Composition of Design Patterns : from the modeling of RTOS synchronization tools to schedulability analysis

Vincent Gaudel, Frank Singhoff, Alain Plantec
Université Européenne de Bretagne, France
Université de Bretagne Occidentale
Lab-STICC/CNRS UMR 6485
20 avenue Le Gorgeu, 29238 Brest, France
{gaudel,singhoff,plantec}@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. We investigate scheduling analysis of multi-tasks applications running on real-time operating systems (RTOS in this article). Scheduling analysis on these types of system can be performed with the real-time scheduling theory, but applying it is a complicated task. To allow designer to automatically apply this theory, we propose several architectural design patterns. Each architectural design pattern models a classical task synchronization or communication protocol available in RTOSes. In this article, we focus on those design patterns composition. We show how to compose the proposed design patterns and how scheduling analysis can be run with them.

1. INTRODUCTION

This article deals with critical real-time systems. The validation of such systems is crucial and various means have been proposed for such purpose. Real-time scheduling provides tools to validate such systems, especially analytic 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*. Unfortunately, the large number of feasibility tests and applicability constraints make the use of such methods difficult. That may explain why they are unused in many practical cases, although it could be profitable [1].

We propose a method to assist designers in the process of selecting feasibility tests. To do so, we present five architectural design patterns. Each of these design patterns models a communication or a synchronization protocol between tasks that usually exist in real-time operating systems. We define those design patterns by sets of assumptions on properties of architectural models that models both the software and the executive environment (e.g. RTOS and the underlying hardware) of a critical real-time system. We propose an algorithm able to verify whether an architecture model is compliant with one of our design patterns, i.e. whether the architecture model meets all the assumptions defining the design pattern. In case of compliance, feasibility tests can be automatically suggested to the architectural model designer.

In this article, the contribution is double. We provide a detailed definition of these architectural design patterns. We also focus on the analysis of architecture models composed of several of our design patterns. Indeed, an application running on top of a real-time operating system may use several communication and synchronization tools.

This paper is organized as follows. In section 2 we present our approach. Section 3 contains a detailed description of our architectural design patterns. Section 4 discuss the composition of such design patterns. In section 5 we expose an evaluation of the approach. Related works follow in section 6, before conclusion in section 7.

2. APPLYING REAL-TIME SCHEDULING THEORY TO REAL-TIME ARCHITECTURE MODELS

In this section, we present our approach. First we introduce what we mean by feasibility test. Then, we present Cheddar, the schedulability analysis tool we use to verify schedulability and to model architectures of real-time systems. Then, we describe how we make possible to automatically apply Cheddar on real-time system architecture models.

2.1 Feasibility Tests

Real-time scheduling theory enables designers to analyze the temporal behavior of a set of tasks with the use of analytic methods called feasibility tests. For example, Liu and Layland [2] have defined a simple model of tasks, called periodic tasks, that are only characterized by three parameters : a deadline (D_i), a period (P_i), and a capacity (C_i). A periodic task models a function of the system that has the following behavior : each time a periodic task T_i is activated, it has to perform a job whose execution time is bounded by C_i units of time and this job must be completed before D_i units of time after the corresponding task release time.

Those parameters can be used to compute 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 a simple feasibility test shown by equation (1). This feasibility test computes the processor utilization factor for a system compliant with the following assumptions: all tasks are periodic, independent and synchronous; the scheduler is a preemptive *Earliest Deadline First*.

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

If this equation is met, it implies that all task's deadlines will be met at execution time. This feasibility test is a necessary and a sufficient condition when $\forall\ T_i : \mathrm{D}_i = \mathrm{P}_i$.

We see that applying a feasibility test to a system requires 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 can express various properties of architecture components such as task periodicity, scheduling protocol, communication protocol, etc.

2.2 Cheddar: a schedulability tool

Our approach is developed within the Cheddar Project [3], an analysis environment for real-time systems. Cheddar already implements numerous feasibility tests. Yet, an architecture designer has to select feasibility tests applicable to his architecture model, which is complicated. Cheddar provides an architecture language, called Cheddar ADL, to model real-time systems. This architecture language is based on a meta-model. The Cheddar ADL meta-model is written in the EXPRESS language [4] and provides all the tools we need to model our architectural design patterns. EXPRESS enables the definition of *OCL(Object Constraint Language)-like* constraints [5] on the meta-model instances.

2.3 Automatic selection of feasibility tests to verify architecture models

To help a system designer to automatically select feasibility tests that are compliant with his architecture model, we have defined five real-time architectural design patterns called: *Time-Triggered*, *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 that is implemented in usual RTOS. There exists numerous other synchronization paradigms that could justify such specifications. In a first approach, we have selected inter-task synchronization/communication paradigms that are either classic industrial practices, or defined by standards related to RTOS.

For example, the *Ravenscar* design pattern is related to the Ada 2005 Ravenscar profile [6] that provides asynchronous communication between tasks with mutexes or semaphores. The *Blackboard* and *Queued buffer* design patterns model communication services that exist in standard RTOS API ARINC 653 [7]. And the *time-triggered* design pattern models a classical mean to implement lock free task communication [8].

With these design patterns, the process to select and apply feasibility tests to an architecture model can be decomposed in three steps. (1) We proceed to an analysis of the architecture model in order to verify its compliance with the one of the architectural design patterns proposed above [9]. This verification allows us to propose a first list of feasibility tests. (2) Then, we evaluate additional lists of applicability constraints in order to select the relevant feasibility tests. (3) Finally, the selected feasibility tests are automatically computed by Cheddar in order to assess scheduling analysis of the architecture model.

An architecture model can be a composition of those design patterns, which makes this analysis complex. In this article, we propose a simple method to master architectural design pattern composition in this context. We therefore focus on the step (1).

3. REAL-TIME ARCHITECTURAL DESIGN PATTERNS

In this section, we present several design patterns modeling synchronization and communication mechanisms of RTOSes. Mechanisms considered here are mutexes, semaphores and also queues of messages that may used to implement classical producer-consumer or readers-writers synchronizations. Those mechanisms are provided by most of RTOSes such as VxWorks, RT-Linux or RTEMS.

In the sequel, we first define what we mean by architectural design pattern. Then, we present how those design patterns are expressed. Finally, we propose five design patterns modeling the synchronization and communications mechanisms introduced above.

3.1 Defining architectural design patterns

In the literature, a pattern is defined as *"a solution to a recurring problem in a context"* [10] and can be seen as a *"three part rule, which expresses a relationship between a context, a problem and a solution"* [11].

In the context of real-time systems, [12] and [13] proposed to group design patterns in three levels corresponding to three phases of design: architectural, mechanistic and detailed designs:

Architectural design deals with large-scale strategic decisions such as scheduling policies, global system properties, interactions between components, global behavior among others.

Mechanistic design deals with the construction of group of components that interact to design a particular mechanism (a communication buffer for instance) or function.

Detailed design deals with lower concerns as data typing, internal algorithm and other details of a particular component.

Mechanistic design patterns can be seen as ways to instantiate solutions to architectural design patterns, and detailed design patterns as ways to instantiate solutions to mechanistic ones. The design patterns we proposed are part of the architectural level as we do no assumptions on how RTOS synchronization and communication mechanisms are imple-

mented.

Furthermore, their context is related to the design of real-time systems in which task synchronization and communication are implemented thanks to a RTOS. Finally, the problem addressed by each pattern is the analysis that has to be performed in order to validate the temporal behavior of the targeted system. Then, each of our design pattern proposes a solution to evaluate the system schedulability by the selection of relevant feasibility tests.

3.2 How we model architectural design patterns

Riehle *et al.* defines the form of a pattern as *"a finite number of visible and distinguishable components and their relationships"* [14].

To express the component of each of our architectural design patterns, we use Cheddar ADL. Cheddar ADL is an architecture language that allows Cheddar's users to model the architectures on which they expect to perform scheduling analysis. Figure 1 is a summary of the main concepts of Cheddar ADL. In the sequel, we assume that architecture models are composed of instances of entity **processor**, **task**, **buffer**, **shared resource** or **dependency**.

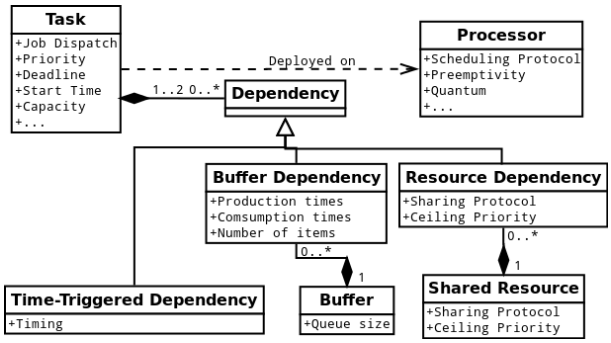

Figure 1: Part of the Cheddar ADL entities

An architectural design pattern is defined by a set of constraints on the elements of the architecture upon which we expect to run scheduling analysis. We use different types of constraints:

- **(a)** constraints on the type and number of components,
- **(b)** constraints on the connections between those components,
- **(c)** constraints on properties of those components.

Constraints of types (a) and (b) are contained in the architecture level and fit perfectly the Riehle *et al.* definition. Usually, one does not use constraints of type (c) in the definition of architectural design patterns, because they belong at the mechanistic level, but the context of our design patterns (hard real-time critical systems) forces us to use them. Indeed, some extra properties of the communication protocol need to be constrained. For instance, the protocol to share a data with the use of a mutex protected access has to implement a mechanism to prevent priority inversion.

The meta-model of Cheddar ADL is extended by the models of all applicability constraints of the feasibility tests that may be assigned to each design pattern. Each applicability constraint is modeled using EXPRESS.

3.3 Examples of architectural design patterns

Let see now some examples of design pattern expressed according to Cheddar ADL. We propose here five patterns modeling classical synchronization or communications mechanisms provided by RTOSes.

Each of these examples assumes the same uniprocessor execution environment. It is also assumed that this execution environment meets the following applicability constraints:

- **R0:** There is a unique processor.
- **R1:** The scheduling policy of the processor must be either earliest deadline first, fixed priority with rate monotonic priority assignment, deadline monotonic priority assignment or any other fixed priority assignment.
- **R2:** Preemptivity of the scheduler must be specified.
- **R3:** The scheduler is not allowed to use any quantum.
- **R7:** There is no hierarchical scheduling.

Most of RTOSes are compliant with these constraints. Indeed, many RTOSes are compliant with POSIX 1003 and this standard is able to meet the execution environment constraints presented above. The POSIX 1003 scheduling model assumes a preemptive fixed priority scheduler (then rules R1 and R2 are met). If all tasks have different priorities, then R7 and R3 are also met.

Finally, constraints R0 is not related to RTOS, but is most of time also met since multi-processor architectures are mostly unused in current real-time critical systems.

3.3.1 Time-triggered communications

The first design pattern is modeling time-triggered communications between tasks [8].

Context: With time-triggered communications, task communications are achieved without any RTOS synchronization tool. The communications between tasks require a shared memory but there is no need to protect it with a mutex or a semaphore. Task communication is achieved with timing synchronization. For example, each task can read its input from shared memory at dispatch time and writes its output on shared memory at completion time.
Problem: This design pattern is simple to analyze: from the perspective of real-time scheduling theory, tasks can be seen as independent. It requires the designer to apply numerous simple and efficient feasibility tests based or processor utilization factor or on worst case response time.
Solution: The constraints defining this design pattern can be expressed as follow:

- **R4:** All tasks are periodic.
- **R5:** There is no buffer entity.
- **R6:** There is no shared resource entity.

3.3.2 Ravenscar

The second design pattern is related to asynchronous communications between tasks.

Context: In Ravenscar, tasks communicate with shared resource under the control of inheritance ceiling priority protocol. Ravenscar assumes that shared resources are protected by semaphores. Semaphores can be used to build multiple synchronization protocols such as critical sections, readers/writers, producers/consumers, etc. But Ravenscar restricts the use of semaphores to mutex protected access to shared resources.

Problem: Here, additional analysis need to be performed: threads are not independent and the worst case response time analysis must take into account the waiting time due to critical sections and shared resource accesses.

Solution: The constraints defining this design pattern are the following:

R8: All tasks are either periodic or sporadic.
R9: There is at least one shared resource entity.
R5: There is no buffer entity.
R10: For each shared resource, there is at least two tasks that are accessing it.
R11: Allowed sharing resource protocols are PIP, IPCP or PCP.
R12: If PCP or IPCP are used, resource's ceiling priority must be higher or equal to all priorities of resource dependent tasks.
R13: if PIP is used, dependent tasks cannot share more than one shared resource.

3.3.3 Blackboard

The two next design patterns are related to task communication mechanisms that exist in the ARINC 653 RTOS API standard.

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

Problem: In this case, before computing the same analysis than the ones of Ravenscar, one has to analyze shared resource blocking time. Indeed, accesses to shared resources are more complex (i.e. different semaphores for readers and writers for instance).

Solution: The constraints defining this design pattern are the following:

R4: All tasks are periodic.
R5: There is no buffer.
R14: There is at least one readers/writers communication.
R15: Readers and writers cannot perform the same semaphore accesses.

3.3.4 Queued buffer

Context: Queued buffer implements a producers/consumers communication protocol. We assume that messages are handled according to a FIFO protocol.

Problem: This pattern requires the same schedulability analysis than Blackboard. Moreover, one has to analyze the memory footprint of buffers and also to check that production rates do not exceed consumption rates in order to prevent loss of data.

Solution: This extra analyze can be done using queuing theory models[15].

R4: All tasks are periodic.
R16: There is at least one buffer entity.
R17: For each buffer, the queue size must be bounded.
R18: Number of items produced and consummated by each task at each of its dispatch must be specified.
R19: Times of messages production and consumption must be specified.

3.3.5 Unplugged

Finally, this last design pattern simply models independent tasks, i.e. tasks that do not communicate nor synchronize with others.

Context: The unplugged design pattern models independent tasks.

Problem: The analysis of such systems is the same that for the time-triggered design pattern.

Solution: The constraints defining this design pattern are the following:

R4: All tasks are periodic.
R19: All tasks are independent.

4. COMPOSITIONS OF REAL-TIME ARCHITECTURAL DESIGN PATTERNS

In the previous section, we have introduced our design patterns. In practice, applications running on top of a RTOS are usually a composition of these design patterns. In this section, we introduce the notion of architectural design pattern composition. Then we expose a static analysis of possible compositions for feasibility tests selection.

4.1 Architectural design patterns composability

A real-time system architecture is a *compound system* of two architectural design patterns, if it is composed of two parts compliant with two different architectural design patterns. i.e. a system mixing two synchronization or communication protocols. Two architectural design patterns are *composable* if it is possible to use a strategy enabling a schedulability analysis of a system composed of those two patterns. The architectural design patterns presented here suppose an uniprocessor environment with no hierarchical scheduling. Hence the system depends on a single scheduler and schedulability analyzes apply to all tasks it schedules. Therefore, feasibility tests enabling the analysis of the whole system at once needs to be selected.

One solution to select feasibility tests for a compound system is to consider that it is compliant with one of the design pattern composing it. For example, a system composed of a set of tasks compliant with the Time-Triggered design pattern and another set of tasks compliant with Ravenscar, is analyzed as compliant with Ravenscar. Ravenscar is said *dominant* over Time-Triggered.

A design pattern A is *dominant* over a design pattern B if the compound system of A and B can be analysed as if it was compliant with A only. Then, a compound system is analyzable if there is a unique instance design pattern dominant over the design patterns of all other instances. The feasibility tests selection is done as if the whole system is compliant with the dominant design pattern.

4.2 Static analysis of architectural design patterns composition

The correctness of the analysis coupled with a dominant design pattern on a compound system needs to be proven. The compound system needs to fulfill all applicability constraints of the dominant design patterns. Hence, we proved that the combination of the two parts of the model meets all the applicability constraints of the dominant design pattern. Figure 2 gives an example of proof by contradiction for the composition of Ravenscar with Time-Triggered.

Proof R8 Let $S = Sub_1 \cup Sub_2$ a real-time system composed of T_i tasks. Let $Sub_1 = \{T_1, ..., T_n\}$ the subpart of S compliant with Time-Triggered. Let $Sub_2 = \{T_{n+1}, ..., T_m\}$ the subpart of S compliant with Ravenscar.
Lets consider that the restriction *"R8: All tasks are either periodic or sporadic."* is not met by S. Since Sub_2 is compliant with Ravenscar, it meets R8.
Therefore, $\exists\ T_{fail} \in Sub_1$ a task neither periodic nor sporadic.
Sub_1 is compliant with Time-Triggered. Thus, it is compliant with the applicability constraint *"R4: All tasks are periodic."*.
So Sub_1 contains T_{fail} and meets R4, which is a contradiction. Therefore the applicability constraint R8 must be met by S.

Figure 2: Example of proof by contradiction that Ravenscar is dominant over Time-Triggered. The same proof must be made for all applicability constraints of Ravenscar.

Figure 3 provides all proven design patterns combinations. Each combination of composable patterns have been proven: the compliance of the compound system to all constraints of the dominant design pattern have been checked.

ADPs	Unpl	T-T	Rav	B-B	Q-B
Unpl	Unpl	T-T	Rav	B-B	Q-B
T-T	T-T	T-T	Rav	B-B	Q-B
Rav	Rav	Rav	Rav	∅	∅
B-B	B-B	B-B	∅	B-B	∅
Q-B	Q-B	T-T	∅	∅	Q-B

Figure 3: Table of dominant design patterns for each supported composition. The first line and column contain the composed architectural design patterns (ADPs), and intersection between each line and column contains the compound design pattern. Empty squares are the combinations for which the design patterns are not composable, or not proven to be composable. In this table, Design patterns are Unplugged(Unpl), Time-Triggered(T-T), Ravenscar(Rav), Blackboard(B-B) and Queued Buffer(Q-B).

5. 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 proposed approach. The evaluation consists in testing the implementation of a design pattern recognition prototype with generated architectures covering all applicability constraints and design patterns. The prototype used for this evaluation is an update of the one used in our previous works [9]

We have designed an architecture generator. Architectures are generated following input parameters: the types of architecture elements and their numbers. The elements taken into account are any type of task, buffer, dependency, message and resource. The execution environment is uniprocessor.

ADPs	Unpl	T-T	Rav	B-B	Q-B
Unpl	25	25	25	25	25
T-T	∅	25	50	50	50
Rav	∅	∅	25	50	50
B-B	∅	∅	∅	25	50
Q-B	∅	∅	∅	∅	25
Uncompliant	10	10	10	10	10

Figure 4: Table of generated architectures.

First, we generate architectures compliant with one design pattern, with a number of tasks and communication varying from 10 to 500. Second, for each previously generated architectures, we add a random number of communication or dependency which are part of another design pattern. For instance, buffers were added to an architecture compliant with the Time-triggered design pattern.

Third, we modified architecture models that were compliant with our design patterns to evaluate each applicability constraint. For example, a Time-Triggered architecture model with an extra sporadic task is generated in order to evaluate the constraint *R4*. Figure 4 gives an overview of the number of generated architectures.

In the sequel, we detected the compliance (or non-compliance) of each generated architecture to an architectural design pattern. The names of the recognized design pattern, or the constraints that were not met are stored in a file. The selection or non-selection of design patterns for each architecture is then manually validated.

This evaluation has been performed under Ubuntu 12.04 LTS on a processor Intel Core^{TM} i5-2430M CPU @ 2.40GHz $\times$ 4 with 6,0GiB 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 and varies from 10ms for the smallest systems (10 tasks $\times$ 10 dependencies) to 1950ms for the biggest systems (500 tasks $\times$ 1000 dependencies). Finally, all the unmet applicability constraints of the third set of architectures have been found.

6. RELATED WORKS

Numerous works deal with the use of design patterns and analysis of application running on top of RTOSes. Douglass et al. [12] define a handbook for real-time systems design

patterns. They define both architectural and mechanistic design patterns. 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 [16]. These approaches describe communication protocols but do not consider the binding with analysis. Vardanega depicts the Ravenscar profile using applicability constraints [17] in order to reduce the gap between industrial practice and theory. Filali et al. [18] define the Time-Triggered architecture as an AADL subset. Each of those methods studies the validation of a set of real-time architectures with a static number of design patterns and does not cope with the composition issue. Design patterns detection has also been subject to numerous studies. For instance, in [19] and [20], authors describe methods to verify and detect classic mechanistic design patterns. These works are useful to detect which communication protocols are used. In our case, we consider them as known and we focus on additional constraints required for analysis.

7. CONCLUSION

Schedulability analysis is an intrinsic step to the design of critical real-time systems. Such analysis depends on the system architecture characteristic and its use is difficult. This article propose architectural design patterns modeling communication and synchronization tools met in classical RTOSes. These design patterns are bound to scheduling analysis methods called feasibility tests. We have shown how those design patterns can be composed, in order to analyze real-time applications mixing different RTOS communication or synchronization tools. Finally, we have proposed an evaluation of this approach upon a large set of generated architectures. The following of this work is to study and classify feasibility tests. This approach aims at enabling a multi-criteria comparison of feasibility tests. The objectives are to (1) propose an analysis method for feasibility tests selection for unknown compositions and (2) refine the selection process for known situations. This will (1) widen the scope of analysable systems and (2) classify selected feasibility tests according to a given objective such as their complexity or their pessimism for instance.

Acknowledgment

We would like to thank Ellidiss Technologies and Conseil Regional de Bretagne for their support to this project.

8. REFERENCES

[1] 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 *Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation-Volume Part I*. Springer-Verlag, 2010, pp. 4–17.

[2] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[3] F. Singhoff, A. Plantec, P. Dissaux, and J. Legrand, "Investigating the usability of real-time scheduling theory with the Cheddar project," *Real-Time Systems*, vol. 43, no. 3, pp. 259–295, 2009.

[4] I. T. N. WD, *EXPRESS Language Reference Manual*, 1997.

[5] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[6] A. Burns, B. Dobbing, and G. Romanski, "The Ravenscar tasking profile for high integrity real-time programs," in *Reliable Software Technologies-Ada-Europe*. Springer, 1998, pp. 263–275.

[7] Arinc, *Avionics Application Software Standard Interface*. The Arinc Committee, Jan. 1997.

[8] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, 2003.

[9] V. Gaudel, F. Singhoff, A. Plantec, S. Rubini, P. Dissaux, and J. Legrand, "An ada design pattern recognition tool for aadl performance analysis," in *Proceedings of the 2011 ACM annual international conference on Special interest group on the ada programming language*, ser. SIGAda '11. New York, NY, USA: ACM, 2011, pp. 61–68.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Elements of reusable object-oriented design," 1995.

[11] C. Alexander, *The timeless way of building*. Oxford University Press, USA, 1979, vol. 1.

[12] B. P. Douglass, *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[13] R. Monroe, A. Kompanek, R. Melton, and D. Garlan, "Architectural styles, design patterns, and objects," *Software, IEEE*, vol. 14, no. 1, pp. 43 –52, jan/feb 1997.

[14] D. Riehle and H. Züllighoven, "Understanding and using patterns in software development," *Theory and Practice of Object Systems*, vol. 2, no. 1, pp. 3–13, 1996.

[15] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Scheduling and memory requirements analysis with aadl," in *ACM SIGAda Ada Letters*, vol. 25, no. 4. ACM, 2005, pp. 1–10.

[16] A. Burns and A. J. Wellings, "Hrt-hood: A structured design method for hard real-time systems," *Real-Time Systems*, vol. 6, pp. 73–114.

[17] T. Vardanega, "When theory meets technology," in *Proceedings of a conference organized in celebration of Pro-fessor Alan Burns' sixtieth birthday*, p. 178.

[18] M. Filali-Amine and J. Lawall, "Development of a synchronous subset of aadl," in *Abstract State Machines, Alloy, B and Z*. Springer, 2010, pp. 245–258.

[19] W. Wang and V. Tzerpos, "Design pattern detection in eiffel systems," in *Reverse Engineering, 12th Working Conference on*, nov. 2005, p. 10 pp.

[20] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe, "Automatic design pattern detection," in *Program Comprehension, 2003. 11th IEEE International Workshop on*. IEEE, 2003, pp. 94–103.