Cheddar Architecture Description Language

Christian Fotsing, Frank Singhoff, Alain Plantec, Vincent Gaudel, Stéphane Rubini, Shuai Li, Hai Nam Tran, Laurent Lemarchand
Lab-STIC/UMLR 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

November 20, 2018

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

1 Introduction ............................................. 7
2 Requirements ............................................ 8
3 Cheddar ADL Concepts ..................................... 8
4 Semantic of Components .................................. 9
   4.1 Hardware Components ................................ 9
   4.1.1 Cache ........................................... 9
   4.1.2 Core .......................................... 14
   4.1.3 Processor ....................................... 17
   4.1.4 Memory ......................................... 19
   4.1.5 Network ......................................... 21
   4.2 Software Components ................................ 23
   4.2.1 Address space ................................... 23
   4.2.2 Task ............................................ 25
   4.2.3 Resource ........................................ 29
   4.2.4 Buffer .......................................... 35
   4.2.5 Message ......................................... 40
   4.2.6 Dependency ...................................... 42
5 Notion of Deployments ................................... 45
   5.1 Generic Deployments ................................ 45
   5.2 Static Deployments ................................ 47
   5.3 Dynamic Deployments ................................ 48
6 Applications of Cheddar ADL .............................. 53
7 Related work ............................................ 56
8 Conclusion ............................................... 58
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The DTD of entity Cache</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>An Example of entity Cache</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>The DTD of entity Core</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>An example of Core description</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>The DTD of entity Processor</td>
<td>19</td>
</tr>
<tr>
<td>6</td>
<td>An example of Mono_core_processor description</td>
<td>20</td>
</tr>
<tr>
<td>7</td>
<td>An example of Multi_cores_processor description</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>The DTD of entity Network</td>
<td>22</td>
</tr>
<tr>
<td>9</td>
<td>An example of Network description</td>
<td>22</td>
</tr>
<tr>
<td>10</td>
<td>The DTD of entity Address_Space</td>
<td>25</td>
</tr>
<tr>
<td>11</td>
<td>An Example of entity Address_space described using Cheddar ADL</td>
<td>25</td>
</tr>
<tr>
<td>12</td>
<td>The DTD of entity Task</td>
<td>30</td>
</tr>
<tr>
<td>13</td>
<td>An example of Task description in Cheddar ADL</td>
<td>31</td>
</tr>
<tr>
<td>14</td>
<td>The DTD of entity Resource</td>
<td>34</td>
</tr>
<tr>
<td>15</td>
<td>An example of Resource description in Cheddar ADL</td>
<td>35</td>
</tr>
<tr>
<td>16</td>
<td>The DTD of entity Buffer</td>
<td>39</td>
</tr>
<tr>
<td>17</td>
<td>An example of entity Buffer</td>
<td>39</td>
</tr>
<tr>
<td>18</td>
<td>The DTD of entity Message</td>
<td>42</td>
</tr>
<tr>
<td>19</td>
<td>An example of entity Message</td>
<td>43</td>
</tr>
<tr>
<td>20</td>
<td>The DTD of entity Dependency</td>
<td>45</td>
</tr>
<tr>
<td>21</td>
<td>Examples of Dependency</td>
<td>46</td>
</tr>
<tr>
<td>22</td>
<td>The DTD of entity Generic_Deployment</td>
<td>47</td>
</tr>
<tr>
<td>23</td>
<td>The DTD of entity Static_Deployment</td>
<td>48</td>
</tr>
<tr>
<td>24</td>
<td>An example of Static_Deployment</td>
<td>48</td>
</tr>
<tr>
<td>25</td>
<td>The DTD of entity Dynamic_Deployment</td>
<td>51</td>
</tr>
<tr>
<td>26</td>
<td>An example of Dynamic_deployment description in Cheddar ADL</td>
<td>52</td>
</tr>
<tr>
<td>27</td>
<td>How interoperability with other ADLs is ensured: the particular case of AADL and Cheddar ADL</td>
<td>53</td>
</tr>
<tr>
<td>28</td>
<td>Example of an application specified using Cheddar ADL</td>
<td>55</td>
</tr>
<tr>
<td>29</td>
<td>A Cheddar scheduling simulation of our example</td>
<td>56</td>
</tr>
</tbody>
</table>
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.

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).

\footnote{The lcm periods}
After that, the sections 4 and 5 precise the semantic of the basic entities of Cheddar ADL, and section 6 is dedicated to the way to apply it. Section 7 gives the related works, by describing some ADLs using in real-time domains, and we conclude in section 8, 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 \textit{Processor}, \textit{Cache}, \textit{Core}, \textit{Memory} and \textit{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 5 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 Cache

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

(1) It is 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.

(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 is instruction cache, data cache or a combined one.

**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 \((\text{Associtivity} = 1)\).

(A13) A way set associative cache: when a memory block can reside in exactly \(A\) location in the cache \((\text{Associtivity} = A)\).

In this case, we have

\[
\text{Number of set of the cache} = \frac{\text{Number of block}}{A}
\]

(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 possible 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.
Figure 1: The DTD of entity Cache Implementation

The figure 1 gives the DTD of entity Cache.
Figure 2: An Example of entity Cache

The figure 2 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.

4.1.2 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 read and execute program instructions.

Standard attributes

Name: It is the unique name of Core.

Speed: It is a real, which gives the exchange rate of flow.

$L1_{cache\_system\_name}$: It is a string, which corresponds to the primary cache. It is faster, generally smaller, and located in core.

The $L1_{cache\_system}$ 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 scheduling. It is the type of `Scheduling_Parameters` (see Annexes for definitions of `Scheduling_Parameters`).

**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 ≠ Empty)` and `(A_Scheduler ≠ Pipeline_User_DEFINED_Protocol)` and `(A_Scheduler ≠ 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 ≠ 0)` and `(A_Scheduler ≠ Posix_1003_Highest_Priority_First_Protocol)` and `(A_Scheduler ≠ Round_Robin_Protocol)` and `(A_Scheduler ≠ Hierarchical_Round_Robin_Protocol)` and `(A_Scheduler ≠ 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.

(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 3 gives the DTD of entity `Core`.

**Example**

The figure 4 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 `core1`, and it is of `Preemptive` Type. The `L1_cache_system_name` is not represented.
Figure 3: The DTD of entity Core

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

Figure 4: An example of Core description

```
<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>1.00000</speed>
</core_unit>
```
4.1.3 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.

(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.
Figure 5: The DTD of entity Processor

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 5 gives the DTD of entity Processor.

Example

The example of figure 6 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 7 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.

4.1.4 Memory

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

(1) It is a deployment unit for a software component.
Figure 6: An example of Mono_core_processor description

```
<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>
  </mono_core_processor>
</processors>
```

Figure 7: An example of Multi_cores_processor description

```
<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>
```
(2) It is everything a process can address, code, data, stack, heap.

(3) In uniprocessor designs, the memory system was 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 just one 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.

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.

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 8 gives the DTD of entity *Network*.

Example

The figure 9 gives an example of entity *Network* described using Cheddar ADL.

The type of *Network* in this case is *BOUNDDED_DELAY*.
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).

**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**


**Implementation**

The figure 10 gives the DTD of entity Address space.

**Example**

The figure 11 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.
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.

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.

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.
Legality rules

(L1) The Task name must not be empty.

(L2) The Task name must be a 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 ≠ 0 Policy = Sched_Others.
We can’t have simultaneously $\text{Priority} = 0 \text{ Policy} \neq \text{Sched-Others}$. 

Annexes

(A1) The types of $\text{Generic\_Task}$.

(A11) $\text{Periodic\_Type}$: In this case, the $\text{Task}$ is periodic, and we have two more attributes in order to characterize the $\text{Task}$:

(A111) $\text{Period}$: 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 $\text{Processor}$ that owns a Rate Monotonic or a Deadline Monotonic scheduler, you have to give a period for each of its tasks.

(A112) $\text{Jitter}$: The $\text{Jitter}$ of $\text{Task}$ is an upper bound on the delay that $\text{Task}$ 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) $\text{Aperiodic\_Type}$ In this case, the $\text{Task}$ is called aperiodic task. An aperiodic task is only activated once.

(A13) $\text{Sporadic\_Type}$: A sporadic task is a task which is activated many times with a minimal delay between two successive activations.

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

(A14) $\text{Poisson\_Type}$. In this case, the task is called poisson task.

It is a subtype of $\text{Periodic}$ task, with two more attributes.

(A141) $\text{Seed}$: 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 $\text{Seed}$ 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 $\text{gettimeofday}$.

(A142) $\text{Predictable}$: It is a boolean, which guides the seed value (see the Seed definition).

(A15) $\text{Parametric\_Type}$. In this case, the task is called parametric task, and it is characterized by one more attribute.

(A151) $\text{Activation\_Rule}$: The name of the rule which defines the way the task should be activated. Only used with user-defined task.
(A16) Scheduling_Type: It is one of the types of Task.

(A17) Frame_Type. In this case, the task 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.

(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 12 gives the DTD of entity Task.

Example

We give at figure 13 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 processor1, and addr1 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.

(2) It is statically defined in an Address space.
Figure 12: The DTD of entity Task
(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
A critical section is defined by two relative instants: \textit{Begin} and \textit{End}. \textit{Begin} is the first unit of time of the task capacity when the task needs the resource, i.e. before running \textit{Begin}th unit of time of its capacity, the task locks the \textit{Resource} component. \textit{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 \textit{End}th unit of time of its capacity, the task unlocks the \textit{Resource} component.

\textit{Cpu\_Name}: Each shared resource has to be located on a given Processor.

\textit{Address\_Space\_Name}: Its stores the name of \textit{Address space} which hosted the \textit{Resource}.

\textit{Priority}: Is of type of \textit{Priority\_Range} and it defines the ceiling priority of the \textit{Resource}. This attribute is currently only used with the PCP and ICPP protocols.

\textit{Priority\_Assignment}: It is an enumerated type, and characterize the way that Cheddar assigns ceiling priority to the \textit{Resource}.

\textbf{Legality rules}

(L1) The resource name is mandatory.

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

(L3) The \textit{Cpu\_Name} is mandatory.

(L4) The \textit{Address\_Space\_Name} is mandatory.

(L5) The types of \textit{Protocol} are specified in Annexes.

(L6) The \textit{Size} must be greater than or equal to 0.

(L7) The \textit{Address} must be greater than or equal to 0.

(L8) The \textit{State} must be greater than or equal to 0.

\textbf{Annexes}

(A1) Each critical section is defined by:

(A11) \textit{Task\_begin}: The time at which the critical section is started. Task is the \textit{Task name} accessing the shared \textit{Resource}.

(A12) \textit{task\_end}: The time at which the critical section is completed. Task is the \textit{Task name} accessing the shared \textit{Resource}.

Each of this data are relative to the task capacity. Finally, several critical sections can be defined for a given task on a given resource.
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 up to the ceiling priority level of the resource. At request time, a task is blocked when its priority level is not strictly 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 its dynamic priority level increased up to the ceiling priority level of the Resource component. Similarly, the Task priority is decreased when the Resource is unlocked. Again, the ceiling 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 14 gives the DTD of entity Resource.
<![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 14: The DTD of entity Resource
An example of resource described in Cheddar ADL is given in figure 15. 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

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.

(L10) The time of the read/write operation is also given: 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) $Q_{s-Pp1}$: 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) $Q_{s-Mm1}$: 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) $Q_{s-Md1}$: 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) $Q_{s-Mp1}$: compliant with the classical $M/P/1$ queueing system model.

(A15) $Q_{s-Mg1}$: compliant with the classical $M/G/1$ queueing system model.

(A16) $Q_{s-Mms}$: compliant with the classical $M/M/S$ queueing system model.

(A17) $Q_{s-Mds}$: compliant with the classical $M/D/S$ queueing system model.

(A18) $Q_{s-Mps}$: compliant with the classical $M/P/S$ queueing system model.

(A19) $Q_{s-Mgs}$: compliant with the classical $M/G/S$ queueing system model.

(A110) $Q_{s-Mm1n}$: compliant with the classical $M/M/1/N$ queueing system model.

(A111) $Q_{s-Md1n}$: compliant with the classical $M/D/1/N$ queueing system model.

(A112) $Q_{s-Mp1n}$: compliant with the classical $M/P/1/N$ queueing system model.

(A113) $Q_{s-Mg1n}$: compliant with the classical $M/G/1/N$ queueing system model.

(A114) $Q_{s-Mmsn}$: compliant with the classical $M/M/S/N$ queueing system model.

(A115) $Q_{s-Mdsn}$: compliant with the classical $M/D/S/N$ queueing system model.
(A116) \(Q_s - M_{psn}\) : compliant with the classical M/P/S/N queueing system model.

(A117) \(Q_s - M_{gsn}\) : 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.

(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.

(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 16 gives the DTD of entity Buffer.

Example

The figure 17 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.
Figure 16: The DTD of entity Buffer

```xml
<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>
      </buffer_role>
    </roles>
  </buffer>
</buffers>
```

Figure 17: An example of entity Buffer
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 UserDefined_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.

- **Response_Time**: Amount of time a message takes to reach the receiver after it has been sent by the sender. 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.

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.

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

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.

(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.
Figure 18: The DTD of entity Message

(A23) Name: The name of the parameter.

Implementation

The figure 18 gives the DTD of entity Message.

Example

The figure 19 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.

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 precedence 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 or received. The attribute Asynchronous Communication Protocol 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* 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 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.

**Implementation**

The figure 20 gives the DTD of entity *Dependency*.

**Examples**

The figure 21 gives examples of *Dependency*.

We illustrate with an example of *QUEUING BUFFER DEPENDENCY*, *RESOURCE DEPENDENCY*, *ASYNCHRONOUS COMMUNICATION DEPENDENCY* and *PRECEDENCE DEPENDENCY*.
5 Notion of Deployments

A *Deployment* models a relationship between two sets of entities: consumer entities and resource entities.

It is specified by the following definitions.

(1) It may be static or dynamic.

(2) We distinguish *Generic Deployments*, which is composed by *Static Deployments* and *Dynamic Deployments*.

5.1 Generic Deployments

(1) A *Generic Deployment* characterizes the deployment notion.

(2) A *Generic Deployment* is a relationship between two set of entities: consumer entities and resource entities.

Standard attributes

*Name*: The unique name of Deployment.

*Consumer Entities*: The set of entities requesting access to the resource.

*Resource Entities*: The set of resources that are made available for resource consumers.
<dependencies>
<dependency>
<type_of_dependency>QUEUING_BUFFERDEPENDENCY
</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>ASYNCHRONOUSCOMMUNICATION_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 21: Examples of Dependency
Figure 22: The DTD of entity Generic_Deployment

Legality rules

(L1) The Generic_Deployment name must not be empty.

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

Annexes

Implementation

The figure 22 gives the DTD of entity Generic_Deployment.

Example

5.2 Static Deployments

(1) A Static Deployment contains a table which defines how the resources are statically allocated by the resource consumers.

(2) It can model a task off-line scheduling for example or a partition off-line scheduling of an ARINC 653 architecture.

Standard attributes

Allocation_description: It defines how the resources are statically allocated by the resource consumers.

This table may be a off-line scheduling of Task or a set of addresses statically defined for each software component inside an Address space.

Legality rules

(L1) The Static_Deployment name must not be empty.

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

Annexes

Implementation
The figure 23 gives the DTD of entity Static_Deployment.

Example

An example of Static_Deployment described in Cheddar ADL is given in figure 24.
In this example, the consumer_entities, a multi_cores_processor needs a periodic_task to work. An off-line scheduling is given in file scheduling_sequence.xml.

5.3 Dynamic Deployments

(1) A Dynamic_Deployment contains an algorithm.

(2) The algorithm defines how the resources are dynamically allocated by the resource consumers.

(3) It allows to define a dynamic resource allocation between the two sets of components.
(4) It may be an on-line scheduling algorithm for Task component or a mal-
loc algorithm for a set of software components inside an Address space.

Standard attributes

Allocation parameters: It is the type of Scheduling Parameters. It specifies how the resources are shared between consumers.
Allocation parameters store parameters of an algorithm, which defines how the resources are scheduling among the resource consumers.

Legality rules

(L1) The Dynamic_Deployment name must not be empty.

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

Annexes


A scheduler is responsible for selecting the running task for each unit of time from among the set of ready tasks. There are various ways to make this choice.

We distinguish:

(A11) Scheduler_type: Which defines the type of scheduler, and may be [17] [18] [19], [20]:

(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: ...

(A1118) Hierarchical_Round_Robin_Protocol: ...

(A1119) Hierarchical_Fixed_Priority_Protocol: ...

(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) Proporionate_Fair_PF_Protocol: ...

(A1125) Proporionate_Fair_PD_Protocol: ...

(A1126) Proporionate_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 process can run on the Processor before being pre-empted by another process 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
(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*. 

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

**Implementation**

The figure 25 gives the DTD of entity *Dynamic_Deployment*.

**Example**

An example of *Dynamic Deployment* is given at figure 26. 

This *dynamic deployment* specify the way that three periodic tasks, referred by *ref 71, 72* and 73 request a *multi-cores_processors*, refered by *ref 69*.

The allocation strategy is *RATE_MONOTONIC*, with another characteristics like *Preemptive_type* which is *PREEMPTIVE*, *capacity* and *period*.
Figure 26: An example of Dynamic deployment description in Cheddar ADL
6 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 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 27 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:

- The designer models a system using AADL inspector.
- The system is then transformed toward the Cheddar ADL used by the analysis tool.

AADL inspector is a product of Ellidiss Technologies http://www.ellidiss.com
This transformation extracts information relevant to the schedulability analysis only.
The example of figure 28 is a result of transformation.
It is composed of a set periodic tasks, with \textit{start\_time} equal to 0.
The tasks run on a multi-cores processors, with two identical cores, with the scheduler \textit{posix\_1003\_highest\_priority\_first\_protocol}.
One address space, linked to the processor, allows to model logical memory.
Finally, a \textit{static deployment} is used to give the relationships between the multi-cores processors and tasks. It is the off-line scheduling, and \textit{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 29 a Cheddar scheduling simulation of our example, on the hyper-period.
Figure 28: Example of an application specified using Cheddar ADL
Let us remark that in this case, our application is not feasible.

7 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.

8 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 these 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


