Specification of Schedulability Assumptions to Leverage Multiprocessor Analysis[⋆]

Stéphane Rubini[a,*], Valérie-Anne Nicolas[a], Frank Singhoff[a], Alain Plantec[a], Hai Nam Tran[a], Pierre Dissaux[b]

[a] *Univ Brest, Lab-STICC, CNRS, UMR 6285, F-29200 Brest, France*
[b] *Ellidiss Technologies, 24 quai de la douane, F-29200 Brest, France*

Abstract

In order to ease the early verification of uniprocessor real-time systems, the tool *Cheddar* provides a service that guarantees the applicability of a schedulability analysis method for a given architecture model. This verification service uses a catalog of design patterns.

In this article, we propose to extend these patterns to multiprocessor architectures. Designing such extension is a challenge because the knowledge of both the software and the hardware architectures are essential to decide on the schedulability of a task set in that context. Indeed, parallel execution of tasks involves hardware resource sharing, that has in turn an effect on the task execution times. Currently, no general method is able to assess the schedulability of a high-performance multicore system with a limited level of pessimism, except if assumptions or usage restrictions are set to simplify the system analysis.

So, the research community is developing multiple schedulability tests based on various assumptions which constrain the task models and their execution platforms. In this article, we propose a framework based on *Prolog* that allows engineers to verify the conditions to apply a test are met. *Prolog* facts model the software and hardware architecture, and the inference engine checks whether these facts conform to a design pattern associated to a given verification method.

The design pattern compliance framework is integrated with the Cheddar tool. Three examples of multiprocessor analyses illustrate the proposal. A scalability analysis shows the tool is able to verify the compliance of architectures composed of 600 tasks and 60 cores, in less than 140s on a desktop computer.

[⋆]Cheddar has been supported by Univ. of Brest, Ellidiss Technologies, Brest Métropole, CR de Bretagne, CD du Finistère, Campus France PESSOA programs number 27380SA and 37932TF, MOSAR (Horizon 2020 grant agreement No 821996) and Thales.

[*]Corresponding author

Email addresses: `Stephane.Rubini@univ-brest.fr` (Stéphane Rubini), `Valerie-anne.Nicolas@univ-brest.fr` (Valérie-Anne Nicolas), `Frank.Singhoff@univ-brest.fr` (Frank Singhoff), `Alain.Plantec@univ-brest.fr` (Alain Plantec), `Hai-Nam.Tran@univ-brest.fr` (Hai Nam Tran), `dissaux@ellidis.com` (Pierre Dissaux)

1. Introduction

In [1], the NIST has shown that 70% of the anomalies during the engineering of critical real-time systems are detected lately, leading to a high increase of their production cost. In such context, early verification of timing constraints with schedulability analysis can contribute to reduce this cost overhead. Many models and methods of schedulability have been proposed. However, due to the diversity and the rapid evolution of the runtime environments, due to the variability of the task models to deal with, schedulability analysis remains a difficult problem.

The *Cheddar* project [2] aims to facilitate the application of schedulability analysis in the engineering practices of critical real-time systems. As part of the *Cheddar* project, the software framework of the same name aims at helping engineers to apply this analysis. For this purpose, it implements several scheduling algorithms, feasibility tests, simulation engines and various features to perform system architecture explorations.

The *Cheddar* tool notably relies on a catalog of design patterns and integrates a service to check the compliance of an architectural model to these patterns [3]. A design pattern is a set of constraints on the software architecture; these constraints derive from the assumptions that must be met to apply a schedulability analysis. Using a design pattern therefore allows the designer to make sure that a schedulability analysis is adequate for the architecture she/he wants to check and then, that the analysis results are sound.

Problem statement. More and more often the engineering of critical real-time systems harnesses numerous processors. For example, an airplane or a car can embed from several tens upto hundreds of interconnected computers [4]. Some of their functions may need more computing and communication capabilities that a single processor can supply [5]. In such new execution platforms, hardware shared resources imply new interference on the software components. Such interference lead to new delays that have to be taken into account during schedulability analysis. Then, schedulability analysis of such systems is a major challenge to enable them on real critical real-time systems, and today few methods have been proposed to leverage their use.

Contributions. This article presents a software framework whose purpose is to guide the activities of real-time scheduling early verifications and analyses on multiprocessor[1] systems. The framework is based on a set of design patterns. In particular, the design patterns discriminate the systems according to the deployment of tasks on the available computing units, but also according to the additional hardware resources these computing units use. The extension

[1] We choose the term "multiprocessor" in the article, in the sense that a multiprocessor system is a system that includes several computing units. The computing units may share some hardware resources. We use the term "multicore" when we want to emphasis that the multiprocessor system is implemented on a single silicon chip.

of the *Cheddar* tool in that way requires more than a simple update of the design patterns, because the modeling of the runtime environment may require to express detailed features about the underlying hardware architecture.

We propose to express the software and hardware architecture of the system to verify as a set of *Prolog* facts, and the design patterns as *Prolog* rules that can be checked on the set of facts that models the system. The approach allows for easily appending new patterns. This ability is mandatory in a research context characterized by a high variability of the assumptions made in the different contributions.

Initially defined for uniprocessor platforms, the features of *Cheddar* are extended to deal with multiprocessor systems, possibly integrated onto a single silicon chip. However, this article does not propose new analysis methods, but rather attempts to define a scalable framework for hosting different scheduling analysis methods which are useful in a system engineering process.

Outline of the article. Next section, we remind the notion of design pattern and its application in the field of schedulability analysis, as it was implemented in the Cheddar tool version for uniprocessor environments. In Section 3, we introduce the overall approach proposed in this article, also based on design patterns, in order to select appropriate schedulability analysis methods for multiprocessor systems. The end of this section focuses on the modeling of the multiprocessor runtime environments. Section 4 presents some design patterns for multiprocessor real-time systems and how they have been prototyped inside the *Cheddar* framework. Section 5 addresses the validation of the approach through examples and a scalability study, before concluding in Section 6.

2. Previous Work

The object oriented design community has widely adopted the design pattern concept, especially since [6]. A design pattern is a well established and documented solution to a recurring problem in a given context. Sanz [7] and Pont [8] have advocated the use of a specific catalog of patterns for the design of real-time systems, covering both functional and non-functional requirements. Software engineers may not master all the aspects of a complex embedded design, that requires skills in hardware, control system algorithms, reliability etc. The reuse of known well accepted solutions decreases the design time, and should also improve the quality of the design.

In the field of embedded systems, the specialization of the execution platform is high. Physical and/or cost constraints, and the ongoing interaction with various environments, explain that the hardware is often dedicated to a product. The design of such systems requires to take care of both their software parts, their hardware parts and the mapping of the first on the second. The Architecture Description Languages (AADL [9] and UML/Marte [10]) have been proposed at that end among other things.

The design pattern approach we advocate is an application of the work initiated with the Ravenscar profile [11], which is a set of restrictions on Ada programs to ease verification and validation activities. This approach has been applied on various systems, and for various properties such as performance, safety or security [12, 13, 14]; sometimes it has even been extended by the concept of contracts [15].

To the best of our knowledge, very few studies have proposed to apply a pattern-based approach on the verification and validation of multiprocessor embedded systems when it comes to taking into account the shared hardware resources beyond the processing units themselves.

Transportation systems, including avionics, are one of the application domains where schedulability analysis is mandatory. Currently, there is no established certification procedure that applies to the use of multicore systems in civil aircraft. However, certification authorities lead an exploratory work on the subject. In [16], the CAST (*Certification Authorities Software Team*) exposes some work tracks, and indicates that memory cache sharing in multicore processors is a key point of the problem. The design pattern modeling that we propose focuses in particular on this point.

More generally, several scheduling analysis tools for multiprocessor systems have been proposed in recent years, e.g. STORM [17], RealtssMP [18], Yartiss [19], SimSo [20] or MAST [21]. Some propose to study shared hardware resources and the interference to which they can lead. However, these tools do not provide mechanisms to ensure that the analyzed models are consistent with the applicability assumptions of schedulability analysis methods.

2.1. Applying Design Patterns in the Schedulability Analysis Domain

In this part, we present the concept of design patterns used as reference designs for helping the schedulability analysis activities. Then, we give examples of patterns that have been previously implemented in the *Cheddar* tool.

Each schedulability analysis method requires that the architecture model to analyze complies with a set of assumptions called *applicability constraints*. The greater the number of applicability constraints or the number of usable schedulability analysis methods are, the harder it is for a designer to choose the schedulability method to apply. Each schedulability method may have different characteristics in terms of result accuracy (exact or pessimistic), sustainability or scalability [12], which increase again the complexity of the choice.

Design patterns can help in choosing a schedulability method. Indeed, a design pattern specifies the applicability constraints of an analysis method. For the designer, the problem is then to ensure that the architectural model she/he wishes to check is compliant with a design pattern.

Fig. 1 summarizes the relations between the design patterns, the schedulability analysis methods, and the architectures to verify.

2.1.1. Formalization of Design Patterns for Schedulability Analysis

In [3], Gaudel formalizes the concept of design pattern previously used in *Cheddar*. The following sets together define a design pattern:

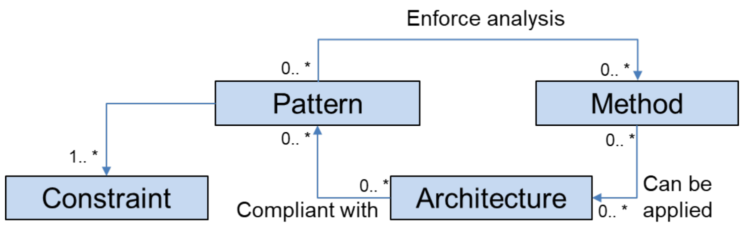


Figure 1: Relations between design patterns, schedulability analysis methods, and architectures. We note that a given architecture may not be compliant with any model or that no analysis method may be applicable. However a schedulability analysis method may be applicable to an architecture whereas no design pattern is usable to guarantee its applicability.

1. **Environmental constraints.** This set of constraints characterizes the runtime environment. It describes the hardware part of the architecture model and the software for the exploitation of its resources (operating system, device drivers etc). In his work, Gaudel focuses on a uniprocessor environment described by the applicability constraints of Table 1.

Env1	The runtime environment has a single processing unit.
Env2	The scheduling policy can be: *EDF*, *LLF* or fixed-priority.
Env3	The preemptive level of the scheduler must be explicit.
Env4	The scheduler does not use quantum.
Env5	There is no hierarchical scheduling.

Table 1: This table (from [22]) describes a uniprocessor runtime environment. The left column contains the identifier of the constraint. The right column describes the constraint in natural language.

2. **Communication and synchronization constraints.** This second set of constraints characterizes the software part of the architectural model to analyze.
These constraints focus on concurrency, and specifically on task communication and synchronization. Task is the main concept used by schedulability analysis. A task is characterized by a set of timing parameters [23]. Among these timing parameters, most come from the requirement of the design of the system and do not depend on the runtime environment. For example, the deadline specifies the time beyond which the execution of a *job* may disrupt the expected behavior of the system. Others, and specifically the maximum time required for a *job* execution (or Worst Case Execution Time) of a task, are related to the runtime environment.
Five sets of communication and synchronization constraints have been defined in [24, 12], and then formalized in [3]. These sets are described later in this section.

3. **Schedulability analysis methods.** Each pattern describes an association between schedulability analysis methods and a type of architecture whose characteristics comply with the applicability constraints of the analysis methods. The analysis methods are feasibility tests such as processor utilization factor tests, simulations on the feasibility interval, worst case response time analysis.

In the next section, we recall the main sets of communication and synchronization constraints allowing the *Cheddar* tool to perform scheduling analyses automatically.

2.1.2. Task Communication and Synchronization Models

Five models of communication and synchronization between tasks have been proposed in [12, 22]. These models have been specified with AADL [9], a standard architecture language used in avionics, or with *Cheddar-ADL* [25] that is the native architecture description language of the *Cheddar* tool. Each of these models expresses an usual method of synchronization and communication that can be found in languages, standards or operating systems used by the actors of the real-time domain. Below, the parenthesized name associated to each model is defined for the purpose of later reference in this article.

1. **Synchronous data-flow** (LOG_SYNC). This first model corresponds to a common synchronization and/or communication in AADL: communications between *thread* components via *data ports* in immediate mode. With AADL, a *thread* component models a control flow, i.e. a thread consists of a sequential instruction sequence scheduled together with the other threads deployed on the runtime environment. When two AADL threads communicate by a data port, every thread reads and writes the data at times specified by the AADL standard. Data writing and reading times are known and the runtime environment handles the synchronization issues involved for sharing this information without data loss.
2. **Ravenscar** (LOG_RAV). In this model, tasks can exchange data asynchronously via shared memory. The accesses to shared memory are protected by a priority inversion avoidance protocol, i.e. PCP [26]. The tasks must respect several additional constraints that we do not detail here for the sake of brevity. All of these constraints come from the Ravenscar profile of the 2012 Ada standard [27].
3. **Blackboard** (LOG_BB). This is a readers/writers model. Readings and writings are done asynchronously and only the last written value can be read. This set of constraints models a synchronization/communication mechanism defined in the ARINC653 standard [28].
4. **Queued buffer** (LOG_QB). This model expresses a producer/consumer synchronization where the messages are produced and consumed according to a FIFO order. *Queued buffer* also models a synchronization/communication mechanism defined in the ARINC653 avionic standard.
5. **Unplugged** (LOG_UPG). This last model assumes that tasks are independent, i.e. they are not synchronized and they do not share any data.

In other words, each task can be scheduled as early as it is ready (periodic wake up in the case of a periodic task) without any other condition than the availability of a computing unit.

Cheddar automates the conformity verification of an architectural model to a design pattern [3]. In this article, we propose an extension of the *Cheddar* design patterns, especially by defining execution environment models in order to take into account the actual hardware in a multiprocessor context.

3. Scheduling Analysis on Actual Runtime Environments

With the growing number of transistors implanted on silicon chips, complex integrated runtime environments are now available for digital system designers. The wide commercial offer of multicore processors, manycore processors, or heterogeneous MultiProcessor System-on-Chip (MPSoCs) illustrates this trend.

One of the major issues, raised by the shift from single-core to multicore processors, is the variability of the task parameter "capacity" depending on the schedule of the system as a whole. Hence, the usual two-phase process, WCET analysis followed by scheduling analysis, can produce very pessimistic results if all the potential interference is taken into account in the task capacity. The answer to this problem implies evaluating the effective interference with regards to the possible schedulings, or mitigating the interference at the runtime level.

In this context, there is a strong need for modeling the runtime environment to support accurate scheduling analyses. Especially identifying non-functional interference related to the concurrent usage of hardware resources is a critical issue to ensure the analysis results are sound and accurate. We describe this interference as *non-functional*, in the sense that it is not expressed in the software architecture model of the system, i.e. as application dependencies between the software tasks.

3.1. The Overall Approach

The overall approach we propose is shown Fig. 2.

Three models, the schedulability analysis model, the runtime interference model and the design patterns, constitute the inputs of the process, and each of them is related to a different area of expertise:

- The software engineers build the **schedulability analysis model** i.e. software tasks, functional dependencies and computing units;
- The hardware engineers are able to identify the usage interference from the hardware specification of the execution platform, and write the **runtime interference model**;
- The scheduling theory expert implements new methods in the scheduling analysis tool. The expert knows the design rules allowing for these analysis methods to be applied to hardware/software systems [29], and maintains a **design pattern catalog**. Especially, the design pattern specifies the

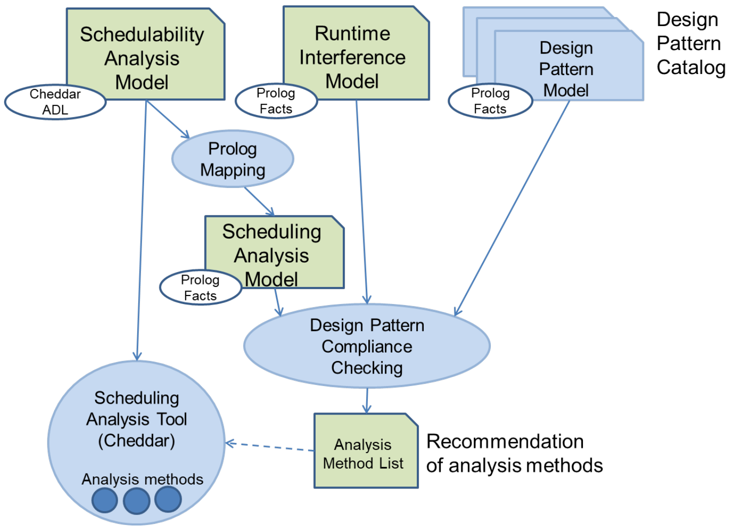

Schedulability Analysis Model
Cheddar ADL
Runtime Interference Model
Prolog Facts
Design Pattern Model
Prolog Facts
Design Pattern Catalog
Prolog Mapping
Scheduling Analysis Model
Prolog Facts
Design Pattern Compliance Checking
Scheduling Analysis Tool (Cheddar)
Analysis methods
Analysis Method List
Recommendation of analysis methods

Figure 2: Scheduling Analysis Method Selection Process

hardware architecture and/or the interference that the analysis method implemented in the Cheddar tool is able to take into account.

From these inputs, the *Design Pattern Compliance Checker* program, called *dp_check* in the sequel, produces a list of design patterns the system architecture complies with. Next, this allows for selecting a list of analysis methods that can be applied to the system.

Notice that the scheduling analysis by itself is driven only by the specification issued from the scheduling analysis model.

The *Prolog Mapping* program generates a Prolog model from the relevant information of the schedulability analysis model used as Cheddar tool configuration; this information includes the list of the tasks, the list of the execution units, the scheduling parameters and protocols, the allocations of tasks on the execution units, and various parameters that influence the scheduling analysis. Beyond this mapping, it detects whether the software architecture conforms to one of the software design pattern described in Section 2.1.2.

The process assumes the consistency of the entity or attribute names extracted from the scheduling analysis model and those coming from the interference model; the hardware entities are considered to be the same if they have the same name in both models.

The *Prolog Mapping* program is written in Ada, and reuses already existing functions from the Cheddar framework to parse the Scheduling Analysis Model and search the software design patterns.

Implementing a new method into the scheduling analysis tool, might require additional parameters (e.g. the number of memory accesses of a task and their duration); in this case, the Prolog mapping program might need to be updated to reflect these new parameters as Prolog facts.

The *dp_check* program is a Prolog implementation based on Prolog facts and inference rules. Then, the set of facts is obtained by merging this Prolog model and the runtime interference model. The inference rules are the predicates dp_i defined for each DP_i design pattern (see Sections 4.3 and 4.4) and its constraints in the design pattern catalog. The *dp_check* main predicate sequentially checks the compliance of the system with all the design patterns specified in the catalog.

3.2. Runtime Environment Interference Model

A runtime environment can be modeled by a set of entities, whose attributes clarify the interference within the hardware architecture that could lead to WCET variability.

Here, a runtime environment is made of *Processing Element* and *Resource* entities, as well as their relations that could lead to potential interference.

Processing Element. A *Processing Element* (PE) is a hardware component allowing for the execution of a sequential flow of execution.

A PE can be a core in a multicore processor or MPSoC architecture, a single-core processor, a hardware thread or a dedicated operator (a hardware task).

Resource. A *Resource* (R) is an additional hardware component required, directly or indirectly, by some PEs for executing their flow of execution. A resource is said to be shared when several PEs use it; a shared resource may lead to functional or non-functional (e.g. temporal) interference between the PEs that use it. For example, an instance of R can model a memory cache, a bus, a *Network-On-Chip* (NoC).

Fig. 3 presents the model describing the interference between entities in a runtime environment.

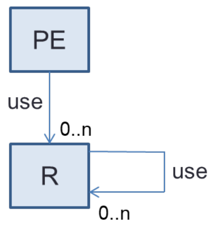


Figure 3: Interference Model of the Runtime environments

When the use of a hardware shared resource does not imply any interference, the relations between the PEs and this shared resource may be left out of the runtime environment model. For example, if a shared cache is partitioned to avoid conflicts when accessing to it, it is not mandatory to model the relations between this hardware resource and the related PEs.

Fig. 4 shows four typical examples of runtime environment models. Fig. 4(a) comprises two independent PEs (*pe1* and *pe2*): these computing units, that do not use any shared resources, or that use hardware resources without interference, do not lead to a WCET variability. This model has been extensively investigated in schedulability analysis with identical processors [30].

In Fig. 4(b), we now have a hardware resource shared by two PEs. A possible example for this model is the share of a memory bus (*r1*) required by *pe1* and *pe2*.

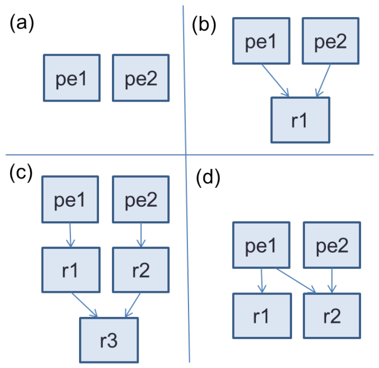


Figure 4: Examples of runtime environment interference

Fig. 4(c) exposes a third example where several shared and private hardware resources are used by *pe1* and *pe2*. Here, *r3* models a level 2 cache shared between all the PEs, and *r1* and *r2* respectively model the level 1 caches of *pe1* and *pe2*.

Finally, Fig. 4(d) also presents two PEs with shared and private resources. The hardware resource *r1* can model a level 1 cache for *pe1*, while *r2* can model a communication bus shared by *pe1* and *pe2*.

From the previous models, one can classify the different PEs according to the *non-functional interference* that they may undergo during task execution. Table 2 shows predicates that define whether a PE belongs to a given interference scheme.

A set of attributes, presented in the next section, characterizes the PE and R entities. Each attribute is specified using a relation of the form:

$attribute\_relation(e, val)$

where e represents an entity or a set of entities. The relation is True if the value of the attribute of the entity e is the value val, False otherwise. In the sequel, we use the term *attribute* at place of *attribute relation* for the readability of the text.

3.3. Attributes of the Runtime Environment Entities

These attributes (prefixed by A_) specify the intrinsic properties of a runtime environment entity, that is either a resource R or a processing element PE.

Name	**Semantics**
HA_independent(PE)	True if the PE does not use any hardware resources whose sharing could cause interference between tasks.
HA_isolation(PE)	True if the PE does not use any hardware resources whose sharing could produce interference impossible to predict/prevent at the system *runtime* level.
HA_bounded(PE)	True if the PE does not use any hardware resources whose sharing can produce interference which *a priori* cannot be bounded in time.
HA_dependent(PE)	True if the PE uses hardware resources whose sharing can produce unbounded time interference.

Table 2: PE interference models

They characterize the associated entity regardless of its context of use in the system.

We can distinguish different kinds of attributes. Some attributes provide information about the hardware entity type ; an entity can provide a computing, a storing or a communication mean. The *A_type* attribute of an entity E then defines its category C using the relation: $A\_type(E, C)$ with $C \in \{processing, memory, interconnect\}$

Other attributes specify the implementation, structural, and temporal characteristics of the entity. The tables 3, 4 and 5 provide a list of attributes classified according to the type of entity to which they apply. Obviously this list is not exhaustive and can be extended for a particular analysis method.

Attributes are used at two levels while defining a design pattern. Some take part in the selection of a specific analysis method, and others in the settings of the selected method. For example, *A_mem_cache_associativity* allows to check whether a method based on direct-mapped cache is appropriate, or in the case of methods dealing with any type of cache, is used to calculate data placement in the cache.

3.4. Access Attributes

Access attributes (prefixed by AM_) define when and how hardware entities access resources R. Modeling the architecture of the runtime environment is useful for identifying potential interference due to hardware resource sharing. However, a hardware resource can be a part of an entity that can be used independently with the other parts of such entity. Furthermore, access to the resource by the PEs can be timely isolated. In these both cases, accesses to the hardware resource are not likely to create interference. If the system design does not prohibit concurrent access, the design pattern must express how the access conflict is handled.

The two first attributes of the table below models relations *use* shown in the Fig. 3. The next three attributes represent respectively the time intervals

Attribute name (prefix A_mem_ is omitted for brevity)	**Semantics**
type(R,V)	entity R sub-type (memory, memory bank, data cache, instruction cache. . .) V ∈ {memory, bank, DCache, ICache, IDCache, hierarchy}
cache_associativity(R,V)	cache associativity V is an integer ≥ 1
cache_replacement_policy (R,V)	associative cache replacement policy V ∈ {LRU, LRR, random}
cache_miss_time(R,V)	cache line loading time (*or Block Reload Time*) in case of miss V is a time in ns
cache_size(R,V)	cache or cache partition full size V is a number of bytes
cache_line_size(R,V)	size of a cache line V is a number of bytes
cache_level(R,V)	cache level with respect to the processor V is an integer ≥ 1
cache_coherency(R,V)	writing strategy, coherency protocol V ∈ {copy_back, write_through}
memory_access_time(R,V)	access time to a memory word V is a time in ns

Table 3: Examples of attributes applied to "memory" resources

Attribute name	**Semantics**
A_proc_type(PE,V)	entity sub-type (processor or MPSoC core, physical *thread*, physical processor, specialized operator) V ∈ {core, processor, thread, dedicated}
A_proc_isa(PE,V)	supported instruction set V is a label
A_proc_speed(PE,V)	speed, or speed range in case of DVFS control (Dynamic Voltage and Frequency Scaling), possibly relative to the PEs supporting the same ISA. V is a range [operation/s, operation/s] or scalar

Table 4: Examples of attributes applied to "processing" resources

Attribute name	**Semantics**
A_conn_type(R,V)	communication entity sub-type (bus, *Network-on-Chip*, star, peer-to-peer…) V $\in$ {bus, NoC, star, p2p}
A_conn_throughput(R,V)	maximum transfer rate on a bus V is a throughput in words/s
A_conn_latency(R,V)	maximum latency of a transfer once the bus is available V is a time in ns

Table 5: Examples of attributes applied to "interconnect" resources

during which a PE is allowed to access a hardware resource, the partition of the hardware resource allocated to the PE, and in case of access conflict, the used arbitration policy.

Attribute name	**Semantics**
AM_PE_use(PE,R)	PE is authorized to use a set R of hardware resources
AM_R_use(R1, R2)	R1 is authorized to use/access a set R2 of hardware resources
AM_time(PE,R,V)	V is the set of time intervals where the PE is authorized to access the hardware resource R
AM_space(PE,R,V)	V is a subpart or a partition of R to which PE is allowed to access
AM_arbitration ({PE,…,PE},R,V)	V is an arbitration policy in case of concurrent accesses of a set of PEs to R.

For example, the following logic formula characterizes a set of two processing units *pe1* and *pe2*, accessing a bus according to a TDM frame:
$AM\_time(PE1, Bus, V1) \wedge AM\_time(PE2, Bus, V2) \wedge (V1 \cap V2 = \emptyset)$

Note that the access attributes do not provide the same information as the attributes describing the scheduling of software tasks on PEs. A task may be active on a PE while the PE is stuck waiting for a hardware resource. Conversely, a PE may have an exclusive access to a hardware resource while no software task requires access to this resource.

3.5. Deployment Attributes

Deployment attributes (prefixed by DM_) indicate how hardware resources are allocated to software entities. As in AADL [9], they define either an effective assignment, or an allowed assignment.

Attribute name	**Semantics**
DM_PE_actual(T,V)	V is the PE where task T is actually executed
DM_PE_allowed(T,V)	V is the set of PEs authorized to perform task T

Even if a task is allowed to run on several PEs, a *job* of this task is only processed by a single PE at a given time, i.e. the task code is not parallelized.

A priori, a task running on a PE can use the overall set of hardware resources accessible by this PE in its environment. The deployment attributes of a task allow one to define this set explicitly, and therefore to restrict it if necessary.

Attribute name	**Semantics**
DM_R_actual(T,V)	V is the set of hardware resources actually used by task T
DM_R_allowed(T,V)	V is the set of hardware resources whose access is authorized to task T

The deployment of tasks on shared hardware entities is specified by a scheduling policy that governs the time intervals during which the entity is assigned to the task. The *DM_PE_scheduling* and *DM_R_scheduling* deployment attributes define the parameters of the scheduling policy.

Attribute name	**Semantics**
DM_PE_scheduling(PE,T,V)	V is the set of parameters that defines the scheduling policy of a task T on a PE
DM_R_scheduling(R,T,V)	V is the set of parameters that defines the scheduling policy of a task T during its access to a resource R

Effective access to a shared resource R at time t first depends on the scheduling on a PE of the task requesting its access, and second by allowing this PE to use the resource R. In other words, we assume here a hierarchical scheduling, most often managed at very dissimilar time scales. For example, in the case of a dual-core processor with shared memory, the memory read instruction of a task executes without waiting (1) if the task is scheduled on a core and (2) if the memory bus arbitration allows it to access the memory.

3.6. Updating Cheddar *Tool Design Patterns*

In the previous sections, we proposed a runtime environment model in order to clarify interference within these environments. We now describe how the design patterns of the *Cheddar* tool are adapted so that they can be applied to multiprocessor runtime environments. In this context, a design pattern is now made of:

1. **Attributes characterizing an intrinsic functionality of a hardware entity**, PE or R resources, in the runtime environment (Section 3.3).
2. **A hardware architecture model, possibly supplemented with access attributes**. These models allow us to exhibit non-functional interference between hardware entities (Section 3.4).

3. **Deployment attributes** which indicate how the software entities (i.e. the tasks) are deployed on the PEs and resources R (Section 3.5).
4. **Synchronization and communication constraints between tasks**. In the rest of the document, we consider the models *Synchronous data-flow*, *Ravenscar*, *Blackboard*, *Queued buffer* and *Unplugged* (see Section 2.1.2).
5. And finally **a set of scheduling methods** that can be applied to systems complying with the constraints of the items (1), (2), (3) and (4).

In the next section, we use these modeling elements to define runtime environments as design patterns.

4. Patterns for Multiprocessor Scheduling Analysis

In this section, we list the analysis methods available in the *Cheddar* tool and that are applicable in the multiprocessor context. The conditions of their use are specified by a design pattern expressed according to the previously defined attributes.

4.1. Multiprocessor Analysis Functions Implemented in Cheddar

Prior to the redesign of the design pattern checking, the move to multiprocessor systems of the Cheddar analyzer required several updates to support multiprocessor scheduling protocols.

First, we completed the modeling language used to express the architecture of the system to be analyzed. We add new entities that represent the multiple execution units, and their basic characteristics (the relative speed for instance). It was also mandatory to extend the communication/synchronization features to support the software interaction between the tasks assigned to different processors. Moreover, some properties have been appended to structurally account for the memory hierarchy (level 1 and 2 caches) which is significant in multiprocessor systems.

A second work was the re-factoring of the Cheddar event-driven scheduling simulation engine. Indeed, the parallel activity of the processors raised issues for a correct computation of the future simulation state. The simulator also had to be able to manage the task/job migrations.

To quantify the amount of work done for updating Cheddar, notice that the multiprocessor shift required the creation of about twenty new entities, and as many new types in its internal data structure. These changes impacted both the simulation engine and the Cheddar input language, but were mandatory to support multiprocessor simulations and to configure the multiprocessor feasibility tests.

Cheddar analysis functions for multiprocessor real-time systems can be classified in three categories.

The first category is based on simulation. It consists to producing a simulation of the task set scheduling on the feasibility interval [31], and then, to computing various performance metrics (worst case response time, worst case blocking time, absence of deadlock, priority inversion, number of preemption, number of context switching etc). To date, the available simulation algorithms apply multiprocessor global or partitioned scheduling, with classical algorithms (fixed-priority, EDF, LLF etc) or specific algorithms such as EDZL or Proportionate-Fair. Simulations can be configured via parameters related to the scheduling policy (preemption level, quantum) or to the studied architecture entities (jitter, offset, and more generally, on the task model and shared resource access protocol).

The second category of analysis methods is based on the use of feasibility tests. In the context of multiprocessor or distributed architectures, the available methods mainly compute task worst response times on tree or linear transactions [32, 33, 34, 35, 36] or processor utilization rates [30]. Some tools can also calculate various data to extend the worst case response times, e.g. the Cache-Related Preemption Delay (CRPD) [37], or the shared resource blocking time.

Finally, the last category of methods contains architecture exploration tools. In a multiprocessor or distributed context, they can be, for example, partitioning methods, methods assigning various task parameters for taking into account hardware resources (e.g. priority allocation according to the CRPDs), or distribution and communication precedence constraints.

Specifically, the scheduling analysis methods that are already implemented in the Cheddar tool and that consider hardware interference are the following:

- For the cache usage interference, the computation of CRPD is included in simulation, feasibility tests, and priority assignment algorithms for fixed priority preemptive scheduler [38]; the CRPD quantification uses various state-of-the-art methods like UCB-only, ECB-only, or multiset [37].
- For the interference between messages, the ECTM model computes the communication times inside a Store-and-Forward or a wormhole Network-on-Chip and deals with direct interference between messages [39]. The arbitration policy of the NoC routers must avoid the indirect interference.
- For the memory access interference, the feasibility test proposed in [40] is implemented. This test applies on a multiprocessor architecture with a shared DRAM memory, and a predefined allocation of the tasks on memory banks.

Thus, to sum up, the Cheddar tool implements about 80 scheduling protocols if we consider the variation of the algorithms with the job migration policy.

4.2. Design Pattern Structuring

Defining a design pattern implies to model constraints on different system layers. In order to simplify their formalization, we define several sets of constraints, possibly reusable, to express the patterns.

The constraint sets that are used in the rest of this article are presented in Fig. 5. The level to which they occur is underlined by their name prefix:

LOG Synchronization and communication constraints of tasks

DEP Deployment constraints

EXE Interference between hardware entities and hardware resource access constraints

FEA Hardware entity constraints

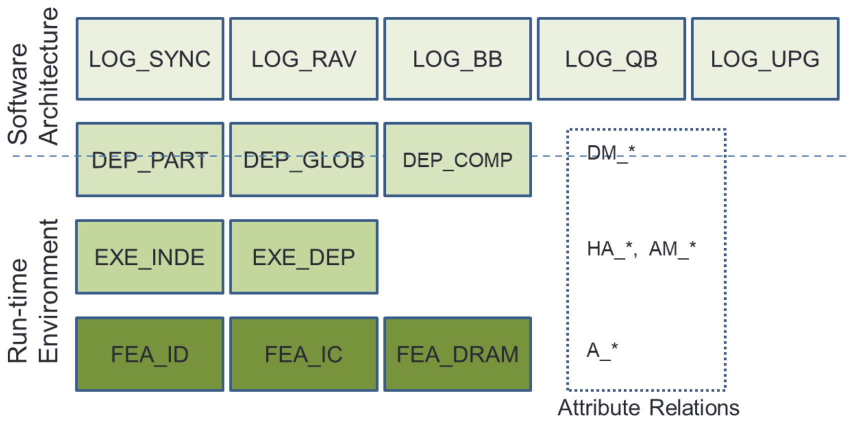


Figure 5: Current Constraint Set Structuring based on System Layers.

We now define these sets of constraints where T denotes the set of tasks of the system to be analyzed, P the set of PEs and R the set of resources.

Compatible tasks (DEP_COMP).
This set constrains task deployment by requiring that the tasks are compatible with a set of PEs. This compatibility must be seen as in terms of system design (i.e. an affinity task-PE), and not in terms of strict software-hardware concordance, even if the first implies the second.

$$\begin{array}{l} \forall t, s \in T, \\ \exists \text{ two single sets of PEs } v_t, v_s \subset P \setminus \\ \quad (\, DM\_PE\_allowed(t, v_t) \wedge DM\_PE\_allowed(s, v_s) \\ \quad \wedge \; (\|v_t\| \geq 1) \wedge (\|v_s\| \geq 1) \\ \quad \wedge \; (\, (v_t = v_s) \vee (v_t \cap v_s = \emptyset)) \,) \end{array}$$

This formula stands for the fact that if a task is compatible with a set v_t of PEs (following the relation $DM\_PE\_allowed$), another task is compatible with the same set or, otherwise, with another set v_s that must be disjoint of the first one.

Task partitioning by PE (DEP_PART).

The set of tasks is partitioned on the processing elements, and the task scheduling on each PE must be managed by the same algorithm.

$$\boxed{\begin{array}{l}
\forall t, s \in T, \\
\quad \exists! p_t, p_s \in P \setminus \\
\qquad (DM\_PE\_actual(t, p_t) \land DM\_PE\_actual(s, p_s) \land \\
\qquad\; (\exists \text{ two single scheduling policies } v_t, v_s \setminus \\
\qquad\quad (DM\_PE\_scheduling(p_t, t, v_t) \\
\qquad\qquad \land\ DM\_PE\_scheduling(p_s, s, v_s) \\
\qquad\qquad \land\ (p_t = p_s \Rightarrow v_t = v_s))))
\end{array}}$$

Task partitioning by PE set (DEP_GLOB).
The PE sets defined by the deployment must be the same, or otherwise, disjoint. All the tasks of the system to be analyzed must be deployed on at least one PE. Within a PE set, a unique scheduling policy must be used.

$$\boxed{\begin{array}{l}
\forall t, s \in T, \\
\quad \exists \text{ two single sets of PEs } v_t, v_s \subset P \setminus \\
\qquad (DM\_PE\_allowed(t, v_t) \land DM\_PE\_allowed(s, v_s) \\
\qquad\; \land\ (\|v_t\| \geq 1) \land (\|v_s\| \geq 1) \\
\qquad\; \land\ (\ (v_t = v_s) \lor (v_t \cap v_s = \emptyset)\ ) \\
\qquad\; \land\ (\ (v_t = v_s) \Rightarrow \\
\qquad\qquad\quad (\forall p, q \in v_t,\ \ \exists \text{ a single scheduling policy } x \setminus \\
\qquad\qquad\qquad (DM\_PE\_scheduling(p, t, x) \land DM\_PE\_scheduling(q, s, x))))
\end{array}}$$

Note that DEP_GLOB $\Rightarrow$ DEP_COMP because DEP_GLOB checks in addition that a unique scheduling policy is used within the PE set. Obviously, the tasks must be compatible with all the PEs belonging to the set managed by the global scheduling.

Note also that, for a model specified without $DM\_PE\_allowed$ attribute, DEP_PART $\Rightarrow$ DEP_GLOB. Actually, DEP_PART uses $DM\_PE\_actual(t, p_t)$ from which we can deduce $DM\_PE\_allowed(t, \{p_t\})$ (used in DEP_GLOB) for any task t. Thus, a global scheduling on a singleton set of PEs is a partition scheduling (obviously, again).

Independent multiple PEs (EXE_INDE).
A runtime environment that complies with this set guarantees that the temporal and functional behavior of the PEs that perform a task, is independent of the status of other PEs, in the absence of functional interference between the tasks.

$$\boxed{\begin{array}{l}
\forall t \in T, \\
\quad \exists \text{ a single set of PEs } v_t \subset P \setminus \\
\qquad (DM\_PE\_allowed(t, v_t) \\
\qquad\; \land\ (\forall p \in v_t,\ HA\_independent(p)) \\
\qquad\; \land\ (\forall p \in P,\ (DM\_PE\_actual(t, p) \Rightarrow HA\_independent(p))))
\end{array}}$$

Dependent multiple PEs (EXE_DEP).
This constraint set defines a hardware architecture where the activity of a PE

temporally interferes with those of the others PEs. The impact of the interference may be difficult to accurately bound except with a dedicated scheduling analysis method that also considers the task model.

```
$\exists t \in T \setminus$
  $(\exists$ a set of PEs $v_t \subset P,\ \exists p \in v_t \setminus$
        $DM\_PE\_allowed(t, v_t)\ \wedge\ HA\_dependent(p))$
  $\vee\ (\exists p \in P \setminus DM\_PE\_actual(t,p) \wedge HA\_dependent(p))$
```

Identical PEs (FEA_ID).
The PEs are only characterized from the point of view of their performances in terms of execution speed, and type of code they can execute. The execution speed can possibly be defined relatively to the others PEs.

```
$Let\ S = Allowed \cup Actual$
 $with\ Allowed = \bigcup_{t \in T}\{v \setminus DM\_PE\_allowed(t,v)\}$
 $and\ Actual = \bigcup_{t \in T}\{p \setminus DM\_PE\_actual(t,p)\},$
$\forall p, q \in S,$
 $(\ ((\exists s \setminus A\_PE\_speed(p,s) \wedge A\_PE\_speed(q,s))$
     $\vee\ \neg(\exists s \setminus A\_PE\_speed(p,s))\ \vee\ \neg(\exists s \setminus A\_PE\_speed(q,s)))$
  $\wedge$
   $((\exists i \setminus A\_PE\_isa(p,i) \wedge A\_PE\_isa(q,i))$
     $\vee\ \neg(\exists i \setminus A\_PE\_isa(p,i)) \vee\ \neg(\exists i \setminus A\_PE\_isa(q,i)))$
```

If the speed or type attribute does not exist for a PE, it is by default considered identical to the others PEs.

If speed and type are not defined for any PEs, then these are considered as abstract entities providing predefined computing power units, to put in relation with the analyzed task model configuration.

PE with private instruction cache (FEA_IC).
These PEs contain a direct-mapped level 1 private instruction cache.

```
$Let\ S = Allowed \cup Actual$
 $with\ Allowed = \bigcup_{t \in T}\{v \setminus DM\_PE\_allowed(t,v)\}$
 $and\ Actual = \bigcup_{t \in T}\{p \setminus DM\_PE\_actual(t,p)\},$
$\forall p \in S,$
 $\exists! r \in$ set of resources $R \setminus$
   $(\ \ AM\_PE\_use(p, \{r\})$
    $\wedge A\_mem\_type(r, Icache)$
    $\wedge A\_mem\_cache\_associativity(r, 1)$
    $\wedge A\_mem\_cache\_miss\_time(r, constant\_value)$
   $)$
```

PEs sharing a DRAM memory through a cache (FEA_DRAM).
These PEs share a Last Level Cache (LLC). The cache miss processing causes

block exchanges with a DRAM memory, which allows by bank accesses. The data/instructions of each tasks are stored on a known set of memory banks.

$$\begin{array}{l}
\textit{Let } S = Allowed_S \cup Actual_S \\
\ \textit{with } Allowed_S = \bigcup_{t \in T} \{v \setminus DM\_PE\_allowed(t, v)\} \\
\ \textit{and } Actual_S = \bigcup_{t \in T} \{p \setminus DM\_PE\_actual(t, p)\}, \\
\textit{Let } U = Allowed_U \cup Actual_U \\
\ \textit{with } Allowed_U = \bigcup_{t \in T} \{v \setminus DM\_R\_allowed(t, v)\} \\
\ \textit{and } Actual_U = \bigcup_{t \in T} \{p \setminus DM\_R\_actual(t, p)\}, \\
(\forall u \in U, A\_mem\_type(u, bank)) \\
\ \wedge \\
(\exists !\ r \in \text{ set of resources } R \setminus \\
\quad (\forall p \in S,\ AM\_PE\_use(p, \{r\})) \\
\quad \wedge\ A\_mem\_type(r, IDcache) \\
\quad \wedge\ (\exists \text{ a set of resources } B \setminus AM\_R\_use(r, B)\ \wedge\ U \subseteq B))
\end{array}$$

4.3. Design Patterns and Achievable Analyses

In what follows, we specify five design patterns based on the previously defined set of constraints, and respectively named DP1, DP2, DP3, DP4 and DP5.

DP1. A runtime environment compliant with this pattern allows a group of tasks to run on a group of identical and independent PEs. The task groups and PE groups are disjoint.

$$\begin{array}{lll}
DP1 & := & (LOG\_UPG \vee LOG\_SYNC) \wedge \\
 & & DEP\_COMP(T, PE) \wedge \\
 & & EXE\_INDE(T, PE) \wedge \\
 & & FEA\_ID(T)
\end{array}$$

Observing this pattern permits the use of the task set partitioning methods available in *Cheddar*. Several independent task set partitioning heuristics are implemented: *Best Fit*, *First Fit*, *Next Fit*, *Small Task* and *General Task* [41, 42].

DP2. A runtime environment consistent with this pattern defines a single PE for the execution of each task, and a scheduling policy for each PE. The PEs are independent and identical.

$$\begin{array}{lll}
DP2 & := & (LOG\_UPG \vee LOG\_RAV \vee LOG\_SYNC) \wedge \\
 & & DEP\_PART(T, PE) \wedge \\
 & & EXE\_INDE(T, PE) \wedge \\
 & & FEA\_ID(T)
\end{array}$$

This pattern defines a partitioned multiprocessor system. Depending on the compliance of the software architecture, this pattern allows to apply many *Cheddar* implemented analysis methods.

For example, if the architecture to check complies with LOG_RAV, LOG_SYNC or LOG_UPG, we can compute worst case response times [43].

DP3. A runtime environment compliant with this pattern allows a group of tasks to run on a group of identical and independent PEs. The task groups and PE groups are disjoint. All PEs in a group are managed by the same scheduling policy.

$$\boxed{\begin{aligned} DP3 := \; & (LOG\_UPG \lor LOG\_RAV \lor LOG\_SYNC) \land \\ & DEP\_GLOB(T, PE) \land \\ & EXE\_INDE(T, PE) \land \\ & FEA\_ID(T) \end{aligned}}$$

This pattern specifies a multiprocessor system with a global scheduling. Various algorithms are available in *Cheddar* to dynamically control the task scheduling on a set of PEs. Some are adaptations of classical algorithms used in uniprocessor environments (RM, DM, EDF . . .), and others have been specifically developed for multiprocessor systems (EDZL, Proportionate-Fair, LLREF. . .) [44].

Note that DP3 ⇒ DP1 for architectures compliant with `LOG_UPG` or `LOG_SYNC` (which are the only architectures compatible with DP1, and because DEP_GLOB ⇒ DEP_COMP).

DP4. A runtime environment consistent with this pattern defines a single PE for the execution of each system task, and a temporal scheduling policy for each PE. The PEs are independent, identical, and contain a direct-mapped level 1 private instruction cache.

$$\boxed{\begin{aligned} DP4 := \; & (LOG\_UPG \lor LOG\_SYNC) \land \\ & DEP\_PART(T, PE) \land \\ & EXE\_INDE(T, PE) \land \\ & FEA\_ID(T) \land FEA\_IC(T) \end{aligned}}$$

Compliance with this pattern grants access to methods that, in *Cheddar*, take into account the CRPD (*Cache Related Preemption Delay*), i.e. the simulation with CRPD [45], the computation of interference between tasks due to the instruction cache [46], and the extension of the optimal algorithm proposed by Audsley [47] for priority assignment.

Note that DP4 ⇒ DP2 for architectures compliant with `LOG_UPG` or `LOG_SYNC` (which are the only architectures compatible with DP4). DP4 is more constraining than DP2 as it also checks FEA_IC.

DP5. The hardware side of the systems represented by this pattern is symmetrical multiprocessor systems. The cores share a LLC (Last-Level Cache) and a DRAM memory organized in bank. A single channel connects the cache and the memory controller. Tasks are partitioned on PEs.

$$\begin{aligned} DP5 := {} & (LOG\_UPG \vee LOG\_SYNC) \wedge \\ & DEP\_PART(T, PE) \wedge \\ & EXE\_DEP(T, PE) \wedge \\ & FEA\_ID(T) \wedge \\ & FEA\_DRAM(T, R) \end{aligned}$$

Currently, in the Cheddar tool, only one feasibility test based on the work of Kim et al [40], is available to analyze architectures conforming to this design pattern.

4.4. Design Pattern Implementation

In this section, we present a *Prolog* implementation of the design patterns and the Design Pattern Checking Algorithm (*dp_ check*) to test the compliance of an architecture model with the different design patterns defined in Section 4.3. These design patterns are specified as first order logic formulas involving predicates for task communication and synchronization modeling (see Section 2.1.2) and predicates representing sets of constraints for runtime environment modeling (see Section 4.2). Actually, these sets of constraints are themselves specified as first order logic formulas using predicates associated with the different runtime environment model attributes (see Sections 3.3, 3.4 and 3.5).

Prolog is a logic programming language based on the first order logic paradigm [48]. Therefore, the design pattern implementation in *Prolog* is a straightforward translation of their specification. Here, we use the *Prolog* implementation provided by the ECLiPSe tool[2].

Here after is the *Prolog* ECLiPSe implementation of the predicate for the DP1 design pattern.

```
dp1 :- sw_archi(A),
       member(A,[log_upg,log_sync]),
       tasks(LTasks),
       processing_elements(LPE), !,
       dep_comp(LTasks,LPE), !,
       exe_inde(LTasks,LPE), !,
       fea_id(LTasks), !.
```

The predicates `processing_elements(LPE)` and `tasks(LTasks)` respectively define the set of PEs *LPE* available in the runtime environment, and the set of tasks *LTasks* of the analyzed architecture model. The predicate `sw_archi(A)` allows to check the task communication and synchronization model of the software part of the architecture model. For sake of conciseness, we here only give the details of the `dep_comp(LTasks,LPE)` predicate used in the DP1 pattern. The predicates `exe_inde(LTasks,LPE)` and `fea_id(LTasks)` are presented in Appendix Appendix A. The predicates that implement the DP2, DP3, DP4

[2]https://www.eclipseclp.org/

and DP5 patterns are constructed in the same way, and can also be found in Appendix Appendix A.

```
dep_comp([],_).
dep_comp([T],LPE) :-
   dm_PE_allowed(T,[C|LC]), !,
   (foreach(P,[C|LC]), param(LPE) do member(P,LPE), !), !.
dep_comp([T1|LT],LPE) :-
   dm_PE_allowed(T1,LP1), !,
   LP1 = [_|_],
   (foreach(P,LP1), param(LPE) do member(P,LPE), !), !,
   (foreach(T2,LT), param(LP1) do
       dm_PE_allowed(T2,LP2), !,
       (subtract(LP1,LP2,[]),
        subtract(LP2,LP1,[])
       ;
        intersection(LP1,LP2,[])), !
   ), !,
   dep_comp(LT,LPE).
```

Finally, we introduce the *dp_check* algorithm to test the compliance of an architecture model with the different design patterns:

```
Algorithm dp_check(system_model) {
    system_model_load(system_model)
    for $dp \in \{dp1, dp2, dp3, dp4, dp5\}$
        if dp then system_model is dp compliant}
```

5. Validation: Examples and Study of the Scalability of the Approach

In the first part of this section, we show how the proposal can be applied through examples, and next we evaluate the scalability of the proposed tools, i.e. how the tools are able to deal with models of various sizes.

5.1. Analysis Examples

We illustrate our approach with 3 examples of multiprocessor systems. We explain for each example how the model is able to describe the hardware architecture and run a schedulability analysis. The first example is a multiprocessor architecture without any hardware shared resource, managed by a global scheduling policy. The second shows how our model can handle shared resources as cache units. Finally, the third example addresses shared memory accesses.

5.1.1. Example 1: Global Scheduling

Here we present the analysis of a software architecture made of four periodic tasks $t0$, $t1$, $t2$ and $t3$, scheduled on a runtime environment including two processors $c0$ and $c1$. Both processors are identical and independent. The software architecture complies with `LOG_UPG`. Here is a *Prolog* set of facts to represent these first characteristics of the system architecture:

```
sw_archi(log_upg).
tasks([t0,t1,t2,t3]).
processing_elements([c0,c1]).
```

The facts below represent the attributes characterizing the hardware part of the runtime environment:

```
a_type(c0,processing).              a_type(c1,processing).
am_PE_use(c0,[]).                   am_PE_use(c1,[]).
ha_independent(c0).                 ha_independent(c1).
a_proc_type(c0,processor).          a_proc_type(c1,processor).
a_proc_isa(c0,i386).                a_proc_isa(c1,i386).
a_proc_speed(c0,100000000).         a_proc_speed(c1,100000000).
```

The designer of this architecture has chosen a (*Proportionnate-Fair*) global scheduling policy. The *Prolog* set of facts below represents the deployment rules. These rules show that the designer allows software tasks to run on any processors and define the computing resource sharing protocol, i.e. the scheduling policy.

```
dm_PE_allowed(t0,[c0,c1]).          dm_PE_allowed(t1,[c0,c1]).
dm_PE_allowed(t2,[c0,c1]).          dm_PE_allowed(t3,[c0,c1]).
dm_PE_scheduling(c0,t0,sched(pfair,preemptive,timeUnMig)).
// + last line repeated 3 times (for t1, t2, t3)
dm_PE_scheduling(c1,t0,sched(pfair,preemptive,timeUnMig)).
// + last line repeated 3 times (for t1, t2, t3)
```

As shown by the execution of the *dp_ check* algorithm defined in Section 4.4, the architecture model actually complies with the DP3 pattern and it is therefore possible to use *Cheddar* to simulate task execution from these hypotheses (see Figure 6). Moreover with this example, we can verify that DP3 $\Rightarrow$ DP1; obviously, the processors targeted by the global scheduling policy and the task codes must be compatible.

```
?- dp_check('example1_model').
     Yes (0.00s cpu)
example1_model is DP1 compliant
example1_model is DP3 compliant
```

5.1.2. Example 2: Instruction Cache Impact on Scheduling

For this second example, we seek to analyze a real-time system implemented on a dual-core runtime environment ($c0$ and $c1$). The program instructions are stored in a shared memory, and each processor has a level 1 private instruction cache. The Figure 7 schematizes the expected runtime environment. The

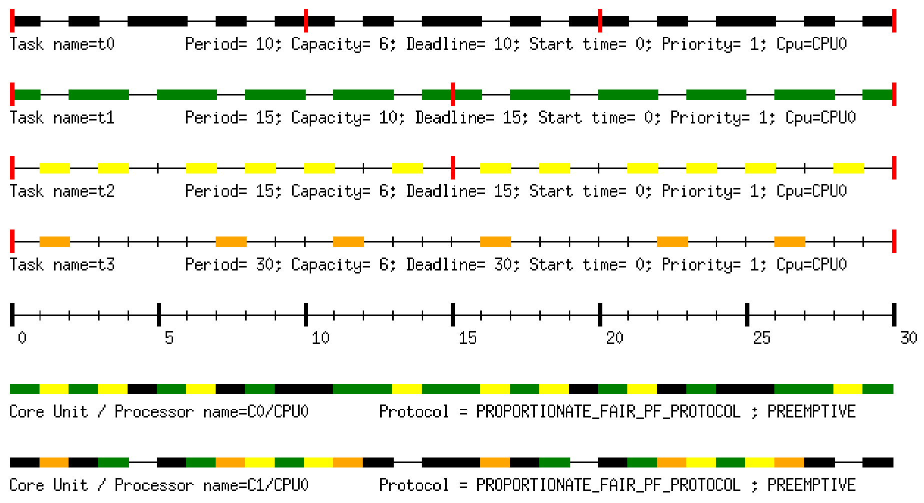


Figure 6: PFair global scheduling, on two identical processors. The four first time-lines show when the associated tasks are executing, while the two last ones represent the processor allocation.

Scratchpad Memories (SPM) are used to record data and task execution context. So the shared memory only stores the instructions. The bandwidth of the memory bus is fairly allocated to both cores by a TDM (Time Division Multiplexing) bus frame constituted of two slots of equal duration.

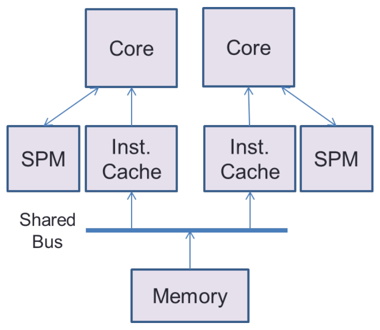


Figure 7: Runtime environment. Scratch Pad Memories are dedicated to data and context storage.

The list of attributes which characterizes this architecture, and which will be used to set some scheduling analysis parameters, is given below.

```
processing_elements([c0,c1]).
resources([ic0,ic1]).
a_type(c0,processing).              a_type(c1,processing).
a_type(ic0,memory).                 a_type(ic1,memory).
```

```
a_proc_type(c0,processor).         a_proc_type(c1,processor).
a_proc_isa(c0,sparc_V8).           a_proc_isa(c1,sparc_V8).
a_proc_speed(c0,100000000).        a_proc_speed(c1,100000000).
am_PE_use(c0,[ic0]).               am_PE_use(c1,[ic1]).
a_type(mb0,interconnect).
a_conn_type(mb0,bus).
a_mem_type(ic0,ICache).
a_mem_cache_size(ic0,1024).
a_mem_cache_line_size(ic0,16).
a_mem_cache_level(ic0,1).
a_mem_cache_associativity(ic0,1).
a_mem_cache_miss_time(ic0,1000). // =500+500, see below
// + the same 6 previous lines but for ic1
```

The interference related to hardware resource sharing is then defined. The cores use the same memory to store their instructions and access it by the same bus (*mb*0). The effective access to the memory being dependent on the access to the bus, the model represents the access rules to the latter only, that is to say, its TDM frame. The additional blocking time that can suffer the cache to process a miss depends on the duration of the slot assigned to the other processor. As this time is static and known, it is possible to infer that the interference duration is bounded if the number of misses is bounded too (*HA_isolation* interference pattern).

As the implicit interference arrives at the time of the cache misses, it is possible to include it in the miss processing time, and to use the *HA_independent* interference pattern. In other words, for this architecture, we are able to abstract the interference related to the memory bus. Note that this updated time must also be taken into account when calculating task WCET.

```
ha_independent(c0).
ha_independent(c1).
am_time(c0, mb0, interval(0,500,1000)).
am_time(c1, mb0, interval(500,500,1000)).
```

The `interval` term defines a time slot by three parameters, respectively its start, its duration and its period. Hence, in the first clause of the `am_time` predicate `interval(0,500,1000)` means the set of time intervals $\{[0+1000.k, 500+1000.k[\mid k \geq 0\}$.

For this example, the software architecture consists of four tasks, called $t0$, $t1$, $t2$ and $t3$. Their code and their memory location are known which allows us to compute the values of added CRPD during simulation when preemptions occur [45]. The software architecture complies with *LOG_UPG*. The designer has chosen to assign the tasks $t0$ and $t1$ to the first processor, and the tasks $t2$ and $t3$ to the second one, as shown in the next deployment model:

```
sw_archi(log_upg).
tasks([t0,t1,t2,t3]).
```

```
dm_PE_actual(t0,c0).          dm_PE_allowed(t0,[c0]).
dm_PE_actual(t1,c0).          dm_PE_allowed(t1,[c0]).
dm_PE_actual(t2,c1).          dm_PE_allowed(t2,[c1]).
dm_PE_actual(t3,c1).          dm_PE_allowed(t3,[c1]).
dm_PE_scheduling(c0, t0, sched(fp,preemptive)).
dm_PE_scheduling(c0, t1, sched(fp,preemptive)).
dm_PE_scheduling(c1, t2, sched(fp,preemptive)).
dm_PE_scheduling(c1, t3, sched(fp,preemptive)).
```

As expected, the execution of the *DP_check* algorithm on this second example shows that this model complies with the DP4 design pattern (and also with DP1, DP2 and DP3). The schedulability analysis can be performed by a simulation with CRPD. Figure 8 shows the simulation result produced by the *Cheddar* tool.

Moreover, on this example compliant with *LOG_UPG*, we can verify that DP4 ⇒ DP2. As this second example model is specified without *DM_PE_allowed* attribute, we can thus verify that DP2 ⇒ DP3. And consequently DP3 ⇒ DP1 also holds.

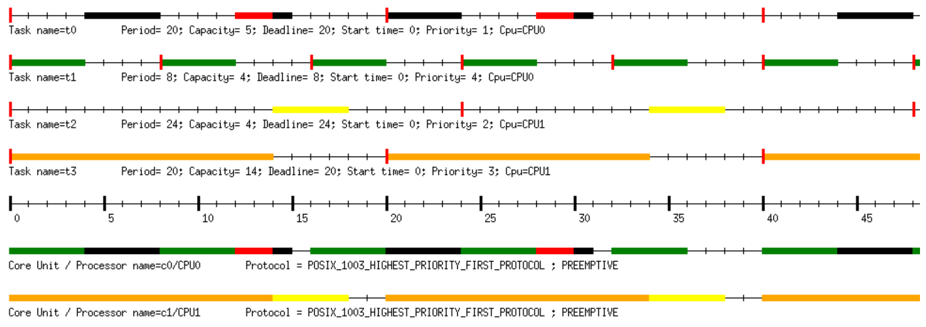


Figure 8: Partitioned scheduling, on two processors (each one with a private instruction cache). Red rectangles represent the additional time due to interference when using the instruction cache in case of preemption. In the case of processor C1, no preemption is observed between tasks $t2$ and $t3$.

5.1.3. Example 3: Memory Sharing Impact on Scheduling

In this last example, we reproduce an example of schedulability analysis of a multiprocessor architecture with shared memory units leading to interference at task execution time [40]. Again, we show we can model both the hardware and software parts. The correctness of those models is assessed by showing that the schedulability test proposed by [40] can be applied.

From the hardware side, we have a 4 core processor sharing 3 banks of memory (Fig. 9). Core 0 may access to any memory bank while any other core i (with $i > 0$) only access to one memory bank (see the *DM_R_actual* and *DM_PE_actual* attributes page 28). The list of attributes which characterizes this architecture, and which will be used to set some schedulability analysis parameters, is given below as a *Prolog* set of facts:

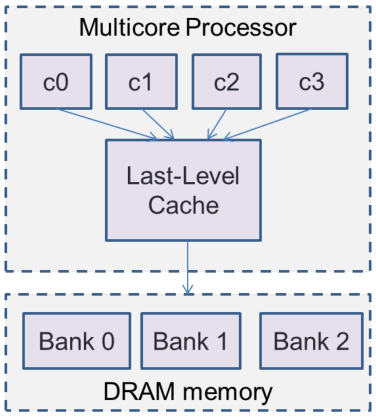


Figure 9: Runtime environment. The processor's cores access a DRAM through a shared cache.

```
processing_elements([c0,c1,c2,c3]).
a_type(c0,processing).                  a_type(c1,processing).
a_type(c2,processing).                  a_type(c3,processing).
resources([mc,b0,b1,b2]).
a_type(mc, memory).
a_type(b0,memory).
a_type(b1,memory).
a_type(b2,memory).
a_mem_type(mc,IDCache).     // LLC
a_mem_type(b0,bank).
a_mem_type(b1,bank).
a_mem_type(b2,bank).
am_PE_use(c0,[mc]).                     am_PE_use(c1,[mc]).
am_PE_use(c2,[mc]).                     am_PE_use(c3,[mc]).
am_R_use(mc,[b0,b1,b2]).
ha_dependent(c0).                       ha_dependent(c1).
ha_dependent(c2).                       ha_dependent(c3).
```

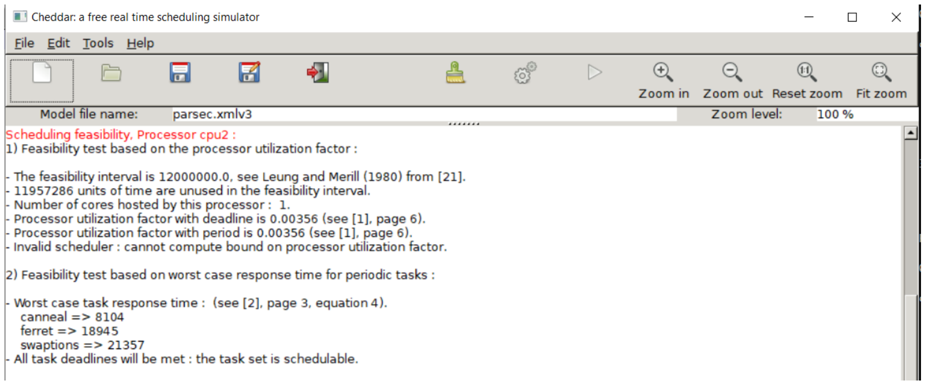
Some specific timing parameters configure the test, for example the worst-case service time for consecutive DRAM row-hit requests; the value of these parameters is provided as entries in the schedulability analysis model. Even if dedicated Prolog attributes could easily represent them in the interference model, the interest to adopt these attributes at that level must be considered on case-by-case basis depending on the reusability of the parameters. Thus, in this analyze example, the timing parameters of the DRAM controller are not represented in the interference model, and then the DP checker might not detect an discrepancy between the execution platform and some assumptions on which the schedulability analysis is based.

The software architecture is modeling a subset of the PARSEC benchmark [49], composed of 9 independent periodic tasks. To model memory interference, a task is defined to generate memory accesses. We assume all task codes and memory locations are known. The software architecture complies with *LOG_UPG*.

PARSEC tasks are mapped on cores $c1$ to $c3$. Core $c0$ is running the task *interference* which is responsible for generating contentions on the memory banks. This deployment is shown with the following model:

```
sw_archi(log_upg).
tasks([interference, blackscholes,x264,vips,swaptions,
      ferret,canneal,bodytrack,freqmine,fluidanimate]).
a_proc_type(c0,processor).
a_proc_type(c1,processor).
a_proc_type(c2,processor).
a_proc_type(c3,processor).
dm_PE_actual(interference,c0).
dm_PE_actual(blackscholes,c1).
dm_PE_actual(x264,c1).
dm_PE_actual(vips,c1).
dm_PE_actual(swaptions,c2).
dm_PE_actual(ferret,c2).
dm_PE_actual(canneal,c2).
dm_PE_actual(bodytrack,c3).
dm_PE_actual(freqmine,c3).
dm_PE_actual(fluidanimate,c3).
dm_PE_allowed(interference,[c0, c1, c2, c3]).
dm_PE_allowed(blackscholes,[c1, c1, c2, c3]).
// and so on for all the tasks
dm_R_actual(interference,[b0,b1,b2]).
dm_R_actual(blackscholes,[b0]).
dm_R_actual(x264,[b0]).
dm_R_actual(vips,[b0]).
dm_R_actual(swaptions,[b1]).
dm_R_actual(ferret,[b1]).
dm_R_actual(canneal,[b1]).
dm_R_actual(bodytrack,[b2]).
dm_R_actual(freqmine,[b2]).
dm_R_actual(fluidanimate,[b2]).
dm_PE_scheduling(c0, interference, sched(fp,preemptive)).
dm_PE_scheduling(c1, blackscholes, sched(fp,preemptive)).
dm_PE_scheduling(c1, x264, sched(fp,preemptive)).
dm_PE_scheduling(c1, vips, sched(fp,preemptive)).
dm_PE_scheduling(c2, swaptions, sched(fp,preemptive)).
dm_PE_scheduling(c2, ferret, sched(fp,preemptive)).
dm_PE_scheduling(c2, canneal, sched(fp,preemptive)).
dm_PE_scheduling(c3, bodytrack, sched(fp,preemptive)).
dm_PE_scheduling(c3, freqmine, sched(fp,preemptive)).
dm_PE_scheduling(c3, fluidanimate, sched(fp,preemptive)).
```

As expected, the execution of the *DP_check* algorithm on this third example shows that this system model complies with the DP5 design pattern, and the Cheddar tool is able to assess the schedulability of this model. As an example, worst case response times of the PARSEC tasks are shown in Fig. 10.



```
Model file name: parsec.xmlv3                     Zoom level: 100 %
Scheduling feasibility, Processor cpu2 :
1) Feasibility test based on the processor utilization factor :

- The feasibility interval is 12000000.0, see Leung and Merill (1980) from [21].
- 11957286 units of time are unused in the feasibility interval.
- Number of cores hosted by this processor : 1.
- Processor utilization factor with deadline is 0.00356 (see [1], page 6).
- Processor utilization factor with period is 0.00356 (see [1], page 6).
- Invalid scheduler : cannot compute bound on processor utilization factor.

2) Feasibility test based on worst case response time for periodic tasks :

- Worst case task response time : (see [2], page 3, equation 4).
  canneal => 8104
  ferret => 18945
  swaptions => 21357
- All task deadlines will be met : the task set is schedulable.
```

Figure 10: Partitioned scheduling, on 4 cores with shared memory banks. Task worst case response times shown in this figure include delays due to the shared memory.

5.2. *Scalability of the Approach*

In order to validate the scalability of the approach, we defined and generated three samples of larger multiprocessor architectures, including some counter examples (to mimic modeling errors). The three scalabilty validation samples are built as generalizations of the 5.1.1 example for the two first samples, and the 5.1.2 example for the third sample, with larger number of tasks and PEs.

The three samples thus respectively illustrate three kinds of architecture with the following constraints:

- global scheduling with compatible tasks (DEP_COMP), task partitioning by PE sets (DEP_GLOB), and the deployment defines only one PE set for all the tasks, i.e. all the tasks are allowed on all the PEs;
- global scheduling with compatible tasks (DEP_COMP), PE set task partitioning (DEP_GLOB) and the deployment defines several disjoint PE sets, i.e. each task is associated to a set of PEs;
- partitioned scheduling (DEP_PART), independent PEs (EXE_INDE) and each processor has a level 1 private instruction cache and access to shared

memory, i.e. the temporal and functional behavior of the PEs that perform a task is independent of the status of other PEs, in the absence of functional interference explicitly modeled.

We ran the *dp_check* algorithm on the *Prolog* architecture model samples in order to verify the compliance with the five design patterns (DP1, DP2, DP3, DP4 and DP5) while measuring execution time. For each item of the samples, the given execution times are calculated by averaging the execution times over 10 executions (for times lower than $2s$), or 3 otherwise. We ran this experiment on a standard laptop with an i7-8665U Processor (Intel Core 8th, 4 cores, HT, 4.8Ghz, 16Go 2400MHz DDR4 RAM).

Hereafter are presented the obtained results in terms of:

- full *dp_check* time execution (including architecture model loading and compiling, and DP1 to DP5 compliance checking);
- compliance checking time execution for each design pattern (DP1 to DP5);
- size of the *Prolog* architecture model (Lines of Code);
- detection of erroneous models.

5.2.1. Sample 1: Global Scheduling

The first sample architecture models are built by extending the example 5.1.1, using a larger number of tasks and PEs. It is therefore a multiprocessor system without any hardware shared resource and a Proportionnate-Fair global scheduling. The software architecture is made of periodic tasks scheduled on a runtime environment including several processors, and complies with `LOG_UPG`. All processors are identical and independent. All tasks are allowed to execute on all PEs. As confirmed by the execution of our *dp_check* algorithm, the architecture model actually complies with DP1 and DP3 patterns.

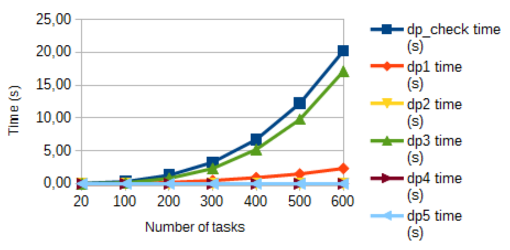
The first experiment measures the impact of the number of tasks on the execution time of *DP_cheker* and on the size of the *Prolog* model. The results are shown Fig. 11.

The second experiment is nearly the same except we analyze the effect of the number of processors (Fig. 12).

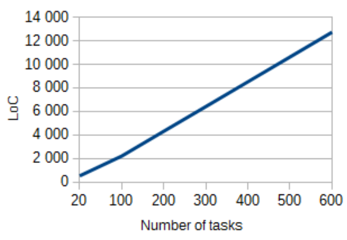
Finally, for a last experiment on this sample (Fig. 13), we make the assumption that the system designer adapts the number of processors she/he uses, to the complexity of her/his software architecture in term of number of tasks. So, the ratio between the number of tasks and the numbers of PEs is set to 10.

These 3 experiments show that:

- The execution time of the checker remains under $2s$ for models that manage upto 200 tasks on 20 processors following a global scheduling policy.
- On the design patterns we check, the execution time grows faster with the number of PEs than with the number of tasks.

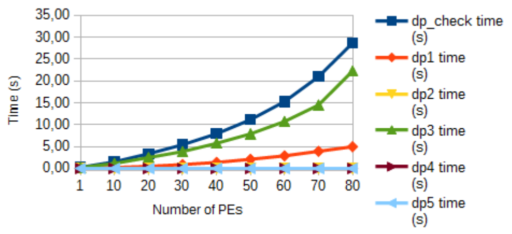


(a) Execution Time

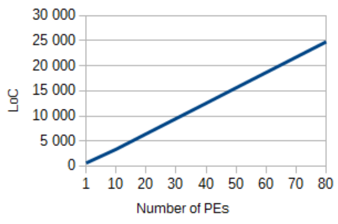


(b) LoC

Figure 11: Execution Time and Prolog Model LoC depending on the number of tasks. The number of PEs is set to 20, whereas the number of tasks varies from 20 to 600. In the drawing, the curves for DP2, DP4 and DP5 overlap.



(a) Execution Time



(b) LoC

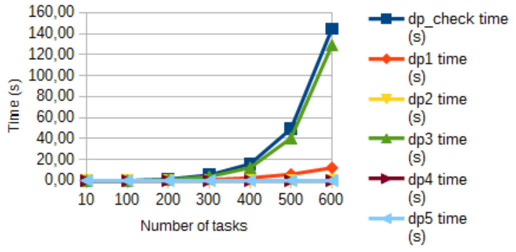
Figure 12: Execution Time and Prolog Model LoC depending on the number of processors. The number of tasks is set to 300, whereas the number of processors varies from 1 to 80. In the drawing, the curves for DP2, DP4 and DP5 overlap.

- The execution times of each individual design pattern dp_i are measured as a part of the overall execution of *dp_check*. Hence, these times do not take into account the initial time related to the loading of the model (i.e. $time_{dp\_check} = time_{load} + \Sigma_i time_{dp_i}$). The load time $time_{load}$ is about $3s$ when the size of the test model is larger than 20000 Lines of Code.
- The detection of non-compliant patterns takes less time than the detection of the other ones. This property is interesting in the perspective of a large catalog of design patterns that contains many specialized patterns.

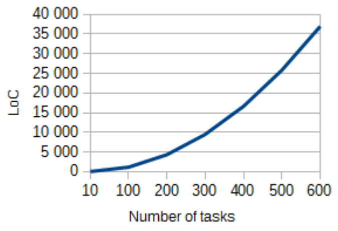
5.2.2. Sample 2: Global Scheduling on Multiple PE *Sets*

The second sample architecture models are also built from the example 5.1.1. The only difference with the first sample is that the deployment defines several disjoint PE sets instead of a single one. Thus, each task is associated to a set of k PEs. In this sample, the architecture models are built for $k = 4$, with $Task\_Nb$ (ranging from 10 to 1500) tasks and PE_Nb (ranging from 10 to 375) PEs.

As shown by the execution of our *dp_check* algorithm, the model actually complies with DP1 and DP3 patterns.



(a) Execution Time



(b) LoC

Figure 13: Execution Time and Prolog Model LoC depending on the number of tasks and processors. The ratio between the number of tasks and the number of processors is fixed to around 10 (but the actual load of each processor depends of the global scheduling algorithm). In the drawing, the curves for DP2, DP4 and DP5 overlap.

Fig. 14 shows the execution time of the checker in function of the number of tasks. With respect to the previous sample, the number of PEs that may execute a task is reduced, and therefore the number of $DM\_PE\_scheduling$ attributes in the model. Reducing the number of such attributes is important, because the checking process to assess the consistency of the scheduling policy takes a significant time. For instance, the execution time for the couple $(Tasks\_Nb, PE\_Nb) = (300, 80)$ is $28,7s$ for a global scheduling policy of all PEs (Fig. 12a), and only $0.3s$ if we apply the same policy on multiple sets of 4 PEs.

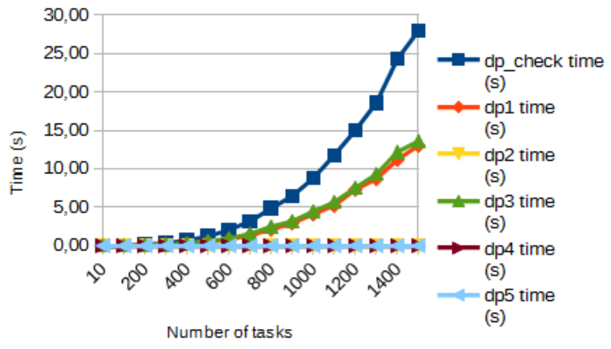


Figure 14: Execution time depending on the number of tasks. In the drawing, the curves for DP1 and DP3 on the one hand, and DP2, DP4 and DP5 on the other hand, overlap.

5.2.3. Sample 3: Instruction cache impact on scheduling

The last sample architecture models are built from the 5.1.2 example. In this sample, the architecture presented Section 5.1.2 has been scaled from a dual-processor to a multiprocessor architecture (with a PE_Nb processor number ranging from 1 to 100). The software architecture consists of $Task\_Nb$ (ranging from 10 to 1000) tasks and complies with `LOG_UPG`. In the sample, the architecture models are built with $Task\_Nb = 10 \times PE\_Nb$ in order to assign to each processor a group of 10 tasks. Each processor has a level 1 private instruction cache and access to shared memory. The bandwidth of the memory bus is fairly allocated to all processors by a TDM bus frame constituted of equal duration slots. Thus, the temporal and functional behavior of the PEs that perform a task is independent of the status of other PEs, in the absence of functional interference explicitly modeled in the system.

The curves drawn Fig. 15 show that the time to check the 5 considered patterns remains less than $20s$ even on software architectures composed of 1000 tasks and 100 processors.

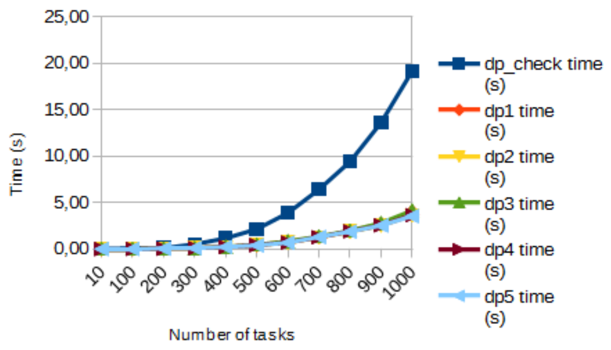


Figure 15: Execution Time. The number of processors is equal to the number of tasks divided by 10. In the drawing, the curves for all the patterns overlap.

5.2.4. Erroneous Model Detection

Finally, we generated erroneous models by corrupting instances of the previous samples. Inconsistent deployment attributes (e.g. $DM\_PE\_allowed$ and/or $DM\_PE\_scheduling$ facts) have been included in the biggest models of each sample used for this evaluation. The checker detects the badly formed models in the same range of time than the correct models ($28.97s$ in the worst case for our tries).

To sum up, we expect the temporal behavior of our proposal is adapted to the scale of the real-time systems we want to analyze. The growing of the execution time when the checker verifies the compliance of a model to numerous design patterns remains to investigate. However, as noted earlier, the time for checking non-compliant models was lower or similar to the time for checking compliant models during our experiments, and this observation suggests a moderate increase of execution time in function of the number of design patterns.

6. Conclusion

With the rapid spread of multicore runtime environments, being able to verify temporal behavior of systems running in these environments, is a major challenge for the real-time schedulability analysis community.

Today, the *Cheddar* tool includes several schedulability analysis methods (by simulation, by feasibility tests), and several design space exploration methods, adapted to multiprocessor architectures.

However, using these analysis methods remains difficult. Indeed, a tool like *Cheddar* may offer many analysis methods, each requiring that the system to be analyzed complies with several applicability assumptions. Moreover, the multiprocessor context enforces to integrate a model of the hardware execution support to explain the interference due to hardware resources. This interference is difficult to exhibit and understand by the scheduling analysis tool user.

The purpose of this article is to formalize the applicability constraints of the analysis methods implemented in *Cheddar*, and in particular the interference due to the runtime environment hardware resources.

From this formalization, we developed a tool implemented in the *Prolog* language, that checks the compliance of an application model (including the specification of the runtime environment), with a catalog of design patterns. The outcome is a list of analysis methods that can be applied on the input system. Today, 5 design patterns are available and recognized by the tool, but the way we express the constraints on the system allows for a rather simple extension by appending new *Prolog* inference rules. Our approach has been applied on 3 representative examples, and also on large automatically generated systems. The checking time is short enough to deal with the complexity of the current real-time applications.

To spread more thoroughly these proposals, we consider to include these design pattern checking approach within the *AADL Inspector* product[3] which already incorporates the *Cheddar* tool and LMP (Logic Model Processing) [50, 51]. LMP is currently used for integrating Cheddar into *AADL Inspector* and expresses the descriptive elements of the studied architecture. Adapting our *Prolog* implementation of *dp_check* to LMP may provide design pattern verification capabilities to *AADL Inspector*.

7. Artefact

All experiment data and programs presented in this paper are available at `http://beru.univ-brest.fr/svn/CHEDDAR/trunk/artefacts/DPCHECK22`.

[3]`https://www.ellidiss.com/aadl-inspector-1-6`

Programs and scripts written to produce these experimental data are available at http://beru.univ-brest.fr/svn/CHEDDAR/trunk/src.

References

[1] G. Tassey, The economic impacts of inadequate infrastructure for software testing, Tech. rep., National Institute of Standards and Technology (NIST) (2002).

[2] F. Singhoff, J. Legrand, L. Nana, L. Marcé, Cheddar: a flexible real-time scheduling framework, ACM SIGAda Ada Letters 24 (4) (2004) 1–8 (December 2004).

[3] V. Gaudel, F. Singhoff, A. Plantec, S. Rubini, P. Dissaux, J. Legrand, An Ada design pattern recognition tool for AADL performance analysis, ACM SIGAda Ada Letters 31 (3) (2011) 61–68 (November 2011).

[4] S. Fürst, Challenges in the design of automotive software, in: Proceedings of the Conference on Design, Automation & Test in Europe (DATE), European Design and Automation Association, IEEE, 2010, pp. 256–258 (Mar 2010).

[5] S. Saidi, R. Ernst, S. Uhrig, H. Theiling, B. D. de Dinechin, The shift to multicores in real-time and safety-critical systems, in: Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis, IEEE Press, 2015, pp. 220–229 (2015).

[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, Mass., 1994 (1994).
URL http://www.worldcat.org/search?qt=worldcat_org_all&q=0201633612

[7] R. Sanz, J. Zalewski, Pattern-based control systems engineering, IEEE Control Systems 23 (3) (2003) 43–60 (2003).

[8] M. J. Pont, Control system design using real-time design patterns, in: proceeding of the UKACC International Conference on Control, IET, 1998, pp. 1078–1083 (1998).

[9] P. H. Feiler, B. A. Lewis, S. Vestal, The SAE architecture analysis & design language (AADL) a standard for engineering performance critical systems, in: Proceedings of the Conference on Computer Aided Control System Design, of the International Conference on Control Applications, and of the International Symposium on Intelligent Control, IEEE, 2006, pp. 1206–1211 (2006).

[10] M. Z. Iqbal, S. Ali, T. Yue, L. Briand, Experiences of applying uml/-marte on three industrial projects, in: R. B. France, J. Kazmeier, R. Breu, C. Atkinson (Eds.), Model Driven Engineering Languages and Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, p. 642–658 (2012).

[11] P. Amey, B. Dobbing, Static analysis of ravenscar programs, in: ACM SIGAda Ada Letters, Vol. 23, ACM, 2003, pp. 58–64 (2003).

[12] A. Plantec, F. Singhoff, P. Dissaux, J. Legrand, Enforcing applicability of real-time scheduling theory feasibility tests with the use of design-patterns, in: Proceedings of the 4^{th} international conference on Leveraging applications of formal methods, verification, and validation-Volume Part I, Springer-Verlag, 2010, pp. 4–17 (2010).

[13] M. De Sanctis, C. Trubiani, V. Cortellessa, A. Di Marco, M. Flamminj, A model-driven approach to catch performance antipatterns in ADL specifications, Information and Software Technology 83 (2017) 35–54 (2017).

[14] A. Motii, B. Hamid, A. Lanusse, J.-M. Bruel, Guiding the selection of security patterns for real-time systems, in: Proceedings of the 21^{st} International Conference on Engineering of Complex Computer Systems (ICECCS), IEEE, 2016, pp. 155–164 (2016).

[15] G. Brau, J. Hugues, N. Navet, A contract-based approach to support goal-driven analysis, in: Proceedings of the IEEE 18th International Symposium on Real-Time Distributed Computing (ISORC), IEEE, 2015, pp. 236–243 (2015).

[16] Multi-core processors (rev 0), position paper (CAST32a), Tech. rep., Certification Authorities Software Team (2016).

[17] R. Urunuela, A.-M. Déplanche, Y. Trinquet, STORM: a simulation tool for real-time multiprocessor scheduling evaluation, in: Proceedings of the 15^{th} IEEE conference on Emerging Technologies and Factory Automation (ETFA), IEEE, 2010, pp. 1–8 (2010).

[18] A. Díaz-Ramírez, D. K. Orduño, P. Mejía-Alvarez, A multiprocessor real-time scheduling simulation tool, in: Proceeedings of the 22^{nd} International Conference on Electrical Communications and Computers (CONIELECOMP), IEEE, 2012, pp. 157–161 (2012).

[19] Y. Chandarli, F. Fauberteau, D. Masson, S. Midonnet, M. Qamhieh, Yartiss: A tool to visualize, test, compare and evaluate real-time scheduling algorithms, in: Proceedings of the 3^{rd} International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS), 2012, pp. 21–26 (2012).

[20] M. Chéramy, P.-E. Hladik, A.-M. Déplanche, Simso: A simulation tool to evaluate real-time multiprocessor scheduling algorithms, in: Proceedings of

the 5^{th} International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS), 2014 (2014).

[21] M. G. Harbour, J. G. García, J. P. Gutiérrez, J. D. Moyano, MAST: Modeling and analysis suite for real time applications, in: Proceedings of the 13^{th} Euromicro Conference on Real-Time Systems (ECRTS), 2001, pp. 125–134 (2001).

[22] V. Gaudel, Applicabilité des méthodes d'analyse et interopérabilité des outils de développement pour systèmes embarqués temps-réel critiques, Ph.D. thesis, Université de Bretagne Occidentale, Brest, France, (in french) (December 2014).

[23] F. Cottet, J. Delacroix, C. Kaiser, Z. Mammeri, Scheduling in Real-Time Systems, Wiley Online Library, 2002 (2002).

[24] P. Dissaux, F. Singhoff, Stood and cheddar: AADL as a pivot language for analysing performances of real time architectures, in: Proceedings of the European Real Time System conference. Toulouse, France, Vol. 32, 2008 (2008).

[25] C. Fotsing, F. Singhoff, A. Plantec, V. Gaudel, S. Rubini, S. Li, H. N. Tran, L. Lemarchand, P. Dissaux, J. Legrand, Cheddar architecture description language, Tech. rep., Lab-STICC,Université de Bretagne Occidentale (2014).

[26] L. Sha, R. Rajkumar, J. P. Lehoczky, Priority inheritance protocols: An approach to real-time synchronization, IEEE Transactions on Computers 39 (9) (1990) 1175–1185 (Sep. 1990).

[27] A. Burns, B. Dobbing, G. Romanski, The ravenscar tasking profile for high integrity real-time programs, in: Proceedings of the 1998 Ada-Europe International Conference on Reliable Software Technologies, Springer-Verlag, London, UK, 1998, pp. 263–275 (1998).

[28] W. Barnes, ARINC 653 and why is it important for a safety-critical RTOS, Boards & Solutions (2004) 16 (2004).

[29] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, R. I. Davis, A survey of timing verification techniques for multi-core real-time systems, Tech. Rep. TR-2018-9, Verimag Research Report (2018).

[30] J. Goossens, Ordonnancement temps réel multiprocesseur (in french), in: N. Navet (Ed.), Systèmes temps réel T. 2 - Ordonnancement, réseaux et qualité de service, Traité IC2, Information - Commande - Communication, Hermès - Lavoisier, 2006, Ch. 2, p. 336 (2006).

[31] J. Goossens, E. Grolleau, L. Cucu-Grosjean, Periodicity of real-time schedules for dependent periodic tasks on identical multiprocessor platforms, Real-time systems 52 (6) (2016) 808–832 (2016).

[32] N. C. Audsley, K. Tindell, A. Burns, The end of the line for static cyclic scheduling?, in: Proceedings fo the Fifth Euromicro Workshop on Real-Time Systems, IEEE, 1993, pp. 36–41 (jun 1993).

[33] K. Tindell, Adding time-offsets to schedulability analysis, Tech. rep., University of York, Department of Computer Science (1994).

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

[35] J. C. Palencia, M. G. Harbour, Schedulability analysis for tasks with static and dynamic offsets, in: Proceedings of the 19^{th} IEEE Real-Time Systems Symposium, IEEE, 1998, pp. 26–37 (1998).

[36] S. Li, F. Singhoff, S. Rubini, M. Bourdellès, Scheduling analysis of tasks constrained by TDMA: Application to software radio protocols, Journal of Systems Architecture 76 (2017) 58–75 (2017).

[37] S. Altmeyer, R. I. Davis, C. Maiza, Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems, Real-Time Systems 48 (5) (2012) 499–526 (2012).

[38] H. N. Tran, S. Rubini, J. Boukhobza, F. Singhoff, Feasibility interval and sustainable scheduling simulation with crpd on uniprocessor platform, Journal of Systems Architecture 115 (2021) 102007 (2021).

[39] M. Dridi, F. Singhoff, S. Rubini, J.-P. Diguet, Ectm: A network-on-chip communication model to combine task and message schedulability analysis, Journal of Systems Architecture 114 (2021) 101931 (2021).

[40] H. Kim, D. De Niz, B. Andersson, M. Klein, O. Mutlu, R. Rajkumar, Bounding memory interference delay in cots-based multi-core systems, in: 2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), IEEE, 2014, pp. 145–154 (2014).

[41] Y. Oh, S. H. Son, Tight performance bounds of heuristics for a real-time scheduling problem, Tech. Rep. CS-93-24, University of Virginia, Dept. of Computer Science (1993).

[42] A. Burchard, J. Liebeherr, Y. Oh, S. H. Son, Assigning real-time tasks to homogeneous multiprocessor systems, Tech. rep., University of Virginia, Dept. of Computer Science (1994).

[43] M. Joseph, P. Pandya, Finding response times in a real-time system, The Computer Journal 29 (5) (1986) 390–395 (1986).

[44] R. I. Davis, A. Burns, A survey of hard real-time scheduling for multiprocessor systems, ACM computing surveys (CSUR) 43 (4) (2011) 35 (2011).

[45] H. N. Tran, F. Singhoff, S. Rubini, J. Boukhobza, Cache-aware real-time scheduling simulator: implementation and return of experience, ACM SIGBED Review 13 (1) (2016) 22–28 (2016).

[46] H. N. Tran, Cache memory aware priority assignment and scheduling simulation of real-time embedded systems, Ph.D. thesis, Université de Bretagne Occidentale, Brest, France (Jan 2017).

[47] N. C. Audsley, Optimal priority assignment and feasibility of static priority tasks with arbitrary start times, in: Technical Report YCS 164, Dept. Computer Science, University of York, UK, 1991 (1991).

[48] A. Colmerauer, P. Roussel, History of programming languages—II, ACM, New York, NY, USA, 1996, Ch. The Birth of Prolog, pp. 331–367 (1996).

[49] C. Bienia, S. Kumar, J. P. Singh, K. Li, The parsec benchmark suite: Characterization and architectural implications, in: Proceedings of the 17th international conference on Parallel architectures and compilation techniques, 2008, pp. 72–81 (2008).

[50] P. Dissaux, B. Hall, Merging and processing heterogeneous models, in: Proceedings of the 8^{th} European Congress on Embedded Real Time Software and Systems (ERTSS), 2016 (2016).

[51] P. Dissaux, P. Farail, Model verification: Return of experience, in: Proceedings of the 6^{th} European Congress on Embedded Real-Time Software and System conference (ERTSS), 2014 (2014).

Appendix A. Annexe

Prolog implementations of the design patterns DP2, DP3, DP4 and DP5:

```
dp2 :- sw_archi(A),
       member(A,[log_upg,log_rav,log_sync]),
       tasks(LTasks),
       processing_elements(LPE), !,
       dep_part(LTasks,LPE), !,
       exe_inde(LTasks,LPE), !,
       fea_id(LTasks), !.

dp3 :- sw_archi(A),
       member(A,[log_upg,log_rav,log_sync]),
       tasks(LTasks),
       processing_elements(LPE), !,
       dep_glob(LTasks,LPE), !,
       exe_inde(LTasks,LPE), !,
       fea_id(LTasks), !.
```

```
dp4 :- sw_archi(A),
       member(A,[log_upg,log_sync]),
       tasks(LTasks),
       processing_elements(LPE), !,
       dep_part(LTasks,LPE), !,
       exe_inde(LTasks,LPE), !,
       fea_id(LTasks), !,
       fea_ic(LTasks),!.

dp5 :- sw_archi(A),
       member(A,[log_upg,log_sync]),
       tasks(LTasks),
       processing_elements(LPE), !,
       dep_part(LTasks,LPE), !,
       exe_dep(LTasks,LPE), !,
       fea_id(LTasks), !,
       resources(LR), !,
       fea_dram(LTasks,LR),!.
```

Prolog implementations of the "constraint sets" checkers:

```
dep_part([],_).
dep_part([T],LPE):-
  dm_PE_actual(T,P),
  member(P,LPE), !,
  dm_PE_scheduling(P,T,_), !.
dep_part([T1|LT],LPE) :-
  dm_PE_actual(T1,P),
  member(P,LPE), !,
  dm_PE_scheduling(P,T1,S), !,
  (foreach(T2,LT), param(P,S) do
      (dm_PE_actual(T2,P) -> dm_PE_scheduling(P,T2,S)
      ;
       true)
      ), !,
  dep_part(LT,LPE).
```

```
dep_glob([],_).
dep_glob([T],LPE):-
  dm_PE_allowed(T,[C|LC]),
  dm_PE_scheduling(C,T,S),
  (foreach(P,[C|LC]), param(LPE,T,S) do
      member(P,LPE), !,
      dm_PE_scheduling(P,T,S), !
  ), !.
dep_glob([T1|LT],LPE) :-
```

```
  dm_PE_allowed(T1,LP1), !,
  LP1 = [P1|_],
  dm_PE_scheduling(P1,T1,S1), !,
  (foreach(P,LP1), param(LPE,T1,S1) do
     member(P,LPE), !,
     dm_PE_scheduling(P,T1,S1), !
  ), !,
  (foreach(T2,LT), param(LP1,P1,S1) do
     dm_PE_allowed(T2,LP2),
     (subtract(LP1,LP2,[]),
      subtract(LP2,LP1,[]),
      dm_PE_scheduling(P1,T2,S1)
      ;
      intersection(LP1,LP2,[]))
  ), !,
  dep_glob(LT,LPE).
```

```
exe_inde(LT,_) :-
  (foreach(T,LT) do
    dm_PE_allowed(T,LP), !,
    LP = [_|_],
    (foreach(P,LP) do ha_independent(P)), !,
    (dm_PE_actual(T,P) -> ha_independent(P) ; true), !).
```

```
exe_dep([T],_) :-
  dm_PE_allowed(T,LP),
  LP = [_|_],
  member(P,LP),
  ha_dependent(P), !.
exe_dep([T],LP) :-
  dm_PE_actual(T,P),
  member(P,LP),
  ha_dependent(P), !.
exe_dep([T|_],_) :-
  dm_PE_allowed(T,LP),
  LP = [_|_],
  member(P,LP),
  ha_dependent(P), !.
exe_dep([T|_],LP) :-
  dm_PE_actual(T,P),
  member(P,LP),
  ha_dependent(P), !.
exe_dep([_|LT],LP) :-
  exe_dep(LT,LP).
```

```
fea_id(LT) :-
  build_set_DM_PE_allowed_actual(LT,LPaa), !,
  (find_a_PE_speed(_,S,LPaa) ->
    (foreach(P,LPaa), param(S) do
       (a_PE_speed(P,S2) -> S=S2 ; true))
    ;
    true), !,
  (find_a_PE_isa(_,I,LPaa) ->
    (foreach(P,LPaa), param(I) do
       (a_PE_isa(P,I2) -> I=I2 ; true))
    ;
    true), !.
```

```
fea_ic(LT) :-
  build_set_DM_PE_allowed_actual(LT,LPaa), !,
  (foreach(P,LPaa) do
    am_PE_use(P,[R]),
    a_mem_type(R,instruction_cache_type),
    a_mem_cache_associativity(R,1),
    a_mem_cache_miss_time(R,_), !).
```

```
fea_dram(LT,LR) :-
  build_set_DM_PE_allowed_actual(LT,LP_allowed_actual), !,
  build_set_DM_R_allowed_actual(LT,LR_allowed_actual), !,
  (foreach(R,LR_allowed_actual) do a_mem_type(R,bank)), !,
  LP_allowed_actual = [P1|LP_allowed_actual2],
  am_PE_use(P1,[R]), !,
  member(R,LR), !,
  a_mem_type(R,idCache), !,
  (foreach(P,LP_allowed_actual2), param(R) do
    am_PE_use(P,[R])), !,
  am_R_use(R,B), !,
  (foreach(R,LR_allowed_actual), param(B) do
    member(R,B)), !.
```