Enforcing Applicability of Real-time Scheduling Theory Feasibility Tests with the use of Design-Patterns

Alain Plantec[1], Frank Singhoff[1], Pierre Dissaux[2], and Jérôme Legrand[2]

[1] LISyC, University of Brest, UEB, 20 av. Le Gorgeu, 29238 Brest, France
alain.plantec,frank.singhoff@univ-brest.fr
[2] Ellidiss Technologies, 24 quai de la douane, 29200 Brest, France
pierre.dissaux,jerome.legrand@ellidiss.com

Abstract. This article deals with performance verifications of architecture models of real-time embedded systems. We focus on models verified with the real-time scheduling theory. To perform verifications with the real-time scheduling theory, the architecture designers must check that their models are compliant with the assumptions of this theory. Unfortunately, this task is difficult since it requires that designers have a deep understanding of the real-time scheduling theory. In this article, we investigate how to help designers to check that an architecture model is compliant with this theory. We focus on feasibility tests. Feasibility tests are analytical methods proposed by the real-time scheduling theory. We show how to explicitly model the relationships between an architectural model and feasibility tests. From these models, we apply a model-based engineering process to generate a decision tool what is able to detect from an architecture model which are the feasibility tests that the designer can apply.

1 Introduction

Performance verifications of embedded real-time architectures can be performed with the real-time scheduling theory. Real-time scheduling theory provides analytical methods, called *feasibility tests*, which make possible timing constraints verifications. A lot of feasibility tests have been elaborated during the last 30 years in order to provide a way to compute different performance criteria such as worst case task response time, processor utilization factor and worst case blocking time on shared resources.

Each criterion requires that the target system fulfils a set of specific assumptions that are called *applicability constraints*. Thus, due to the large number of feasibility tests and due to the large number of applicability constraints, it may be difficult for a designer to choose the relevant feasibility test for a given architecture to analyze. Then, it appears that in many practical cases, no such analysis is performed with the help of real-time scheduling theory although experience shows that it could be profitable.

In order to help the designer, we have proposed, in [4], a set of architecture design-patterns that allows early performance verifications of architecture models. These design-patterns model usual communication paradigms of multi-tasked real-time software. Given a particular architecture model, these design-patterns are used in order to choose the relevant set of feasibility tests. We have defined four design-patterns called *Synchronous data flows*, *Ravenscar*, *Blackboard* and *Queued buffer*. For each design-pattern, several feasibility tests can be applied. For example, in the case of the *Synchronous data flows* design-pattern, we have listed 10 feasibility tests that can be applied in 64 possible cases, depending on the parameters of each architecture components (tasks, processors, shared resources, etc). It implies that only defining a set of design-patterns may not be enough to really help the designer to automatically perform performance verifications with feasibility tests.

In this article, we investigate how to automatically check that an architecture model is compliant with a design-pattern, in order to ensure that a particular set of feasibility tests is relevant. We show how to explicitly model the relationships between an architectural design-pattern and the compliant feasibility tests. From these models, we apply a model-based engineering process to generate a decision tool which is able to identify, from an architecture model, the feasibility tests the designer is allowed to compute. Then, this decision tool helps the designer to choose the feasibility tests that he is allowed to apply to his architecture models.

This article is organized as follows. In section 2, we introduce our design-pattern approach. Section 3 presents an example: the *Synchronous data flows* design-pattern. In section 4, we explain how the decision tool is currently implemented and how we plan to integrate it into a schedulability tool called *Cheddar*. Then, section 5 is devoted to related works and we conclude and present future works in section 6.

2 The design-pattern approach

During the last decades, a lot of emphasis has been given to software modeling techniques, in a continuous move from traditional coding activities to higher level of abstractions. Several standardized languages such as the MARTE profile for UML [17] or the AADL language [21] provide a set of categorized components that are appropriate for real-time system and software modeling activities. These modeling languages allow not only the applicative architecture to be described, but also its interaction with the underlying executive. As an example, a thread is a kind of AADL component which can be scheduled by the run-time executive.

Nevertheless, although it becomes now easier to describe real-time architectures, their validation still remains a subject of investigation. For instance, the lack of a single property may sometimes be enough to prevent a real-time architecture from being properly processed by a schedulability analysis tool.

This is why, the next step in the improvement of the development process of real-time systems consists in providing to the end user a set of predefined

composite constructs that match known real-time scheduling analysis methods. The composite constructs we have studied correspond to the various inter-task communication paradigms that can be applied in an architecture and that can be considered as real-time design-patterns.

Four design-patterns that are compliant with the real-time scheduling theory are proposed in [4]. These design-patterns are:

1. **Synchronous data flows design-pattern:** this first design-pattern is the simplest one. Task share data by clock synchronizations: each task reads data at dispatch time or writes data at complete time. This design-pattern does not require the use of shared data components.
2. **Ravenscar design-pattern:** the main drawback of the previous pattern is its lack of flexibility at run time. Each task will always execute read and write data at pre-defined times, even if useless. In order to introduce more flexibility, asynchronous inter-task communications is proposed with this design-pattern: tasks access shared data components asynchronously according to priority inheritance protocols.
3. **Blackboard design-pattern:** Ravenscar allows task to share data protected by semaphores. Semaphores can be used to build various synchronization protocols such as critical section, barrier, readers-writers, private semaphore, producers-consumers and others [28]. The blackboard design-pattern implements a readers-writers synchronization protocol.
4. **Queued buffer design-pattern:** in the blackboard design-pattern, at any time, only the last written message is made available to the tasks. Queued buffer allows to store all undelivered messages in a memory unit.

Each of these design-patterns models a typical communication and synchronization paradigm of multi-tasked real-time software. Design-patterns are specified with a set of constraints on the architecture model to verify. There are two types of constraints expressed in a design-pattern: *Architectural constraints* and *Property constraints*.

- *Architectural constraints* are restrictive rules which specify the kind of architectural element that are allowed. As an example, a design-pattern can forbid declaration of a shared data component or a buffer in a model. They may also constraint component connections.
- *Property constraints* are related to the architecture components properties and constraint further their value. As an example, a design-pattern can assume that the *quantum*[3] property of a processor component must be equal to zero.

For each design-pattern, according to their architectural and property constraints, we have identified which feasibility tests the designer can compute to perform the verification of his architecture.

[3] A quantum is a maximum duration that a task can run on a processor before being preempted by another.

With this approach, the designer can verify its real-time system architecture in two steps: (1) he first looks for the design-pattern which is matching his architecture. Then (2), assuming that a matching design-pattern is found, the designer can compute all the feasibility tests associated with this design-pattern.

3 Example of the *Synchronous data flows* design-pattern

In order to specify our architecture models, we are using the AADL modeling language. AADL is a textual and graphical language for model-based engineering of embedded real-time systems that has been published as SAE Standard AS-5506 [21]. AADL is used to design and analyze software and hardware architectures of embedded real-time systems. Many tools provide support for AADL: Ocarina implements Ada and C code generators for distributed systems [10], TOPCASED, OSATE and Stood provide AADL modeling features [5,3,22], the Fremont toolset and *Cheddar* implement AADL performance analysis methods [26,25]. An updated list of supporting tools can be found on the official AADL web site http://www.aadl.info.

An AADL model describes both the hardware part and the software part of an embedded real-time system. Basically, an AADL model is composed of components with different categories: data, threads or processes (components modeling the software side of a specification), processors, devices and buses (components modeling the hardware side of a specification). A data component may represent a data structure in the program source text. It may contain subprograms such as functions or procedures. A thread is a sequential flow of control that executes a program and can be implemented by an Ada task or a POSIX thread. AADL threads can be dispatched according to several policies: a thread may be periodic, sporadic or aperiodic. An AADL process models an address space. In the most simple case, a process contains threads and data. Finally, processors, buses and devices represent hardware components running one or several applications.

This section presents one of the simplest design-patterns: the *Synchronous data flows* design-pattern. First, we specify the design-pattern by its architectural and property constraints. Then, we present the feasibility tests that are assigned to this design-pattern. In the sequel, we also illustrate this design-pattern with a compliant AADL model.

3.1 Specification of the *Synchronous data flows* design-pattern

The *Synchronous data flows* design-pattern is inherited from *Meta-H*. An AADL architecture model is compliant with this design-pattern if it is only composed of process, thread, sub-program and processor components. We also assume that the architecture model meets the constraints expressed by the following seven rules:

Rule 1: All threads are periodic.

Rule 2: We assume that threads are scheduled either by a fixed priority scheduler or by EDF [14]. In the case of a fixed priority scheduler, any kind of priority assignment can be used (*Rate Monotonic* or *Deadline Monotonic*) but we assume that all threads have different priority levels.

Rule 3: The scheduler may be either fully preemptive or non preemptive.

Rule 4: We assume that the scheduler do not use *quantum* (see the POSIX 1003 scheduling model [7]).

Rule 5: Thread communications do not make use of any data component, of any shared resource or buffer and there is no connection between threads and data components.

Rule 6: Threads are independent: thread dispatches are not affected by the inter-thread communications. In this synchronization schema, communications between threads are achieved by pure data flows with AADL data ports: each thread reads input data at dispatch time and writes output data at completion time.

Rule 7: Each processor owns only one process and there is no virtual processor. This rule expresses that there is no hierarchical scheduling (see the ARINC 653 standard [1]).

3.2 Feasibility tests assigned to the *Synchronous data flows* design-pattern

For an AADL model compliant with *Synchronous data flows*, we can perform performance analysis with real-time scheduling theory feasibility tests. For this design-pattern, we can check performances by computing two performance criteria: (1) the worst case response time of each thread and (2) the processor utilization factor. For such a purpose, we have assigned 10 feasibility tests to compute these performance criteria [23].

However, it does not mean that for each AADL model compliant with this design-pattern, we can apply the 10 feasibility tests. For a given AADL model, depending on the value of the AADL component properties, we will be able to apply one or several feasibility tests among this set of feasibility tests. We have identified 64 different cases depending on component properties values, which represent applicability assumptions of the feasibility tests. This shows that even for this simplest design-pattern, choosing the right feasibility test to apply may be difficult for architecture designers.

One of these feasibility tests is called the "worst case response time feasibility test" and consists in comparing the worst case response time of each thread with its deadline.

For this feasibility test, the thread components of the *Synchronous data flows* design-pattern are defined by three parameters: their deadline (D_i), their period

(P_i) and their capacity (C_i). P_i is a fixed delay between two release times of the thread i. Each time the thread 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 thread release time.

Joseph and Pandia [13] have proposed a way to compute the worst case response time of a thread with pre-emptive fixed priority scheduling by equation:

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

Where r_i is the worst case response time of thread i and $hp(i)$ is the set of threads that have a higher priority level than thread i.

This feasibility test is one of the most simple tests which can be applied to the *Synchronous data flows* design-pattern. This test has several applicability assumptions. For example, this test assumes that deadlines are equal to periods and that all threads have an equal first release time.

3.3 Example of an AADL model compliant with the *Synchronous data flows* design-pattern

```
thread T1 end T1;

thread implementation T1.impl
properties
    Dispatch_Protocol => Periodic;
    Compute_Execution_time => 1 ms .. 2 ms;
    Deadline => 10 ms;
    Period => 10 ms;
    Cheddar_Properties::Fixed_Priority => 128;
end T1.impl;

thread T2 end T2;

thread implementation T2.impl
properties ....
end T2.impl

process implementation process0.impl
subcomponents
  a_T2 : thread T2.impl; ....
end process0.impl

processor implementation rma_cpu.impl
properties
   Scheduling_Protocol => Rate_Monotonic_Protocol;
   Cheddar_Properties::Preemptive_Scheduler => True;
   Cheddar_Properties::Scheduler_Quantum => 0 ms;
end rma_cpu.impl;
```

Fig. 1. Part of an AADL model

Figure 1 shows a part of an AADL model. This example is composed of several periodic threads defined into a process and that are run on a single pro-

cessor. AADL component are always specified by a type definition (line number 1) and a component implementation (from line 2 to line 9). The model also contains several AADL properties. Some properties of thread components define deadlines, periods, capacities and the thread priority. The properties of the processor component defines how the scheduler works. In this example, the processor embeds a preemptive Rate Monotonic scheduler with a quantum equal to zero (from line 24 to line 27).

This AADL model is compliant with the *Synchronous data flows* design-pattern because all rules of the section 3.1 are met. We assume that the designer has previously checked the compliance of this model with this set of rules.

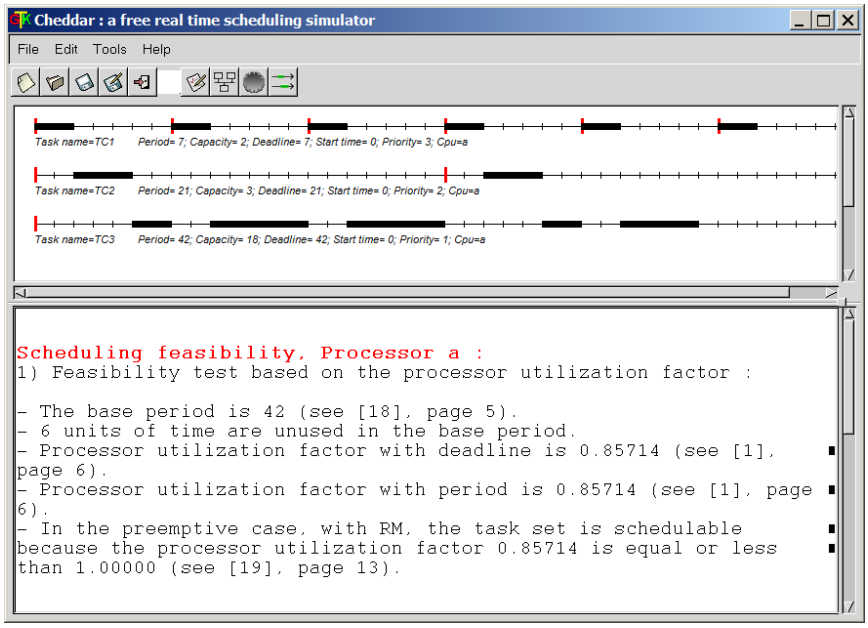

Fig. 2. A screenshot of Cheddar, a tool which implements several feasibility tests

To compute feasibility tests of the *Synchronous data flows* design-pattern, we are using *Cheddar*. *Cheddar* is a framework which aims at providing performance analysis of concurrent real-time applications [25]. With *Cheddar*, a real-time application is modeled as a set of processors, shared resources, buffers and tasks. *Cheddar* is able to handle architecture models described by an AADL specification and also by its own simplified architecture design language. *Cheddar* already implements numerous feasibility tests [23]. Figure 2 shows a screenshot of *Cheddar*. The top part of this window displays the scheduling analysis of the model of figure 1 and the bottom part shows the feasibility tests results computed for this AADL model.

4 A decision tool to check the compliance of an AADL model with the design-patterns

In the previous section, we have presented a design-pattern and an AADL model that is compliant with it. To automatically perform performance verifications of an AADL model with *Cheddar*, we have assumed that designers are able to check the compliance of their AADL models with a design-pattern. This task is, however, not easy for users who are not experts on real-time scheduling theory. To facilitate this task, we propose a second tool, called *decision tool*, which is able to automatically perform this compliance analysis.

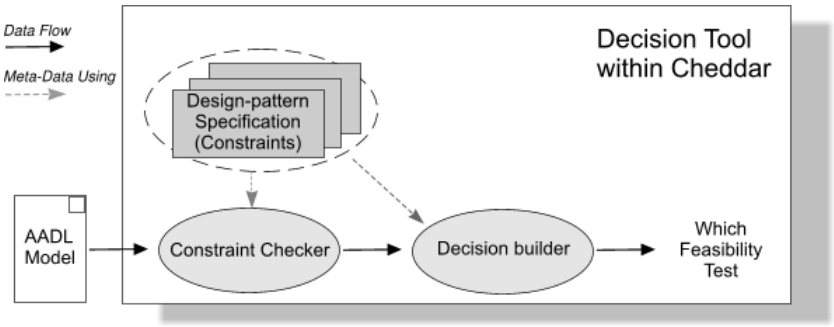

Fig. 3. AADL model analyzer overview

We plan to integrate this decision tool as a new functionality of *Cheddar*. As depicted by figure 3, within *Cheddar*, the decision tool will be able to automatically detect which design-pattern a real-time system is compliant with and then to automatically compute the relevant feasibility tests.

So far, the design-patterns are themselves currently elaborated. Thus, for now we are using a prototype of the decision tool. This prototype is built with the *Platypus* tool. In this section we first briefly describes the *Platypus* tool and its usage for prototyping the decision tool. Then we explain how *Cheddar* will be enriched with the decision tool.

4.1 Prototyping within Platypus

Platypus [20] is a software engineering tool which embeds a modeling environment based on the STEP standard [11]. First of all, *Platypus* is a STEP environment, allowing data modeling with the EXPRESS language [12] and the implementation of STEP exchange components automatically generated from EXPRESS models. *Platypus* includes an EXPRESS editor and checker as well as a STEP file reader, writer and checker.

In *Platypus*, a meta-model consists in a set of *EXPRESS* schemas that can be used to describe a language. The main components of the meta-model are types and entities.

From an EXPRESS schema and a data set made of instances of entities described by the EXPRESS schema, *Platypus* is able to check the data set conformity by evaluating the constraint rules specified in the EXPRESS model.

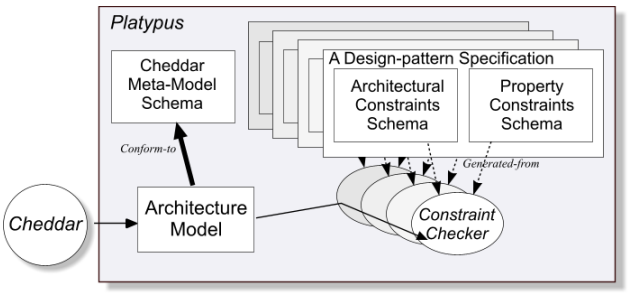

Fig. 4. The decision tool prototype within Platypus

Thus, given that the design-patterns are specified with EXPRESS, the decision tool prototype directly benefits from the *Platypus* STEP generic framework. The figure 4 shows the prototype components and the data flow when an architecture model is analyzed. The prototype is first made of the shared meta-model named *Cheddar meta-model schema*. This meta-model specifies the internal *Cheddar* representation of a real-time architecture to verify. Then, each design-pattern is composed of two models which are defined in order to further constraint the *Cheddar* meta-model. These models correspond to the two kinds of constraints as explained in the section 2. In the figure 4, they are specified by the *architectural constraints* and *property constraints* schemas.

Due to the current usage of the prototype, in order to be analyzed, an AADL architecture model must be encoded as an XML or a STEP data exchange file conforming to the *Cheddar meta-model*. *Cheddar* can be used for that purpose. Then, each design-pattern is evaluated separately. Evaluating a design-pattern consists in interpreting all rules specified in the *architectural constraints* and *property constraints* schemas.

4.2 Design-pattern modeling framework

As depicted by figure 5, the modeling framework is made of three main layers: (1) the *Architecture resources*, (2) the *Feasibility test resources* and (3) the *Feasibility test design-patterns* layers. Each of these layers are made of one or several EXPRESS schemas. This section briefly describes them and gives some illustrative EXPRESS samples.

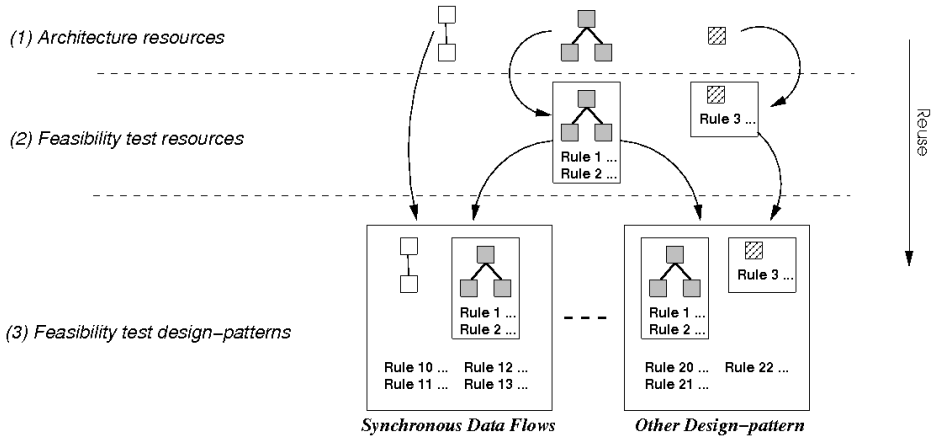

Fig. 5. The three layers of the Design-pattern modeling framework

The *Architecture resources* layer This layer is the most generic one, it contains the specification of all domain entities which are used for architectures modeling. Indeed, it is specified independently of any design-pattern. It mainly contains the Cheddar meta-model. A particular real-time architecture is made of instances of this meta-model because it specifies the internal representation of a real-time architecture within Cheddar. As depicted by the figure 6, it is composed of entities such as *Generic_Task* or *Generic_Scheduler* but also *Buffer* or *Processor*. These entities store all attributes that are required for the analysis of an AADL architecture model.

```
SCHEMA Tasks;
  ENTITY Periodic_Task SUBTYPE OF ( Generic_Task );
    Period : Natural_type;
    Jitter : Natural_type;
  END_ENTITY; ...
END_SCHEMA;

SCHEMA Schedulers;
  TYPE Preemptive_Type = ENUMERATION OF
    (fully_preemptive, non_preemptive, partially_preemptive);
  END_TYPE;

  ENTITY Generic_Scheduler;
    Quantum : Natural_type;
    Preemptivity : Preemptive_Type;
  END_ENTITY; ...
END_SCHEMA;

SCHEMA Processors ... END_SCHEMA;
SCHEMA Buffers ... END_SCHEMA;
```

Fig. 6. Part of the Cheddar Meta-model

The *Feasibility test resources* layer This is the intermediate layer which makes use of entities of the *Architecture resources* layer and in addition is made of new entities, functions and rules which are reusable across several design-patterns.

```
SCHEMA Period_Equal_Deadline_Constraint;
  USE FROM Tasks;

  RULE Period_Equal_Deadline FOR ( Periodic_Task );
  WHERE
    SIZEOF ( QUERY ( p <* Periodic_Task | p.Period <> p.Deadline ) ) =0;
  END_RULE;
END_SCHEMA;
```

The *Period_Equal_Deadline* constraint concerns only the set of all *Periodic_Task* instances. This constraint is satisfied if there is no *Periodic_Task* instance which have a period value which is different from its deadline.

Fig. 7. EXPRESS model of the Feasibility tests resources layer

As an example, the *Period_Equal_Deadline* constraint is shown in the figure 7. This constraint is typically reusable for several design-patterns. It is specified by an EXPRESS rule which ensures that, for each instance of the *Periodic_Task* entity, the value of the *Period* and of the *Deadline* attributes are equal.

The *Feasibility test design-patterns* layer Each model of this layer is called a *design-pattern model*, it concerns a particular design-pattern and is made of one or several EXPRESS schemas which reuse parts of the other layers. In addition, each *design-pattern model* defines very specific rules that represent constraints which have to be checked for the related design-pattern.

```
SCHEMA Data_Flow_Constraints;
  USE FROM Tasks; USE FROM Schedulers; USE FROM Buffers;

  RULE All_Tasks_Are_Periodic FOR ( Generic_Task );
  WHERE
    R1 : SIZEOF ( QUERY ( t <* Generic_Task |
      NOT ('TASKS.PERIODIC_TASK' IN TYPEOF ( t ) ) ) ) =0;
  END_RULE;
  ...
  RULE No_Shared_Ressource FOR ( Generic_Resource, Buffer );
  WHERE
    R5 : ( SIZEOF ( generic_resource ) =0 ) AND ( SIZEOF ( Buffer ) =0 );
  END_RULE;
END_SCHEMA;
```

Fig. 8. EXPRESS model for the Synchronous data flows design-pattern

As an example, the *Data_Flow_Constraints* EXPRESS schema shown in figure 8 is for the *Synchronous Data Flows* design-pattern. Only the rules 1 and 5 given for this design-pattern are shown (see section 3.1 page 4): the *Rule 1* is specified by the *All_Tasks_Are_Periodic* EXPRESS constraint and the *Rule 5* by the *No_Shared_Ressource* one.

4.3 Toward an implementation within Cheddar

As we are currently elaborating EXPRESS models for the work presented in this article. It is very efficient to be able to directly test them from the *Platypus* modeling environment itself.

But having to use *Cheddar* together with *Platypus* is not comfortable for end-users and *Platypus*, as a STEP based data modeling environment, is not user friendly enough. We plan to use a model driven engineering process in order to automatically generate the decision tool in Ada, the implementation language of *Cheddar*. For such a purpose, *Platypys* will have to handle EXPRESS models of figures 6, 7 and 8. Today, *Cheddar* is already partly automatically generated by *Platypus* from EXPRESS models of the figure 6. As an example, the *Cheddar meta-model* schema is used in order to produce the core components of *Cheddar* [19,24].

5 Related works

This article has shown an approach to check that an architectural model of a real-time system is compliant with a set of constraints. Many other approaches also investigated how to perform such verifications.

First, UML together with its standard constraint language OCL could be used for the purpose of designing and building feasibility test checkers.

Second, in [8], Gilles and al. have proposed a similar constraint language for AADL. The proposed language is called REAL (REAL stands for Requirement Enforcement Analysis Language). REAL is developed by Télécom-Paris-Tech and ISAE. It should be adopted as an annex of the AADL standard. This language is then specifically designed for the modeling of real-time architectures. REAL allows to express various type of constraints on AADL architecture and their authors have shown that it can express some of the applicability constraints of the real-time scheduling theory.

Another approach of a similar move towards more analyzable constructs built on top of a modeling language can be found in the history of the HOOD method [15]. The first versions of this modeling approach defined a quite basic concept of component (called HOOD objects) which aimed at representing more or less an Ada 83 package. In 1995, two specializations of HOOD were specified: HOOD 4 [16] which targets Object-Oriented programming languages and especially Ada 95, and HRT-HOOD [2] which goal is to comply with the Ada Ravenscar model (now included into Ada 2005 [27]). In both cases, the original concepts and principles of the HOOD methodology have been kept, and

specific composite constructs have been identified in order to support properly Ada 95 tagged types or Ravenscar cyclic, sporadic and protected objects. More recently, in the context of the IST-ASSERT project, Panunzio and al. [18] proposed to integrate some HRT-HOOD components with UML models. For such a purpose, they have proposed an engineering process based on a meta-model called RCM (RCM stands for Ravenscar Computational Model). In this process, performance verifications are performed with the MAST framework [9], which also implements several feasibility tests.

Finally, PPOOA proposes a similar approach [6]. PPOOA is an architectural style for concurrent object oriented architectures. PPOOA is implemented as an extension of UML and provide several coordination mechanisms such as buffers, semaphores, transporters, Ada rendezvous and others. All these coordination mechanisms are similar to our design-patterns.

6 Conclusion

Feasibility tests of real-time scheduling theory may be difficult to be used by system designers. In this article, we investigate how to increase their usability with an approach based on design-patterns.

In this approach, we have defined a list of design-patterns and a set of feasibility tests is assigned to each design-pattern. When a designer wants to perform a performance analysis of an AADL model, he must check that his model is compliant with one of these design-patterns. If the model is actually compliant with a design-pattern then he can call *Cheddar* to automatically compute the feasibility tests assigned to the selected design-pattern.

However, checking compliance of his models to the design-patterns may be difficult to achieve, especially if designers are not expert on real-time scheduling theory. To automatically check compliance, we propose a framework, called decision tool, which relies on the Platypus environment.

In the current implementation of our approach, designers have to handle two different tools: *Cheddar* and the decision tool. In a next step, we plan to integrate the decision tool into *Cheddar*. Having only one tool to deal with should facilitate the performance analysis of AADL models.

A second future work is related to the list of design-patterns. Indeed, we only have investigated how to check compliance with the *Synchronous data flows* design-pattern. In the next months, we will do the same work for the other design-patterns: *Ravenscar*, *Queued Buffer* and *BlackBoard* [4].

Finally, in this approach, we expect to verify if an AADL model is fully compliant with a set of design-patterns. But in some cases, architectural models of practitioners may be compliant with none of the proposed design-patterns. Then, we plan to investigate how the designers can eventually be helped with a set of metrics. These metrics should allow the designers to compare their AADL models with our design-patterns and to improve their models in order to be compliant with the real-time scheduling theory.

References

1. Arinc: Avionics Application Software Standard Interface. The Arinc Committee (January 1997)
2. Burns, A., Wellings, A.: HRT-HOOD: A Design Method for Hard Real-time Systems. Real Time Systems journal 6(1), 73–114 (1994)
3. Dissaux, P.: Using the AADL for mission critical software development. 2nd European Congress ERTS, EMBEDDED REAL TIME SOFTWARE Toulouse (January 2004)
4. 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 (January 2008)
5. Farail, P., Gaufillet, P., Canals, A., Camus, C.L., Sciamma, D., Michel, P., Crégut, X., Pantel, M.: TOPCASED : An Open Source Development Environment for Embedded Systems. Chapter 11, From MDD Concepts to Experiments and Illustrations, ISTE Editor pp. 195–207 (September 2006)
6. Fernandez, J.L., Marmol, G.: An Effective Collaboration of a Modeling Tool and a Simulation and Evaluation Framework. 18th Annual International Symposium, INCOSE 2008. Systems Engineering for the Planet. The Netherlands. (June 2008)
7. Gallmeister, B.O.: POSIX 4 : Programming for the Real World . O'Reilly and Associates (January 1995)
8. Gilles, O., Hugues, J.: Expressing and enforcing user-defined constraints of AADL models. pp. 337–348. International workshop on AADL and UML. In the proceedings of the 15th IEEE International Conference on Engineering of Complex Computer Systems, University of Oxford, UK (March 2010)
9. Harbour, M.G., Garcìa, J.G., Gutiérrez, 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, (June 2001)
10. Hugues, J., Zalila, B., Pautet, L., Kordon, F.: 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 (Jun 2007)
11. ISO 10303-1: Part 1: Overview and fundamental principles (1994)
12. ISO 10303-11: Part 11: edition 2, EXPRESS Language Reference Manual (2004)
13. Joseph, M., Pandya, P.: Finding Response Time in a Real-Time System. Computer Journal 29(5), 390–395 (1986)
14. 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 (January 1973)
15. Masson and Prentice-Hall: HOOD Reference Manual release 3.1, HOOD User Group (1993)
16. Masson and Prentice-Hall: HOOD Reference Manual release 4.0, HOOD User Group (1995)
17. OMG: A UML Profile for MARTE, Beta 1. OMG Document Number: ptc/07-08-04 (Aug 2007)
18. 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 (June 2007)

19. 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 (November 2006)
20. Platypus Technical Summary and download. http://cassoulet.univ-brest.fr/mme/ (2007)
21. 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 2.0 (January 2009)
22. SEI: OSATE : An extensible Source AADL Tool Environment. SEI AADL Team technical Report (December 2004)
23. Singhoff, F.: A taxonomy of real-time scheduling theory feasibility tests. LISyC Technical report, number singhoff-01-2010, Available at http://beru.univ-brest.fr/~singhoff/cheddar (Feb 2010)
24. 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 (June 2007)
25. Singhoff, F., Plantec, A., Dissaux, P., Legrand, J.: Investigating the usability of real-time scheduling theory with the Cheddar project. Journal of Real-Time Systems, Springer Verlag 43(3), 259–295 (November 2009)
26. Sokolsky, O., Lee, I., Clark, D.: Schedulability Analysis of AADL models . International Parallel and Distributed Processing Symposium, IPDPS 2006, Volume 2006, (Apr 2006)
27. 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)
28. Tanenbaum, A.: Modern Operating Systems. Prentice-Hall (2001)