An Ada design pattern recognition tool for AADL performance analysis

Vincent Gaudel
Frank Singhoff
Alain Plantec
Stéphane Rubini
LISyC/EA 3883, University of Brest
20, av Le Gorgeu
CS 93837, 29238 Brest Cedex 3, France
{gaudel,singhoff,plantec,rubini}@univ-brest.fr

Pierre Dissaux
Jérôme Legrand
Ellidiss Technologies
24, quai de la douane
29200 Brest, France
{pierre.dissaux,jerome.legrand}@ellidiss.com

ABSTRACT

This article deals with performance verification of architecture models of real-time embedded systems. Although real-time scheduling theory provides numerous analytical methods called feasibility tests for scheduling analysis, their use is a complicated task. In order to assist an architecture model designer in early verification, we provide an approach, based on real-time specific design patterns, enabling an automatic schedulability analysis. This analysis is based on existing feasibility tests, whose selection is deduced from the compliance of the system to a design pattern and other system's properties. Those conformity verifications are integrated into a schedulability tool called Cheddar. We show how to model the relationships between design patterns and feasibility tests and design patterns themselves. Based on these models, we apply a model-based engineering process to generate, in Ada, a feasibility test selection tool. The tool is able to detect from an architecture model which are the feasibility tests that the designer can apply. We explain a method for a designer willing to use this approach. We also describe the design patterns defined and the selection algorithm.

Keywords

AADL, Cheddar, Ada framework, Design Patterns, Real-time schedulability analysis, Platypus

General Terms

Performance, Reliability, Verification.

Categories and Subject Descriptors

SOFTWARE ENGINEERING [**Software/Program Verification**]: Validation

SIGAda'11, November 6–10, 2011, Denver, Colorado, USA.

1. INTRODUCTION

A real-time system is called critical when its dysfunction could lead to serious damages upon persons or the environment. The validation of such systems is therefore crucial. Real-time scheduling theory provides tools to validate such systems, especially analytical methods called *feasibility tests*. Yet, their use implies a high level of expertise in the real-time scheduling theory: the common draw to all these feasibility tests is that they need the system to fulfill a set of specific assumptions called *applicability constraints*. Those are based on architectural and environmental properties. Unfortunately, the large number of feasibility tests and applicability constraints complicates the use of such methods. That may explain why they are unused in many practical cases, although it could be profitable.

In our approach, we aim to help real-time architecture designers by automatically select relevant feasibility tests. To do so, we explicitly model relationships between architectural models and real-time scheduling analysis. The modeling of relationships is provided by a set of real-time design patterns for whom we are able to select applicable feasibility tests [5, 13].

We have provided five design patterns based on AADL (*Architecture Analysis and Description Language* [15]) communication and synchronization protocols between tasks. We model those design patterns by sets of assumptions on properties of architectural models. We analyze architectural models to verify their compliance to design patterns. When an architecture is compliant with one design pattern, we automatically propose to the designer to compute a set feasibility tests assigned to the corresponding design pattern.

In this article, we present a prototype which is able to select feasibility tests applicable to an architecture model compliant to a design pattern. This prototype is integrated with the performance analysis environment Cheddar and is generated by a model driven engineering process. We present our approach from a designer's perspective. We show how to model relationships between architectural properties and a set of feasibility tests. Then, we present an algorithm to analyze compliance of an architecture model to a design pattern and finally, we explain how we can generate our Ada prototype from our feasibility tests and design pattern models.

This article is organized as follows. In section 2 we present

the approach from the designer's point of view. Section 3 contains a description of our five design patterns. Section 4 presents the architecture model of a simplified automotive system as a case study. Then, in section 5 the algorithm is presented. In section 6 we explain our engineering process and present the prototype evaluation. Related works follow in section 7, before conclusion in section 8.

2. PROPOSED APPROACH

In this section, we describe what we call feasibility tests and applicability constraints. Then we give a brief presentation of our five design patterns, followed by the description of the proposed method from the designer's perspective.

2.1 The issue of critical system schedulability analysis

Real-time scheduling theory enables designers to analyze the temporal behavior of a set of tasks with the use of analytical methods called feasibility tests. For example, Liu and Layland [10] have defined periodic tasks characterized by three parameters: deadline (D_i), period (P_i), and capacity (C_i). Each time a task i is activated, it has to perform a job whose execution time is bounded by C_i units of time before D_i units of time after the task release time. Those parameters are used to compute some feasibility tests. Feasibility tests evaluate different performance criteria: processor utilization factor, worst case response time, deadlocks and priority inversions due to data access, memory footprint analysis, etc. Liu *et al.* have proposed equation (1) to perform schedulability analysis. This feasibility test computes the processor utilization factor for a system compliant with the following assumptions: all tasks must be periodic, independent and synchronous [7]; the scheduling protocol must be preemptive *Earliest Deadline First* or preemptive *Least Laxity First*.

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1 \quad (1)$$

This feasibility test is a necessary and a sufficient condition if $\forall i : \mathrm{D}_i = \mathrm{P}_i$.

We see that applying a feasibility test to a system implies that the system has to meet a set of architectural assumptions. In the next sections, those assumptions will be called *"applicability constraints"*. An applicability constraint expresses properties of architecture components such as task periodicity, scheduling protocol, communication protocol, etc.

2.2 Design pattern description

In order to select applicable feasibility tests, we have defined five real-time design patterns: *Synchronous data-flow*, *Ravenscar*, *Blackboard*, *Queued buffer* and *Unplugged*. Each of those design patterns provides an architectural solution to a synchronization problem between tasks by defining an inter-task communication protocol. There exists numerous other synchronization paradigms that could justify such specification. We use a pragmatic approach: the previous design patterns are all related to standards or industrial practices. For example, *Synchronous data-flow* implements Meta-H communication paradigms [20]. *Ravenscar* is related to the Ada 2005 Ravenscar profile [2]. *Blackboard* and *Queued buffer* model communication services that exist in ARINC 653 [1].

We assume that architecture models potentially compliant with our design patterns are written in AADL (*Architecture Analysis and Description Language* [15]). AADL is a textual and graphical architecture design language for model-based engineering of real-time systems.

The five design patterns work as follows:

1. **Synchronous data-flow** achieves data sharing through AADL data ports. Each task reads its input data ports at dispatch time and writes its output data ports at completion time. The execution platform deals with the data synchronization: there is no explicit protocol needed.

2. In **Ravenscar**, data are shared asynchronously, under the control of inheritance ceiling priority protocol. Ravenscar allows tasks to share data protected by semaphores. Semaphores can be used to build multiple synchronization protocols such as critical sections, readers/writers, producers/consumers, etc.

3. **Blackboard** implements the readers/writers communication protocol: only the last value produced can be consumed by tasks.

4. **Queued buffer** implements the producers/consumers communication protocol: messages are handled by a FIFO protocol.

5. **Unplugged** models independent tasks: tasks that do not communicate with each other.

2.3 Method from designer perspective

In the previous section, we have described feasibility tests and real-time design patterns. In this one, we define how we intend to assist real-time architecture designers. Figure 1 describes the method for designing an architecture model with the use of our design patterns. (1) A designer provides an architecture model. (2) An analysis is performed upon the architecture model in order to check its compliance to one of our design patterns. (4) Once the compliance to a design pattern is met, we provide a list of feasibility tests, (6) and we compute them to perform schedulability analysis.

During each of steps 2, 4 and 6, the designer may have to come back to the design phase. First at step (3), when we check the compliance of the architecture model to a design pattern: if it is not compliant, we are not able to provide a list of feasibility tests, so we give to the designer information (metrics and non-met applicability constraints) in order to assist him to fulfill the compliance of his architecture model to a real-time design pattern. During step (4), if the list of feasibility tests does not satisfy the designer (he might want to use a particular feasibility test or the list of selected feasibility tests may not be sufficient to prove the schedulability of the system), we provide alternative lists of feasibility tests and the reasons for their non-selection (5). Finally, once the schedulability analysis has proceeded, the designer may have to loop if the model is not schedulable (7).

3. HOW WE MODEL DESIGN PATTERNS

In order to perform the analysis explained in the previous section, we model different entities implied in the feasibility

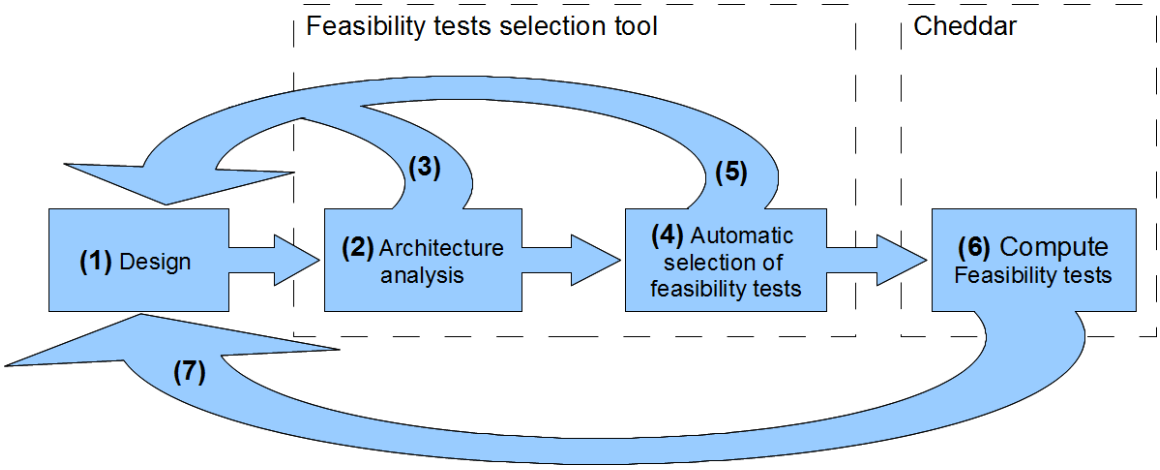

Figure 1: Method from the designer's perspective.

tests selection. In this section, we first present the language we use to model our design patterns. Then we describe how we model the relationships between architectural models and feasibility tests.

```
SCHEMA Tasks;
  ENTITY Generic_Task
    ...
    Capacity : Natural;
    Deadline : Natural;
    Start_Time : Natural;
    Priority : Priority_Range;
    ...
  END_ENTITY;
...
  ENTITY Periodic_Task SUBTYPE OF ( Generic_Task );
    Period : Natural_type;
    Jitter : Natural_type;
  END_ENTITY; ...
END_SCHEMA;
```

Figure 2: Excerpt from Cheddar meta-model: Generic_Task and Periodic_Task entities.

Our approach is developed within the Cheddar Project [18], an analysis environment for real-time applications written in Ada. Cheddar already implements numerous feasibility tests. Yet, an architecture designer has to select feasibility tests applicable to his architecture model, which is a hard task.

Cheddar provides an architecture language to model real-time systems. The architecture language is based on a meta-model. An instance within this meta-model is composed of several sets of entities: tasks, resources, dependencies, processors, buffers, networks and address spaces. The Cheddar meta-model is written in the EXPRESS language [19] and provides all the tools we need to model real-time design patterns. Figure 2 gives an excerpt of this meta-model. A generic task is defined by its capacity, its deadline, its release time and priority among others. A periodic task is defined as a generic task for which period and jitter are defined.

3.1 Structure of the EXPRESS models

In order to make explicit the relationships between architectural models and feasibility tests, we model design patterns by the applicability constraints they meet. Thus,

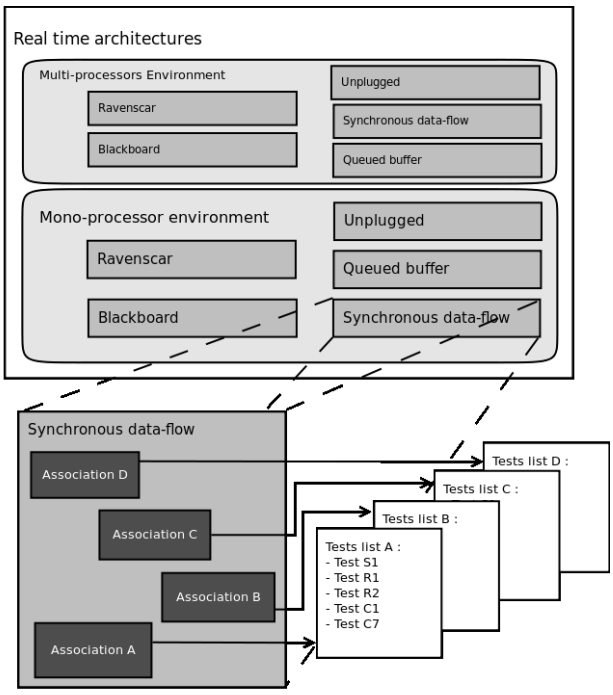

Figure 3: List of applicability constraints for feasibility tests selection.

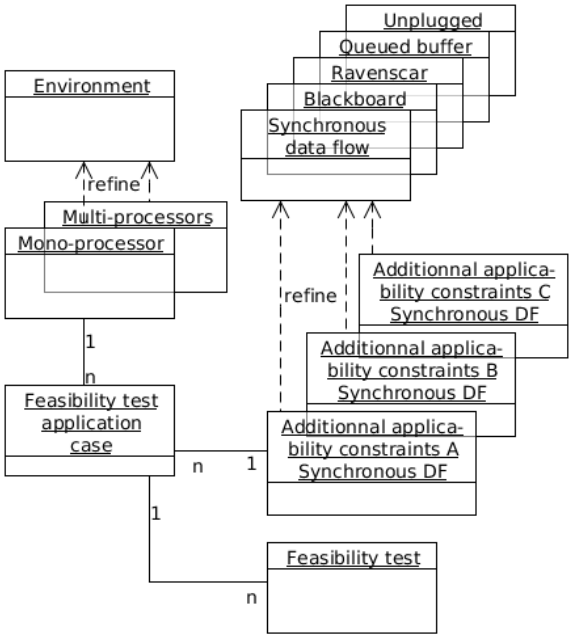

Figure 4: Association Model.

design patterns are modeled, with EXPRESS, by sets of applicability constraints structured as follows.

Figure 3 represents the set of all real-time architecture types we can model with the Cheddar meta-model. Each box contains a subset of real-time architectures that meets the same applicability constraints. The more the background color of the boxes is dark, the more the number of met applicability constraints increases. There are various types of applicability constraints. Some are relative to the deployment environment: one cannot use the same feasibility tests for a mono-processor environment or a multi-processor environment (light gray). Others are relative to the used design pattern (medium gray). For each design pattern, we define several *associations* (dark gray) between additional applicability constraints and feasibility tests (126 for Synchronous data-flow). Applicability constraints are structured by environments, design patterns and associations. Figure 4 shows the association between those entities and feasibility tests.

3.2 How we model applicability constraints

For each subset of real-time systems (see. 3.1), we model all the applicability constraints with EXPRESS. For example, Synchronous data-flow applicability constraints on an architecture model are: (1) all tasks are periodic, (2) there is no buffer, (3) there is no data component, (4) data sharing protocol is sampled, immediate or delayed[1] and (5) there is no hierarchical scheduler (which means no shared address space between processors). EXPRESS enables the definition of *OCL(Object Constraint Language)-like* constraints [21] on the meta-model instances. Figure 5 gives the example of Synchronous data-flow applicability constraint number (1). This constraint is checked for all generic_task instances. It returns true when the size of the set of non-periodic tasks of the architecture model is equal to zero. The Cheddar meta-model is enriched by an *OCL-like* constraint for each applicability constraint.

[1]Sampled, immediate and delayed are AADL communication protocols [15].

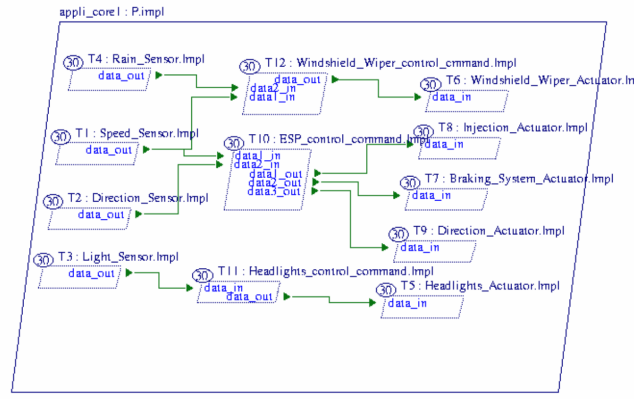

Figure 6: AADL graphic representation of the automotive architecture model.

4. CASE STUDY: EXAMPLE OF AN ARCHITECTURE MODEL COMPLIANT WITH SYNCHRONOUS DATA-FLOW

In this section, we present a case study of an architecture model compliant with Synchronous data-flow that will be used to illustrate the feasibility tests selection algorithm.

Let's consider a simplified version of an automotive embedded system. The architecture model is written in AADL. Figure 6 gives an AADL graphic representation of its architecture model. This system implements three functions: headlights control, windshield wiper control and ESC (Electronic Stability Control) which is a trajectory correction system. It contains twelve tasks communicating with each other through AADL data-ports. Each task reads its input memory slots at dispatch time and writes its output memory slots at completion time. There are four tasks reading sensor values (rain, speed, direction and light sensors) and five tasks sending instructions to actuators (windshield wiper, injection, braking system, direction and headlights actuators). The windshield wiper control task receives data from rain and speed sensors reading tasks and transmits a value to the task communicating with the windshield wiper actuator. The headlights control task receives data from light sensor reading task and transmits a value to the task communicating with the headlights actuator. The ESC control task receives data from speed and direction sensors reading tasks and transmits values to tasks communicating with injection, braking system and direction actuators.

All tasks have a 30ms period, a 30ms deadline and a 2ms capacity. The system is deployed on a mono-processor environment. All tasks are simultaneously released and the scheduling protocol is preemptive deadline monotonic. All five applicability constraints relative to the Synchronous data-flow design pattern are met: no buffer, nor data component, data sharing protocol is sampled and there are no shared address space between processors.

```
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;
```

Figure 5: Example of applicability constraint modeled in EXPRESS: Synchronous data-flow applicability constraint number (1).

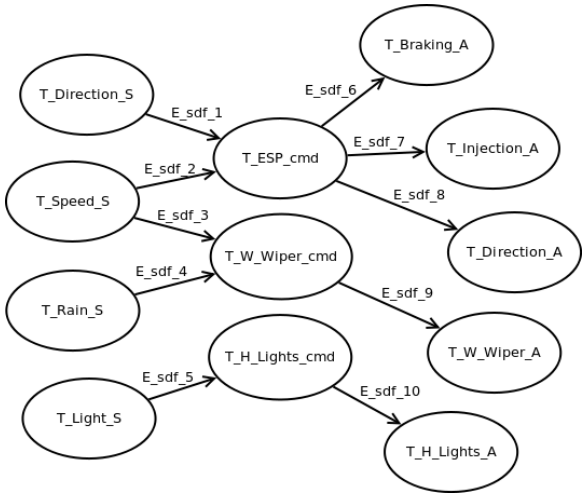

Figure 7: Graphic representation of dependency graph built in step 1.

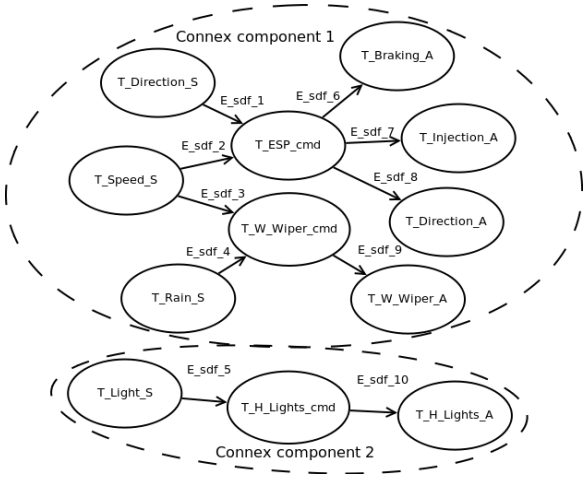

Figure 8: Connectivity components for synchronous data-flow edge in dependency graph built in step 1.

5. FEASIBILITY TESTS SELECTION ALGORITHM

In order to evaluate our approach, we implemented in Ada a prototype integrated into Cheddar. This section describes the algorithm used for the prototype. As we explained in section 2.3, the prototype analyses the compliance of an AADL architecture model to a design pattern, informs the designer of potential modifications of his architecture model and selects a list of feasibility tests. This algorithm is composed of four steps. In this section, we will describe it step by step and apply the steps to the case study presented in the previous section.

5.1 Step 1 : Model analysis to build Graph

The AADL architecture model is parsed by Cheddar, and stored as a set of Cheddar meta-model entities. Cheddar also computes dependencies between tasks for each data port communication, precedence constraint, shared data, message queue, etc.

An architecture model instance can be composed of several design patterns. Thus we have to identify which part of the system is a design pattern instance and the number and types of each design pattern instances. To do so, the first step builds a directed graph containing all tasks of the system and dependencies between those tasks. Thus, we create one node per task and one edge per dependency. Each dependency models a task synchronization or communication. There is one type of node and one type of edge for each dependency type (and then for each type of synchronization/communication).

Figure 7 gives the graphical representation of the dependency graph resulting of step 1 for the example of section 4. It contains twelve nodes and ten Synchronous data-flow edges.

5.2 Step 2 : Graph analysis to extract potential instances

Once the dependency graph is built, we look for subgraphs which are design pattern instances. Extracted subgraphs are called *potential instances*. We extract connectivity components for each type of edge. A connectivity component for one type of edge with no intersection with any other connectivity component is considered as a design pattern potential instance. If there is no potential instances, nodes and edges shared between two connectivity components are identified. Tasks and dependencies corresponding to those nodes and edges are the ones the designer has to modify in order to make his architecture model compliant with a design pattern.

Figure 8 represents the two connectivity components of the case study. The first one contains the three nodes relative to the headlights control tasks, and the second one the rest of the graph. Thus, there are two potential synchronous data-flow instances in the example.

5.3 Step 3 : Instantiation Confirmation

Each design pattern has a set of constraints. Extracted potential instances need to be analyzed in order to confirm actual design patterns instances. To do so, all potential instances are checked one by one by a verification of the applicability constraints relative to the design pattern.

For the simplified car system case study, all five applicability constraints of synchronous data-flow (see section 3.2)

```
...
SCHEMA ASSOCIATION_17;
  USE FROM Mono_Processor_Environment;
  USE FROM Synchronous_Data_Flow;
  USE FROM Simultaneous_Release_Time_Constraint;
  USE FROM Period_Equal_Deadline_Constraint;
  USE FROM Preemptive_Deadline_Monotonic ;
  USE FROM Feasibility_tests_Taxonomy ( test_S1, test_C5,
      test_C7, test_R1, test_R2 );

END_SCHEMA;
....
```

Figure 9: EXPRESS model of an association between feasibility tests, execution environment, design pattern and applicability constraints.

```
package body All_Tasks_Are_Periodic is

  function R1_QUERY1_Condition (t : Generic_Task_Ptr) return
      boolean is
  begin
    return not (Element_In_List (To_Unbounded_String("TASKS.
        PERIODIC_TASK"),Type_Of(t)));
  end R1_QUERY1_Condition;

function R1 (Sys : System) return boolean is
begin
  Context := Sys;
  return (Get_Number_Of_Elements(Select_And_Copy (sys.Tasks,
      R1_QUERY1_Condition'Access)) = Generic_Task_Set.
      Element_Range(0));
end R1;

end All_Tasks_Are_Periodic;
```

Figure 10: Generated Ada package for a given applicability constraint: "All task are periodic".

are checked for the two potential instances. Synchronous data-flow applicability constraints are met in our automotive system example. Then, two Synchronous data-flow instances are identified in this example.

5.4 Step 4 : Applicability constraints verification for feasibility tests selection

At the previous steps, we have verified that our architecture model is compliant with a design pattern. Now, we must find the feasibility tests that the designer is allowed to apply.

An association between a design pattern, its applicability constraints, an execution environment and a list of feasibility tests is modeled by an EXPRESS entity. Figure 9 gives an example of this type of entity for our automotive case study. Each **USE FROM** EXPRESS statement models one or several applicability constraints. If all **USE FROM** statements are met, it means that the feasibility tests listed in the EXPRESS entity can be applied on the architecture model.

In our automotive case study, five feasibility tests have been selected: one based on exhaustive simulation (test_S1), two based on processor utilization factor (test_C5, test_C7) and two based on worst case response time (test_R1, test_R2)[16].

6. PROTOTYPING AND EVALUATION

6.1 The prototype engineering protocol

One goal of our approach is to enable the creation of new design patterns at a meta-level. Thus, one can model a new design pattern and automate additional real-time scheduling theory knowledge. Figure 11 presents the prototype engineering/evaluation protocol. Based on the design patterns modeling, we first implemented manually a prototype integrated to Cheddar. This manually written version has been used to validate the design patterns modeling and the feasibility test selection algorithm. Once the prototype is evaluated, the Cheddar meta-model is enriched and checked within Platypus [14]. The Ada code of a new version of the prototype is then generated [12, 17]. For example, the Ada code generated for the applicability constraint presented in figure 5 is shown in figure 10.

At the time of the writing of this article, the prototype was able to recognize and select feasibility tests for the three design patterns: Synchronous Data Flow, Ravenscar and Unplugged, in the case of a mono-processor environment.

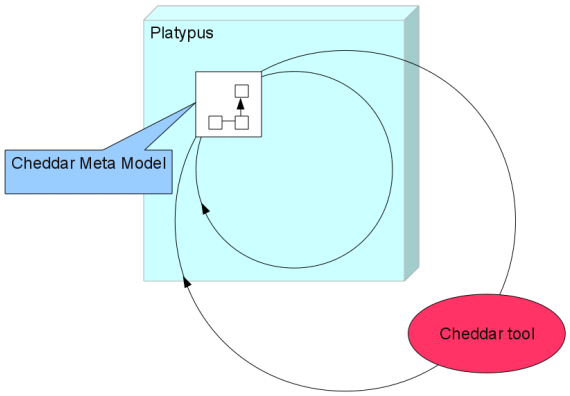

Figure 11: Prototype engineering protocol.

6.2 Evaluation

For the evaluation of our approach and the prototype, we aim to validate multiple points: the prototype itself (robustness, scaling, generated applicability constraints, ...) and the approach. Thus, the evaluation is divided in two parts. The first one consists in testing the implementation of the prototype with generated architectures covering all applicability constraints and design patterns. The second one consists in testing our approach with case studies.

6.2.1 Evaluation of generated architectures

We have designed an architecture generator. Architectures generation is parameterizable. The parameters are the elements of generated architectures: for each generation, we choose which elements to generate and their numbers. The elements taken into account are the following: processor, core, task, buffer, dependency, message, resource and address space.

First, we generate architectures compliant with one design pattern, with a varying number of tasks and communication. Second, we generate architectures that are not compliant with any design patterns: an architecture model is generated for each applicability constraint. For example, a synchronous data flow instance with an extra sporadic task is generated in order to evaluate the constraint *all tasks*

are periodic. At last set of architecture model is generated randomly. Figure 12 gives an overview of the number of generated architectures.

In the sequel, we selected feasibility tests for each generated architecture. The names of selected feasibility tests for each architecture, or the non-compliance to a design pattern, are stored in a file. Based on the AADL documentation, the selection or non-selection of feasibility tests for each architecture is then manually validated.

Design Patterns	Number of Generated architectures
Synchronous_data_flow	40
Ravenscar	40
Unplugged	40
Uncompliant	16
Random	40

Figure 12: Number of generated architectures compliant, or not, with a given design pattern.

Figure 13 gives the response time of the design pattern recognition algorithm, depending on the number of tasks and dependencies. This evaluation has been performed under Ubuntu 10.04 on a processor Intel(R) Dual CPU T2330 1.60GHz with 2,0GiB RAM memory. This evaluation shows that our prototype is robust to scaling, which is important from an industrial perspective. Moreover, the computation time required to check the compliance and to select feasibility tests is linear to the number of tasks in the system. For the second set of generated architectures, all the unmet applicability constraints have been found. As pointed by figure 13, the response time of the design pattern recognition algorithm for large systems (1000 tasks and 1000 dependencies) is reasonable for an interactive design tool.

6.2.2 *Evaluation of case studies*

We have investigated two case studies : the automotive system presented in section 4 and the mars pathfinder architecture [4]. The feasibility tests selected for the automotive system are described in section 5.4. Those five feasibility tests are enough to prove the schedulability of the system.

The mars pathfinder architecture is an instance of the Ravenscar design pattern. It contains seven tasks. Four of them share a data component. The access to the shared data is made by a mutex. The tasks's priorities are static and the scheduling protocol is Rate Monotonic. With the use of our approach, we select five feasibility tests : one based on exhaustive simulation (test_S1), two based on processor utilization factor (test_C3, test_C4) and two based on worst case response time (test_R1, test_R2)[16].

7. RELATED WORKS

Multiples approaches also investigate the domain of constraints checking within real-time systems.

Gilles *et al.* define a constraint language for AADL called REAL (Requirement Enforcement Analysis Language)[8]. REAL is developed by Telecom-Paris-Tech and ISAE and should be adopted as an AADL standard annex. REAL enables one to express different kinds of constraints directly on AADL architecture models. Authors have shown it could be used to express applicability constraints similar to those we aim to model. OCL (Object Constraint Language) enables to express the same constraints we use, but for UML models[21]. Another specification approach related to our design patterns can be found in the HOOD method and in the definition of HRT-HOOD whose goal is to comply with the Ada Ravenscar model [3]. Panunzio *et al.* define an engineering process based on the RCM meta-model (Ravenscar Computation Model)[11]. Performance verifications are proceeded by the MAST framework [9]. Like Cheddar, MAST implements several feasibility tests. Finally, PPOOA proposes an approach similar to our design patterns [6]. PPOOA is implemented as an UML extension and provides a set of predefined synchronization mechanisms. Moreover, the authors underline the importance of applying feasibility tests early in the design process and use Cheddar to do so. Each of those methods studies the validation of a set of real-time architectures with a static number of design patterns. In our approach, we would like to let the designer specify new design patterns and automatically generate the validation tool.

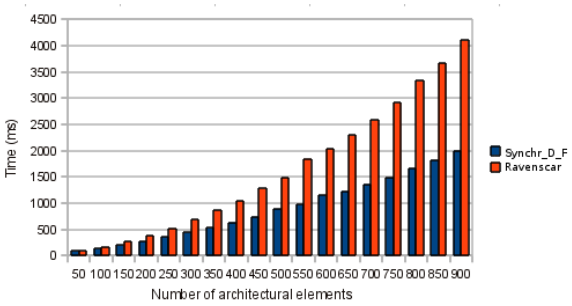

Figure 13: Response times for the design pattern recognition algorithm in function of the number of tasks and dependencies (number of tasks = number of dependencies).

8. CONCLUSION

In this article, we have presented an approach enabling the automatic selection of feasibility tests in order to do the verification of real-time systems. We have proposed a design method. We use design patterns based on inter-task communication protocols to reduce the diversity of architecture models. Thus, an algorithm was defined to check the compliance of the architecture model to a design pattern and to select a specific set of feasibility tests. In order to evaluate our work, we have generated code within Cheddar and made a prototype. Now, we will look for the analysis of more complex systems, resulting of design patterns composition. For now, we are able to compose design pattern two by two based on static rules. A second future work consists in designing a simple process to define new design patterns.

9. ACKNOWLEDGMENTS

We would like to thank Ellidiss Technologies and Region Bretagne for their support to this project.

10. REFERENCES

[1] Arinc. *Avionics Application Software Standard Interface*. The Arinc Committee, January 1997.

[2] A. Burns, B. Dobbing, and G. Romanski. The ravenscar tasking profile for high integrity real-time programs. In Lars Asplund, editor, *Reliable Software Technologies Ada-Europe*, volume 1411 of *Lecture Notes in Computer Science*, pages 263–275. Springer Netherlands, 1998.

[3] A. Burns and A.J. Wellings. *HRT-HOOD: a structured design method for hard real-time Ada systems*, volume 3. Elsevier Science, 1995.

[4] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Scheduling in real-time systems*. Wiley Online Library, 2002.

[5] P. Dissaux and F. Singhoff. 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.

[6] J. Fernández Sánchez and G. Mármol Acitores. Modelling and Evaluating Real-Time Software Architectures. *Reliable Software Technologies–Ada-Europe 2009, LNCS Springer, Volume 5570*, pages 164–176, 2009.

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

[8] O. Gilles and J. Hugues. Expressing and enforcing user-defined constraints of AADL models. In *2010 15th IEEE International Conference on Engineering of Complex Computer Systems*, pages 337–342. IEEE, 2010.

[9] G. Harbour, G. Garcia, P. Gutierrez, D. Moyano, et al. MAST: Modeling and analysis suite for real time applications. In *Real-Time Systems, 13th Euromicro Conference on, 2001.*, pages 125–134. IEEE, 2002.

[10] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[11] M. Panunzio and T. Vardanega. A metamodel-driven process featuring advanced model-based timing analysis. In *Reliable software technologies: Ada-Europe 2007: 12th Ada-Europe International Conference on Reliable Software Technologies, Geneva, Switzerland, June 25-29, 2007: proceedings*, pages 128–141. LNCS Springer-Verlag New York Inc, Volume 4498, 2007.

[12] A. Plantec and F. Singhoff. 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.

[13] A. Plantec, F. Singhoff, P. Dissaux, and J. Legrand. Enforcing applicability of real-time scheduling theory feasibility tests with the use of design-patterns. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6415 of *Lecture Notes in Computer Science*, pages 4–17. Springer Berlin / Heidelberg, 2010.

[14] Platypus Technical Summary and download. http://cassoulet.univ-brest.fr/mme/, 2007.

[15] SAE. Architecture Analysis and Design Language (AADL) AS-5506A. Technical report, The Engineering Society For Advancing Mobility Land Sea Air and Space, Aerospace Information Report, Version 2.0, January 2009.

[16] F. Singhoff. 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, February 2010.

[17] F. Singhoff and A. Plantec. 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.

[18] F. Singhoff, A. Plantec, P. Dissaux, and J. Legrand. Investigating the usability of real-time scheduling theory with the Cheddar project. *Real-Time Systems*, 43(3):259–295, 2009.

[19] P. Spiby. ISO 10303 industrial automation systems–product data representation and exchange–part 11: Description methods: The express language reference manual. *ISO DIS*, pages 10303–11, 1992.

[20] S. Vestal. Meta-H User's Manual, Version 1.27. Technical report, download at http://www.htc.honeywell.com/metah/uguide.pdf, 1998.

[21] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.