Can we increase the usability of real time scheduling theory ? The Cheddar project

Frank Singhoff*, Alain Plantec*, Pierre Dissaux+

* LISyC/University of Brest, 20, av Le Gorgeu, 29238 Brest Cedex 3, France
+ Ellidiss Technologies, 24 quai de la douane, 29200 Brest, France
{singhoff,plantec}@univ-brest.fr, pierre.dissaux@ellidiss.com

Abstract. The Cheddar project deals with real time scheduling theory. Many industrial projects do not perform performance analysis with real time scheduling theory even if the demand for the use of this theory is large. The Cheddar project investigates why real time scheduling theory is not used and how its usability can be increased. The Cheddar project was launched at the University of Brest in 2002. This article presents a summary of its contributions and ongoing works.

1 Introduction

Real time scheduling theory provides algebraic methods and algorithms in order to predict the temporal behavior of real time systems. The foundations of real time scheduling theory were proposed in 1970 [1] and it leads to extensive researchs. Since 1990, it makes it possible the analysis of systems composed of periodic tasks sharing resources and running on a single processor [2]. Numerous operating systems provide features allowing the implementation of such applications. Some standards and compilers also provide tools to enforce that an application meets real time scheduling theory assumptions. The Ravenscar profile defined in the Ada 2005 standard allows this assumption checking [3].

Real time scheduling theory was successfully used in many projects [4]. Nevertheless, many practical cases also do not perform analysis with such a method even if our experience shows that the demand for the use of this analysis method is large.

Several reasons can explain why real time scheduling analysis is not applied as much as it could be. Of course, there exists some architectures on which real time scheduling analysis is difficult. For example, few analytical methods were proposed for the analysis of distributed systems [5]. Sometimes, there is no analytical method for architectures made of complex schedulers or task models. In these cases, a real time scheduling toolset should at least provide means to model the system and to run simulations.

Furthermore, we believe that this theory is not so easy to understand and to be applied for many engineers. Many analytical methods and algorithms were proposed during the last 30 years. Each analytical method allows to compute different performance criteria. Each criterion requires that a set of assumptions must be meet by the investigated system. Then, it may be difficult for a designer to choose the relevant analytical method. Unfortunately, there is currently few supports by design languages and CASE tools which can help him to automatically apply real time scheduling theory.

This article presents three possible ways investigated by the Cheddar project in order to increase the usability of real time scheduling theory. Section 2 presents a set of tools which aims at helping the designer to automatically apply real time scheduling theory on an architecture model. Section 3 depicts how the use of an architecture design language can help the designer to apply real time scheduling theory. Section 4 presents a domain specific language and a set of tools that the designer can use when no analytical method can be applied in order to investigate performances of a specific architecture. Finally, section 5 is devoted to a conclusion and presents Cheddar project ongoing works.

2 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 which help the system designer to analyze the timing behaviour of his architecture. With the Liu and Layland real time task model [1], 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 wake up time. From this 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 to analyze. Usually, simulations can not lead to a proof. However with deterministic schedulers and periodic tasks, scheduling simulation may lead to a schedulability proof if the designer is able to compute the scheduling during the base period [6]. 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 their deadline. Joseph, Pandia, Audsley et al. [7] have proposed a way to compute the worst case response time of a task with pre-emptive fixed priority scheduler by:

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

Where r_i is the worst case response time of the task i and $hp(i)$ is the set of tasks which have a higher priority level than i. This feasibility test must be extended to take into account task waiting time on shared resources, jitter on task release time or task precedency relationships. To apply a feasibility test, the designer must check that his design and his executive fulfill all the feasibility test assumptions. As an example, with the feasibility test of the equation (1), D_i must be less or equal than P_i and all tasks must have the same first release time. Then, for a designer who has not a deep knowledge of real time scheduling theory, verifying an architecture with feasibility tests becomes a difficult task because, for each part of the architecture to verify, he must (see figure 1):

1. Choose the performance criterion 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 which takes into account properties required for analysis.
3. Select a feasibility test which is able to compute the criterion chosen in (1) and which is compliant with the models chosen in (2). For such a purpose, he must check that his model is compliant with the feasibility test assumptions.

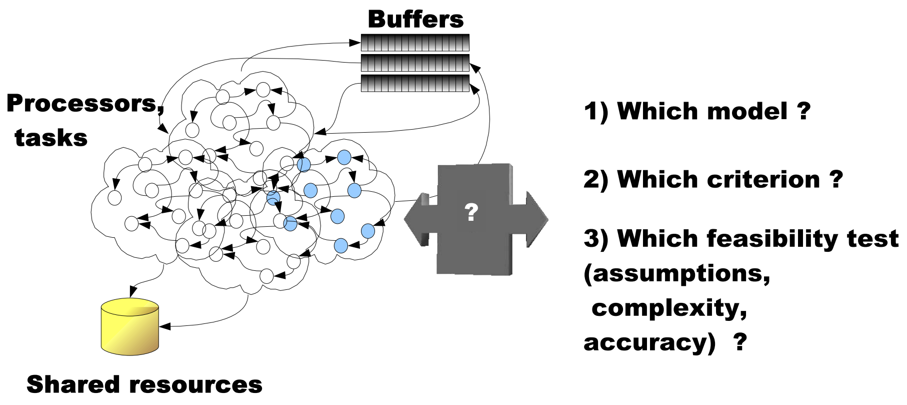

Fig. 1. From the modelling to the analysis

But of course, in many cases, this work can be quite simple since the studied architecture is simple too. A real time scheduling analysis toolset should actually provide several using levels. Several real time scheduling tools exist such as MAST [8], Rapid-RMA [9] or Cheddar. Cheddar is a toolset composed of an editor and of a framework. The designer can specify his architecture model with the Cheddar editor. However, it is expected that designers perform modelling with dedicated CASE tools. The Cheddar framework consists in a set of Ada packages which includes most current feasibility tests and most of the classical real time scheduling algorithms. This framework also offers a domain specific language together with an interpreter and a compiler, for the design and the analysis of schedulers which are not already implemented into the framework.

Cheddar offers different using levels depending on the architecture to analyze, on the CASE tool Cheddar is supposed to work with or on the knowledge of the designer. Typical use cases are:

– Just load an architecture model into the Cheddar editor and simply push a button to perform its analysis. In this case, Cheddar chooses the feasibility test, 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. For example, the designer can model his architecture with the Ravenscar design pattern. Ravenscar is a part of the Ada 2005 standard [3]. It is a set of Ada program restrictions usually enforced at compilation time, which guaranties that the software architecture is real time scheduling theory compliant. Ravenscar is an Ada subset from which one can write applications 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. This first way to use Cheddar is also the best suited for students who have to understand real time scheduling foundations.
– A second way is to let the designer choose which performance criteria to compute. The designer must handle the Cheddar editor menus to customized which criteria the Cheddar framework has to compute. In this case, feasibility test assumptions are always automatically checked by Cheddar.
– Third, if the scheduling algorithms or the feasibility tests implemented into Cheddar can not be applied, then the designer must extend the Cheddar framework. Two ways exist for such a purpose. The framework can be extended by the Cheddar domain specific language with the process explained in section 4. Otherwise, the designer manually implements the performance analysis tools. In this case, he must well understand the Cheddar framework design.

There exists many other ways to use a toolset such as Cheddar. As an example, Cheddar can be embedded into CASE tools such as Stood [10] or Ocarina [11] in order to increase its usability. In this case, the designer does not use the Cheddar editor anymore and the Cheddar framework is directly called by embedding CASE tools. Cheddar exports analysis results as an XML data stream which can be displayed back by the CASE tools. The next section presents how an architecture language can be used to achieve CASE tool and analysis tool interoperability.

3 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 such a knowledge into the engineering process with the help of design languages and design patterns.

Panunzio and Vardanega have proposed a metamodel which permits the execution of timing analysis [12]. An UML profile called MARTE which allows such a timing analysis is also currently investigated by Frédéric et al. [13]. 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 [14]. AADL is used to design and analyze software and hardware architecture of embedded real-time systems. In the context of the Cheddar project, 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 primilary used in order to check that the first AADL standard can be actually analyzed with real time scheduling theory tools. Then, we have investigated how memory footprint analysis can be conducted with AADL [15] and finally, some design patterns expressed in AADL were proposed in order to ease interoperability between AADL tools [10].

3.1 Investigating AADL suitability for real time scheduling theory

An AADL model is a set of hardware and software components such as data, threads, processes (the software side of a specification), processors, devices and busses (the hardware side of a specification). A data component may represent a data structure in the program source. An AADL data component can be implemented by an Ada tagged record. A thread is a sequential flow of control that executes a program. An AADL thread can be implemented by an Ada task. AADL threads can be released according to several policies: a thread may 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. Properties can be defined for most of AADL components. A property is defined by a name, a value and a type. Information provided by component properties can be related to the component behavior, its state, the way it will be implemented in Ada or anything else that makes it possible to perform analysis.

Figure 2 shows an AADL specification. This specification contains a shared resource (called $R1$) accessed by two threads (threads $TH1$ and $TH2$). The threads and the shared resource are defined into one address space (process *proc*0). The process *proc*0 is bound to a processor called *cpu*0.

The first release of the AADL standard provides component properties required in order to apply 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. We have proposed a set of property extensions [16] to model:

– Usual properties of real time schedulers (eg. quantum, preemptivity, POSIX 1003.1b policies).

```
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
        can_access : requires data access shared_resource_type;
end task_type;
thread implementation task_type.Impl
    properties
        Dispatch_Protocol => Periodic;
        Period => 50;
        Compute_Execution_time => 3 ms .. 3 ms;
        Cheddar_Properties::POSIX_Scheduling_Policy => SCHED_FIFO;
        Cheddar_Properties::Fixed_Priority => 5;
        Cheddar_Properties::Dispatch_Jitter => 10;
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;
        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.can_access;
        data access R1 − > TH2.can_access;
end a_proc.Impl;
system a_system
end a_system;
system implementation a_system.Impl
    subcomponents
        cpu0 : processor a_cpu.Impl;
        proc0 : process a_proc.Impl;
    properties
        Actual_Processor_Binding => reference cpu0 applies to proc0;
end a_system.Impl;
```

Fig. 2. Example of an AADL model

- Usual thread properties such as fixed priority, jitter, offset, shared resource blocking time, ...
- Properties to define when shared resources are accessed by threads.
- And finally, the current AADL standard leading to some ambiguities, some properties to express thread precedency relationships which can not be computed from standard AADL connections.

Some of the lacks presented above will be fixed in the next AADL standard with the Behavioral Annex [17] and with some of the Cheddar properties which will be included in the standard AADL property set.

3.2 Memory footprint analysis with AADL

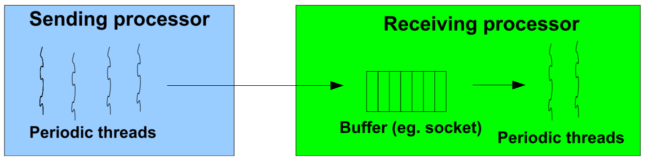

Fig. 3. Part of a distributed system

One of the most interesting part of an architecture design language as AADL, is that it allows performance analysis on multiple resources. This is especially mandatory with distributed real time systems which may be composed of several processors, memory units and communication devices. The figure 3 shows a distributed system composed of two processors exchanging messages througth a TCP/IP socket. With such a system, performance analysis on processors and memory units can not be performed independently:

- In one hand, if the periodic receiving/sending threads have a high priority level, and then a short worst case response time, the required memory in the socket to store messages may be low.
- In the other hand, when sending/receiving threads have a long worst case response time, the memory requirement into the socket may be high if no message have to be lost.

By defining all the parts of a system, AADL allows such an analysis. As an example, in [18], Legrand et al. have proposed a set of feasibility tests based on queueing system. These feasibility tests were adapted to AADL in [15]. It was shown how to perform memory footprint analysis with AADL models containing event data ports. Event data ports represent connection points for transfer of messages that may be queued. For example, if both producers and consumers are periodic AADL thread exchanging messages through an event data port, L, the worst case number of messages in the event data port is equal to $L = 2.n$

if threads are harmonic, or $L = 2.n + 1$ otherwise. Where n is the number of producers. As any feasibility test, this memory footprint feasibility test has to meet several assumptions (eg. Kirchhoff's law).

3.3 About interoperability between AADL 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 a standard use of the AADL all along the tool-chain. In the sequel, we show how AADL can be used as a pivot language between Cheddar and a modelling tool called Stood.

Stood is a software design tool that provides an extended support for AADL in addition to its compliancy with the HOOD methodology. Stood makes it possible to manage a complete software project by building libraries of reusable components, reversing legacy code and specifying the real time application as well as its execution platform. Most of the modelling activities can be performed graphically and the corresponding AADL code is automatically generated by the tool.

To ease interoperability between Stood and Cheddar, in [10], we have proposed a set of AADL design patterns which models usual real time synchronization/threads-communication paradigms (eg. ARINC 653 [19]):

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 [14] 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 complete time. This design pattern does not require the use of a shared data component. In this simple case, the execution platform consists in one processor running a scheduler such as Rate Monotonic [1].
2. **Ravenscar design pattern:** main drawback of the previous pattern is its lack of flexibility at run time. Each thread will always execute, read and write data at pre-defined times, even if useless. In order to introduce more flexibility, asynchronous inter-thread communications can be proposed. An example of such a run-time environment is given by the Ravenscar profile. In Ravenscar, threads access shared data components asynchronously according to priority inheritance protocols.
3. **Blackboard design pattern:** Ravenscar allows a thread to allocate/release several shared resources (eg. AADL data). Real time scheduling theory usually models such a shared resource as a semaphore, to represent, for example, a critical section. In classical operating systems, there exists many synchronization design patterns such as critical section, barrier, readers-writers, private semaphore, and various producers-consummers. 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, that 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 executives provide communication features which allow to store all written messages in a memory unit. AADL also propose such a feature with event data ports or shared data components.

For each pattern, an applicative test case was described under the form of an AADL model which has 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, ...) [10].

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

Many practical cases can not be analyzed by real time scheduling theory feasibility tests. Complex industrial real time architectures frequently make use of specific task models or schedulers. In this case, no feasibility tests exists and building new feasibility tests is a difficult and expensive work. Furthermore, industrial real time systems may be composed of a large number of entities (eg. tasks, processors, memory units ...). These large scale systems can not be efficiently analyzed with model-checking. The only way people can expect to verify performances of such real time systems is to perform analysis with extensive simulations.

Languages and models were proposed for such a purpose. CPN tools [20] provides simulation features based on Petri Net for example. 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 reusability of the simulation programs. The Cheddar framework 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 schedulers and tasks.

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 Platypus [21].

4.1 A language for the modelling of real time schedulers

Real time schedulers are composed of two different aspects:

1. Arithmetic and logical statements which allow to select a task amoung a set of ready tasks or to compute task priorities.
2. Temporal constraints and synchronizations between entities (eg. 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) a subset of Ada for the modelling of arithmetic and logical statements of the schedulers and 2) a timed automaton language for the synchronizations modelling scheduler and task relationships. A detailed description of this language is given into the Cheddar users's guide [22].

An Ada subset language 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 (eg. 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 sub-programs called sections. These sub-programs are typed:

– Some sub-programs are devoted to data simulation declaration and initialization. They are called *start_section*.
– Some sub-programs allow to select a task amoung a set of ready tasks according to simulation data (eg. priority). These sub-programs are called *election_section*.
– Finally, some sub-programs contain statements which have to be ran 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 assignements. 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 last 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.

A timed automaton language The second part of a Cheddar scheduler model is a network of timed automata. A scheduler model can contain timed automata similar to those proposed by UPPAAL [23, 24]. UPPAAL is a toolbox for the modelling and the verification of real time systems.

A network of timed automata models timing and synchronization between schedulers and tasks. The Ada subset 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 makes it possible the modelling of simple schedulers like Earliest Deadline First, Rate Monotonic ou Maximum Urgency First. However, some real time schedulers require the modelling of complex synchronizations. This is the case of hierarchical schedulers. An architecture based on hierarchical scheduling is an architecture in which several entities work all together for the processor sharing. 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 (eg. fair process scheduling [25]). Today, hierarchical scheduling also exists in several real time system standards such as ARINC 653, POSIX 1003 or Ada 2005 [26, 3, 19].

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 subset sections in order to compute task priorities or to choose the next task to run.

For further readings, a model of an ARINC 653 hierarchical scheduling modelled with the Cheddar language is given in [27].

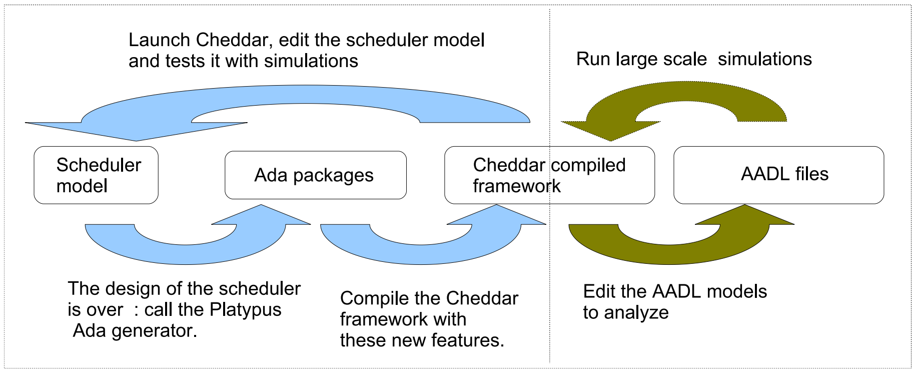
Launch Cheddar, edit the scheduler model and tests it with simulations
Run large scale simulations
Scheduler model
Ada packages
Cheddar compiled framework
AADL files
The design of the scheduler is over : call the Platypus Ada generator.
Compile the Cheddar framework with these new features.
Edit the AADL models to analyze

Fig. 4. A process to perform simulations from Cheddar scheduler models

4.2 Engineering process of a Cheddar scheduler model: from the model to the scheduling simulation

Figure 4 depicts the process that a designer runs to perform scheduling simulations with specific scheduler or task models:

1. With the Cheddar toolset the designer models a new scheduler. This model can be directly interpreted using the Cheddar framework. This feature eases the design step and allows the designer to perform small scheduling simulations.
2. When his scheduler has been tested, the designer can generate Ada packages implementing his scheduler into the Cheddar framework. The Ada package generator is implemented within Platypus. Platypus [21] is a meta-environment suitable for model driven engineering activities.
3. The generated Ada packages can be integrated into the Cheddar framework. The Cheddar framework is then compiled in order to enrich it with this new scheduler.
4. The designer can actually run large scale simulations with this new Cheddar framework embedding his scheduler. The designer makes use of his scheduler through this enriched Cheddar framework in the same way he will make use of standard schedulers manually implemented into Cheddar (eg. Rate Monotonic).

5 Conclusion and ongoing works

This article presents three possible ways investigated by the Cheddar project in order to increase the usability of real time scheduling theory. We have presented a set of tools which help the designer to apply real time scheduling theory. This toolset allows several levels of use and is able to perform analysis of models written with design languages such as AADL. We also have presented a domain specific language to investigate performances of architectures on which real time scheduling theory does not propose analytical method.

At the time we are writing this article, it is difficult to state if Cheddar has actually helped people to apply real time scheduling theory on practical cases. The toolset has been used to build many real time scheduling courses. It has been experimented in different research and development projects related to avionic or robotic applications, with different design languages. Besides these first encouraging results, the Cheddar project have raised several interesting open research questions.

First, Ellidiss technologies will distribute Cheddar with its modelling tool Stood. We expect to spread the use of real time scheduling theory on practitioners. For such a purpose, we have started to investigate how to apply Cheddar to modelling design patterns that practitioners usually handle with Stood [10]. For this project, we have chosen AADL as a pivot language between Stood and Cheddar.

Second, the Cheddar language we have defined to model schedulers was experienced in several projects. We know that this language is well suited for this purpose. The language is based on an Ada subset, which allows static analysis (eg. SPARK [28]) and on a timed automaton language which allows dynamic analysis (eg. model-checking with UUPPAL). 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 resource requiring deep and accurate analysis was the processor. But now, many real time systems are distributed over several processors and several resources have to be managed all together: processors, communication networks and memory units. In the next months, we plan to focus on memory footprint analysis with queueing system models.

6 Acknowledgments

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 [11]. We also would like to thank the Ocarina's Team (B. Zalila, J. Hugues, L. Pautet and F. Kordon).

References

1. 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) (1973) 46–61
2. Sha, L., Rajkumar, R., Lehoczky, J.: Priority Inheritance Protocols : An Approach to real-time Synchronization. IEEE Transactions on computers **39**(9) (1990) 1175–1185
3. 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)
4. SEI: The Rate Monotonic Analysis. Technical report, In the Software Technology Roadmap. http://www.sei.cmu.edu/str/descriptions/rma_body.html (2003)
5. Tindell, K.W., Clark, J.: Holistic schedulability analysis for distributed hard real-time systems. Microprocessing and Microprogramming **40**(2-3) (1994) 117–134
6. Leung, J., Merril, M.: A note on preemptive scheduling of periodic real time tasks. Information processing Letters **3**(11) (1980) 115–118
7. George, L., Rivierre, N., Spuri, M.: Preemptive and Non-Preemptive Real-time Uni-processor Scheduling, INRIA Technical report number 2966 (1996)
8. Harbour, M.G., Garca, J.G., Gutirrez, J.P., Moyano, J.D.: MAST: Modeling and Analysis Suite for Real Time Applications, Proc. of the 13th Euromicro Conference on Real-Time Systems, Delft, The Netherlands, (2001) 125–134
9. Tri-Pacific: Rapid-RMA : The Art of Modeling Real-Time Systems. http://www.tripac.com/html/prod-fact-rrm.html (2003)
10. 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)
11. Hugues, J., Zalila, B., Pautet, L.: Rapid Prototyping of Distributed Real-Time Embedded Systems Using the AADL and Ocarina, In 18th IEEE/IFIP International Workshop on Rapid System Prototyping (RSP'07), Porto Allegre, Brazil (2007)

12. 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)
13. 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)
14. Inc., S.: Architecture Analysis and Design Language (AADL) AS 5506. Technical report, The Engineering Society For Advancing Mobility Land Sea Air and Space, Aerospace Information Report, Version 1.0 (2004)
15. Singhoff, F., Legrand, J., Nana, L., Marcé, L.: Scheduling and Memory requirements analysis with AADL, ACM SIGAda Ada Letters, volume 25, number 4, pages 1-10. Edited by ACM Press, New York, USA, ISSN:1094-3641 (2005)
16. 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)
17. Inc., S.: AADL Annex Behavior (draft V1.6), AS 5506. Technical report, The Engineering Society For Advancing Mobility Land Sea Air and Space, Aerospace Information Report (2007)
18. 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)
19. Arinc: Avionics Application Software Standard Interface. The Arinc Committee (1997)
20. Wells., L.: Performance Analysis using CPN Tools, Proceedings of the First International Conference on Performance Evaluation Methodologies and Tools 2006. ACM Press, ValueTools'06 (2006)
21. : Platypus Technical Summary and download. http://cassoulet.univ-brest.fr/mme/ (2007)
22. 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)
23. Alur, R., Dill, D.L.: Automata for modeling real time systems, Proc. of Int. Colloquium on Algorithms, Languages and Programming, Vol 443 of LNCS (1990) 322–335 (1990)
24. 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)
25. Kay, J., Lauder, P.: A Fair Share Scheduler. In: Communications of the ACM. Volume 31. (1988) 44–45
26. Gallmeister, B.O.: POSIX 4 : Programming for the Real World . O'Reilly and Associates (1995)
27. Singhoff, F., Plantec, A.: AADL Modeling and Analysis of a hierarchical schedulers, ACM SIGAda Ada Letters, volume 27, number 3, pages 41-50. Edited by ACM Press, New York, USA, ISSN:1094-3641 (2007)
28. Barnes, J.: High integrity software: The Spark approach to safety and security. Addison-Wesley Publishing Company (2003)