick]-> s1 { Long_Action!; };
**};
-- With a Behavior Annex clause, an analysis tool will set the
-- capacity to a more precise value corresponding to each case.
END Switch.BA;
```

Figure 5: Modelling precise thread capacity

(called $switch.NoBA$) only statements of the core language are used, whereas with the second implementation (called $switch.BA$), a Behavior Annex clause has been added to refine the functional description.

If only the core language is used, the functional behavior description of the subprogram is limited to a call sequence. For the purpose of real-time analysis, only the worst case execution time will be considered. In this case, it will probably be the value of the $Compute\_Execution\_Time$ property associated to the $Long\_Action$ subprogram. On the contrary, if a Behavior Annex clause is added, an analysis tool may be able to allocate a smaller execution time to this subprogram when the parameter is known to be equal to True, thus resulting in a finer real-time analysis.

```
THREAD update
   FEATURES
      buffer : REQUIRES DATA ACCESS T_buffer;
END update;

THREAD IMPLEMENTATION update.NoBA
   CALLS {
      c1 : SUBPROGRAM Long_Computation;
   };
-- Without a B.A. clause, an analysis tool will set the critical section
-- duration to the same value as the complete thread duration.
END update.NoBA;

THREAD IMPLEMENTATION update.BA
ANNEX behaviour_specification {**
   STATE VARIABLES
      v1, v2 : T_buffer;
   STATES
      s0 : INITIAL STATE;
      s1, s2 : STATE;
      s3 : COMPLETE STATE;
   TRANSITIONS
      s0 -[]-> s1 { v1 := buffer; };
      s1 -[]-> s2 { Long_Computation!(v1,v2); };
      s2 -[]-> s3 { buffer := v2; };
**};
-- With a B.A. clause, an analysis tool will isolate the actual
-- critical sections.
END update.BA;
```

Figure 6: Modelling precise shared data access

A second usage of the AADL Behavior Annex deals with a more precise definition of critical sections (see example of figure 6). When the action block of a transition contains an explicit access to a shared data component, then only the corresponding action block will be considered as a critical section. If no Behavior Annex clause is defined, the risk is that an analysis tool evaluates the critical section duration to the complete execution time of the thread.

A third possible use of the Behavior Annex comes from the fact that the evaluation of the transition guards can be used to define a more precise thread dispatch condition. In the example of figure 7, the thread dispatch condition can be specified either by the predefined AADL property *Dispatch_Protocol* or by a Behavior Annex clause. In the first case, it may be understood that the thread can always be dispatched on the arrival of any of its input ports. In the second case, a more precise behavior can be expressed to explicitly show that the receptivity of each port can depend on the internal state of the serving thread, thus changing its actual dispatch conditions.

With these three examples, we have seen how the Behavior Annex can be used to improve the expressiveness of AADL architectures to perform proper real-time analysis. Additional improvements will be brought by AADL version 2

```
THREAD FlipFlop
   FEATURES
      set : IN EVENT PORT;
      reset : IN EVENT PORT;
END FlipFlop;

THREAD IMPLEMENTATION FlipFlop.NoBA
   PROPERTIES
      Dispatch_Protocol => Sporadic;
-- Without a B.A. clause, an analysis tool will consider that
-- the Thread will be dispatched on any event port.
END FlipFlop.NoBA;

THREAD IMPLEMENTATION FlipFlop.BA
ANNEX behavior_specification {**
   STATES
      On, off : INITIAL COMPLETE STATE;
   TRANSITIONS
      on -[set?]-> off { };
      off -[reset?]-> on { };
**};
-- With a B.A. clause, dispatch conditions will vary according to
-- the internal state of the Thread.
END FlipFlop.BA;
```

Figure 7: Modelling thread dispatching rules

which was released at the end of year 2008. These enhancements include in particular new predefined scheduling protocols and a new category of components to manage software components in a distributed system.

5 Increasing the usability of real-time scheduling theory: when no feasibility test exists

In the previous section, we assumed that the architecture to analyze can be designed as a set of design-pattern instances. Since each design-pattern can be analyzed with a set of feasibility tests, the overall architecture can actually be analyzed with such analytical tools.

But some practical architectures can not be modelled as a set of design-pattern instances. Then, analysis based on real-time scheduling theory feasibility tests is thus not applicable. Complex industrial real-time architectures frequently make use of specific task models or schedulers. In this case, no feasibility tests exist and building new ones is a difficult and expensive work. Furthermore, industrial real-time systems may be composed of a large number of entities (e.g. tasks, processors, memory units). For now, these large scale systems can not be efficiently analyzed with model-checking. The only way people can expect to verify performances of these real-time systems is to perform analysis with extensive simulations.

Languages and models were proposed for such a purpose. For example, CPN

tools [85] provides simulation features based on Petri Net. Unfortunately, the use of these general purpose simulation tools usually implies that the designer must model real-time scheduling low level abstractions such as task preemption. A second way is to develop ad-hoc simulation programs, but this solution implies a very low level of reusability. The *Cheddar* library proposes a third way by the use of a domain specific language and a set of tools (compiler, interpreter, code generator). This domain specific language allows the designer to build models of his scheduler and task models.

We also propose an engineering process from which the designer can test his models and automatically generate a simulation program. This model driven engineering process is implemented with the help of a software engineering tool called *Platypus* [58].

5.1 A language for modelling of real-time schedulers

Real-time schedulers are composed of two different parts:

1. Arithmetic and logical statements which allow to select a task among a set of ready tasks or to compute task priorities.

2. Temporal constraints and synchronization between entities (e.g. tasks and schedulers). These synchronizations describe how entities must work all together in order to share processors.

The *Cheddar* language is then defined by two parts: 1) an Ada like language for the modelling of arithmetic and logical statements of the schedulers and 2) a timed automaton language for the synchronizations that model scheduler and task relationships. A detailed description of this language is given in the *Cheddar* User's guide [71].

5.1.1 Modelling arithmetic and logical statements

This part of the *Cheddar* language allows to express the arithmetic and logical statements on simulation data. Simulation data are associated to the entities composing the architecture to analyze (e.g. task release time, scheduler quantum, shared resource protocol). This language allows the designer to express sort rules as Earliest Deadline for example. A *Cheddar* program is organized in subprograms called sections. These subprograms are typed:

- Some subprograms are devoted to data simulation declaration and initialization. They are called *start_section*.

- Some subprograms allow to select a task among a set of ready tasks according to simulation data (e.g. priority). These subprograms are called *election_section*.

- Finally, some subprograms contain statements which have to be run at each unit of time before the task selection. They are called *priority_section*.

The language defines usual operators and statements. Schedulers can be modelled with loops, conditional tests or assignments. This domain specific language also provides statements and operators that are specific to real-time scheduling theory. For example, the $uniform/exponential$ statements customize the way random values are generated during simulations ; the lcm operator computes least common multiplier of simulation data ; the max_to_index operator looks for the ready task which has the highest priority level.

The language is typed and provides usual types as integer, boolean or string. Some types related to real-time scheduling theory are also defined.

5.1.2 Modelling timing and synchronization relationships

The second part of a *Cheddar* program is a network of timed automata. A *Cheddar* scheduler model can contain timed automata similar to those proposed by UPPAAL [4, 9]. UPPAAL is a toolbox for the modelling and the verification of real-time systems.

Timed automata are frequently used to express timing and synchronization requirements of real-time systems. There are some experiments to model and verify real-time schedulers with timed automata [3, 78, 44]. Numerous tools exist (editors, simulators and model-checkers such as UPPAAL [9] or Esterel Studio [10]) and some standards are also based on such a formal model (e.g. UML Statecharts [21]).

A network of timed automata models timing and synchronization between schedulers and tasks. The Ada like language described above is enough to model schedulers which have fixed synchronization relationships between tasks and schedulers. By the past, we have shown that this language allows the modelling of simple schedulers like Earliest Deadline First, Rate Monotonic or Maximum Urgency First. However, some real-time schedulers require the modelling of complex synchronizations. This is the case of hierarchical schedulers for example [2].

In the context of the *Cheddar* language, every automaton may fire a transition separately or synchronize with another automaton. Transitions may be guarded with time constraints. Delays can express time consumption at transition firing. Finally, at transition firing, automata may run Ada like subprograms in order to compute task priorities or to choose the next task to run.

The next section describes an example of *Cheddar* program which models an ARINC 653 hierarchical scheduling [74].

5.2 Cheddar program examples: an illustration with the hierarchical ARINC scheduler

Hierarchical scheduling has been initially proposed in the context of time sharing systems. In time sharing systems, hierarchical schedulers were proposed in order to define user-level scheduling policies (e.g. fair process scheduling [45]

[2] An architecture based on hierarchical scheduling is an architecture in which several entities work all together for the processor sharing.

or user-level and kernel-level threads scheduling into Solaris [84]). If user-level scheduling capability stays a motivation for the use of hierarchical schedulers, system designers mostly focus on hierarchical scheduling in order to reduce system design cost and to increase the sharing resource efficiency. Today, it is usual to share a processor between several applications. This allows old applications to run efficiently on newer processors without being re-designed (e.g. re-design of the scheduling). Applications sharing a processor can have different resource requirements. For example, in a real-time multimedia application, a given scheduler may support critical tasks for audio and video presentation while uncritical tasks can be managed by a different scheduler which does not provide deterministic task response time. Today, hierarchical scheduling also exists in several real-time system standards such as ARINC 653, POSIX 1003 or Ada 2005 [31, 79, 6].

From a performance point of view, real-time hierarchical schedulers raise two difficult challenges.

The first challenge is related to the large number of hierarchical schedulers that were proposed. These hierarchical schedulers have complex and different ways to perform communication and synchronization relationships between the schedulers and the tasks [5]. It is also difficult to express scheduler requirements and behaviors in order to combine them for example [47, 62]. Then, in contrary to the usual schedulers such as Earliest Deadline First or Rate Monotonic, it is difficult to implement into the *Cheddar* library a set of hierarchical schedulers satisfying most of system designer needs.

The second challenge is related to the availability of analytical methods for hierarchical schedulers. Currently few feasibility tests exist in real-time hierarchical scheduling context [2, 37, 70, 17]. Building feasibility tests is usually a difficult work and it is more complex for hierarchical schedulers.

ARINC 653 is an avionic standard which provides such a hierarchical scheduling. ARINC 653 is a specification for an application executive used for integrating avionics systems on modern aircraft. An ARINC 653 system is a set of partitions. Each partition represents a separate application and makes use of memory space that is dedicated to it. Several tasks can be run within a partition and ARINC 653 provides communication services responsible for communication between tasks residing in the same partition.

With an ARINC 653 system, the processor sharing is made according to a two-level hierarchical scheduling:

1. Partitions are cyclically activated. A task can be run only when its partition is activated. This first level of scheduling is fixed at design time. It is usually statically computed.

2. The second scheduling level is related to the task scheduling. Tasks of a given partition are scheduled all together with a fixed priority scheduler. This task scheduling is an online scheduling.

With *Cheddar*, an ARINC 653 hierarchical scheduler is modelled as a set of *Cheddar* programs. A textual representation of those programs is given in the

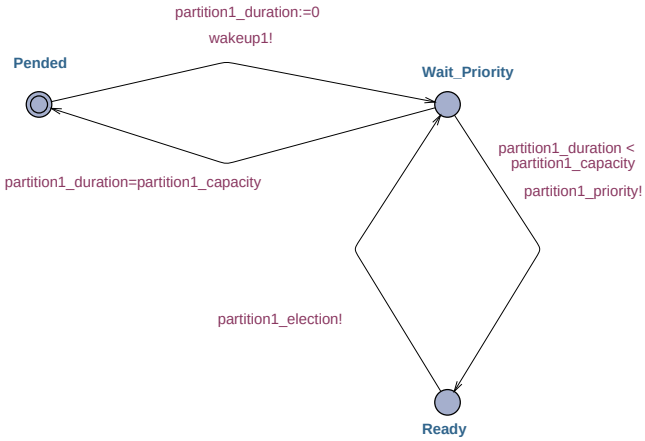

Figure 8: Automaton modelling the ARINC 653 task scheduler of the partition 1

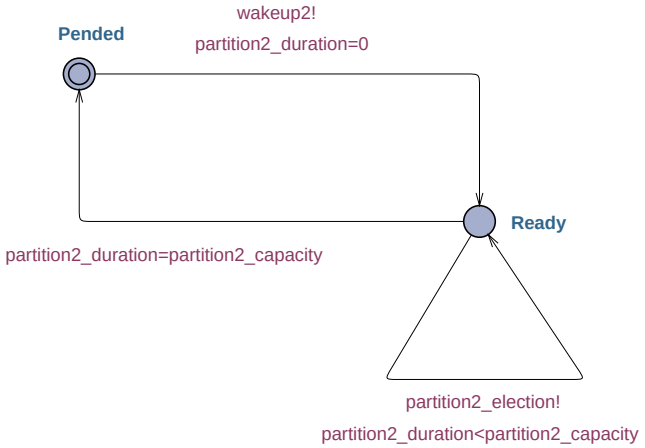

Figure 9: Automaton modelling the ARINC 653 task scheduler of the partition 2

Annex A.1, A.2 and A.3. A graphical representation is also given in figures 8, 9 and 10.

The automaton of the figure 10 specifies when each partition has to be active or not. This automaton models the first level of an ARINC 653 scheduling: the partition scheduling. The automata of the figures 8 and 9 model the schedulers of each partition. Such schedulers are responsible for the tasks scheduling of the modelled ARINC partition. These automata model the second level of an ARINC 653 scheduling: the task scheduling.

The automata modelling the task scheduling of each partition have three types of location:

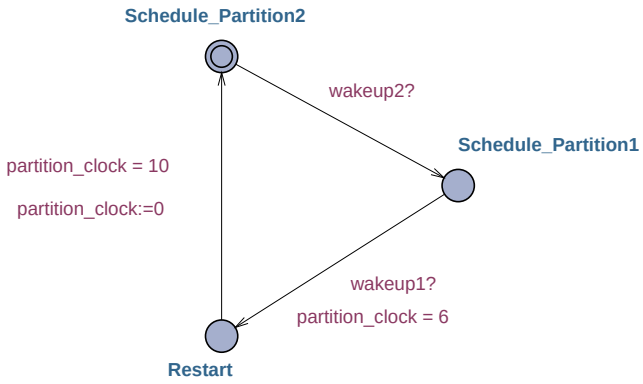

Figure 10: Automaton modelling the ARINC 653 partition scheduler

1. The *Pended* locations. From these locations, the partitions can not get access to the processor in order to run one of their tasks.

2. The *Wait_Priority* locations. *Wait_Priority* is an intermediate location from which the scheduler computes task priorities. Task priorities are computed by the *partition*1*_priority* subprogram during the firing of the *Wait_Priority* outgoing transition (see the *Cheddar* program of Annex A.1).

3. The *Ready* locations. These last locations choose the task to run during the next unit of time. To find the next task to run, the *Cheddar* program interpreter calls the *partition*1*_election* subprogram during the firing of the *Ready* outgoing transition.

The partition scheduler automaton (see figure 10) models the cyclic partition activation. In this example, the partition scheduling is made on a 10 units of time cycle. Each cycle, the partition 2 is activated during the 6 first units of time and the partition 1 is activated during the 4 last units of time. We assume that the partition 2 schedules critical periodic tasks according to Rate Monotonic whereas the partition 1 schedules a set of uncritical tasks according to a round-robin scheduler. The partition scheduler enforces timing isolation between the two partitions which have different processor resource requirements.

5.3 From Cheddar programs to scheduling analysis

Scheduling simulation consists in predicting for each unit of time, the task to which the processor should be allocated. Checking if tasks meet their deadlines can be performed by analysis of the computed scheduling. When an architecture only contains periodic tasks and schedulers such as Rate Monotonic, scheduling simulations leads to a proof whether tasks will meet their deadline. For such a proof, the designer has to run the scheduling simulation during a time interval

called the hyper-period (also called schedule length, base period, major cycle or scheduling period). If the architecture is composed of periodic tasks which have the same first release time, this hyper-period can be computed by [50, 16]:

$$[0, LCM(\forall i : P_i)] \tag{2}$$

where P_i is the period of the task i and LCM is the least common multiplier of all task periods of the system. If the system designer runs a scheduling simulation from the time 0 to the time $LCM(\forall i : P_i)$ and if no task deadlines are missed during such a hyper-period, then, no deadline will be missed during all the task scheduling. With equation 2, we assume that the initial critical instant is used : all tasks start simultaneously at time 0.

From a *Cheddar* program that models a hierarchical scheduler, we generate Ada packages. These packages are part of the *Cheddar* library and allow the designer to run these scheduling simulations. In the following, we describe how such Ada packages are generated.

5.4 Increasing the usability of real-time scheduling theory: Cheddar as an adaptable toolset

Real world problems often need very specific tools to be implemented. In the field of real-time scheduling simulation tools, one very critical part is the scheduling simulation engine. This simulator can heavily change from one application to another and more, from one version to another version of the same system.

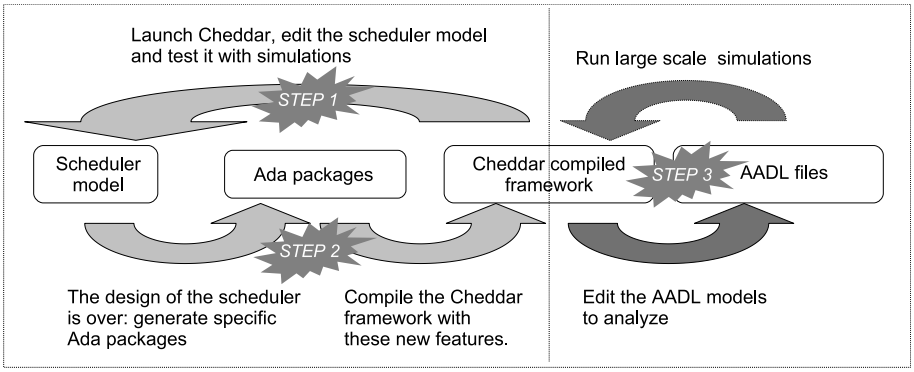

Figure 11: A process to perform simulations from *Cheddar* scheduler models

A very important point for a toolset to be really useable is to allow such adaptation. With *Cheddar*, very specific schedulers can be implemented following the three steps process globally depicted by figure 11:

1. **Specification of the new scheduler**: the designer models a new scheduler with the *Cheddar* language. The resulting model can be directly interpreted using the *Cheddar* library. *Cheddar* toolset allows the designer to perform small scheduling simulations by running its high level specification using *Cheddar* language interpreter. This first running step gives the designer the opportunity to detect critical problems and to review its design very early in the life cycle.

2. **Scheduler integration**: the new scheduler can be integrated into *Cheddar* library as a specific Ada component that is automatically generated from its specification. The *Cheddar* library is then compiled in order to enrich it with this new scheduler.

3. **Scheduler using**: generated scheduler can be used as a built-in one. The designer makes use of his scheduler through this enriched *Cheddar* library in the same way he will make use of standard schedulers manually implemented into *Cheddar* (e.g. Rate Monotonic). Then more tests can take place because he can actually run large scale simulations.

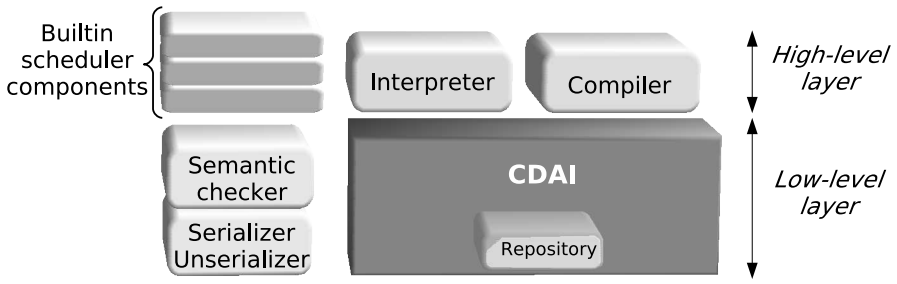

Figure 12: Architecture of the Cheddar framework

As shown by figure 12, the *Cheddar* library is made of six main components and the overall architecture is made of two layers:

- The low-level layer is dedicated to data management and directly depends on manipulated data types and constraints. This layer is highly reusable and evolutive.

- The high-level layer is *Cheddar* domain specific. However, it is made to be evolutive thanks to a model driven engineering process which is able to automatically build a significant part of the layer components.

5.4.1 The low-level layer

It is built around a repository for data and meta-data storage. Data and meta-data reading and writting from the repository are all implemented by

the *Cheddar* Data Acces Interface (*CDAI*). The *CDAI* is a central component that is used by all other *Cheddar* components.

The low-level layer implements some additional data specific components such as a data checker and a data exchange component which is responsible for the creating of XML data files.

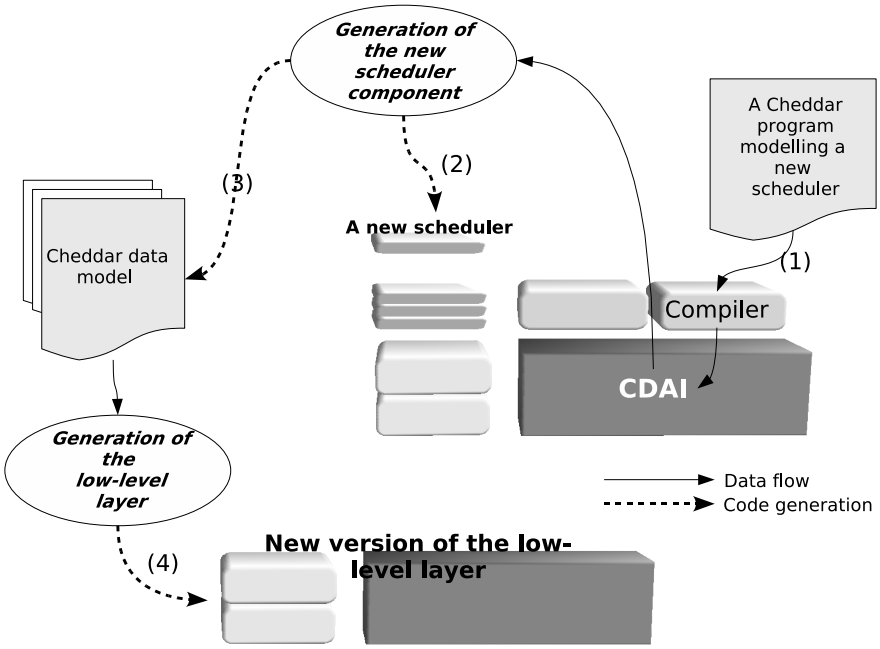

Figure 13: Data flow for the generation of a new *Cheddar* version

5.4.2 The high-level layer

It allows real-time systems simulation at two levels:

1. *Cheddar* natively implements several well known scheduling algorithms. These algorithms can be directly used together with a user defined tasks model. Related components are built-in schedulers. Each scheduler is implementing one and only one algorithm and is made of a specific Ada package.

2. When no built-in scheduler matches the user need, an application specific scheduler can be implemented with the help of the *Cheddar* language. From a *Cheddar* program, an internal representation is built and stored in the repository by the compiler. The *Cheddar* interpreter is then able to run it. An interpreting process runs statements and expressions stored

as meta-data and interacts with *Cheddar* library for data values reading and writing.

5.4.3 Cheddar model driven engineering process

Real-time scheduling simulation can be very time consuming. *Cheddar* language and its interpreter are very useful for new schedulers prototyping. However, in the context of real world application, the interpreter may be too slow to compute simulations. In order to minimize time consuming simulations, a solution consists in implementing a new built-in scheduler into the *Cheddar* Ada framework, which leads to produce a specific version of *Cheddar*. Thank to a model driven engineering process, the *Cheddar* tool makes it possible to automatically produce such a specific *Cheddar* version from an arbitrary *Cheddar* program. This four steps process is depicted by figure 13:

- As for the interpreting process, the input *Cheddar* program is analyzed by the compiler. Produced meta-data are stored into the repository (step 1).
- The scheduler generator makes use of these meta-data and subsequent generating process is twofold:
  - From the automaton specification, a new Ada package is produced (step 2). This Ada package is specifically implementing the user defined scheduler.
  - The *Cheddar* data model is enriched with the new scheduler data types definitions (step 3). This new version of the *Cheddar* data model is used by a second code generator (step 4) which is responsible for the production of the low-level *Cheddar* layer Ada components.

This process is using code generation at two levels of abstraction which are depicted by figure 14. The first level is *Cheddar* library level. This level is related to a particular *Cheddar* library version for which all handled object types (processors, tasks, buffers) are fixed and described by the *Cheddar* data model. A substantial part of the low-level layer tools and especially the *CDAI* is automatically generated from the *Cheddar* data model [59]. The second level is the *Cheddar* language level. It corresponds to *Cheddar* specializations driven by the specification of new schedulers and task models. These new parts are specified using the *Cheddar* programming language. The dedicated code generator is able to parse a *Cheddar* program, to produce a user-defined scheduler implementation and to enrich the *Cheddar* data model. Then, low-level layer packages are regenerated and a new *Cheddar* library version can be compiled [75].

Models, meta-models and code generators are specified and implemented using the software engineering tool *Platypus* [58]. This tool is using the STEP technology and especially the ISO 10303 data modelling language *EXPRESS* [40, 41]: the *Cheddar* data model, the *Cheddar* language meta model and related code generators are all modelled using the *EXPRESS* language (see Annex B).

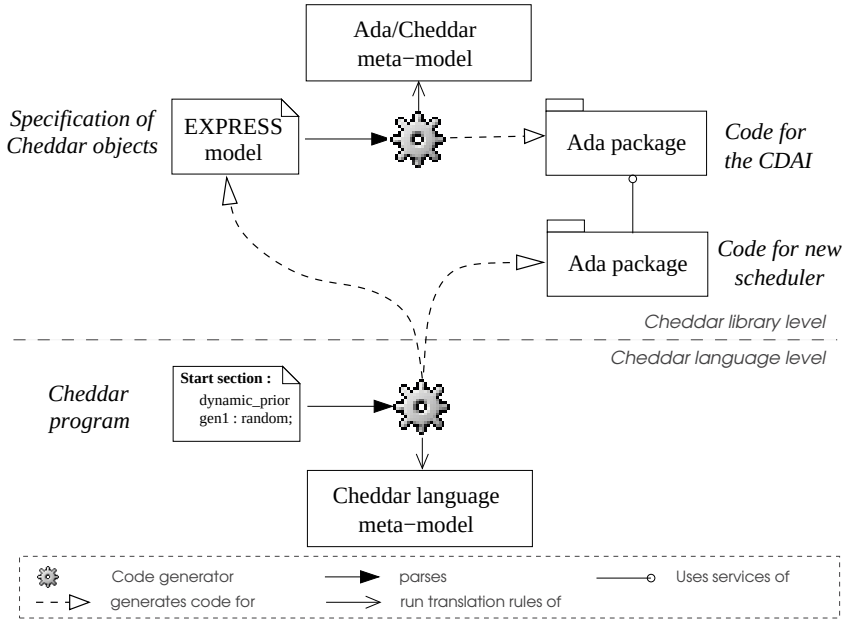

Figure 14: Generation of *Cheddar* code from models and meta-models

6 Conclusion and future work

This article presents three possible directions that are being explored by the *Cheddar* project in order to increase real-time scheduling theory usability. We have presented a set of open-source tools which help the designer to automatically apply this theory to concrete cases. This toolset allows several levels of use and is able to perform analysis of models written with standardized design languages. At this time, the *Cheddar* project has been focused on AADL, an architecture language which aims at modelling and analyzing system and software real-time architectures. We also have presented a domain specific language. It can be used to investigate performances of architectures for which existing real-time scheduling theory does not propose proper analytical methods. The *Cheddar* toolset can be downloaded from http://beru.univ-brest.fr/~singhoff/cheddar.

At the time this article is written, it is difficult to state if *Cheddar* has actually helped people to apply real-time scheduling theory on practical cases. Nevertheless, some real-world applications exist. For example, Airbus Industries has modelled and verified a multi-processor architecture with the *Cheddar* language. The model describes a flight simulator of an Airbus Aircraft [15]. In the same way, Thales has shown that *Cheddar* was able to model and verify a robot architecture [52]. We also have experimented the use of *Cheddar* for the analysis of a supervisor implemented with RTAI (Real-Time Application Interface for Linux). This supervisor was in charge of an automated assembly

line of the Autoliv company [61, 60].

The project has also led to some significant contributions and returns of experience:

- The *Cheddar* project raised an interesting opportunity to make a survey of the feasibility tests proposed by the real-time scheduling theory. Some missing feasibility tests were thus identified.

- New feasibility tests based on the queueing system theory have been proposed. Two queue types called P/P/1 and M/P/1 have been identified from which feasibility tests can be built. These feasibility tests enable memory footprint analysis of real-time architectures and can be also applied on distributed systems.

- By experimenting automatic scheduling verification from AADL models, suitability of AADL version 1 for such a purpose has been investigated. Some extensions to the default AADL properties set were proposed and was integrated into the ADDL standard version 2. Some usual inter-task communication or synchronization paradigms have also been modelled in AADL. These paradigms enable interoperability between modelling tools such as Stood and analysis tools such as *Cheddar*. This experiment can be seen as a proof that AADL can be actually used as an efficient pivot language to build model driven engineering tool chains.

- *Cheddar* has also been used for the same purpose with other modelling languages such as MARTE/UML [56] or PPOOA [24]. Thales RT has developed a set of Eclipse plugins which allows scheduling analysis with *Cheddar* on MARTE/UML models edited with IBM Rational Software Architect [52]. Similarly, PPOOA is a modelling framework based on UML with a customized profile for pipe-line architecture which has been proposed by the University of Madrid. In [24], the University of Madrid has described how PPOOA/UML can be used with *Cheddar*.

- Some other experiments have proved that the specific design language proposed to model new real-time schedulers is well suited for this purpose. We have shown how to use it in order to model ARINC 653 schedulers [74] and sporadic tasks [75]. Several other real-time schedulers have been verified with this language. For example, Ahn et al. have designed and verified with *Cheddar* a real-time multimedia scheduler [1]. In the same way, Muhuri and al. [53] have verified a fuzzy real-time scheduler.

- Furthermore, the *Cheddar* project has also contributed in many research projects related to model driven engineering or real-time system performance analysis [33, 27, 39, 65, 63, 54]. For example, in the context of the european IST ASSERT project, [39] have shown that *Cheddar* can be used in a process in which a real-time system is modelled with AADL and verified by *Cheddar* or CPN-AMI [35]. The verified AADL model

can actually lead to the generation of Ada components that implement the expected real-time system.

- Finally, several universities and engineering schools have built real-time scheduling courses with the help of *Cheddar*: Télécom Paris-Tech, University of Rhodes Island, University of Monash, Universitat Politecnica de Catalunya, University of the West Indies, ... Engineering curriculum with real-time scheduling courses may help people to make use of such analysis method.

These first encouraging experiments may lead to numerous future works. First, Ellidiss Technologies will distribute *Cheddar* with its modelling tool Stood. We expect to experiment the use of the real-time scheduling tools in more industrial contexts. For such a purpose, more operational modelling design-patterns will have to be studied and implemented consistently in Stood for modelling concerns and in *Cheddar* for analysis purposes [20].

Second, the *Cheddar* language that has been defined to model schedulers was experimented in several projects. Today, we know that this language is well suited for this purpose. The language is based on an Ada like language, which allows static analysis (e.g. SPARK [7]) and on a timed automaton language which allows dynamic analysis (e.g. model-checking). We plan to investigate how *Cheddar* scheduler model analysis can help designers to compare their models.

Finally, the complexity of real-time systems has been growing quickly for these 15 last years. In the past, the only kind of resource requiring deep and accurate analysis was the processor. But today many real-time systems are distributed over several processors. It means that several resources have to be managed all together: processors, communication networks and memory units. The work we have done on memory footprint analysis with queueing system models must be extended to cope with distributed system analysis. Some new feasibility tests from P/P/1 or M/P/1 queueing system must also be defined.

7 acknowledgements

Cheddar is an open-source toolset and many people have helped the *Cheddar* team. The *Cheddar* team would like to thank all contributors (see http://beru.univ-brest.fr/~singhoff/cheddar/). *Cheddar* AADL analysis features rely on Ocarina [39]. We would like to thank the Ocarina's Team (B. Zalila, J. Hugues, L. Pautet and F. Kordon). We also would like to thank reviewers of this article who sent us numerous corrections and advices.

References

[1] Ahn, B., Kim, J., Lee, D., Lee, S.: A Real Time Scheduling Method for Embedded Multimedia Applications. Proceedings of the 2006 International

Conference on Pervasive Systems and Computing, Las Vegas, Nevada, USA, June 26-29, CSREA Press, Pages 104-107 (2006)

[2] Almeida, L., Pedreiras, P.: Scheduling within Temporal Partitions : response-time analysis and server design. Proceedings of the EMSOFT'04 conference. September 27-29, Pisa, Italy pp. 95–103 (2004)

[3] Altisen, K., Gossler, G., Sifakis, J.: Scheduler Modeling Based on the Controller Synthesis Paradigm. Real Time Systems journal **23**(1), 55–84 (2002)

[4] Alur, R., Dill, D.L.: Automata for modeling real time systems. Proc. of Int. Colloquium on Algorithms, Languages and Programming, Vol 443, LNCS, pages 322–335 (1990)

[5] Anderson, T.E., Bershad, B.N., Lazowska, E.D., Levy, H.M.: Scheduler activations: Effective kernel support for the user-level management of parallelism. ACM Transactions on Computer Systems **10**(1), 53–79 (1992)

[6] Arinc: Avionics Application Software Standard Interface. The Arinc Committee (1997)

[7] Barnes, J.: High integrity software: The Spark approach to safety and security. Addison-Wesley Publishing Company (2003)

[8] Barreto, J., Muller, G.: Bossa: a Language-based Approach for the Design of Real Time Schedulers . In RTS'2002, pp. 19-31, Paris, France (2002)

[9] Behrmann, G., David, A., Larsen, K.G.: A Tutorial on UPPAAL. Technical Report Updated the 17th November 2004, Department of Computer Science, Aalbord University, Denmark (2004)

[10] Berry, G.: Getting Started with Esterel Studio 5.3. Tech. rep., Esterel technologies SA. Available from http://www.esterel-technologies.com/technology/getting-started/ (2005)

[11] Berthomieu, B., Ribet, P.O., Vernadat, F., Bernartt, J., Farines, J., Bodeveix, J., Farail, P., Filali, M., Padiou, G., Michel, P., Farail, P., Gaufillet, P., Dissaux, P., Lambert, J.: Towards the verification of real time systems in avionics : the COTRE approach. Electronic notes in Theorical Computer Sciences ENTCS, volume 80 (2003)

[12] Berthomieu, B., Vernadat, F.: Time Petri Nets Analysis with TINA. In Proceedings of 3rd Int. Conf. on The Quantitative Evaluation of Systems (QEST 2006), IEEE Computer Society (2006)

[13] Bodeveix, J., Dissaux, P., Filali, M., Gaufillet, P., Vernadat, F.: Behavioural descriptions in architecture description languages, application to AADL. Proceedings of ERTS conference, Toulouse (2006)

[14] Burns, A., Wellings, A.: HRT-HOOD: A Design Method for Hard Real-time Systems. Real Time Systems journal **6**(1), 73–114 (1994)

[15] Castor, M., Casteres, J., Gasmi, F.: Modélisation et simulation de l'Architecture des simulateurs avion pour la mesure de performance. Rapport technique Airbus (2007)

[16] Cottet, F., Delacroix, J., Kaiser, C., Mammeri, Z.: Scheduling in Real Time Systems. John Wiley and Sons Ltd editors (2002)

[17] Davis, R.I., Burns, A.: Hierarchical Fixed Priority Pre-Emptive Scheduling. In the 26th IEEE International Real-Time Systems Symposium (RTSS'05). Miami, Florida, USA. pp. 389–398 (2005)

[18] Debruyne, V., Simonot-Lion, F., Trinquet7, Y.: EAST-ADL - An Architecture Description Language. pp. 181–195. Book on Architecture Description Languages, IFIP International Federation for Information Processing, Springer Verlag, volume 176 (2005)

[19] Dissaux, P., Bodeveix, J., Filali, M., Gaufillet, P., Vernadat, F.: AADL behavioral annex. Proceedings of DASIA conference, Berlin (2006)

[20] Dissaux, P., Singhoff, F.: Stood and Cheddar : AADL as a Pivot Language for Analysing Performances of Real Time Architectures. Proceedings of the European Real Time System conference. Toulouse, France (2008)

[21] Drusinsky, D.: Modeling and Verification using UML StateCharts. Elsevier inc. editor (2006)

[22] Farail, P., Dissaux, P.: COTRE a new approach for modelling real-time software for avionics. Proceedings of DASIA conference, Dublin (2002)

[23] Farail, P., Gaufillet, P.: COTRE as an AADL profile. IFIP TC-2 Workshop on Architectural Description Languages, LNCS volume 175, pp 167-179 (2005)

[24] Fernandez, J.L., Marmol, G.: An Effective Collaboration of a Modeling Tool and a Simulation and Evaluation Framework. 18 th Annual International Symposium, INCOSE 2008. Systems Engineering for the Planet. The Netherlands. 15-19 June 2008. (2008)

[25] Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: Schedulability analysis of fixed-priority systems using timed automata. Theoretical Computer Science **354**(2), 301–317 (2006)

[26] Frana, R.B., Bodeveix, J.P., Filali, M., Rolland, J.F.: The AADL behaviour annex – experiments and roadmap. pp. 377 – 382. 12th IEEE International Conference on Engineering Complex Computer Systems (2007)

[27] Frana, R.B., Rolland, J., Amine, M.F., Bodeveix, J., Chemouil, D.: Assessment of the AADL Behavioral Annex. Journées FAC'2007, Formalisation des Activités Concurrentes (2007)

[28] Frédéric, T., Gérard, S., Delatour, J.: Towards an UML 2.0 profile for real-time execution platform modeling. Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS 06), Work in progress session (2006)

[29] Gagnaire, M., Kofman, D.: Réseaux Haut Débit : réseaux ATM, réseaux locaux, réseaux tout-optiques. Masson-Inter Editions, Collection IIA (1996)

[30] Gai, J., Abeni, L., Giogi, M., Buttazzo, G.: A New Kernel Approach for Modular Real-Time systems Development. Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems, Delft, The Netherlands (2001)

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

[32] George, L., Rivierre, N., Spuri, M.: Preemptive and Non-Preemptive Real-time Uni-processor Scheduling. INRIA Technical report number 2966 (1996)

[33] Gilles, O., Hugues, J.: Applying WCET analysis at architectural level. pp. 113–122. Workshop on Worst-Case Execution Time (WCET'08). Prague, Czech Republic (2008)

[34] Grolleau, E., Choquet-Geniet, A.: Off-Line Computation of Real-Time Schedules by Means of Petri nets. pp. 309–316. Workshop On Discrete Event Systems, Analysis and Control, Ghent, Belgium, Kluwer Academic Publishers (2000)

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

[36] Harbour, M.G., Garca, J.G., Gutirrez, J.P., Moyano, J.D.: MAST: Modeling and Analysis Suite for Real Time Applications. pp. 125–134. Proc. of the 13th Euromicro Conference on Real-Time Systems, Delft, The Netherlands, (2001)

[37] Harbour, M.G., Palencia, J.: Response Time Analysis for Tasks Scheduled under EDF within Fixed Priorities. In: Proceedings of the 24th IEEE Real-Time Systems Symposium, Cancun, Mexico (2003)

[38] Henzinger, T.A., Kirsch, C.M., Sanvido, M.A., Pree, W.: From Control Models to Real-Time Code Using Giotto. IEEE Control Systems Magazine **1**(23) (2003)

[39] Hugues, J., Zalila, B., Pautet, L., Kordon, F.: From the prototype to the final embedded system using the Ocarina AADL tool suite. ACM Transactions on Embedded Computing Systems (TECS) , ACM Press, New York, USA **7**(4), 42:2–42:25 (2008)

[40] ISO 10303-1: Part 1: Overview and fundamental principles (1994)

[41] ISO 10303-11: Part 11: edition 2, EXPRESS Language Reference Manual (2004)

[42] ISO 10303-21: Part 21: edition 2, Implementation method: Clear text encoding of the exchange structure (2001)

[43] ISO 10303-22: Part 22: Implementation method: Standard data access interface specification (1998)

[44] Iversen, T.K., Kristoffersen, K.J., Larsen, K.G., Madsen, R.G., Laursen, M., Mortensen, S.K., Pettersson, P., Thomasen, C.B.: Model-Checking Real Time Control Programs : Verifying LEGO Mindstorm Systems Using UPPAAL. Tech. rep., BRICS RS-99-53 (1999)

[45] Kay, J., Lauder, P.: A Fair Share Scheduler. In: Communications of the ACM, vol. 31, pp. 44–45 (1988)

[46] Kleinrock, L.: Queueing Systems : theory. Wiley-interscience (1975)

[47] Lawall, J., Muller, G., Duchesne, H.: Language Design for implementing Process Scheduling Hierarchies. Proceedings of the PEPM'04 conferences. August 24-26, Veronna Italy pp. 80–90 (2004)

[48] Legrand, J., Singhoff, F., Nana, L., Marcé, L.: Performance Analysis of Buffers Shared by Independent Periodic Tasks. LISyC Technical report, number legrand-02-2004, Available at http://beru.univ-brest.fr/~singhoff/cheddar (2004)

[49] Legrand, J., Singhoff, F., Nana, L., Marcé, L., Dupont, F., Hafidi, H.: About Bounds of Buffers Shared by Periodic Tasks : the IRMA project. In the 15th Euromicro International Conference of Real Time Systems (WIP Session), Porto (2003)

[50] Leung, J., Merril, M.: A note on preemptive scheduling of periodic real time tasks. Information processing Letters **3**(11), 115–118 (1980)

[51] Liu, C.L., Layland, J.W.: Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environnment. Journal of the Association for Computing Machinery **20**(1), 46–61 (1973)

[52] Maes, E.: Validation de systèmes temps-réel et embarqué à partir d'un modèle MARTE. Thales RT, Journée Ada-France 2007, Brest (2007)

[53] Muhuri, P., Shukl, K.: Real-time task scheduling with fuzzy uncertainty in processing times and deadlines. Applied Soft Computing review, Volume 8, Issue 1, Pages 1-13. (2008)

[54] Nemer, F., Cassé, H., Sainrat, P., Awada, A.: Improving the WCET accuracy by inter-task instruction cache analysis. IEEE International Symposium on Industrial Embedded Systems (SIES 2007), Lisbonne, July, p. 25-32 (2007)

[55] Nicholson, M., McDermid, J., Burns, A.: Analysis and Design Synthesis for Hard Real-Time Safety Critical Systems. YCS-237 technical report, Department of Computer Science, University of York (2004)

[56] OMG: A UML Profile for MARTE, Beta 1. OMG Document Number: ptc/07-08-04 (2007)

[57] Panunzio, M., Vardanega, T.: A Metamodel-Driven Process Featuring Advanced Model-Based Timing Analysis . Proceedings of the 12th International Conference on Reliable Software Technologies, Ada-Europe. Geneva, LNCS springer-Verlag (2007)

[58] Plantec, A., Ribaud, V.: EUGENE: a STEP-based framework to build Application Generators. AWCSET'98, CSIRO-Macquarie University (1998)

[59] Plantec, A., Singhoff, F.: Refactoring of an Ada 95 Library with a Meta CASE Tool. ACM SIGAda Ada Letters, ACM Press, New York, USA **26**(3), 61–70 (2006)

[60] Plassart, L., Parc, P.L., Singhoff, F., Marcé, L.: Modelling and Simulation of Interactions Between the Local Command Units and the Supervisor of an Automated Assembly Line: a Case Study. Journal of Machine Engineering. Edited by Jedrzejewski, ISSN 1642-6568. n 1-2, chap. Volume 5 (2005)

[61] Plassart, L., Singhoff, F., Parc, P.L., Marcé, L.: Impact de l'ordonnancement temps réel des tâches d'un superviseur de ligne d'assemblage . 1ères Rencontres des Jeunes Chercheurs en Informatique Temps Réel 2005 (RJCITR'05), conjointement à l'école d'été temps réel 2005 (ETR'05), Nancy (2005)

[62] Regehr, J., Stankovic, J.A.: HLS : a Framework for Composing Soft Real-Time Schedulers. In the 22th IEEE International Real-Time Systems Symposium (RTSS'01). London, UK. pp. 3–14 (2001)

[63] Revest, F., Boniol, F., Pagetti, C.: Aide à la conception multi-points de vue de systèmes embarqués. Journées Formalisation des Activités Concurrentes, 15-16 mars, CERT-ONERA, Toulouse (2007)

[64] Robertazzi, T.G.: Computer Networks and Systems : queueing theory and performance evaluation. Springer-Verlag (1990)

[65] Rolland, J., Thomas, D., Chemouil, D.: Utilisation d'AADL pour la conception de logiciels de vol satellite. Revue Génie logiciel, Number 80, pages 41-44 (2007)

[66] SAE: Architecture Analysis and Design Language (AADL) AS 5506. Tech. rep., The Engineering Society For Advancing Mobility Land Sea Air and Space, Aerospace Information Report, Version 1.0 (2004)

[67] SAE: AADL Annex Behavior (draft V1.6), AS 5506. Tech. rep., The Engineering Society For Advancing Mobility Land Sea Air and Space, Aerospace Information Report (2007)

[68] SEI: The Rate Monotonic Analysis. Tech. rep., In the Software Technology Roadmap. http://www.sei.cmu.edu/str/descriptions/rma_body.html (2003)

[69] Sha, L., Rajkumar, R., Lehoczky, J.: Priority Inheritance Protocols : An Approach to real-time Synchronization. IEEE Transactions on computers **39**(9), 1175–1185 (1990)

[70] Shin, I., Lee, I.: Periodic resource model for compositional real-time guarantees. In: 4th IEEE International Real-Time Systems Symposium (RTSS'03) (2003)

[71] Singhoff, F.: Cheddar Release 2.x User's Guide. LISyC Technical report, number singhoff-01-2007, Available at http://beru.univ-brest.fr/~singhoff/cheddar (2007)

[72] Singhoff, F.: The Cheddar AADL property set (Release 2.x). LISyC Technical report, number singhoff-03-2007, Available at http://beru.univ-brest.fr/~singhoff/cheddar (2007)

[73] Singhoff, F., Legrand, J., Nana, L., Marcé, L.: Scheduling and Memory requirements analysis with AADL. ACM SIGAda Ada Letters, ACM Press, New York, USA **25**(4), 1–10 (2005)

[74] Singhoff, F., Plantec, A.: AADL Modeling and Analysis of hierarchical schedulers. ACM SIGAda Ada Letters, ACM Press, New York, USA **27**(3), 41–50 (2007)

[75] Singhoff, F., Plantec, A.: Towards User-Level extensibility of an Ada library : an experiment with Cheddar. Proceedings of the 12th International Conference on Reliable Software Technologies, Ada-Europe. LNCS springer-Verlag, Volume 4498, pages 180-191, Geneva (2007)

[76] Singhoff, F., Plantec, A., Dissaux., P.: Can we increase the usability of real time scheduling theory ? The Cheddar project. pp. 240–253. 13th International Conference on Reliable Software Technologies, Ada-Europe, Lecture Notes on Computer Sciences, Springer-Verlag editor, volume 5026, Venice (2008)

[77] Sokolsky, O., Lee, I., Clark, D.: Schedulability Analysis of AADL models . International Parallel and Distributed Processing Symposium, IPDPS 2006, Volume 2006, (2006)

[78] Subraminian, V., Gill, C., Sanchez, C., Sipma, H.B.: Reusable Models for Timing and Liveness Analysis of Middleware for Distributed Real-Time and Embedded systems. Proceedings of the 6th ACM and IEEE International conference on Embedded software EMSOFT '06 (2006)

[79] Taft, S.T., Duff, R.A., Brukardt, R.L., Ploedereder, E., Leroy, P.: Ada 2005 Reference Manual. Language and Standard Libraries. International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1 and Amendment 1. LNCS Springer Verlag, number XXII, volume 4348. (2006)

[80] Tanenbaum, A.: Modern Operating Systems. Prentice-Hall (2001)

[81] TimeSys: Using TimeWiz to Understand System Timing before you Build or Buy. White paper, http://www.timesys.com/index.cfm?bdy=home_bdy_library.cfm (2002)

[82] Tindell, K.W., Clark, J.: Holistic schedulability analysis for distributed hard real-time systems. Microprocessing and Microprogramming **40**(2-3), 117–134 (1994)

[83] Tri-Pacific: Rapid-RMA : The Art of Modeling Real-Time Systems. http://www.tripac.com/html/prod-fact-rrm.html (2003)

[84] Vahalia, U.: UNIX Internals : the new frontiers. Prentice Hall (1996)

[85] Wells., L.: Performance Analysis using CPN Tools. ACM International Conference Proceeding Series; Vol. 180, Proceedings of the 1st international conference on Performance evaluation methodologies and tools. (2006)

A Examples of Cheddar programs

A.1 Cheddar program modelling the partition 1 scheduler

```
start_section  start1 :
  partition1_capacity : integer := 4;
  partition1_duration : clock := 0;
  quantum : integer :=2;
  my_prio : array (tasks_range) of integer;
  my_prio:=0;
end section;
```

```
priority_section partition1_priority :
   quantum:=quantum-1;
   if quantum = 0
      then quantum:=2;
           my_prio(previously_elected):=my_prio(previously_elected)+1;
   end if;
end section;

election_section partition1_election :
   return min_to_index(my_prio);
end section;

automaton_section  partition1_scheduler  :
  Pended : initial_state;
  Ready : state;
  Wait_Priority : state;

  transition Pended ==> [,partition1_duration:=0;,wakeup1!] ==> Wait_Priority;
  transition Wait_Priority ==> [partition1_duration<partition1_capacity,
     ,partition1_priority!] ==> Ready;
  transition Ready ==> [,,partition1_election!] ==> Wait_Priority;
  transition Wait_Priority ==> [partition1_duration=partition1_capacity,,]
     ==> Pended;
end section;
```

A.2 Cheddar program modelling the partition 2 scheduler

```
start_section  start2 :
   partition2_capacity : integer := 6;
   partition2_duration : clock := 0;
end section;

election_section  partition2_election :
   return min_to_index(tasks.priority);
end section;

automaton_section  partition2_scheduler  :
  Ready : state;
  Pended : initial_state;

  transition Pended ==> [,partition2_duration:=0;,wakeup2!] ==> Ready;
  transition Ready ==> [partition2_duration<partition2_capacity,
     ,partition2_election!] ==> Ready;
  transition Ready ==> [partition2_duration=partition2_capacity,,] ==> Pended;
end section;
```

A.3 Cheddar program modelling the partition scheduler

```
start_section  start_processor :
      partition_clock : clock:=0;
end section;

automaton_section  processor_scheduler :
  Schedule_Partition2 : initial_state;
  Schedule_Partition1 : state;
```

```
  Restart : state;

  transition Schedule_Partition2 ==> [,,wakeup2?] ==> Schedule_Partition1;
  transition Schedule_Partition1 ==> [partition_clock=6,,wakeup1?] ==> Restart;
  transition Restart ==> [partition_clock=10,
     partition_clock:=0;,] ==> Schedule_Partition2;
end section;
```

B The STEP technology

As described in [40], STEP, namely the ISO 10303 international standard, is developed in order to facilitate product information sharing by specifying sufficient semantic content for data and their usage. Parts of STEP are intended to standardize conceptual structures of information either generic, or within a subject area (e.g. mechanics). A important concept of STEP is the definition of consensus data specifications that describe the data to be exchanged or shared and that cover some particular application domains. These data specifications are called *application protocols*. In order to define and implement *application protocols*, the information modelling language $EXPRESS$ [41], the standard file exchange format [42] and a standard data access interface (SDAI) [43] were developed.

The EXPRESS language [41] is a data-oriented modelling language. The application data are described in schemata. A schema can refer to other schemata. A schema owns the type definitions and the object descriptions of the application called *Entities*. An entity is made of attributes and constraint descriptions. The constraints expressed in an entity definition can be of several kinds:

- The *unique* constraint allows entity attributes to be constrained (e.g. to be unique either solely or jointly).
- The *derive* clause is used to represent computed attributes.
- The *where* clause of an entity constraints each instance of an entity.
- The *inverse* clause is used to specify inverse cardinality constraints.

Entities may inherit attributes and constraints from their super-types. *EXPRESS* allows the definition of global rules. These rules are used when either all instances of a given entity or instances of at least two entities need to be examined concurrently to determine whether a given constraint is satisfied.

Figure 15 depicts an $EXPRESS$ example with four types and three entities used for the modelling of the date concept. Types are specified in order to qualify *day*, *month* and *year* numbers used within a date. Entity *cardinal_date* is inheriting from the *date* entity and is given with a rule which further constraints each *cardinal_date* instance. *calendar_date_depicter* illustrates how a derived attribute can compute data. An instance of *calendar_date_depicter*

```
SCHEMA Example_schema;

TYPE year_number = INTEGER; END_TYPE;
TYPE day_in_month_number = INTEGER; END_TYPE;
TYPE month_in_year_number = INTEGER;
WHERE
  WR1: (1 <= SELF) AND (SELF <= 12);
END_TYPE;

TYPE day_in_year_number = INTEGER; END_TYPE;

ENTITY date;
  year_component : year_number;
END_ENTITY;

ENTITY calendar_date SUBTYPE OF (date);
  day_component   : day_in_month_number;
  month_component : month_in_year_number;
WHERE
  WR1: valid_calendar_date(SELF);
END_ENTITY;

ENTITY calendar_date_depicter;
  source : calendar_date;
DERIVE
  rep : STRING := format(source.year_component, '2I') + '/'
    + format(source.month_component, '2I') + '/'
    + format(source.day_component, '2I');
END_ENTITY;

FUNCTION valid_calendar_date (date: calendar_date): LOGICAL;
  IF NOT ((1 <= date.day_component) AND (date.day_component <= 31)) THEN
    RETURN(FALSE);
  END_IF;
  CASE date.month_component OF
    4 : RETURN((1 <= date.day_component) AND (date.day_component <= 30));
    6 : RETURN((1 <= date.day_component) AND (date.day_component <= 30));
    9 : RETURN((1 <= date.day_component) AND (date.day_component <= 30));
    11 : RETURN((1 <= date.day_component) AND (date.day_component <= 30));
    2 :
    BEGIN
      IF leap_year(date.year_component) THEN
        RETURN((1 <= date.day_component) AND (date.day_component <= 29));
      ELSE
        RETURN((1 <= date.day_component) AND (date.day_component <= 28));
      END_IF;
    END;
  END_CASE;
  RETURN(TRUE);
END_FUNCTION;

END_SCHEMA;
```

Figure 15: An *EXPRESS* example: the calendar_date entity

can be associated with a *calendar_date* instance in order to compute a string representation for the date.

The STEP physical file format defines an exchange structure using a clear text encoding of product data for which a conceptual model is specified in

the *EXPRESS* language. The mapping from the *EXPRESS* language to the syntax of the exchange structure is specified in [42].

The Standard Data Access Interface (SDAI) [43] defines an access protocol for *EXPRESS*-defined databases and is defined independently from any particular system and language. The representation of this functional interface in a particular programming language is referred to as a language binding in the standard. As an example, ISO 10303-23 is the STEP part describing the C++ SDAI binding.

The main goals of the SDAI are:

- To access and manipulate data which are described using the *EXPRESS* language so that access to a database happens through a conceptual schema, not a physical schema.
- To allow access to multiple data repositories by a single application at the same time.
- To allow commit and rollback on a set of SDAI operations.
- To allow access to the *EXPRESS* definition of all data elements that can be manipulated by an application process.
- To allow the validation of the constraints defined in *EXPRESS*.

An SDAI can be implemented as an *EXPRESS* interpreter or as a specialized data interface. In the standard [43], the interpreter implementation is referred to the SDAI late binding. An SDAI late binding is generic in nature. In the standard, the specialized data interface implementation is referred to the SDAI early binding.