Cheddar Architecture Description Language

Christian Fotsing, Frank Singhoff, Stéphane Rubini,
Hai Nam Tran, Alain Plantec, Laurent Lemarchand,
Shuai Li, Mourad Dridi, Ill-ham Atchamdam, Vincent Gaudel
Lab-STICC/UMR CNRS 6285, University of Brest
20, av Le Gorgeu, CS 93837, 29238 Brest Cedex 3, France
Email: {$first\_name.last\_name$}@univ-brest.fr

Pierre Dissaux, Jérôme Legrand
Ellidiss Technologies
24, quai de la douane
29200 Brest, France
Email: {$first\_name.last\_name$}@ellidiss.com

August 16, 2023

Abstract

The aim of this paper is to give a complete and fine definition of the Cheddar Architecture Design Language.

Cheddar is a free real time scheduling tool composed of a graphical editor used to describe a real-time applications, a framework which includes most of classical real time scheduling/feasibility algorithms/tests. It is designed for checking temporal constraints of real-time applications.

To perform this type of scheduling analysis with Cheddar, systems to analyse can be described with AADL or with a dedicated ADL, the Cheddar Architecture Design Language, called Cheddar ADL.

Cheddar ADL aims to write, analyse and validate real-time applications handled in the context of Cheddar.

Our objective in this report is to describe it formally, to also show how it may be implemented and used.

Keywords: ADL; Cheddar; Real-Time Systems; Validation; Simulation;

Table of Contents

List of Figures

List of Tables

Guidelines to read

(1) For each entity, we have several sub-paragraphs.

Standard attributes

We define in this part the main properties of entity.

Legality rules

(L1) We define here the constraints of entity.

Annexes

(A1) This part completes the standard attributes definitions. It allows to define precisely the attributes.

(A2) The annexes of an entity allow to define the sub attributes of this entity.

Implementation

This corresponds to a table, which give the precise denomination of attributes with their types.

Example

We present here an example of use of entity.

Definitions

(1) We define here a set of usual terms in this paper.

Definition 1 *Valid identifier.*
An identifier is valid if it is composed only by the letters of the alphabet (upper-case or lower-case), digits from 0 to 9 and characters ., :, /, \ *and* _.

1 Introduction

We consider real-time applications, dedicated to process control, which are modelled by a set of tasks. They are characterized by the presence of temporal constraints, induced by the dynamics of the controlled process. The tasks can be periodic or not, have same first release time or not, may shared resources or exchanges messages.

For study schedulability analysis, each task T_i is usually modelled by four temporal parameters [10]: its first release time r_i, its worst case execution time (WCET) C_i, which is the highest computation time of the task, its relative deadline D_i, which is the maximum acceptable delay between the release and the completion of any instance of the task, and its period P_i.

From this simplified model of task, real-time scheduling theory provides two ways to perform schedulability analysis: feasibility tests and scheduling simulation on the hyperperiod[1].

We have a necessary condition for a system to be feasible [11]: if a system of n tasks $T_1 \ldots T_n$ is feasible, then its utilization factor, defined by

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i}$$

is less than or equal to 1.

Our general aim is the validation, using simulation, of the real-time applications.

For that, we consider the Cheddar project. Cheddar [9] is a GPL open-source schedulability tool composed of a graphical editor and a library of schedulability analysis modules. The library of analysis modules implements various analysis methods based on the real-time scheduling theory.

In order to perform schedulability analysis in Cheddar context, several approaches have been investigated [12]: The MARTE based approach [37] [13] where a MARTE to Cheddar transformation have been proposed, and an experimentation on an industrial software radio system have been done; The Model Driven Engineering (MDE) [53] based approach, combined to ADL have been experimented by showing how scheduling analysis tools can be automatically produced from an ADL with MDE [38] [39] [40]; An AADL based approach [41] where the relevant hardware features are expressed in order to have better scheduling analysis.

[1] The lcm periods

We focus on this paper to the ADLs approach, by presenting Cheddar ADL. It will allow us to finely specify and validate the real-time application, in the Cheddar context [12]. Indeed, Cheddar ADL, especially dedicated to scheduling analysis, provides tools to design and validate (by using Cheddar tool) real-time applications.

The paper presents an Architecture Description Language (ADL) that has been designed to model software architecture in the perspective of scheduling analysis. This ADL illustrates how an ADL may provide to model a real-time application on which designers expect to perform scheduling analysis.

The rest of the paper is organized as follows: we begin by enumerate the requirements of Cheddar ADL (section 2), in order to lay the foundations of our language, then, we describe the general concepts of the language (section 3). After-that, the sections 4 and **??** precise the semantic of the basic entities of Cheddar ADL, and section 5 is dedicated to the way to apply it. Section 6 gives the related works, by describing some ADLs using in real-time domains, and we conclude in section 7, by giving some perspectives.

2 Requirements

In order to allow interoperability with other ADLs, we define some requirements for Cheddar ADL:

- Applicable in several areas: It should be take various concepts like FPGA, Multiprocessor architectures with caches/cores and shared memory, N-level of hierarchical scheduling into account.
- Should be as close as possible with real-time scheduling and queueing system theories: Classic models, synchronization/dependencies, scheduler parameters.
- Must stay simple and easy/quick to use/understand.
- Maintain existing features: allow transformation to AADL/Marte/others supported ADLs.
- Provide isolation : spatial and temporal. The aim is to enable independently spatial analysis (take the behaviour of system into account, model checking ...) and temporal analysis (compute the respond time, the schedulability analysis ...).

- Both hardware/software modelling and software deployment: required for real-time systems analysis, user/designer understanding.

The next section aims to present the general concepts of Cheddar ADL, based on these requirements.

3 Cheddar ADL Concepts

Cheddar ADL defines basic entities which model usual concepts of the real-time scheduling theory. We have two kinds of entities:

1. Components: There are the reusable units. A component has a type, an unique name and attributes. It is a part of a system to analyse.

2. Bindings: the bindings define relationships between components. They Model a resource allocation between n providers and m consumers, where n and m are integers.

These basic entities can be grouped into 3 types:

1. Hardware components: They model resources provided by the execution environment. We have $Processor$, $Cache$, $Core$, $Memory$ and $Network$.

2. Software components: They model the design of the software. They are deployed onto hardware components. In Cheddar, we have $Task$, $Resource$, $Buffer$, $Dependency$ and $Message$.

3. Bindings: Their role is to enforce either temporal or spatial isolation. They allow to model the relationships between components.

In order to finely describe the Cheddar ADL, we clarify in sections 4 and **??** the semantic of basic entities.

4 Semantic of Components

We distinguish in Cheddar two types of components: hardware components and software components.

4.1 Hardware Components

(1) We define in this section the following hardware components: *Cache* , *Core* , *Processor* , *Memory* and *Network* .

(2) *Cache* which models a hardware cache unit.

(3) *Core* which models an entity providing a resource to sequentially run flow of controls.

(4) *Processor* which corresponds to the deployment unit for a software component.

(5) *Memory* which models an entity providing a physical memory unit.

(6) *Network* which models any entity allowing tasks located in different processors to exchange messages.

4.1.1 Core

A *Core* is specified by the following definitions [16]:

(1) It is a deployment unit for a software component.

(2) It is the unit that reads and executes program instructions.

Standard attributes

Name : It is the unique name of *Core* .

Speed: It is an positive integer, which scales execution time of any task run by this *Core* .

*L*1_*cache*_*system*_*name*: It is a string, which corresponds to the primary cache. It is faster, generally smaller, and located in core.
The *L*1_*cache*$_s$*ystem* is a list of 0 or several *Caches* .
Each *Cache* can be of 2 types:

- $Unified$: in this case, it corresponds to $Data\_Instruction\_Cache$.
- $Separated$: in this case, it corresponds to 1 $Data\_Cache$ and 1 $Instruction\_Cache$.

Scheduling: It defines all parameters of the mean tasks will be scheduling on the core.

The type of this data is $Scheduling\_Parameters$ and it is defined the Annex of this section.

Legality rules

(L1) The $Core$ name must not be empty.

(L2) The $Core$ name must be valid identifier.

(L3) We must not have simultaneous ($A\_Scheduler = Pipeline\_User\_Defined\_Protocol$) and ($File\_Name\ Empty$).

(L4) We must not have simultaneous ($File\_Name \neq Empty$) and ($A\_Scheduler \neq Pipeline\_User\_Defined\_Protocol$) and ($A\_Scheduler \neq Automata\_User\_Defined\_Protocol$

(L5) The $Period$ must be greater than or equal to 0.

(L6) The $Capacity$ must be greater than or equal to 0.

(L7) We must not have ($Priority < Priority\_Range'First$) or ($Priority < Priority\_Range'Last$).

(L8) We must not have simultaneous ($Quantum \neq 0$) and ($A\_Scheduler \neq Posix\_1003\_Highest\_Priority\_First\_Protocol$) and ($A\_Scheduler \neq Round\_Robin\_Protocol$ and ($A\_Scheduler \neq Hierarchical\_Round\_Robin\_Protocol$) and ($A\_Scheduler \neq Hierarchical\_Cyclic\_Protocol$).

(L9) The $Quantum$ must be greater than or equal to 0.

(L10) The $Speed$ must be greater than or equal to 0.

```
<!ELEMENT core_units (core_unit)+>
<!ELEMENT core_unit (object_type | name
  | scheduling | speed | l1_cache_system_name)*>
<!ATTLIST core_unit id ID #REQUIRED>
```

Figure 1: The DTD of entity *Core*

(L11) *A_Scheduler* must be different of *No_Scheduling_Protocol*.

Annexes

(A1) See Annexes of *Dynamic_Deployment* for attributes of *Scheduling_Parameters*.

Implementation

The figure 1 gives the DTD of entity *Core* .

Example

The figure 2 gives an example of entity *Core* described using Cheddar ADL.

In this case, the scheduler *POSIX_1003_HIGHEST_PRIORITY_FIRST_PROTOCOL* is fixed on the core named *core*1, and it is of *Preemptive* Type. The *L1_cache_system_name* is not represented.

Annexes

(A1) The *Scheduling_Parameters* attributes.

A scheduler is responsible for selecting the running task for each unit of time from among the set of ready tasks on a given core. A scheduler is defined by a policy and various parameters. The available policies are:

(A11) *Scheduler_type*: Which defines the scheduling policy, and may be [17] [18] [19], [20]:

```
<core_unit id="_67" >
        <object_type>CORE_OBJECT_TYPE</object_type>
        <name>core1</name>
        <scheduling>
                <scheduling_parameters>
                        <scheduler_type>
              POSIX_1003_HIGHEST_PRIORITY_FIRST_PROTOCOL
                        </scheduler_type>
                        <quantum>100</quantum>
                        <preemptive_type>PREEMPTIVE
                        </preemptive_type>
                        <capacity>101</capacity>
                        <period>102</period>
                        <priority>103</priority>
                        <start_time>0</start_time>
                </scheduling_parameters>
        </scheduling>
        <speed>1</speed>
</core_unit>
```

Figure 2: An example of *Core* description

(A111) $Compiled\_User\_Defined\_Protocol$: ...

(A112) $Automata\_User\_Defined\_Protocol$: ...

(A113) $Pipeline\_User\_Defined\_Protocol$: ...

(A114) $User\_Defined\_Protocol$: ...

(A115) $Earliest\_Deadline\_First\_Protocol$: Tasks can be periodic or not and are scheduled according to their $Deadline$.

(A116) $Least\_Laxity\_First\_Protocol$: Tasks can be periodic or not and are scheduled according to their laxity.

(A117) $Rate\_Monotonic\_Protocol$: He attributes the higher priority to the task which has the smallest period.

(A118) $Deadline\_Monotonic\_Protocol$: At every scheduling point, the task having the shortest deadline is taken up for scheduling.

Tasks have to be periodic and are scheduled according to their deadline. You have to be aware that the value of the priority field of the tasks is ignored here.

(A119) $Round\_Robin\_Protocol$: ...

(A1110) $Time\_Sharing\_Based\_On\_Wait\_Time\_Protocol$: ...

(A1111) $Posix\_1003\_Highest\_Priority\_First\_Protocol$: ...

(A1112) $D\_Over\_Protocol$: ...

(A1113) $Maximum\_Urgency\_First\_Based\_On\_Laxity\_Protocol$: ...

(A1114) $Maximum\_Urgency\_First\_Based\_On\_Deadline\_Protocol$: ...

(A1115) $Time\_Sharing\_Based\_On\_Cpu\_Usage\_Protocol$: ...

(A1116) $No\_Scheduling\_Protocol$: ...

(A1117) $Hierarchical\_Cyclic\_Protocol$: This policy runs cyclically the set of address spaces. The address space scheduling is computed with the order of the address space in the CheddarADL model. The cycle is always the same. Each time an address space is activated, it stays activated during its capacity. So each address space may be activated during a different amount of time.

(A1118) $Hierarchical\_Round\_Robin\_Protocol$: This policy runs cyclically the set of address spaces. The address space scheduling is computed with the order of the address space in the CheddarADL model. The cycle is always the same. Each time an address space is activated, it stays activated during the quantum value. So all address space are activated during the same amount of time.

(A1119) $Hierarchical\_Fixed\_Priority\_Protocol$: The address space scheduling is computed with the fixed priority of the address space in the CheddarADL model. The scheduling policy use the period and start time attributes to compute the address space scheduling. The core quantum is not used with this scheduling policy.

(A1120) $Hierarchical\_Polling\_Aperiodic\_Server\_Protocol$: ...

(A1121) $Hierarchical\_Priority\_Exchange\_Aperiodic\_Server\_Protocol$:...

(A1122) $Hierarchical\_Sporadic\_Aperiodic\_Server\_Protocol$:
...

(A1123) $Hierarchical\_Deferrable\_Aperiodic\_Server\_Protocol$:
...

(A1124) $Proportionate\_Fair\_PF\_Protocol$: ...

(A1125) $Proportionate\_Fair\_PD\_Protocol$: ...

(A1126) $Proportionate\_Fair\_PD2\_Protocol$: ...

(A12) $Quantum$: It is the quantum value associated with the $Scheduler$. It is a natural, which defines the smallest unit of execution time of a task.

A time quantum is a maximum duration that a task can run on the $Processor$ or $Core$ before being pre-empted by another task of the same queue.

This information is useful if a scheduler has to manage several tasks with the same dynamic or static priority : in this case, the simulator has to choose how to share the processor between these tasks. The quantum is a bound on the delay a task can hold the processor (if the quantum is equal to zero, there is no bound on the processor holding time).

(A13) $Preemptive\_type$: It characterizes the scheduler type. We have two types:

(A131) $Preemptive$: When the running task is interrupted for some time and resumed later when the priority task has finished its execution.

(A132) $Not\_preemptive$: In this case, a running task is executed till completion. It cannot be interrupted.

(A14) $Automaton\_name$: ...

(A15) $Capacity$: It is the worst case execution time of a $Task$.

(A16) $Period$: It is duration between two periodic release times.

In this case, a task starts a job at each release time.

(A17) $Priority$: It is a priority range. It is an integer, which allows the scheduler to choose the task to run.

(A18) $User\_Defined\_Scheduler\_Source$: ...

(A19) $User\_Defined\_Scheduler\_Source\_File\_Name$: the file name of a file which contains the source code of a $User\_Defined\_Scheduler$.

```
<scheduling_parameters>
 <scheduler_type>RATE_MONOTONIC_PROTOCOL
     </scheduler_type>
 <quantum>0</quantum>
 <preemptive_type>PREEMPTIVE
     </preemptive_type>
 <capacity>0</capacity>
 <period>0</period>
 <priority>0</priority>
 <start_time>0</start_time>
 </scheduling_parameters>
```

Figure 3: An example of Scheduling parameters of a Core Unit

(A110) *Start_Time*: It is the first release time of a *Task*.

4.1.2 Processor

The entity *Processor* is specified by the following definitions:

(1) It is a deployment unit for a software component.

(2) It is composed by a set of *Cores* and *Caches* .

(3) We distinguish in Cheddar two separate cases: **Mono_Core_Processor** and **Multi_Core_Processor**.

Standard attributes

Name : It is the unique name of *Processor*.

Network: It is a string, which corresponds to the name of entity *Network* connected to the *Processor* .

Processor_Type: It is an enumeration, which defines the type of considered *Processor* .

Migration_Type: It is an enumeration, which defines the type of *Tasks* migration between the *Cores* of the *Processor* .

In Cheddar, we assume that a *Task* may be migrated only between jobs.

Legality rules

(L1) The *Processor* name must not be empty.

(L2) The *Processor* name must be valid identifier.

(L3) *Cores* should not be empty in the *Multi_cores_processor* case.

(L4) If the *Processor* type is *Monocore_Type*, it must exist at least one *Core* .

(L5) If the *Processor* type is *Multicore_Type*, it must exist several *Cores* .

Annexes

(A1) The types of *Processors* [30] [29] [32] [33] [31].

(A11) *Monocore_Type*: It is a *Processor* with only one *Core* .
In this case, the *Processor* only executes one instruction flow, i.e. one *Task* , at a time.

The *Multicore_Type* references to a *Processor* with two or more processing units, i.e. *Core* components.

(A12) *Identical_Multicores_Type*: In this case, all processors are identical.

That means processors have the same computing capability and run task at the same rate.

(A13) *Uniform_Multicores_Type*: Each *Processor* P is characterized by a single parameter *speed* (or computing capacity), named $Speed(P)$, with the interpretation that a job that executes on processor P for t time

units completes $Speed(P) \times t$ units of execution.

(A14) $Unrelated\_Multicores\_Types$: In this case, $Processors$ differ in area, performance, power dissipated, speed, ...

An execution rate is defined for every uplet ($r_{i,j}$, i, j): the work i requires $r_{i,j}$ units of time on the processor j.

(A2) The types of Migrations [24] [25] [26].

(A21) $No\_Migration\_Type$: When no migration is allowed between the **Cores** of a $Processor$.

A $Task$ that begins its execution on a $Core$ cannot migrate and must always run on this one.

(A22) $Job\_Level\_Migration\_Type$: a $Task$ can run its successive jobs in different $Cores$.

When a job is started in one $Core$, the task cannot migrate before the job is completed, i.e. a job starting in a given $Core$ must be completed on the same $Core$.

Job-level parallelism is forbidden (i.e., a job may not execute concurrently with itself on two or more different $Cores$.

(A23) $Time\_Unit\_Migration\_Type$: A $Task$ can migrate at any time on any $Cores$ of the processor.

Job-level parallelism is also forbidden in this case.

(A3) **Mono_Core_Processor** is defined by the following parameter:

(A31) $Core$: It is the corresponding core to the **Mono_Core_Processor**.

It is characterized by:

(A311) $Scheduling$: See the $Core$ description.

(A312) $Speed$: See the $Core$ description.

(A313) $L1\_Cache\_System\_Name$: See the $Core$ description.

```
<!ELEMENT processors (generic_processor
  | mono_core_processor | multi_cores_processor)+>
<!ELEMENT generic_processor (object_type
  | name | network_name | processor_type |
  migration_type)*>
<!ATTLIST generic_processor id ID #REQUIRED>
<!ELEMENT mono_core_processor (object_type
  | name | network_name | processor_type |
  migration_type | core)*>
<!ATTLIST mono_core_processor id ID #REQUIRED>
<!ELEMENT multi_cores_processor (object_type
  | name | network_name | processor_type |
  migration_type
  | cores | l2_cache_system_name)*>
<!ATTLIST multi_cores_processor id ID #REQUIRED>
```

Figure 4: The DTD of entity *Processor*

(A4) **Multi_Core_Processor** is a single computing component with two or more independent *Cores* , which are the units that read and execute program instructions [15].

It allows to schedule tasks globally with a set of cores.

It is defined by the following parameters:

(A41) *Cores*: The list of *Cores* of the *Processor*.

(A42) $L2\_Cache\_System\_Name$: It is a list of 0 or several *Caches*. When it exists, it is $Unified$: it corresponds to $Data\_Instruction\_Cache$. In this case, all corresponding $L1\_Cache\_System$ are separated.

Implementation

The figure 4 gives the DTD of entity *Processor* .

Example

```
<processors>
        <mono_core_processor id="␣77" >
        <object_type>PROCESSOR_OBJECT_TYPE</object_type>
                <name>processor1</name>
                <network>a_network</network>
                <processor_type>MONOCORE_TYPE
                </processor_type>
                <migration_type>NO_MIGRATION_TYPE
                </migration_type>
                <core ref="␣75" />
        </mono_core_processor>
</processors>
```

Figure 5: An example of *Mono_core_processor* description

The example of figure 5 describes a *Mono_core_processor*, with one core referenced by *id* 75.

Notice the type of *Processor* (*MONOCORE_TYPE*) and the type of *Migration* (*NO_MIGRATION_TYPE*).

The example of figure 6 describes a *Multi_cores_processor*, with two cores referenced by their *id* (67, 68).

It has two identical cores (that means that the cores have the same computing capability and run *Task* at the same rate), and migrations between cores is of the type *time_unit_migration*; that means that a job requires one time unit for task migration between two cores.

4.1.3 Cache

The *Cache* is specified by the following definitions [2] [3] [4] [5] [6] [14]:

(1) It is a small high speed memory usually Static RAM (SRAM) that contains the most recently accessed pieces of main memory.

(2) Cache memories are small, high-speed buffer memories used in modern computer systems to hold temporarily those portions of the contents of main memory which are (believed to be) currently in use.

```
<processors>
        <multi_cores_processor id="␣69">
        <object_type>PROCESSOR_OBJECT_TYPE</object_type>
                <name>processor1</name>
                <network>a_network</network>
                <processor_type>IDENTICAL_MULTICORES_TYPE
                </processor_type>
                <migration_type>TIME_UNIT_MIGRATION_TYPE
                </migration_type>
                <cores>
                        <core_unit ref="␣67"/>
                        <core_unit ref="␣68"/>
                </cores>
        </multi_cores_processor>
</processors>
```

Figure 6: An example of *Multi_cores_processor* description

(3) In Cheddar ADL, the *Cache* is named *Generic_Cache* in the xml schema.

Standard attributes

Name : the unique name of a *Cache*.

Cache_Size: the size of a cache - the number of bytes that a cache can contain.

Line_Size: the number of contiguous bytes that are transferred from main memory on a cache miss. When the cache does not contain the memory block requested, the transaction is said to be a cache miss.

Associativity: the number of cache locations where a particular memory block may reside.

Block_Reload_Time: the time to access a memory block that is not in the cache.

Replacement_Policy:a policy which decides which cache block should be replaced when a new memory block needs to be stored in the cache.

Coherence_Protocol: a protocol which maintains the consistency between all the caches in a system of distributed shared memory.

Cache_Category: specifies a cache as instruction cache, data cache or a combined one.

Cache_Blocks: TO BE COMPLETED

Write_Policy: TO BE COMPLETED

Legality rules

(L1) The cache name must not be empty.

(L2) The cache name must be valid identifier.

(L3) $Cache\_size > 0$.

(L4) $Block\_size > 0$.

(L5) $Cache\_Size\ MOD\ Block\_Size = 0$.

(L6) $Hit\_Time > Miss\_Time > 0$.

(L7) The $Coherence\_Protocol$ of $Instruction\_Cache$ can only be $Shared\_Cache\_Protocol$ or $Private\_Cache\_Protocol$.

Annexes

(A1) The kinds of *Associativity* :

(A11) $Fully\_Associative\_Cache$: when a memory block can reside in any locations in the cache ($Associativity = Cache\_Size/Block\_Size$).

(A12) $Direct\_Mapped\_Cache$: when a memory block can reside in exactly one location in the cache ($Associtivity = 1$).

(A13) A way set associative cache: when a memory block can reside in exactly A location in the cache ($Associativity = A$).

In this case, we have

$$Number\_of\_set\_of\_the\_cache = \frac{Number\_of\_block}{A}$$

.

(A2) The replacement policies :

(A21) $Random$: this policy randomly replace a selected block among all blocks currently in cache.

(A22) $Least\_Recently\_Used(LRU)$: it replaces the block in cache that has not been used for the longest time.

(A23) $Least\_Recently\_Replaced(LRR)$: it replaces the block in cache that has not been replaced for the longest time.

(A24) $First\_in, First\_out(FiFo)$: it evicts the block that has been in the cache the longest.

(A3) The types of coherence protocols.

(A31) $Shared\_Cache\_Protocol$: cache is shared between cores.

(A32) $Private\_Cache\_Protocol$: each core has its private cache.

(A33) $Private\_Invalid\_Cache\_Protocol$: when a core writes into a memory block in the cache,all copies of this memory block in other cores' cache are invalidated.

(A34) *Private_MSI_Cache_Protocol*: M,S,I stand for Modified, Shared and Invalid.

They are three possibles states that a block inside the cache can have. This protocol is used in the 4D machine [7].

(A35) *Private_MESI_Cache_Protocol*: MESI stand for Modified, Exclusive, Shared and Invalid. This cache coherence protocol is derived from MSI protocol. More information about the MESI protocol can be found in [8]

(A4) The *Cache_Category*:

(A41) *Data_Cache*: cache which is used in order to speed up data fetch and store.

(A42) *Instruction_Cache*: cache which is used in order to speed up executable instruction fetch. *Instruction_Cache* is in fact a cache where coherence protocol is *Private_Cache_Protocol* or *Shared_Cache_Protocol*.

(A43) *Data_Instruction_Cache*: both data and instructions are stored in cache.

(A5) Type of write policies.

(A51) *Write−Back* (called also *Copy−Back*, *Write_Behind*): the information is written only to the block in the *Cache* .

The modified block is written in the memory only when the cache is replaced.

(A52) *Write_Through*: the information is written both in the block to cache and the block in the main-memory.

(A521) *Write_Through_With_Allocation*: memory block at the missed-write location is loaded to cache then followed by a write operation in the *Cache* .

(A522) *Write_Through_Without_Allocation*: memory block at the missed-write location is not loaded to the cache and written directly in the higher level memory.

```
<!ELEMENT caches (generic_cache | data_cache |
        instruction_cache | data_instruction_cache |
        cache_system)+>

<!ELEMENT generic_cache (object_type | name |
        cache_size | line_size | associativity |
        replacement_policy | block_reload_time |
        coherence_protocol | cache_category)*>
<!ATTLIST generic_cache id ID #REQUIRED>

<!ELEMENT data_cache (object_type | name | cache_size |
        line_size | associativity | cache_replacement |
        block_reload_time | coherence_protocol |
        cache_category | write_policy)*>
<!ATTLIST data_cache id ID #REQUIRED>

<!ELEMENT instruction_cache (object_type | name |
        cache_size | line_size | associativity |
        replacement_policy | block_reload_time |
        coherence_protocol | cache_category)*>
<!ATTLIST instruction_cache id ID #REQUIRED>

<!ELEMENT data_instruction_cache (object_type | name |
        cache_size | line_size | associativity |
        replacement_policy | block_reload_time |
        coherence_protocol | cache_category |
        write_policy)*>
<!ATTLIST data_instruction_cache id ID #REQUIRED>

<!ELEMENT cache_system (object_type | name | caches)*>
<!ATTLIST cache_system id ID #REQUIRED>
```

Figure 7: The DTD of entity *Cache*

Implementation

The figure 7 gives the DTD of entity *Cache* .

```
<instruction_cache id="␣1" >
        <object_type>CACHE_OBJECT_TYPE</object_type>
        <name>Cache_01</name>
        <cache_size>2048</cache_size>
        <line_size>8</line_size>
        <associativity>2</associativity>
        <replacement_policy>LRU</replacement_policy>
        <block_reload_time>10.00000</block_reload_time>
        <coherence_protocol>
                PRIVATE_CACHE_PROTOCOL
        </coherence_protocol>
        <cache_category>
                INSTRUCTION_CACHE_TYPE
        </cache_category>
</instruction_cache>
```

Figure 8: An Example of entity *Cache*

Example

The figure 8 gives an example of entity *Cache* described using Cheddar ADL. It describes an 2-ways set associative ($Associativity = 2$) instruction cache with cache size 2048 bytes (2 KB) and block size 8 bytes. The replacement policy is Least Recently Used.

4.1.4 Memory

A *Memory* is specified by the following definitions [23]:

(1) It is a deployment unit for a software component.

(2) It models any entity a process can address: code, data, stack, heap.

(3) In uniprocessor designs, the memory system can be a rather simple component, consisting of a few levels of cache to feed the single processor

with data and instructions.

(4) In multi-cores, the caches are can be part of the memory system. The other components include the consistency model, cache coherence support, and the intra-chip interconnect.

Standard attributes

$Name$: It is the unique name of $Memory$.

$size$:
$access\_latency$:
$memory\_category$:
$shared\_access\_latency$:
$private\_access\_latency$:
l_rw_inter :
l_act_inter :
l_pre_inter :
$n\_reorder$:
l_conhit :
l_conf :
nb_bank :
$partition\_mode$:

Legality rules

(L1) The $Memory$ name must not be empty.

(L2) The $Memory$ name must be valid identifier.

Implementation

Example

4.1.5 Network

A *Network* is specified by the following definitions:

(1) It is any communication link between any hardware components.

(2) It is used to simulate message scheduling.

Standard attributes

Name: It is the unique name of entity *Network* .

Network_type: It is the technique of taking into account the communication.

Legality rules

(L1) The network name must not be empty.

(L2) The network name must be valid identifier.

(L3) The *Network_type* is mandatory.

Annexes

(A1) The *Network_type* techniques [21] [34] [35] [22] is in fact the way to characterize delay when we consider the network.
We distinguish:

(A11) *Bounded_Delay*: In this case, the delay is bounded.
It is an effective search prioritization strategy for concurrent programs that handles both statically-known and dynamically-created tasks.
The sending of message is characterized by a bounded time.

(A12) *Jitter_Delay*: In this case, the delay is a function of jitter.

```
<!ELEMENT networks (network)+>
<!ELEMENT network (object_type | name
  | network_type)*>
<!ATTLIST network id ID #REQUIRED>
```

Figure 9: The DTD of entity *Network*

```
<networks>
 <network id="id_89">
  <object_type>NETWORK_OBJECT_TYPE</object_type>
  <name>a_network</name>
  <network_type>BOUNDED_DELAY</network_type>
 </network>
</networks>
```

Figure 10: An example of *Network* description

The sending of message is characterized by an bounded interval (max and min).

(A13) *Parametric_Delay*: In this case, the delay is parametric. The user may define its own delay.

Implementation

The figure 9 gives the DTD of entity *Network* .

Example

The figure 10 gives an example of entity *Network* described using Cheddar ADL.

The type of *Network* in this case is $BOUNDED\_DELAY$.

4.1.6 Battery

A *Battery* is specified by the following definitions :

(1) It is an hardware component storing energy.

(2) A battery is connected to a *Processor* and each task from this processor may or may not specify an energy comsumption rate.

Standard attributes

Name : it is the unique name of the *Battery* .

Cpu_name: It is the name of *Processor* which is connected to the *Battery* .

Capacity : It stores the maximum amount of energy that can be stored at any.

Rechargeable_Power : It models that amount of energy the battery is able to store/recover at each unit of time.

E_Max : It is the units of energy consumed during one unit time-slot when a task is scheduled.

Initial_Energy : It stores the current amount of energy stored by a battery at start time (i.e. scheduling start time).

Legality rules

(L1) The *Battery* name must not be empty.

(L2) The *Battery* name must be valid identifier.

(L3) An *Battery* must be connected to a *Processor* .

(L4) Several *Battery* may be connected to a *Processor* .

(L5) The *Capacity* must be greater than 0.

```
<!ELEMENT batteries (battery)+>
<!ELEMENT battery (object_type | name
  | cpu_name | capacity | rechargeablz_power
  | initial_energy | e_max)*>
<!ATTLIST battery id ID #REQUIRED>
```

Figure 11: The DTD of entity *Battery*

(L6) The *Rechargeable_Power* must be greater than 0.

(L7) The E_Max must be greater than 0.

(L8) The *Initial_Energy* must be greater than 0.

Annexes

Implementation

The figure **??** gives the DTD of entity *Battery* .

Example

The figure 14 gives an example of *Battery* .

This *Battery* , named $B1$ is based on *Processor processor*1. The others parameters must be greater than 0.

```
<battery id="100">
  <object_type>BATTERY_OBJECT_TYPE</object_type>
  <name>B1</name>
  <capacity>4</capacity>
  <rechargeable_power>2</rechargeable_power>
  <cpu_name>processor1</cpu_name>
  <e_max>1</e_max>
  <initial_energy>4</initial_energy>
 </battery>
```

Figure 12: An Example of entity *Battery* described using Cheddar ADL

4.2 Software Components

(1) We define in this section the following software components: *Address space* , Task, Resource, Buffer, Message and Task group.

(2) *Address space* models a logical unit of memory.

(3) $Task$ models a flow of control.

(4) *Resource* models asynchronous communication between tasks of the same address space.

(5) $Buffer$ models queued data exchanges between tasks of the same address space.

(6) $Message$ models queued data exchanges between tasks located in different address spaces.

(7) *Task group* models a subset of tasks organized in transactions.

(8) *Dependency* which models relationships between tasks and other software entities.

4.2.1 Address space

An *Address space* is specified by the following definitions [27] [28]:

(1) It is the range of virtual addresses that the operating system assigns to a user or separately running program.

(2) The range of addresses which a *Processor* or process can access, or at which a device can be accessed.

(3) It refers to either physical address or virtual address.

(4) An *Address space* may be associated to an address protection mechanism.

(5) An *Address space* defines a range of discrete addresses, each of which may correspond to a network host, peripheral device, disk sector, a memory cell or other logical or physical entity.

(6) It allows to model a logical unit of memory.

Standard attributes

Name : it is the unique name of the *Address space* .

Cpu_name: It is the name of *Processor* which contain *Address space* .

Text_Memory_Size: It is the size of text segment. A text segment contains the executable image of the program.
It is used to perform a global memory analysis.

Stack_Memory_Size: It is the size of stack segment. A stack segment contains the function-call stack.
This segment is extended automatically as needed.

Data_Memory_Size: It is the size of data segment. A data segment contains the *heap* of dynamically allocated data space.

Heap_Memory_Size: It is the size of logical memory reserved for the heap.

Scheduling: It defines all parameters of scheduling.
It is the type of *Scheduling_Parameters* (see Annexes for definitions of *Scheduling_Parameters*).

mils_confidentiality_level : It defines the level of confidentiality of an address space. *mils_confidentiality_level* can be UnClassified, or *Classified*, or *Secret*, or *Top_Secret*

mils_integrity_level : It defines the level of integrity of an address space. *mils_integrity*$_l$*evel* can be *Low*, or *Medium*, or *High*

mils_component : It defines the type of an address space according to the classification of components in MILS architecture. It can have the SLS value for Single Level Secure component, or MLS for Multi-Level Secure component, or $MSLS$ for Multi Single-Level Secure component.

mils_partition : It defines the kind of MILS component modeled by the address space. It can have either the *Device* value or the *Application* value. *mils_compliant* : It is a boolean that specifies if an address space models a component of MILS or not.

Legality rules

(L1) The *Address space* name must not be empty.

(L2) The *Address space* name must be valid identifier.

(L3) An *Address space* must be linked to a *Processor* .

(L4) The $Text\_Memory\_Size$ must be greater than or equal to 0.

(L5) The $Stack\_Memory\_Size$ must be greater than or equal to 0.

(L6) The $Data\_Memory\_Size$ must be greater than or equal to 0.

(L7) The $Heap\_Memory\_Size$ must be greater than or equal to 0.

Annexes

(A1) See Annexes of $Dynamic\_Deployment$ for attributes of $Scheduling\_Parameters$.

Implementation

The figure 13 gives the DTD of entity *Address space* .

Example

```
<!ELEMENT address_spaces (address_space)+>
<!ELEMENT address_space (object_type | name
  | cpu_name | text_memory_size | stack_memory_size
  | data_memory_size | heap_memory_size
  | scheduling | mils_confidentiality_level | mils_integrity_level
  | mils_component | mils_partition | mils_compliant)*>
<!ATTLIST address_space id ID #REQUIRED>
```

Figure 13: The DTD of entity *Address_Space*

The figure 14 gives an example of *Address space* .

This *Address space* , named $addr1$ is based on *Processor* $processor1$. The others parameters are fixed on 0, and the scheduling parameters have a *quantum* equal to 0, and is the type $PREEMPTIVE$.

```
<address_space id="␣18" >
 <object_type>ADDRESS_SPACE_OBJECT_TYPE</object_type>
 <name>addr1</name>
 <cpu_name>processor1</cpu_name>
 <text_memory_size>0</text_memory_size>
 <stack_memory_size>0</stack_memory_size>
 <data_memory_size>0</data_memory_size>
 <heap_memory_size>0</heap_memory_size>
 <scheduling><scheduling_parameters>
  <scheduler_type>NO_SCHEDULING_PROTOCOL</scheduler_type>
  <quantum>0</quantum>
  <preemptive_type>PREEMPTIVE</preemptive_type>
  <capacity>0</capacity>
  <period>0</period>
  <priority>0</priority>
  <start_time>0</start_time>
  </scheduling_parameters>
 </scheduling>
 <mils_confidentiality_level>TOP_SECRET</mils_confidentiality_level>
 <mils_integrity_level>HIGH</mils_integrity_level>
 <mils_component>SLS</mils_component>
 <mils_partition>DEVICE</mils_partition>
 <mils_compliant>TRUE</mils_compliant>
</address_space>
```

Figure 14: An Example of entity *Address space* described using Cheddar ADL

4.2.2 Task

A task, named *Generic_Task* is specified by the following definitions:

(1) Run any type of program (including any operating system function such as a scheduler).

(2) Statically defined in an *Address space* .

Standard attributes

Name : It is the unique name of the *Task* .

Task_Type: It defines the type of the task.
Annexes (A1) give the different types of task.

Cpu_Name: It is a string, which defined the *Processor* where is running the *Task* .

Address_Space_Name: It is a string, which defines the name of the *Address space* hosting the task.

Core_Name: It is a string, which defined the *Core* where is running the *Task* . This attribute is restricted to multicore architectures where tasks to core affinities have to be enforced.

Capacity: It is a natural, and it corresponds to the worst case execution time of the task.

Deadline: The *Task* must end its activation before its deadline. A deadline is a relative information : to get the absolute date at which a task must end an activation, you should add to the deadline the time when the task was awoken/activated. The deadline must be equal to the period if you define a Rate Monotonic scheduler.

Start_Time: It is a natural, which defines the first release time of a *Task* .

Completion_Time: It defines the time at which a cyclic task is completed, i.e. from when the task must not be dispatched. This attribute has 0 as default value. If set with 0, the dispatching rules of the task are not changed.
Priority: It is a priority range.
It allows the scheduler to choose the *Task* to run.

Blocking_Time: It's the worst case shared resource waiting time of the task. This duration could be set by the user or computed by Cheddar shared resources accesses are described.

Policy: It defines the scheduling policy of a task. Policy can be $SCHED\_RR$, or $SCHED\_FIFO$ or $SCHED\_OTHERS$ and describes how the scheduler chooses a task when several tasks have the same priority level.

$Offsets$: An offset stores two information : an activation number and a value. It allows to change the wake up time of a task on a given activation number. For an activation number, the task wake up time will be delayed by the amount of time given by the value field.

$Text\_Memory\_Size$: Size of the text segment of the task in order to perform memory requirement analysis.

$Stack\_Memory\_Size$: Size of the memory stack of the task in order to perform memory requirement analysis.

Parameters: A parameter is similar to the deadline, the period, to capacity ..., but used by user-defined schedulers.

A user can define new task parameters. A user-defined task scheduled has a value, a name and a type. The types currently available to define user-defined task parameters are : string, integer boolean and double.

Criticality: The field indicates how the task is critical. Currently used by the *MUF* scheduler or any user-defined schedulers.

Context_Switch_Overhead: It is an integer modeling the corst of the context swith for the task

$maximum\_number\_of\_memory\_request\_per\_job$:
$access\_memory\_number$:
cfg_name :
$cfg\_relocatable$:
$cache\_access\_profile\_name$:
$mils\_confidentiality\_level$: It defines the level of confidentiality of a task. $mils\_confidentiality\_level$ can be $UnClassified$, or $Classified$, or $Secret$, or Top_Secret

$mils\_integrity\_level$: It defines the level of integrity of a task. $mils\_integrity\_level$ can be Low, or $Medium$, or $High$

mils_component : It defines the type of a task according to the classification of components in mils architecture. It can be SLS for Single Level Secure component, or MLS for Multi-Level Secure component, or MSLS for Multi Single-Level Secure component.

mils_task : It defines a task as a common application or a particular element of mils architecture. It can be application, or MMR for MILS Message Router, or Guard, or Collator, or Downgrader, or Upgrader

mils_compliant : It is a boolean that specifies if a task models a component of MILS or not.

Energy_Consumption : It is an integer modelling the energy consumption of the task.

Legality rules

(L1) The *Task* name must not be empty.

(L2) The *Task* name must be valid identifier.

(L3) In the case of *Parametric_Type*, The *Activation_Rule* should be a valid identifier.

(L4) The *Cpu_Name* must not be empty.

(L5) The *Address_Space_Name* must not be empty.

(L6) In the case of *Periodic_Type*, the *Period* must be greater than 0.

(L7) In the case of *Periodic_Type*, the *Jitter* must be greater than or equal to 0.

(L8) In the case of *Aperiodic_Type*, the *Period* shouldn't exist.

(L9) In the case of $Sporadic\_Type$, the $Period$ shouldn't exist.

(L10) In the case of $Parametric\_Type$, the $Activation\_Rule$ must not be empty.

(L11) In the case of $Frame\_Task\_Type$, the $Period$ must be greater than or equal to 0

(L12) The $Capacity$ must be greater than 0.

(L13) The $Context\_Switch\_Overhead$ must be greater than or equal to 0.

(L14) The $Criticality$ must be greater than or equal to 0.

(L15) The $Deadline$ must be greater than or equal to 0.

(L16) The $Deadline$ must be less than the $Jitter$.

(L17) The $Start\_Time$ must be greater than or equal to 0.

(L18) The $Blocking\_Time$ must be greater than or equal to 0.

(L19) The $Text\_Memory\_Size$ must be greater than or equal to 0.

(L20) The $Stack\_Memory\_Size$ must be greater than or equal to 0.

(L21) The $Priority$ must be between $Priority\_Range\_First$ and $Priority\_Range\_Last$.

(L22) We can't have simultaneously $Priority \neq 0$ $Policy = Sched\_Others$.

(L23) We can't have simultaneously $Priority = 0$ $Policy \neq Sched\_Others$.

Annexes

(A1) The types of $Generic\_Task$.

(A11) *Periodic_Type*: In this case, the $Task$ is periodic, and we have two more attributes in order to characterize the $Task$:

(A111) *Period*: It is the time between two task activations. The period is a constant delay for a periodic task. It's an average delay for a *poisson* process task. If you have selected a *Processor* that owns a Rate Monotonic or a Deadline Monotonic scheduler, you have to give a period for each of its tasks.

(A112) *Jitter*: The *Jitter* of $Task$ is an upper bound on the delay that $Task$ may suffer between the time it is supposed to be released and the time that it is actually released.

The jitter is a maximum lateness on the task wake up time. This information can be used to express task precedencies and to applied method such as the Holistic task response time method.

(A12) *Aperiodic_Type* In this case, the $Task$ is called aperiodic task.

An aperiodic task is only activated once.

(A13) *Sporadic_Type*: A sporadic task is a task which is activated many times with a minimal delay between two successive activations.

If the $Task$ type is $user\_defined$, the task activation delay is defined by the user.

(A14) *Poisson_Type*. In this case, the task is called *poisson* task.

It is a subtype of *Periodic* task, with two more attributes.

(A141) *Seed*: If you define a *poisson* process task or a user-defined task, you can set here how random activation delay should be generated (in a deterministic way or not). The *Seed* button proposes you a randomly generated seed value but of course, you can give any seed value. This seed value is used only if the Predictable button is pushed. If the Unpredictable button is pushed instead, the seed is initialized at simulation time with *gettimeofday*.

(A142) *Predictable*: It is a boolean, which guides the *seed* value (see the *Seed* definition).

(A15) *Parametric_Type*. In this case, the task is called parametric task, and it is characterized by one more attribute.

(A151) *Activation_Rule*: The name of the rule which defines the way the task should be activated. Only used with user-defined task.

(A16) *Scheduling_Task_Type*: It is one of the types of *Task* .

(A17) *Frame_Task_Type*. In this case, the task type is called frame task, and it is characterized by one more attribute.

(A171) *Interarrival*: It defines the duration between the release of two tasks. It is specified through the *Period* attribute.

(A18) *Periodic_Inner_Periodic_Type*. In this case, the task type is Periodic inner periodic task model. Such model of task allows the modelling of burst of activation. A burst is composed of several periodic releases. For those inner periodic release, the *Period* attribute is used. The tasks is inactive between two bursts. Timing behavior is expressed will the following attributes.

(A181) $Outer-Period$: It is an integer which specifies the number of periodic activitation in a burst.

(A182) *Outer_Duration* : It is an integer which specifies the delay between two bursts.

(A19) $Sporadic\_Inner-periodic\_Type$. In this case, the task is type is Sporadic inner periodic. uch model of task allows the modelling of burst of activation. A burst is composed of several periodic releases. For those inner periodic release, the *Period* attribute is used. The tasks is inactive between two bursts. Delays between burst are expressed with *Outer_Duration* and $Outer-Period$ excepts that $Outer-Period$ is not a fixed amount of time but a random delay computed according to a poisson process with $Outer-Period$ as average delay.

(A2) The types of *Policies* [36].

(A21) $Sched\_Fifo$: With this policy, ready processes in a given priority level get the $Processor$ according to their order in the FIFO queue.

The process at the head of the queue runs first and keeps the processor until it executes some statement that blocks it, explicitly releases the processor, or finishes.

(A22) $Sched\_Rr$: It can be seen as a $Sched\_Fifo$ policy but with a time quantum and some extra rules on the queue management.

When the quantum is exhausted, the preempted thread is moved to the tail of the queue.

(A23) $Sched\_Others$: The behaviour of this policy is not defined in the POSIX standard. It is implementation defined.

Sometimes, this policy provides a time sharing scheduler.

This policy is used by Linux for all processes with a priority level of 0. These processes are put in a $Sched\_Others$ queue.

With Linux, the process in the $Sched\_Others$ queue that has waited longest for the processor is dispatched first.

Implementation

The figure 15 gives the DTD of entity $Task$.

Example

We give at figure 16 an example of $Task$ described in Cheddar ADL.

This task is *periodic* (*task_type* is $PERIODIC\_TYPE$), and its *period* (4) is specified by attribute *period*. We can remark another informations, like by example the task runs on *processor*1, and *addr*1 is $Address\_space$ dedicated for its execution.

4.2.3 Resource

A *Resource* is specified by the following definitions:

(1) It models any synchronized data structure shared by tasks.

```
<!ELEMENT tasks (generic_task | periodic_task | aperiodic_task
  | poisson_task | sporadic_task | parametric_task
  | scheduling_task | frame_task)+>
<!ELEMENT generic_task (object_type | name | task_type
  | cpu_name | address_space_name | capacity | deadline
  | start_time | priority | blocking_time | policy
  | offsets | text_memory_size | stack_memory_size
  | parameters | criticality | context_switch_overhead
  | mils_confidentiality_level | mils_integrity_level
  | mils_component | mils_task | mils_compliant)*>
<!ATTLIST generic_task id ID #REQUIRED>
<!ELEMENT periodic_task (object_type | name | task_type
  | cpu_name | address_space_name | capacity | deadline
  | start_time | priority | blocking_time | policy | offsets
  | text_memory_size | stack_memory_size | parameters
  | criticality | context_switch_overhead | mils_confidentiality_leve
  | mils_integrity_level | mils_component | mils_task | mils_complian
  | period | jitter)*>
<!ATTLIST periodic_task id ID #REQUIRED>
<!ELEMENT aperiodic_task (object_type | name | task_type
  | cpu_name | address_space_name | capacity | deadline
  | start_time | priority | blocking_time | policy | offsets
  | text_memory_size | stack_memory_size | parameters
  | criticality | context_switch_overhead
  | mils_confidentiality_level | mils_integrity_level
  | mils_component | mils_task | mils_compliant)*>
<!ATTLIST aperiodic_task id ID #REQUIRED>
<!ELEMENT poisson_task (object_type | name | task_type
  | cpu_name | address_space_name | capacity | deadline
  | start_time | priority | blocking_time | policy | offsets
  | text_memory_size | stack_memory_size | parameters
  | criticality | context_switch_overhead | mils_confidentiality_leve
  | mils_integrity_level | mils_component | mils_task | mils_complian
  | period | jitter | seed | predictable )*>
<!ATTLIST poisson_task id ID #REQUIRED>
<!ELEMENT sporadic_task (object_type | name | task_type
  | cpu_name | address_space_name | capacity | deadline
  | start_time | priority | blocking_time | policy | offsets
  | text_memory_size | stack_memory_size | parameters
  | criticality | context_switch_overhead |
mils_confidentiality_level
  | mils_integrity_level | mils_component | mils_task | mils_complian
  | period | jitter | seed | predictable)*>
<!ATTLIST sporadic_task id ID #REQUIRED>
<!ELEMENT parametric_task (object_type | name | task_type
  | cpu_name | address_space_name | capacity | deadline
  | start_time | priority | blocking_time | policy | offsets
  | text_memory_size | stack_memory_size | parameters
  | criticality | context_switch_overhead |
```

```
<periodic_task id="␣71">
        <object_type>TASK_OBJECT_TYPE</object_type>
        <name>T1</name>
        <task_type>PERIODIC_TYPE</task_type>
        <cpu_name>processor1 </cpu_name>
        <address_space_name>addr1</address_space_name>
        <capacity>2</capacity>
        <deadline>4</deadline>
        <start_time>0</start_time>
        <priority>1</priority>
        <blocking_time>10</blocking_time>
        <policy>SCHED_FIFO</policy>
        <text_memory_size>0</text_memory_size>
        <stack_memory_size>0</stack_memory_size>
        <criticality >0</criticality >
        <context_switch_overhead>0</context_switch_overhead>
        <context_switch_overhead>0</context_switch_overhead>
        <mils_confidentiality_level>Top_Secret</mils_confidentiality_le
        <mils_integrity_level>HIGH</mils_integrity_level>
        <mils_component>SLS</mils_component>
        <mils_task>APPLICATION</mils_task>
        <mils_compliant>TRUE</mils_compliant>
        <period>4</period>
        <jitter >0</jitter >
</periodic_task>
```

Figure 16: An example of *Task* description in Cheddar ADL

(2) It is statically defined in an *Address space* .

(3) It models asynchronous communication between tasks of the same *Address space* .

Standards attributes

State: It is the initial value of the resource component (similar to a semaphore initial value). During a scheduling simulation, at a given time, if a resource value is equal or less than zero, the requesting tasks are blocked until the semaphore/shared resource is released.

An initial value equal to 1 allows you to design a shared resource that is initially free and that can be used by only one task at a given time.

Size: It defines the size of the *Resource* .

Address: It is the location of the *Resource* .

Protocol: It characterises how the *Resource* is locked and unlocked.

Currently, you can choose between [42] [43] PCP (for Priority Ceiling Protocol), PIP (for Priority Inheritance Protocol), IPCP (Immediate Priority Ceiling Protocol) or *No protocol*.

With PCP, IPCP or PIP, accessing shared resources may change task priorities. With *No protocol*, the tasks are inserted in a FIFO order in the semaphore queue and no priority inheritance will be applied at simulation time.

$Critical\_Sections$: It specifies when each $Task$ must lock or unlock resources. This attribute specifies critical sections defined for each $Task$ and *Resource* .

A critical section is defined by two relative instants : *Begin* and *End*. *Begin* is the first unit of time of the task capacity when the task needs the resource, i.e. before running *Begin*th unit of time of its capacity, the task locks the *Resource* component. *End* is the last unit of time of the task capacity when the task needs the resource, i.e. when the task has ran the *End*th unit of time of its capacity, the task unlocks the *Resource* component.

Cpu_Name: Each shared resource has to be located on a given *Processor* .

Address_Space_Name: Its stores the name of *Address space* which hosted the *Resource* .

Priority: Is of type of *Priority_Range* and it defines the ceiling priority of the *Resource* .
This attribute is currently only used with the PCP and ICPP protocols.

Priority_Assignment: It is an enumerated type, and characterize the way that Cheddar assigns ceiling priority to the *Resource* .

Legality rules

(L1) The resource name is mandatory.

(L2) The resource name must be a valid identifier.

(L3) The *Cpu_Name* is mandatory.

(L4) The *Address_Space_Name* is mandatory.

(L5) The types of *Protocol* are specified in *Annexes*.

(L6) The *Size* must be greater than or equal to 0.

(L7) The *Address* must be greater than or equal to 0.

(L8) The *State* must be greater than or equal to 0.

Annexes

(A1) Each critical section is defined by :

(A11) *Task_begin*: The time at which the critical section is started. Task is the *Task* name accessing the shared *Resource* .

(A12) *task_end*: The time at which the critical section is completed. Task is the *Task* name accessing the shared *Resource* .

Each of this date are relative to the task capacity. Finally, several critical sections can be defined for a given task on a given resource.

(A2) The types of *Protocol* [36].

(A21) *No_Protocol*: The *Resource* is accessed by a FIFO order and no priority inheritance is applied.

(A22) *Priority_Inheritance_Protocol*: A *Task* which blocks a high priority task due to a critical section, sees its priority to be increased to the priority level of the blocked task. *Priority_Inheritance_Protocol* should not be used with more than one shared resource due to deadlock.

(A23) A ceiling priority of a resource is the maximum *Priority* of all the task component which use the resource.

(A24) *Priority_Ceiling_Protocol*:
In the case of *Priority_Ceiling_Protocol*, a ceiling priority is assigned to each resource. The ceiling priority can be automatically computed by Cheddar or provided by the user thanks to the *Priority* attribute of *Resource* components. A *Task* which blocks a high priority task due to a critical section, sees its priority to be increased upto the ceiling priority level of the resource. At request time, a task is blocked when its priority level is not strickly higher than every previously allocated resource ceiling priority, except the resource the requesting has allocated.

(A25) *Immediate_Priority_Ceiling_Protocol*: with IPCP, a task has a static and a dynamic priority level. The task is run according to its dynamic priority level. Its static priority level is the one stored in the *Task Priority* attributes. When a task allocates a IPCP *Resource* , it sees it dynamic priority level increased upto the ceiling priority level of the *Resource* component. Similarly, the *Task* priority is decreased when the *Resource* is unlocked. Again, the ceilin priority can be automatically computed by Cheddar or provided by the user thanks to the *Priority* attribute of *Resource* components.

(A251) Dynamic task priority = maximum of its own static priority and the ceiling priorities of any resources it has locked.

(A3) The types of *Priority_Assignment*:

(A26) *Automatic_Assignment*: In this case, Cheddar assigns automatically ceiling priority to the *Resource* . The attribute *Priority* is then ignored during simulation.

(A27) *Manual_Assignment*: This case corresponds to the manually affectation of *Priority* to the *Resource* . The attribute *Priority* is then used during the simulation.

Implementation

The figure 17 gives the DTD of entity *Resource* .

Example

An example of resource described in Cheddar ADL is given in figure 18.
Since *protocol* has value $NO\_PROTOCOL$, it means that no priority inheritance is applied to the task when it holds the resource $R1$. We can also notice that this resource concerns $processor1$, and runs at $addr1$, and has two critical sections: each task holds and releases the resource at time unit 1.

4.2.4 Buffer

A $Buffer$ is specified by the following definitions:

(1) It is statically defined in an *Address space* .

(2) A $Buffer$ has an unique name, size and is hosted by a *Processor* and an *Address space* .

(3) It allows to model queued data exchanges between $Tasks$ on the same *Address space* .

Standard attributes

```
<!ELEMENT resources (generic_resource | np_resource |
pip_resource |
pcp_resource | ipcp_resource | critical_section)+>
<!ELEMENT generic_resource (object_type | name |
state | size | address | protocol |
critical_sections | cpu_name | address_space_name)*>
<!ATTLIST generic_resource id ID #REQUIRED>
<!ELEMENT np_resource (object_type | name | state |
priority | size | address | protocol | critical_sections |
cpu_name | address_space_name)*>
<!ATTLIST np_resource id ID #REQUIRED>
<!ELEMENT pip_resource (object_type | name | state |
priority | size | address | protocol | critical_sections |
cpu_name | address_space_name)*>
<!ATTLIST pip_resource id ID #REQUIRED>
<!ELEMENT pcp_resource (object_type | name | state |
priority | size | address | protocol | critical_sections |
cpu_name | address_space_name | ceiling_priority)*>
<!ATTLIST pcp_resource id ID #REQUIRED>
<!ELEMENT ipcp_resource (object_type | name | state |
priority | size | address | protocol | critical_sections |
cpu_name | address_space_name | ceiling_priority)*>
<!ATTLIST ipcp_resource id ID #REQUIRED>
<!ELEMENT critical_section (task_begin | task_end)*>
```

Figure 17: The DTD of entity *Resource*

```
<np_resource id="␣44" >
<object_type>RESOURCE_OBJECT_TYPE</object_type>
        <name>R1</name>
        <state>1</state>
        <size>0</size>
        <address>0</address>
        <protocol>NO_PROTOCOL</protocol>
        <critical_sections>
                <task_name> T1 </task_name>
                <critical_section>
                        <task_begin>1</task_begin>
                        <task_end>1</task_end>
                </critical_section>
                <task_name> T2 </task_name>
                <critical_section>
                        <task_begin>1</task_begin>
                        <task_end>1</task_end>
                </critical_section>
        </critical_sections>
        <cpu_name>processor1</cpu_name>
        <address_space_name>addr1</address_space_name>
</np_resource>
```

Figure 18: An example of *Resource* description in Cheddar ADL

$Name$: It is the unique name of a $Buffer$.

Cpu_Name: It corresponds to the name of $Processor$ hosted by a $Buffer$.

$Address\_Space\_Name$: It corresponds to the name of *Address space* hosted by a $Buffer$.

$Queueing\_System\_Type$: It defines the types of $Queueing\_System$. A $Queueing\_System$ model is assigned to each $Buffer$. This model describes the way buffer reads and writes operations will be done at simulation time. This information is also used to apply $Buffer$ feasibility tests.

$Buffer\_Size$: Size of a $Buffer$.

$Buffer\_Initial\_Data\_Size$: Initial data in a $Buffer$.

$Roles$: It is the type of $Buffer\_Roles\_Table$.

$Buffer\_Roles\_Table$ is a list of $Buffer\_Role$.

Legality rules

(L1) $Buffer$ name must not be empty.

(L2) $Buffer$ name must be a valid identifier.

(L3) Cpu_Name must not be empty.

(L4) $Address\_Space\_Name$ must not be empty.

(L5) $Buffer\_Size$ must be greater than 0.

(L6) $Buffer\_Initial\_Data\_Size$ must be greater than 0 and less than $Buffer\_Size$.

(L7) Two types of $Tasks$ can access to a $Buffer$: producers and consumers.

(L8) We suppose that a producer/consumer writes/reads a fixed size of information to a $Buffer$.

(L9) For each producer or consumer, the size of the information produced or consumed has to be defined in the $Buffer\_Role$.

(L10) The time of the read/write operation is also given in the $Buffer\_Role$: this time is relative to the task capacity (e.g. if task T_i consumes a message at time 2, it means that the message will be removed from the $Buffer$ when T_i runs the 2^{nd} unit of time of its *capacity*).

Annexes

(A1) The types of $Queueing\_System$ [45].

(A11) Qs_Pp1: compliant with P/P/1 queueing model [44]. There are several producers but only one consumer $Task$. Producers and the consumers are independent periodic tasks. Producers and the consumer are not blocked when they access the $Buffer$.

(A12) Qs_Mm1 : compliant with the classical M/M/1 queueing system model. There are several producers but only one consumer task. Producers are released according to a Markovian law. The consumer is released on data arrival and its service time is exponential (markovian law too).

(A13) Qs_Md1 : compliant with the classical M/D/1 queueing system model. There are several producers but only one consumer $Task$. Producers are released according to a Markovian law. The consumer is released on data arrival and its service time is deterministic.

(A14) Qs_Mp1 : compliant with the classical M/P/1 queueing system model.

(A15) Qs_Mg1 : compliant with the classical M/G/1 queueing system model.

(A16) Qs_Mms : compliant with the classical M/M/S queueing system model.

(A17) Qs_Mds : compliant with the classical M/D/S queueing system model.

(A18) Qs_Mps : compliant with the classical M/P/S queueing system model.

(A19) Qs_Mgs : compliant with the classical M/G/S queueing system model.

(A110) Qs_Mm1n : compliant with the classical M/M/1/N queueing system model.

(A111) Qs_Md1n : compliant with the classical M/D/1/N queueing system model.

(A112) Qs_Mp1n : compliant with the classical M/P/1/N queueing system model.

(A113) Qs_Mg1n : compliant with the classical M/G/1/N queueing system model.

(A114) Qs_Mmsn : compliant with the classical M/M/S/N queueing system model.

(A115) Qs_Mdsn : compliant with the classical M/D/S/N queueing system model.

(A116) Qs_Mpsn : compliant with the classical M/P/S/N queueing system model.

(A117) Qs_Mgsn : compliant with the classical M/G/S/N queueing system model.

(A2) The $Buffer\_Roles\_Table$ is a list of $Buffer\_Role$. Each $Buffer\_Role$ is characterized by:

(A21) The_Role which is the type of $Buffer\_Role\_Type$.
It describes the behaviour of $Task$ on the $Buffer$.

(A211) No_Role: When any role is defined.

(A212) $Queuing\_Producer$: name of the producer $Task$.

(A213) $Queuing\_Consumer$: name of the consumer $Task$.

(A214) $Sampling\_Writer$: name of the producer $Task$.

(A215) $Sampling\_Reader$: name of the consumer $Task$.

(A216) $UCSDF\_Producer$: name of the producer $Task$ whom the production rate is described by an amplitude function (A25)

(A217) $UCSDF\_Consumer$: name of the consumer $Task$ whom the consumption rate is described by an amplitude function (A25)

(A22) $Size$: size of the data to read/write from/to the $Buffer$.

(A23) $Time$: time at which the data must be read or write on the $Buffer$. This time is relative to the $Task$ capacity

(A24) $Timeout$: specify the maximum blocking time allowed when a read/write is proceeded on a $Buffer$.

(A25) $amplitude\_function$: the number of token produced/consumed by a producer/consumer is an ultimately cyclo-static sequence $s = u(v)$ in which u denotes the initialization part and (v) denotes the cyclic part. The number of produced tokens is depending on the release number of a producer/consumer. For an i^{th} release, the number of token produced/-

```
<!ELEMENT buffers (buffer | buffer_role)+>
<!ELEMENT buffer (object_type | name | cpu_name
  | address_space_name | queueing_system_type | size
  | roles)*>
<!ATTLIST buffer id ID #REQUIRED>
<!ELEMENT buffer_role (the_role | size | time | timeout
   | amplitude_function)*>
<!ATTLIST address_space id ID #REQUIRED>
```

Figure 19: The DTD of entity $Buffer$

consumed is computed as follows:

$$nb\_token = \begin{cases} u[i] & \text{if } i \leq u \\ v[(j - |u|) \mod |v|] & \text{otherwise} \end{cases} \quad (1)$$

This property can only be applied to the producers/consumers with the roles UCSDF_Producer and UCSDF_Consumer

(A26) Buffer Underflow : Underflow event occurs when a task reads from a buffer and the read data size is greater than the current data size in the buffer. When it happens, a task does not read the buffer and current data in a buffer is not consumed.

(A25) Buffer Overflow : Overflow event occurs when a task writes to a buffer and the write data size plus the current data size in the buffer is greater than $Buffer\_Size$. When it happens, a task does not write any data to the buffer.

Implementation

The figure 19 gives the DTD of entity $Buffer$.

Example

The figure 20 gives an example of entity $Buffer$, described using Cheddar ADL.

The considered $Buffer$, named $B1$ is hosted by the $Processor$ $processor1$ and $Address$ $space$ $addr1$.

4.2.5 Message

A $Message$ is specified by the following definitions:

(1) It allows to model data exchanges between $Tasks$ located in different $Processor$.

(2) A $Message$ can be sent by one or several sending $Task$ and can be received by one or several receiving $Tasks$.

(3) $Messages$ can be queued on the receiver side before they are read by sending $Task$.

(4) $Messages$ are read/write according to a protocol specified by the dependency expressed between the tasks and the messages (see dependency entity for further readings.

Standard attributes

$Name$: It is the unique name of the $Message$.

$Message\_Type$: It specifies the type of $Message$.

$Parameters$: It is the type of $User\_Defined\_Parameters\_Table$, which is the list of $Parameter$.

$Deadline$: It is deadline of the $Message$. This corresponds to the last time that the $Message$ should be received or sent.

$Jitter$: The jitter of the $Message$.

$Size$: It is the size of the payload of the $Message$.

```
<buffers>
<buffer id="␣37" >
 <object_type>BUFFER_OBJECT_TYPE</object_type>
 <name>B1</name>
 <cpu_name>processor1 </cpu_name>
 <address_space_name>addr1</address_space_name>
 <queueing_system_type>QS_MM1</queueing_system_type>
 <size>1</size>
 <roles>
 <task_name> T1 </task_name>
 <buffer_role>
  <the_role>QUEUING_PRODUCER</the_role>
  <size>1</size>
  <time>1</time>
  <timeout>1</timeout>
 </buffer_role>
 <task_name> T2 </task_name>
 <buffer_role>
  <the_role>QUEUING_CONSUMER</the_role>
  <size>2</size>
  <time>2</time>
  <timeout>2</timeout>
  <amplitude_functio TO BE COMPLETED
 </buffer_role>
 </roles>
</buffer>
</buffers>
```

Figure 20: An example of entity $Buffer$

Response_Time : Amount of time a message takes to reach the receiver after it has been sent by the sender. A receiver is blocked until this delay is not exhausted during scheduling simulation. It also corresponds to the end to end time between the emission and reception of the $Message$.

Communication_Time: It is the duration of the $Message$ in the $Buffer$. It is not currently used du scheduling simulation

$mils\_confidentiality\_level$:
$mils\_integrity\_level$:

Legality rules

(L1) The $Message$ name must not be empty.

(L2) The $Message$ name must be a valid identifier.

(L3) The $Size$ must be greater than 0.

(L4) The $Deadline$ must be less than or equal to $Jitter$.

(L5) $Message$ is sent when the sender is running its last time unit of its capacity.

(L6) A $Message$ is received at the execution of the at completion time of a sender $Task$.

(L7) A $Message$ cannot be received before the specified number of units of time specified by the attribute $response\_time$ after the message sending time. $response\_time$ can be equal to 0: in that case, the message is immiediatly available for the receiving task.

(L8) A $Message$ is received when ($L7$) holds and when a receiver task is running its first unit of time of its $Capacity$.

(L9) To have message exchanges during the scheduling simulation, a asynchronous communication dependency must exist in the model. Two protocols can be specified during dependency expression:

- *first_message* protocol: in this case, a receiving task waiting for several message is woken up when one message is arrived. This protocol is the default protocol.
- *all_messages* protocol: in this case, a receiving task waiting for several messages is woken up when all messages are arrived.
- With any of these protocol, when a receiving task is run, it consumes only one message whataver the number of arrived messages.

(L10) Period of messages are not used during scheduling simulation.

Annexes

(A1) The types of *Message* .

(A11) *Periodic_Type*: A *periodic* message is automatically sent at each *Period*.

(A12) *Aperiodic_Type*: In this case, the *message* is sent according to *aperiodic* time.

(A13) *Generic_Type*: It is the case when we have no information about the sending time of *Message* .

(A2) Each *Parameter* is characterized by:

(A21) *Discriminant*. It is the type of *Parameter_Type*.

(A211) *Boolean_Parameter*: When the parameter is boolean.

(A212) *Integer_Parameter*: When the parameter is integer.

(A213) *Double_Parameter*: When the parameter is double.

(A214) *String_Parameter*: When the parameter is string.

(A22) *Union*, which is the type of *Parameter_Union*.

```
<!ELEMENT messages (generic_message |
  periodic_message | aperiodic_message)+>
<!ELEMENT generic_message (object_type | name |
  message_type | parameters | deadline | size |
  response_time | communication_time)*>
<!ATTLIST generic_message id ID #REQUIRED>
<!ELEMENT periodic_message (object_type | name |
  message_type | parameters | deadline | size |
  response_time | communication_time | period | jitter)*>
<!ATTLIST periodic_message id ID #REQUIRED>
<!ELEMENT aperiodic_message (object_type | name |
  message_type | parameters | deadline | size |
  response_time | communication_time)*>
<!ATTLIST aperiodic_message id ID #REQUIRED>
```

Figure 21: The DTD of entity *Message*

(A221) *Boolean_Parameter*: When the parameter is boolean.

(A222) *Integer_Parameter*: When the parameter is integer.

(A223) *Double_Parameter*: When the parameter is double.

(A224) *String_Parameter*: When the parameter is string.

(A23) *Name*: The name of the parameter.

Implementation

The figure 21 gives the DTD of entity *Message*.

Example

The figure 22 gives an example of entity *Message* , described using Cheddar ADL.

The example describes a periodic message, named $M1$, with *deadline*, *size*, *response_time*, *period* and *jitter* equal to 1.

```
<messages>
 <periodic_message id="␣58" >
 <object_type>MESSAGE_OBJECT_TYPE</object_type>
 <name>M1</name>
 <message_type>PERIODIC_TYPE</message_type>
 <deadline>1</deadline>
 <size>1</size>
 <response_time>1</response_time>
 <communication_time>1</communication_time>
 <period>1</period>
 <jitter>1</jitter>
 </periodic_message>
</messages>
```

Figure 22: An example of entity *Message*

4.2.6 Dependency

A *Dependency* is specified by the following definitions:

(1) It models an interaction between two software entities which has an impact upon the scheduling of the system. One of those entity is called the sink entity and the other is the source entity.

(2) Software entities handled in a *Dependency* component can either be $Tasks$, and or *Resources* and or $Buffers$ and or *Message* .

Standard attributes

$Type\_of\_dependency$: It specifies the kind of dependency component.

A dependency is a kind of union component : depending on its type, a dependency has different attributes described in the sequel.

Legality rules

(L1) The dependency name must have a type.

Annexes

(A1) The types of *Dependency* are :

(A11) $Precedence\_Dependency$: this dependency models a precedency relationship between two $Task$ components. With this dependency, the sink task must wait for the completion of the source task before being released by the scheduler.

(A12) $Queuing\_Buffer\_Dependency$: this dependency models a producer/consumer relationship between $Task$ components and a $Buffer$ component. When the source component is a task and the sink component is a buffer, the dependency models the producer side. When the sink component is a task and the source component is a buffer, the dependency models the consumer side.
This dependency assume a fixed buffer size. A data can be read once and we assume that no data can be lost.

(A13) $Asynchronous\_Communication\_Dependency$: this dependency models asynchronous exchanges of $Messages$ between $Task$ components. When the source component is a task and the sink component is a message, the dependency models the emitter side. When the sink component is a task and the source component is a message, the dependency models the receiver side.
Different protocols explain how and when messages are sent of received. The attribute $Asynchronous\_Communication\_Prococol\_Property$ models such a semantics and allows to express two protocols:

- $First - Message$ protocol : the receiver task is released when at least one message of any of its sender is available in the queue. Releasing the receiving task leads to read this message and to remove it from the queue.
- $All\_Messages$ protocol : the receiver task is released when at least one message of each its sender is available. Releasing the receiving task leads to read one message per sender and removing them from the queue.

(A14) $Time\_Triggered\_Communication\_Dependency$: this dependency models a synchronous communication between two $Task$ components. Data/messages are read on task release and sent on task completion. Three protocols exist for such dependency:

- $Sampled\_Timing$: with this protocol, emitters and receivers send/receive synchronous messages at their own rate. Messages are not queued : only the last sent message can be read by receivers. This protocol does not imply any constraint on task releases and data consistency. From a scheduling point of view, tasks are independent.

- $Immediate\_Timing$: with this protocol, when an emitter completes its execution, its message is send and the receiver task is immediately released. The actual time at which the receiving task is run depends on the scheduling, e.g this protocol enforces that the receiver task can not be released and run before completion time of the sending task. From a scheduling point of view, this dependency is equivalent to a precedence dependency.

- $Delayed\_Timing$: with this protocol, when an emitter completes its execution, the protocol enforces that the receiver will be released on the next activation of the emitter task. This delay is expressed by the offset of the sink task. From a scheduling point of view, those tasks are independent as we only change task release times by offsets.

The protocol is set by the attribute $Time\_Triggered\_Timing\_Property$.

(A15) $Resource\_Dependency$: this dependency models a shared memory access between several $Task$ components and a $Resource$ component.

(A16) $Black\_Board\_Buffer\_Dependency$: this dependency models a producer/consumer relationship between $Task$ components and a $Buffer$ com-ponent. When the source component is a task and the sink component is a buffer, the dependency models the producer side. When the sink component is a task and the source component is a buffer, the dependency models the consumer side. This dependency assume a buffer size of 1. A message can be read several times or never, e.g. consumers read the last written message produced by producers. From a scheduling point of view, tasks accessing a black board are independent.

```
<!ELEMENT dependencies (dependency)+>
<!ELEMENT dependency (type_of_dependency |
precedence_sink | precedence_source |
buffer_dependent_task | buffer_orientation |
buffer_dependency_object |
asynchronous_communication_dependent_task |
asynchronous_communication_orientation |
asynchronous_communication_dependency_object |
asynchronous_communication_protocol_property |
time_triggered_communication_sink |
time_triggered_communication_source |
time_triggered_timing_property |
resource_dependency_resource |
resource_dependency_task |
black_board_dependent_task |
black_board_orientation |
black_board_dependency_object)*>
```

Figure 23: The DTD of entity *Dependency*

Implementation

The figure 23 gives the DTD of entity *Dependency* .

Examples

The figure 24 gives examples of *Dependency* .

We illustrate with an example of $QUEUING\_BUFFER\_DEPENDENCY$, $RESOURCE\_DEPENDENCY$, $ASYNCHRONOUS\_COMMUNICATION\_DEPEND$ and $PRECEDENCE\_DEPENDENCY$

4.2.7 Task group

A *Task group* is specified by the following definitions:

(1) A *Task group* is a sub-set of *Tasks* .

```
<dependencies>
<dependency>
<type_of_dependency>QUEUING_BUFFER_DEPENDENCY
</type_of_dependency>
<buffer_dependent_task ref="_42" />
<buffer_orientation>FROM_TASK_TO_OBJECT
</buffer_orientation>
<buffer_dependency_object ref="_46" />
</dependency>

<dependency>
<type_of_dependency>RESOURCE_DEPENDENCY
</type_of_dependency>
<resource_dependency_resource ref="_44" />
<resource_dependency_task ref="_42" />
</dependency>

<dependency>
<type_of_dependency>ASYNCHRONOUS_COMMUNICATION_DEPENDENCY
</type_of_dependency>
<communication_dependent_task ref="_43" />
<communication_orientation>FROM_TASK_TO_OBJECT
</communication_orientation>
<asynchronous_communication_protocol_property>ALL_MESSAGES
</asynchronous_communication_protocol_property>
<communication_dependency_object ref="_45" />
</dependency>

<dependency>
<type_of_dependency>PRECEDENCE_DEPENDENCY
</type_of_dependency>
<precedence_sink ref="_42" />
<precedence_source ref="_43" />
</dependency>

</dependencies>
```

Figure 24: Examples of *Dependency*

(2) A *Generic_Task_Group* has an unique name.

(3) A *Generic_Task_Group* may constrain (the attributes of) its *Tasks* .

(4) The manner a *Generic_Task_Group* constrains its *Tasks* depends on the type of the *Task group* .

Standard attributes

Task_List: The set of tasks part of the *Task group* . A list is used because some *Task groups* may constrain its *Tasks* to be in a certain order.

Task_Group_Type: Type of *Task group* .
Currently there exists two types of *Task group* . Annexes (A1) give the different types of a *Task group* .

The other attributes of a *Task group* are the same as those for a *Task*. This way a *Task group* may constrain any attribute of its *Tasks* .

Legality rules

(L1) The *Task group* name must not be empty.

(L2) The *Task group* name must be valid identifier.

(L3) When *Task_Group_Type* is *Multiframe_Type*, the *Task_Type* must not be different of *Frame_Task_Type*.

(L4) When *Task_Group_Type* is *Transaction_Type*, the *Task_Type* must not be different of *Periodic_Type*.

(L5) When *Task_Group_Type*, of a *Task group* , is *Transaction_Type*, the period attribute of its tasks must be equal to the period attribute of the *Generic_Task_Group*

Annexes

(A1) The types of $Generic\_Task\_Group$.

(A11) $Transaction\_Type$: A $Generic\_Task\_Group$ of $Transaction\_Type$ is a transaction [46].

A transaction is group of tasks related by precedence dependency.

A transaction is released by a periodic event. A particular instance of a transaction is called a job. A job of a task in a transaction is released after the event that releases the job of the transaction.

If a job of a transaction is released at t_0, then a job of a $Task$ is released at earliest after $t_0 + O$ with O being the offset of the $Task$.

(A12) $Multiframe\_Type$: A $Generic\_Task\_Group$ of $Multiframe\_Type$ is a Multiframe task [49] or a General Multiframe task [48], i.e. group of tasks of $Frame_Task_Type$. Tasks of $Frame_Task_Type$ represent the instances of the Multiframe/Generam Multiframe task ($Generic\_Task\_Group$ of $Multiframe\_Type$) and the tasks are ordered. Releases of two tasks of $Frame_Task_Type$ are separated at least by the $Interarrival$ attribute of the tasks.

Implementation

Example

We give at figure 25 an example of $Task\ group$ described in Cheddar ADL.

The considered example is a $Transaction$, composed by three periodic tasks.

Note that other parameters are not described here in order to simplify the example.

5 Applications of Cheddar ADL

In this section, we show how Cheddar ADL is used for scheduling analysis in the Cheddar context.

We implement Cheddar ADL through a XML format. XML tags represent the different types of components and attributes. Each real-time application architecture is specified by a XML file, which must be conform to the

```
<transaction_task_group id="_13">
  <object_type>TASK_GROUP_OBJECT_TYPE</object_type>
  <name>TDMA_Frame</name>
  <task_list>
    <periodic_task ref="_14"/>
    <periodic_task ref="_15"/>
    <periodic_task ref="_16"/>
  </task_list>
  <task_group_type>TRANSACTION_TYPE</task_group_type>
  <period>0</period>
</transaction_task_group>
```

Figure 25: An example of *Task group* (of type *Transaction_Type*) description in Cheddar ADL

DTD (Document Type Definition) of Cheddar ADL. Cheddar tools check this XML file format, verify specific consistency rules, and then append an instance of the matching component into the internal system representation.

Notice that the Cheddar ADL file does not indicate what is the type of the scheduling analysis to apply. Users choose the method through a graphical interface, or by calling dedicated programs from the Cheddar toolbox. A tool is also provided to guide users towards the feasibility tests that are usable according to the architecture of their system [40].

Like announced in our requirements, Cheddar ADL should be a gateway with another tool in order to perform schedulability analysis.

The figure 26 shows how, in general case, an analysis tool with it specific language is used to check an another model.

Using an ADL editor or not, the user produces its own model, which is transformed by an ADL Model Translator, in order to generate the specific model, compatible with your analysis tool.

Especially in our case, we show how Cheddar is integrated into an iteration of the development process of a real-time system using AADL inspector, an AADL model editor[2]:

[2]AADL inspector is a product of Ellidiss Technologies http://www.ellidiss.com

Figure 26: How interoperability with other ADLs is ensured: the particular case of AADL and Cheddar ADL

- The designer models a system using AADL inspector.
- The system is then transformed toward the Cheddar ADL used by the analysis tool.
  This transformation extracts information relevant to the schedulability analysis only.
  The example of figure 27 is a result of transformation.

  It is composed of a set periodic tasks, with $start\_time$ equal to 0.

  The tasks run on a multi-cores processors, with two identical cores, with the scheduler $posix\_1003\_highest\_priority\_first\_protocol$.

  One address space, linked to the processor, allows to model logical memory.

  Finally, a *static deployment* is used to give the relationships between the multi-cores processors and tasks. It is the off-line scheduling, and *scheduling_sequence.xml* gives the time moments when each task is pre-empted.
- The analysis tool performs the schedulability analysis and provides an analysis report.
  We give at figure 28 a Cheddar scheduling simulation of our example, on the hyper-period.

```
<cheddar>
 ...
 <core_unit id="␣59" >
   <name>core1</name>
   <scheduler_type>
   POSIX_1003_HIGHEST_PRIORITY_FIRST_PROTOCOL
   </scheduler_type>
   <preemptive_type> PREEMPTIVE </preemptive_type>
  </core_unit>
  ...
  <multi_cores_processor id="␣61" >
   <name>processor1</name>
   <processor_type> IDENTICAL_MULTICORES_TYPE
   </processor_type>
   <migration_type> TIME_UNIT_MIGRATION_TYPE
   </migration_type>
   <cores>
    <core_unit} ref="␣59" />
    <core_unit} ref="␣60" />
   </cores>
  </multi_cores_processor>
  ...
  <periodic_task id="␣63" >
   <name>T1</name>
   <cpu_name>processor1</cpu_name>
   <capacity>2</capacity>
   <deadline>4</deadline>
   <start_time>0</start_time>
   <policy>SCHED_FIFO</policy>
   <period>4</period>
   <jitter>0</jitter>
  </periodic_task>
  ...
  <static_deployment id="␣66" >
   <name>static_example</name>
   <consumer_entities>
    <multi_cores_processor ref="␣61" />
   </consumer_entities>
   <resource_entities>
    <periodic_task ref="␣63" />
    <periodic_task ref="␣64" />
    <periodic_task ref="␣65" />
   </resource_entities>
   <allocation>          scheduling_sequence.xml
   </allocation>
  </static_deployment>
  ...
</cheddar
```

Figure 28: A Cheddar scheduling simulation of our example

Let us remark that in this case, our application is not feasible.

6 Related work

The Dedal [1] context is the development of software based components. The life cycle of the software is composed of three steps: specification, deployment and exploitation, which are closely linked in term of maintenance.

Dedal is an ADL which aim to define independently specification, configuration and assembly of an architecture, in order to coordinate the evolution of different levels of an abstraction. Usually, only two of the three levels are taken into account in the definition of ADLs.

Three dimensions so define the language:

- The abstract specification of an architecture: it is the description of all roles played by components that participate in the realization of the architecture. At this level, the authors focus on the definition of roles of components, connections between component interfaces and behaviour of the architecture.

- The concrete configuration of an architecture: this is to define the classes of components and connectors used to implement the architecture.

- The assembly of an architecture: this step describes all instances, components and connectors that make up the architecture. Here, the authors assign values or constraints to the components.

In the Cheddar context, our objective is the proposition of an architecture description language for real-time applications, in order to perform scheduling analysis.

Although it is not the same context, the description of Cheddar ADL is based on the same logic: software component for abstract specification, hardware component for concrete specification and binding for assembly.

The Architecture Analysis & Design Language (AADL) is a SAE standard (AS-5506), first published in 2004 [47]. AADL targets the design, analysis and integration of distributed real-time systems. An AADL model describes a system as a hierarchy of components with their interfaces and their connections. It allows the modelling of the software components and their interactions, and also of the execution platform. Component categories are *process*, *data*, *thread*, *subprogram* for the software modelling, and *processor*, *memory*, *bus* and *device* for the entities of the execution platform. The deployment of a software application onto an execution platform is specified through *binding* properties. The execution of a software task may be assigned to one or a set of components of the execution platform. The AADL standard includes a large set of properties to precisely model system characteristics. Moreover, new ones may be appended to extend the description with regard to the expected system analysis.

The AADL language is not especially dedicated to analysis of real-time systems. It takes into account more components, and the concepts is not very adapted for a classical designer of real-time systems. This is actually a language too heavy, too wide and not especially dedicated to the schedulability analysis of real-time systems. Moreover, it is not very adapted for a regular user of real-time systems. By example, *device* is an inappropriate term for a classic user of real-time systems, whose objective is the schedulability analysis.

Modelling and Analysis of Real-Time Embedded systems (MARTE) is a standard UML profile promoted by the Object Management Group [50] [51] [52]. The profile adds capabilities to UML for Model-Driven Development [53] of real-time systems. MARTE thus provides support to specify and to

design such a system but also to annotate the model for different kinds of analysis. MARTE was designed to cover a large area of real-time systems, including avionic, automotive or software radio systems. For example, the profile was designed so that all AADL concepts can be modelled in MARTE ([50], Annex A, section 2.3). The profile was also designed to support tools dedicated to real-time system (e.g. modeller, code generator, functional analyser, simulator). Several real-time analysing tools have been developed using MARTE, such as: [54], which presented the MARTE model elements associated with the time model package of MARTE, and illustrated their use on an automotive case study. For that, they extracted the physical timing information and used it to perform a schedulability analysis.

The real-time profile of MARTE is dedicated to model and analyse real-time systems, by cons, MARTE's approach is not to create new analysis methods, but to support existing ones, as opposed to Cheddar ADL which offers possibilities to take into account new analytical techniques.

In the context of automotive application, [55] investigate schedulability of real-time systems at earliest design phases. Their approach is based on the combination of two modelling languages for system design: EAST-ADL2, which addresses modelling and analysis needs of automotive electronic systems and the integration of an open source toolset for scheduling analysis, MAST [56] [57]. On one hand, EAST-ADL2 is an architecture description language defined as a domain specific language for the development of automotive electronic systems. On the other hand, MARTE is known for its rich expressive power for the modelling of system real-time properties and constraints. Their methodology has the objective of completing EAST-ADL models with MARTE entities to enable scheduling analysis at the design level. MAST is used to perform schedulability analysis. Other works based on EAST-ADLs have been done: It is the case of [58], which combines TADL2, Timing Augmented Description Language v.2, EAST-ADL and UPPAAL [59] to perform scheduling analysis of real-time systems.

We can note that, like AADL, EAST-ADL is not initially designed to perform schedulability analysis, and the diversity of languages that are associated do not facilitates neither automatic code generation, nor reactivity about the integration of new components.

To easily cover new real-time scheduling models and techniques, [60] propose an ADL: MoSaRT. This approach may be used to extract the scheduling

information from different design methodologies and ADL used for system design such as UML-MARTE or AADL. The extracted information is then modeled with MoSaRT. Thus, it fills the gap between the conception abstraction level and the analysis abstraction level by capturing information relevant for analysis.

Yet, this language is not dedicated to analysis itself, but to model transformation between ADLs with different purposes. Cheddar ADL is, on its side, built from the analysis methods it supports.

7 Conclusion

This paper presented Cheddar ADL, an Architecture Design Language for the scheduling of real-time systems.

The particularity of this ADL, as compared to other ADLs, is that it allows to capture all required aspects for the schedulability analysis of real-time systems.

After the presentation of the requirements of this ADL, we classified the elements into two categories: software part, which contain *Address space* , $Task$, $Buffer$, *Resource* , $Message$ and *Dependency* , and hardware part, which contain $Core$, $Cache$, $Processor$ and $Network$.

Another category, deployment, is in fact a combination of these two categories.

We then presented in detail each of theses elements, by giving for each, the definitions, the standard attributes, the legality rules, an implementation and an example.

Subsequently, we have shown how the language is used as an entry point to the Cheddar tool, a real-time scheduling simulator.

In order to extend the usability of the language, we have also been interested in interoperability with other tools for analysing real-time systems that are more and more numerous.

For that, we proposed an approach which allow to use our Cheddar tool, in order to perform schedulability analysis of real-time systems described in other languages. This approach consists in fact in transforming the entry ADL, into Cheddar ADL.

The language is meant to evolve. Therefore, our future works will concern the extension, in order to consider new hardware (e.g. heterogeneous multiprocessors), that can provide schedulability results closest to reality; and

then, new feasibilities tests.

We also plan to focus on evaluation of our language, by comparing with other ADLs. The paper of [61] will be the starting point. Indeed, [61] compare different ADLs under certain aspects: syntax; visualization, which concerns graphical representation; variability and extensibility, the capability to model new patterns. The paper of [62] provides also a good basis for this study, because it provides a framework which is used to classify and compare several existing ADLs.

References

[1] Z. H. Yulin and C. Urtado and S. Vauttier. Dedal: Un ADL à trois dimensions pour gérer l'évolution des architectures à base de composants. 4eme Conférence francophone sur les architectures logicielles, Pau, France. CAL 2010.

[2] A. J. Smith. Cache Memories. Computing Surveys, Vol. 14, No. 3, September 1982. ACM Press.

[3] Sebek, Filip. "The state of the art in cache memories and real-time systems." (2001).

[4] Mellor-Crummey, John, David Whalley, and Ken Kennedy. "Improving memory hierarchy performance for irregular applications using data and computation reorderings." International Journal of Parallel Programming 29.3 (2001): 217-247.

[5] Zahran, Mohamed. "Cache replacement policy revisited." Proceedings of the 6th Workshop on Duplicating, Deconstructing, and Debunking. 2007.

[6] Jaleel, Aamer, et al. "High performance cache replacement using re-reference interval prediction (RRIP)." ACM SIGARCH Computer Architecture News. Vol. 38. No. 3. ACM, 2010.

[7] F. Baslett, T. Jermoluk, and D. Solomon, "The 4D-MP Graphics Superworkstataion: Computing+Graphics= 40MIPS+40MFLOPS and 100,000 Lighted Polygons per Second," Proc. 33rd IEEE Computer Society Int'l Conference – COMPCON'88, pp 468-471, February 1988.

[8] Papamarcos, Mark S., and Janak H. Patel. "A low-overhead coherence solution for multiprocessors with private cache memories." ACM SIGARCH Computer Architecture News. Vol. 12. No. 3. ACM, 1984.

[9] F. Singhoff and J. Legrand and L. Nana and L. Marcé. Cheddar: a flexible Real-Time Scheduling Framework. ACM SIGAda Ada Letters. ACM Press, New York, USA. 24 (4). pp 1–8. Dec 2004.

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

[11] G. C. Buttazo. Hard real-time computing systems: predictable scheduling algorithms and applications. Kluwer academics. 1997.

[12] F. Singhoff, A. Plantec, S. Rubini, V. Gaudel, S. Li, C. Fotsing, P. Dissaux, J. Legrand, L. Lemarchand. How architecture description languages help (or not) schedulability analysis : the example of Cheddar. Preprint submitted to Science of Computer Programming. SCP13. 2013.

[13] S. Li and F. Singhoff and S. Rubini and M. Bourdellès. Applicability of real-time schedulability analysis on a software radio protocol. Proceedings of the 2012 ACM conference on High integrity language technology. New York, USA, pp. 81–94. December 2012.

[14] C. Srilatha and C. V. Guru Rao and G. Prabhu. Effective Cache Configuration for High Performance Embedded Systems. American Journal of Computer Architecture, 1(1). pp. 1–5. DOI: 10.5923/j.ajca.20120101.01. 2012.

[15] http://searchdatacenter.techtarget.com/definition/multi-core-processor.

[16] R. Kumar and D. Tullsen and N. Jouppi. Core Architecture Optimization for Heterogeneous Chip Multiprocessors. PACT'06. September 16, 20. Seattle, Washington, USA. ACM 1-59593-264-X/06/0009. 2006.

[17] R. Mall. Real-Time Systems: Theory and Practice. Pearson Education India. September 14, 2006. ISBN-10: 8131700690.

[18] F. Singhoff. Real time scheduling theory and its use with Ada. ACM SIGAda'07 tutorial, Washington DC, USA.

[19] G. Buttazzo. Rate monotonic vs. EDF: Judgment day. In Proc. 3rd ACM International Conference on Embedded Software, Philadephia, USA , October 2003.

[20] J. Zalewski. What Every Engineer Needs To Know About Rate-Monotonic: A Tutorial. Real-Time Magazine. 1995. Edited by Zalewski, IEEE Computer Society Press. Vol 1.

[21] C.Chou and I. Cidon and I. S. Gopal and S. Zaks. Synchronizing Asynchronous Bounded Delay Networks. IEEE TRANSACTIONS ON COMMUNICATIONS, VOL.38, NO. 2, FEBRUARY 1990.

[22] M. Emmi and S. Qadeer and Z. rakamarié. Delay-Bounded Scheduling. PoPL'11, January 26–28, 2011, Austin, Texas, USA. ACM 978-1-4503-0490-0/11/01.

[23] G. Blake and R. G. Dreslinski and T. Mudge. A Survey of Multicore Processors (A review of their common attributes). IEEE SIGNAL PROCESSING MAGAZINE. 1053-5888/09/26.002009.IEEE. November 2009.

[24] M. Katre and H. Ramaprasad and A. Sarkar and F. Mueller. Policies for Migration of Real-Time Tasks in Embedded Multi-Core Systems. RTSS09, WIP. 2009.

[25] N. W. Fisher. The Multiprocessor Real-Time Scheduling of General Task Systems. Phd Thesis. University of North Carolina, USA. 2007.

[26] A. Sarkar and F. Mueller and H. Ramaprasad. Predictable Task Migration for Locked Caches in Multi-Core Systems. LCTES11. Chicago, Illinois, USA. 2011.

[27] http://publib.boulder.ibm.com/infocenter/zos/basics/index.jsp?topic=/com.ibm.zos.zconcepts/zconcepts_82.htm.

[28] http://menehune.opt.wfu.edu/Kokua/More_SGI/007-2478-008/sgi_html/ch01.html.

[29] N. Maculan and S.C.S. Porto and C.C. Ribeiro and C.C. De. Souza. A new formulation for scheduling unrelated processor under precedence constraints. RAIRO Rech. Oper. Vol. 33. Nunmber 1. pp. 87-92. 1999.

[30] E. L. Lawler and J. Labetoulle. On Pre-emptive Scheduling of Unrelated Parallel Processors by Linear Programming. Journal of the Association for Computing Machinery. pp 612–61. Vol 25. No 4. October 1978.

[31] K. Jansen and C. Robenet. Scheduling jobs on identical and uniform processors revisited. In Proceeding WAOA'11. Proceedings of the 9th international conference on Approximation and Online Algorithms. Pages 109-122. 2012.

[32] S. Baruah and J. Goossens. The Static-priority Scheduling of Periodic Task Systems upon Identical Multiprocessor Platforms. Fifteenth IASTED International Conference on Parallel and Distributed Computing and Systems. Pages 427–432. Marina del Rey CA, November. Acta Press. ISBN 0-88986-392-X.

[33] A. Hyari. A Comparative Study on Heterogeneous and Homogeneous Multiprocessors. University of Jordan. 2009.

[34] Q. Li, D. L. Mills, "Investigating the Scaling Behavior, Crossover and Anti-persistence of Internet Packet Delay Dynamics," Proceedings of IEEE Globalcom, Vol. 3, pp. 1843-1852, Rio de Janeiro, Brazil, 1999.

[35] E.J. Daniel and C.M. White and K.A. Teague. An inter-arrival delay jitter model using multi-structure network delay characteristics for packet networks. Sch. of Electr. Comput. Eng., Oklahoma State Univ., Stillwater, OK, USA. pp 1738 - 1742. Vol.2. 2003.

[36] John W. McCormick and Frank Singhoff and Jérôme Hugues. Building Parallel, Embedded, and Real-Time Applications with Ada. Cambridge University Press. ISBN-13: 978-0521197168. 2011.

[37] E. Maes and N. Vienne. MARTE to Cheddar Transformation Using ATL. Technical report. THALES Research and Technologies. 2007.

[38] A. Plantec and V. Ribaud. PLATYPUS: A STEP-based Integration Framework. 14th Interdisciplinary Information Management Talks (IDIMT-2006). pp 261-274. September 2006.

[39] F. Singhoff and A. Plantec. Towards user-level extensibility of an Ada library: an experiment with cheddar. Proceedings of the 12th international conference on Reliable software technologies. Ada-Europe'07. isbn 978-3-540-73229-7. pp 180-191. 2007.

[40] V. Gaudel and F. Singhoff and A. Plantec and S. Rubini, P. Dissaux and J. Legrand. An Ada design pattern recognition tool for AADL performance analysis. Ada Letters.

[41] S. Rubini, F. Singhoff and J. Hugues. Modeling and Verification of Memory Architectures with AADL and REAL. Sixth IEEE International Workshop on UML and AADL. In the proceedings of the 16th IEEE International Conference on Engineering of Complex Computer Systems. Las Vegas, USA. pp 338–343. isbn 978-0-7695-4381-9. April 2011.

[42] M.I. Chen and K.J. Lin. Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems. Journal of Real-Time Systems. 2. pp 325 – 346. 1990.

[43] L. Sha and R. Rajkumar and J.P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. IEEE transactions on computers. 39 (9). pp 1175 – 1185. 1990.

[44] J. Legrand, F. Singhoff, L. Nana and L. Marcé. Performance Analysis of Buffers Shared by Independent Periodic Tasks. LISYC Technical report number legrand-02-2004. January 2004.

[45] L. Kleinrock. Queueing Systems: Theory. Wiley. 1. 1976.

[46] J. C. Palencia and M. G. Harbour. Exploiting precedence relations in the schedulability analysis of distributed real-time systems. The 20th IEEE Real-Time Systems Symposium. pp 328 - 339. 1999.

[47] P. Feiler and B. Lewis and S. Vestal. The SAE AADL standard: A basis for model-based architecture-driven embedded systems engineering. Workshop on Model-Driven Embedded Systems. May 2003.

[48] S.K. Baruah. Feasibility analysis of recurring branching tasks. In proceedings of the 10th Euromicro Workshop on Real-Time Systems. pp 138–145. Jun 98.

[49] A. K. Mok and D. Chen. A Multiframe Model for Real-Time Tasks. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. 23 (10). pp 635 – 645. October 1997.

[50] MARTE Specification. http://www.omg.org/spec/MARTE. Object Management Group. 2005.

[51] UML Specification. http://www.omg.org/spec/UML. Object Management Group. 2011.

[52] OMG Website. http://www.omg.org. Object Management Group. 2013.

[53] Douglas C. Schmidt. Model-Driven Engineering. IEEE Computer. 39 (2). February 2006.

[54] M. Peraldi-Frati and Y. Sorel. From hight level modelling of time in MARTE to real-time scheduling analysis. MODELS 2008.

[55] Anssi, Saoussen and Gérard, Sébastien and Albinet, Arnaud and Terrier, François. Requirements and Solutions for Timing Analysis of Automotive Systems. System Analysis and Modeling: About Models. Lecture Notes in Computer Science. Kraemer, Frank and Herrmann, Peter. Springer Berlin / Heidelberg. 6598. pp 209-220. 2011.

[56] Harbour, Michael Gonzalez and Garcia, Javier Gutierrez and Palencia, J.C. and Drake Moyano, J.M. MAST: Modeling and analysis suite for real-time applications. isbn = 0-7695-1221-6. Proceedings of the 13th Euromicro Conference on Real-Time Systems. IEEE Comput. Soc. pp 125–134. 2001.

[57] J. M. Drake and M. G. Harbour and J. J. Gutiérrez and P. L. Martinez and J. L. Medina and J. C. Palencia. Modeling and Analysis Suite for Real-Time Applications (MAST 1.4.0): Description of the MAST Model. Universidad de Cantabria, SPAIN. 2011.

[58] A. Goknil and J. Suryadevara and M. Peraldi-Frati and F. Mallet. Analysis Support for TADL2 Timing Constraints on EAST-ADL Models. ECSA 2013. pp 89–105. 2013.

[59] G. Behrmann and R. David and K. G. Larsen. 3185. Lecture Notes in Computer Science. A tutorial on UPPALL. International School on

Formal Methods for the Design of Computer, Communication and Software Systems. SFM-RT 2004. pp 200–237. 2004. Springer Verlag. Updated November 28, 2006.

[60] Y. Ouhammou and E. Grolleau and M. Richard and P. Richard. Towards a Simple Meta-model for Complex Real-Time and Embedded Systems. The First International Conference, MEDI 2011. pp 226-236. Obidos, Portugal. 2011.

[61] A. W. Kamal and P. Avgeriou. An Evaluation of ADLs on Modeling Patterns for Software Architecture. RISE07. 2007.

[62] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. Software Engineering, IEEE Transactions. 26 (1). pp 70–93. January 2000.