Semantics of the DOL-Critical XML Schemata

1 Introduction

This section describes the interfaces to the DOL-Critical environment, i.e. the application, platform and mapping specification. In comparison to the original DOL environment (`http://www.tik.ee.ethz.ch/~shapes/dol.html`), the interface as well as the internal implementation of the DOL-environment needs to be completely re-designed. This is due to the extended application models considered in CERTAINTY that deviate from classical data flow models, the fact that several applications are running on the platform with different criticalities, the new mixed-criticality mapping and scheduling methods as well as the newly required performance analysis methods.

DOL has been developed originally for streaming applications which can be represented with data flow formalisms. However, the application domain of CERTAINTY can not be modeled suitably in a data flow model as most tasks are triggered independently by external events or times, and communicate through double buffering mechanisms. This can clearly be seen in the Flight Management System (FMS), which is used as a case study in the CERTAINTY project. Therefore, this section contains the input- and output specification of the DOL-Critical mapping environment that can capture the necessary properties and features of the Flight Management System.

Some of the distinguishing characteristics of the FMS are the following:

- Tasks of different criticality levels coexist in one application.
- Tasks can have complex activation patterns, e.g. pseudo-periodic and restartable.
- Control dependencies can exist between tasks, e.g. a task may be stopped by another task.
- Tasks may execute in different modes depending on the availability of input data.
- Tasks may communicate with each other with double buffering (blackboards) or FIFOs (mailboxes).

All of those features of the Flight Management System application need to be accurately represented by the DOL input specification.

The DOL environment uses the XML format to represent the application specification as well as the platform specification. Given such information, a mapping specification can be generated which contains information about the allocation of hardware resources,

the scheduling parameters on all resources, and the binding of application elements to the hardware resources. The mapping is specified in XML format as well. The application, platform and mapping information is then input to the BIP-environment which performs simulation, formal verification and software generation for the target platform.

In the following, Section 2 describes the DOL-Critical application specification elements and their representation in XML. Section 3 provides information about the specification of a hardware platform and Section 4 about the specification of task mapping in the DOL-Critical XML format.

2 Application specification

The original DOL environment deals with streaming applications, where processes are only triggered by data availability. On the other hand, the Flight Management System application consists of a set of mixed-criticality tasks that are inherently dynamic with complex activation patterns.

Tasks can be periodic, pseudo-periodic, aperiodic, or restartable (please refer to deliverable D1.1 - Criticality and Use Case Requirements for their description and for description of all tasks in the Flight Management System and their dependencies). Moreover, control dependencies can exist between tasks, e.g. the pseudo-periodic task TRAJ_{P1} can start and stop the restartable task TRAJ_{R1}. In addition, there are data dependent executions, e.g. the localization tasks have execution modes that depend on the availability of sensor data. Furthermore, different communication mechanisms are used, such as double buffering and FIFOs.

In DOL-Critical, we can distinguish two layers in an application specification: a functional layer which consists of tasks (processes) and data communication mechanisms between them, and a control layer which consists of task controllers (one per task) and event communication mechanisms. The separation between functional and control information allows a clean way to model applications as considered in CERTAINTY, e.g. the Flight Management System application.

We begin by first showing an example constructed from the Flight Management System application. Figure 1 gives an example of a specification of a subset of the Flight Management System application. Tasks are controlled by corresponding controllers, which specify their activation patterns, execution modes and possible control interactions with other tasks. For example, the SENS_{C1} task is periodically activated by its controller CSENS_{C1}, and it may affect the execution mode of task LOC_{C2} by sending control signals to the controller CLOC_{C2}. Note that in Figure 1, only the high level elements regarding the specification are shown graphically. The XML specification of the application in Figure 1 can be given as follows:

```
01 <app name="FMS">
02   <!--processes-->
03   <process name="SENS_A1" criticality="B">
04     <superblock minRep="1" maxRep="1">
05       <phase name="phase1">
06         <info level="B" minAccess="20" maxAccess="50"
07               minExecution="5" maxExecution="25"/>
```

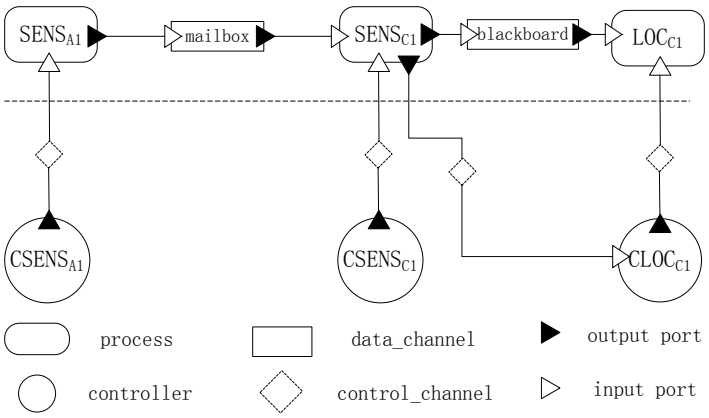


Figure 1: Example Specification

```
08          <info level="C" minAccess="30" maxAccess="45"
09                minExecution="7" maxExecution="18"/>
10          <info level="D" minAccess="35" maxAccess="40"
11                minExecution="10" maxExecution="15"/>
12        </phase>
13      </superblock>
14      <port type="out_data" name="p1"/>
15      <port type="in_event" name="p2">
16        <event name="start"/>
17      </port>
18      <port type="out_event" name="p3">
19        <event name="finish"/>
20      </port>
21      <source location="SENSA1.cpp"/>
22    </process>
23
24    <process name="SENS_C1" criticality="B">
25      <superblock minRep="1" maxRep="1">
26        <phase  name="phase1">
27          <info level="B" minAccess="30" maxAccess="90"
28                minExecution="3" maxExecution="30"/>
29          <info level="C" minAccess="40" maxAccess="60"
30                minExecution="5" maxExecution="20"/>
31          <info level="D" minAccess="50" maxAccess="55"
32                minExecution="8" maxExecution="15"/>
33        </phase>
34      </superblock>
35      <port type="in_data" name="p18"/>
36      <port type="out_data" name="p20"/>
37      <port type="out_event" name="p4"/>
38          <event name="1"/>
39          <event name="2"/>
40        </port>
41      <port type="in_event" name="p5">
42        <event name="start"/>
43      </port>
44      <port type="out_event" name="p6">
45        <event name="finish"/>
46      </port>
```

```
47      <source location="SENSC1.cpp"/>
48    </process>
49
50    <process name="LOC_C1" criticality="B">
51      <superblock mode="1" minRep="1" maxRep="1">
52        <phase name="mode1_phase1">
53          <info level="B" minAccess="20" maxAccess="50"
54                minExecution="5" maxExecution="25"/>
55          <info level="C" minAccess="30" maxAccess="45"
56                minExecution="7" maxExecution="18"/>
57          <info level="D" minAccess="35" maxAccess="40"
58                minExecution="10" maxExecution="15"/>
59        </phase>
60      </superblock>
61      <superblock mode="2" minRep="1" maxRep="1">
62        <phase name="mode2_phase1">
63          <info level="B" minAccess="25" maxAccess="60"
64                minExecution="2" maxExecution="20"/>
65          <info level="C" minAccess="30" maxAccess="45"
66                minExecution="7" maxExecution="18"/>
67          <info level="D" minAccess="35" maxAccess="40"
68                minExecution="12" maxExecution="15"/>
69        </phase>
70      </superblock>
71      <port type="in_data" name="p21"/>
72      <port type="in_event" name="p8">
73            <event name="1"/>
74            <event name="2"/>
75      </port>
76      <port type="out_event" name="p12">
77        <event name="finish"/>
78      </port>
79      <source location="LOCC1.cpp"/>
80    </process>
81
82    <!--controllers-->
83    <controller name="CSENS_A1" deadline="0.2">
84      <activation type="aperiodic">
85        <parameter name="m_max" value="2"/>
86        <parameter name="interval" value="0.2"/>
87      </activation>
88      <port type="out_event" name="p11">
89        <event name="start"/>
90      </port>
91      <port type="in_event" name="p10">
92        <event name="finish"/>
93      </port>
94    </controller>
95
96    <controller name="CSENS_C1" deadline="0.3">
97      <activation type="periodic">
98        <parameter name="period" value="0.2"/>
99      </activation>
100     <port type="out_event" name="p13">
101       <event name="start"/>
102     </port>
103     <port type="in_event" name="p9">
104       <event name="finish"/>
105     </port>
106   </controller>
107
108   <controller name="CLOC_C1" deadline="0.4">
```

```
109     <activation type="periodic_mode">
110       <parameter name="period" value="0.2"/>
111       <parameter name="modes" value="2"/>
112     </activation>
113     <port type="out_event" name="p15">
114           <event name="1"/>
115           <event name="2"/>
116     </port>
117     <port type="in_event" name="p14">
118       <event name="finish"/>
119     </port>
120     <port type="in_event" name="p7">
121           <event name="1"/>
122           <event name="2"/>
123     </port>
124   </controller>
125
126   <!--channels-->
127   <data_channel name="dc1" type="mailbox" size="28" length="2">
128     <port name="p16" type="in_data"/>
129     <port name="p17" type="out_data"/>
130   </data_channel>
131
132   <data_channel name="bb1" type="blackboard" size="340" lifespan="3">
133     <port name="p22" type="in_data"/>
134     <port name="p23" type="out_data"/>
135   </data_channel>
136
137   <control_channel name="sensa1s">
138     <port name="p11"/>
139     <port name="p2"/>
140   </control_channel>
141
142   <control_channel name="sensa1f">
143     <port name="p3"/>
144     <port name="p10"/>
145   </control_channel>
146
147   <control_channel name="sensc1s">
148     <port name="p13"/>
149     <port name="p5"/>
150   </control_channel>
151
152   <control_channel name="sensc1f">
153     <port name="p6"/>
154     <port name="p9"/>
155   </control_channel>
156
157   <control_channel name="locc1s">
158     <port name="p15"/>
159     <port name="p8"/>
160   </control_channel>
161
162   <control_channel name="locc1f">
163     <port name="p12"/>
164     <port name="p14"/>
165   </control_channel>
166
167   <control_channel name="sensc12locc1">
168     <port name="p4"/>
169     <port name="p7"/>
170   </control_channel>
```

```
171
172   <!--global requirements-->
173   <global name="globalCycle">
174     <cycle value="15"/>
175   </global>
176
177   <!--connections-->
178   <connection name="c1">
179     <port name="p1"/>
180     <port name="p16"/>
181   </connection>
182   <connection name="c2">
183     <port name="p17"/>
184     <port name="p18"/>
185   </connection>
186    <connection name="c3">
187     <port name="p20"/>
188     <port name="p22"/>
189   </connection>
190   <connection name="c4">
191     <port name="p21"/>
192     <port name="p23"/>
193   </connection>
194 </app>
```

Listing 1: XML specification for the application example in Figure 1

We proceed with descriptions of the elements that such a specification can contain.

2.1 Elements

In this section, we describe in detail the necessary elements to be able to specify applications as used in CERTAINTY. Namely, we describe the following elements of an XML specification: <process>, <controller>, <data_channel>, <control_channel>, <port>, <connection>, and <global>.

2.1.1 <process>

The <process> element corresponds to a task in an application and represents the basic data processing unit. It contains two compulsory attributes: `name` and `criticality`. The `criticality` attribute specifies the criticality level of a process, and it has one value from the set {`A`, `B`, `C`, `D`, `E`} as specified in the DO-178B standard [1], with `A` being the highest criticality level and `E` being the lowest criticality level. For the Flight Management System application example from the CERTAINTY applications, only the set of criticality levels {`B`, `C`, `D`} exists. Moreover, the <process> element contains the following child elements: <superblock>, <source>, and <port> (described in Section 2.1.6). The actual activation type of the process is defined in its controller, see Section 2.1.2. A process reacts to events sent by its controller instantaneously, i.e., in zero time. There is a one-to-one relation between processes and controllers however, a process may send events to the controllers of other processes.

The compulsory element <superblock> specifies the superblock information of a process, i.e., it contains the memory accesses and execution times information for the

source code of this process [3]. It consists of a list of <phase> elements, each having a unique (in the scope of the corresponding <process>) `name`. Every <phase> element consists further of a list of <info> elements. An <info> element is characterized by several attributes. The `level` attribute specifies under which criticality level assumption the attributes of the <info> element are estimated. This attribute is one value from {`A`, `B`, `C`, `D`, `E`}. The `minAccess` and `maxAccess` attributes specify the minimum and the maximum number of data accesses in one phase. The `minExecution` and `maxExecution` specify the minimum and the maximum execution times of one phase in the superblock model. Each process is considered to have finite, non-zero execution time.

The <process> element may have different modes of activation (see Section 2.1.2). Hence, it may contain several <superblock> elements corresponding to different modes, which specify the different computations it can perform. In this case, the <superblock> element may have the attribute `mode`, of which the value must be an integer. We assume that by default, the `mode` with value "0" corresponds to the degraded mode, while value "1" stands for the normal execution mode. Additionally, for restartable tasks, some superblocks may be needed to execute more than once in a row (depending on how many consecutive "restarts" are allowed). In this case, the optional attributes `minRep` and `maxRep` of the <superblock> element define respectively, the minimum and maximum number of consecutive activations of the superblock that can occur.

The <source> element contains one attribute `location`. The `location` attribute specifies the path and the name of the source code file of the corresponding task.

The <port> elements represent the data and event ports of the process. A port can be of type data input (`in_data`), data output (`out_data`), event input (`in_event`), and event output (`out_event`). Ports will then be connected by corresponding channels which will be described later. Each <process> element must have exactly one input event port for a connection to its controller, and may contain more than one output event port for connections to different controllers (if it needs to notify its controller when finished, or if it influences the execution modes of other controllers and processes).

Listing 2 shows an example XML specification of a process.

```
01 <process name="LOC_C2" criticality="B">
02   <superblock mode="1">
03     <phase name="mode1_phase1">
04       <info level="B" minAccess="20" maxAccess="50"
05             minExecution="5" maxExecution="25" />
06       <info level="C" minAccess="30" maxAccess="45"
07             minExecution="7" maxExecution="18" />
08       <info level="D" minAccess="35" maxAccess="40"
09             minExecution="10" maxExecution="15" />
10     </phase>
11   </superblock>
12   <superblock mode="2">
13     <phase name="mode2_phase1">
14       <info level="B" minAccess="25" maxAccess="60"
15             minExecution="2" maxExecution="20" />
16       <info level="C" minAccess="30" maxAccess="45"
17             minExecution="7" maxExecution="18" />
18       <info level="D" minAccess="35" maxAccess="40"
19             minExecution="12" maxExecution="15" />
20     </phase>
```

```
21   </superblock>
22   <port type="in_event" name="p8">
23     <event name="1"/>
24     <event name="2"/>
25   </port>
26   <port type="out_event" name="p9">
27     <event name="finish"/>
28   </port>
29   <source location="LOCC2.cpp"/>
30 </process>
```

Listing 2: XML specification of one process element

2.1.2 <controller>

Each process in a CERTAINTY application is associated with a controller. Controllers are stateful and describe the activation pattern and the operating modes of a process. They are responsible to trigger the process at the right moment in time, with the correct execution mode.

The <controller> element has two attributes: `name` and `deadline`, where the `deadline` attribute corresponds to the deadline of the modeled task. The <controller> element has a child element <activation> with a compulsory attribute `type`. The possible values of `type` are: `periodic`, `aperiodic`, `pseudoperiodic`, `restartable`, and `periodic_mode`. The <activation> element has one or more child elements <parameter>. The <parameter> element has two attributes, `name` and `value`, which define the type of parameter and its value.

Depending on the activation type of a controller, the allowed types of parameters in the XML specification are:

- `period` (in seconds) for controllers of periodic tasks.
- `m_max, interval` for controllers of aperiodic tasks. Parameter `m_max` denotes the maximum number of task activations within the time interval (in seconds) defined by `interval`.
- `delta` for controllers of pseudo-periodic tasks. Parameter `delta` denotes the time interval (in seconds) between the finish time of a job and the activation of the next one.
- `modes` for controllers of tasks which can be executed at different modes (input data sensitive behavior). Parameter `modes` specifies the number of possible execution modes. The modes of a controller are always denoted as: $\{1, 2, \ldots, \texttt{modes}\}$.

For each type of controller, there are certain events supported which are summarized in Table 1.

A `start` event is sent by a controller to activate the process it controls. A `finish` event is sent by a process to a controller to notify its completion of execution. In the case of a controller with modes, the events are in the range of $\{1,\ldots,\texttt{modes}\}$. When they

Table 1: List of supported events by controller type

controllers	periodic	aperiodic	pseudo-periodic	restartable	periodic_mode
events	`start` `finish`	`start` `finish`	`start` `finish`	`start,restart` `start_res,finish`	{1,...,modes} {1,...,modes},finish

are sent from a controller to a process, they activate the process in the corresponding mode. When they are sent from a process to a controller, they notify the controller in which mode its controlled process should be activated.

Controllers react to events instantaneously, i.e., in zero time. If a controller can receive multiple events, we assume that there is always a sequential ordering among them.

The <controller> element may contain two or more <port> elements. It must contain exactly one input and one output event port to connect with the process it controls, and may contain several input event ports to receive control events from different processes. In addition, a list of <event> elements is attached to the event ports. Note that the events going through the ports must be consistent with the supported events of the connected processes.

Listing 3 shows an example XML specification for a controller.

```
<controller name="CLOC_C2" deadline="0.4">
  <activation type="periodic_mode">
      <parameter name="period" value="0.2" />
      <parameter name="modes" value="2" />
  </activation>
  <port type="out_event" name="p1">
   <event name="1" />
   <event name="2" />
  </port>
  <port type="in_event" name="p2">
   <event name="finish" />
  </port>
  <port type="in_event" name="p3">
   <event name="1" />
   <event name="2" />
  </port>
</controller>
```

Listing 3: XML specification of one controller element

Next, we provide state-based illustrations of the controllers and their interaction with a process with Timed Automata (TA) using Uppaal notation [2] to clarify the semantics of their operation. Note that the timed automata of Figures 2 and 3 are used only for illustration purposes and do not restrict the modeling and implementation of the controller operation.

In the following, we describe some of the specifics of CERTAINTY applications, e.g. the Flight Management System, which a controller needs to be able to model:

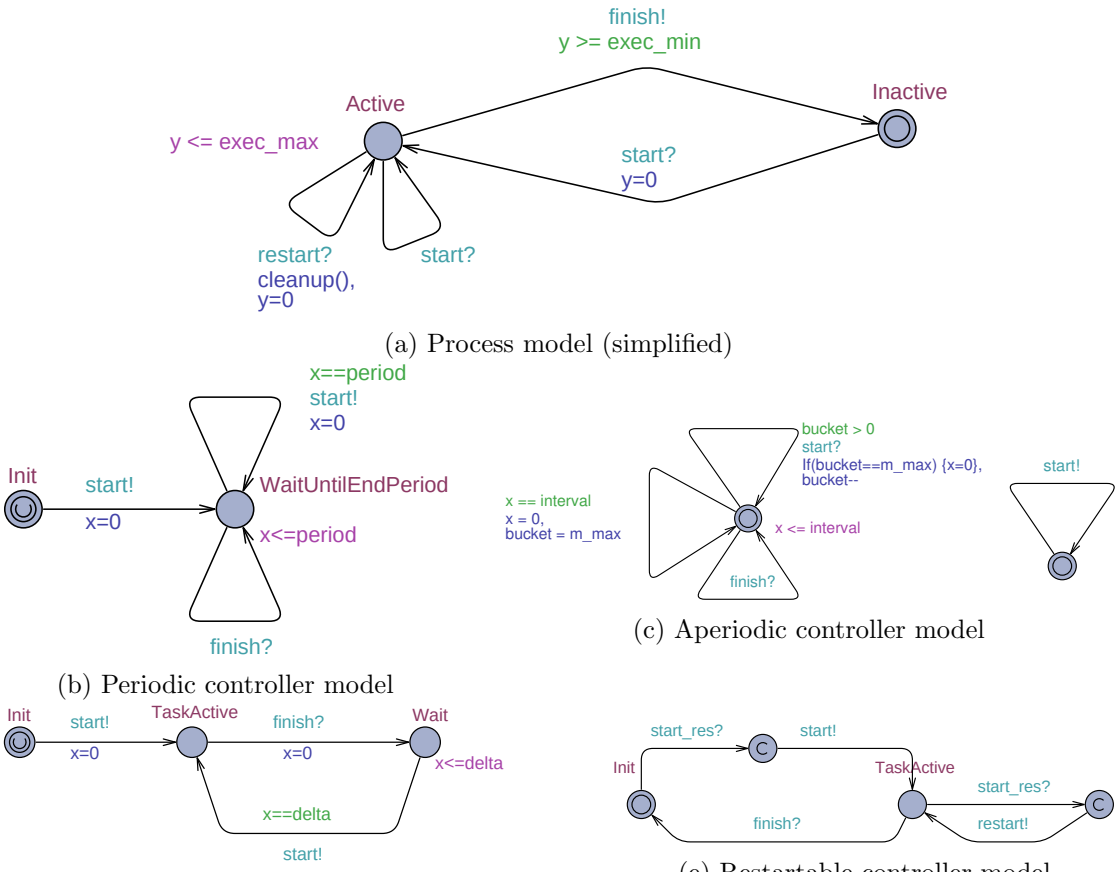
Modeling task activations. Task activations can be of four different types: periodic, aperiodic, pseudo-periodic, and restartable. Each of them is modeled by coupling the process corresponding to the task with a controller of a specific type: periodic (Fig-

ure 2b), aperiodic (Figure 2c), pseudo-periodic (Figure 2d), and restartable (Figure 2e). In the TA models for all four activation patterns, the `start` and `finish` signals are used to activate a process (`start!`: output signal) and get notified upon its completion (`finish?`: input signal), respectively. Variables `x` are clocks, measuring the elapsed time between two successive resets of them (actions `x=0`). Their value is checked within invariants (e.g., `x<=period` in Figure 2b) or transition guards (e.g., `x==period` in Figure 2b). For simplicity, we assume that the first activation of all periodic and pseudo-periodic tasks occurs at time 0 (initial locations denoted by `U`, i.e., urgent).

In the TA of Figure 2c for the aperiodic controller, variable `m` is used to count the process activations within a time interval equal to `interval`. The process activations occur non-deterministically in time, namely their inter-arrival time is not fixed. Also, the number of process activations within `interval` is non-deterministically selected between 0 and `m_max`.

The TA of Figure 2e for the restartable controller employs an additional signal, namely `start_res`. `start_res` is used to notify the controller TA that it must (immediately) activate the corresponding restartable task. In the Flight Management System for example, this information is conveyed by another task of the application. For instance, task TRAJ_{P1} performs some checks and decides whether to activate the restartable task TRAJ_{R1} or not. Therefore, the input signal `start_res`, in this case, is provided by task TRAJ_{P1}. If the signal `start_res` is received while the corresponding restartable task is idle (not executed), then the controller activates it through the output signal `start`. If the restartable task is already being executed, then the controller orders it to immediately stop and restart execution through the output signal `restart`. Note that the task deciding whether a restartable task should be (re)started, here TRAJ_{P1}, needs to know when the restartable task completes its execution, e.g., to evaluate whether the latter has been executing for a time interval larger than a given threshold. We assume that the restartable task communicates with the deciding task over a data channel (mailbox). Before finishing execution, the restartable task inserts a token to this channel to notify its termination to the deciding task and then, it sends the `finish` signal to the restartable controller.

Figure 2a shows a TA model of a process, which is controlled by any of the four previously described controllers. For simplification, the considered process is defined by only one execution phase (no memory accesses) with a known upper and lower bound on execution time, i.e., `exec_max` and `exec_min`, respectively. The process is initially inactive. Once it receives the input signal `start`, it starts executing. Its execution time is measured by clock `y` in location `Active` and it is non-deterministically selected between `exec_min` and `exec_max`. If at any time during execution, the process receives signal `start` (possible for aperiodic tasks), no action is taken, namely the information is neglected. On the contrary, if the process receives signal `restart` (possible for restartable tasks), then the required clean-up operations are performed and clock `y` is reset to measure the elapsed time for the new process execution. Upon completion, the process informs its controller by sending signal `finish` and returns to location `Inactive`.



(a) Process model (simplified)

(b) Periodic controller model

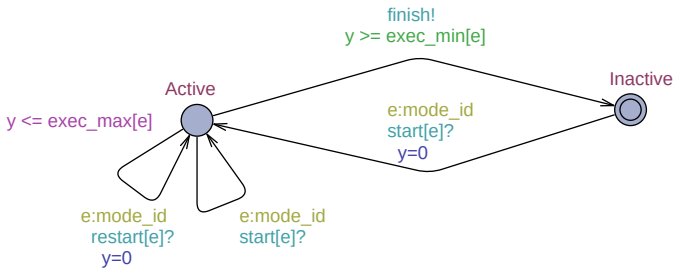
(c) Aperiodic controller model

(d) Pseudo-periodic controller model

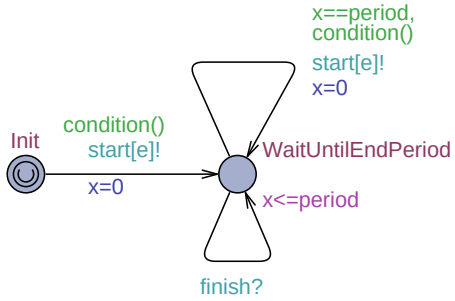
(e) Restartable controller model

Figure 2: TA models for a process and the four controller types

Data sensitive processing that application tasks can exhibit. In the localization task group, depending on the amount of sensor data available, the localization tasks may work in different modes, i.e. tasks perform different functionalities, which exhibit different execution times and memory access parameters. This can be modeled by a special activation type, namely periodic with modes. A TA for the corresponding controller is shown in Figure 3b. Note that by checking a condition (guard `condition()`), the controller specifies the mode `e` at which the controlled process must be executed. Checking the condition may involve synchronization with the TA of other processes of the application and/or their controllers. That is, a task or a controller can send a signal to the controller TA of Figure 3b and depending on the signal, the appropriate execution mode of the controlled processed is selected. The selected mode `e` is then sent along with signal `start`. The controlled process (modeled in Figure 3a) receives this information and is executed according to the execution time (and access) parameters of the respective mode. In the depicted TA, `mode_id` denotes the set of mode identifiers, namely $\{1, \ldots, \texttt{modes}\}$.



(a) Process with modes model (simplified)



(b) Periodic with modes controller model

Figure 3: Data sensitive behaviors - Modified TA for process and periodic controller

2.1.3 <data_channel>

The <data_channel> element is used to describe a data communication channel between processes. It contains 5 attributes: `name`, `type`, `size`, `length`, and `lifespan`. For the FMS application, the `type` can be `blackboard` (double buffer) or `mailbox` (FIFO queue). A `mailbox` connects one reading process with one writing process. A `blackboard` connects one writing process with one or more reading processes. The `lifespan` attribute specifies the number of consecutive periods that an input datum will still be considered valid without being updated by its writer. Once an input datum expires its lifespan, it will be marked as invalid in the data channel. Note that, in the FMS application, all writer tasks to blackboards are periodic. The `lifespan` attribute is only valid for the `blackboard` element. Each <data_channel> must contain exactly one input and one output data <port> elements if it is of mailbox type, and at least two data <port> elements of which exactly one is input and at least one is output if it is of blackboard type. The `size` attribute specifies the size in bytes of a single entry in the FIFO for a `mailbox`, and the size of the buffer in bytes for a <blackboard>. The `length` attribute is specified only for the `mailbox` element, and it defines the length of the FIFO (capacity in terms of entries). Data channels transport data in finite non-zero times.

Listing 4 shows an example XML specification of a blackboard data channel.

```
01 <data_channel name="bb2" type="blackboard" size="1024" lifespan="3">
02   <port type="in_data" name="p1"/>
03   <port type="out_data" name="p2"/>
04   <port type="out_data" name="p3"/>
```

```
05 </data_channel>
```

Listing 4: XML specification of one data channel element Blackboard

2.1.4 <control_channel>

We explicitly distinguish between a <data_channel> and a <control_channel> to mark the difference between control- and data-dependencies. The <control_channel> element specifies a communication channel between a process and a controller. It has one attribute, its `name`. Each <control_channel> specifies two event ports belonging to a controller and a process, i.e., an event input port of a process and an event output port of a controller, or an event output port of a process and an event input port of a controller. The order in which the `port` elements appears defines the direction of the channel (first port of type `out_event`, second port of type `in_event`). The events that are associated with the two ports must be the same. Note, that we allow control channel connections only between a process and a controller. A process may provide output events to more than one controller (through different control channels), and a controller can provide output events to exactly one process (through one control channel). Control channels transport events between a controller and a process in zero time.

Listing 5 shows an example XML specification of a control channel.

```
01 <control_channel name="cc6">
02   <port name="p8"/>
03   <port name="p15"/>
04 </control_channel>
```

Listing 5: XML specification of one control channel element

2.1.5 <global>

Applications as considered in CERTAINTY have global constraints and requirements, i.e. requirements that can not be associated to a single process such as its deadline. The <global> element allows to specify these global requirements that concern several processes. For the Flight Management System specification, we consider the following three requirements:

- Cycle requirement: The FMS application has an explicit cycle requirement, i.e. Requirement 28 in D1.1: The whole Flight Management System software should be able to operate with a basic cycle frequency of 15Hz.
- End-to-end delay requirement: It specifies an upper bound on the time delay of data propagation in a chain of processes. The time delay is defined from the activation time of the first process in the chain to the finish time of the last process. For example, in the Flight Management System, after any lateral or vertical flightplan modifications, the new vertical guidance information should be produced within 15s, i.e., after task FPLN_{A6} has finished modifying the active flightplan, task

$TRAJ_{R1}$ should produce new trajectory information based on these modifications which trajectory information is then processed by the guidance task $GUID_{C2}$.

- Precedence constraint: It specifies the required order of execution in a chain of (periodic) processes with equal periods. It implies that the first process of the chain cannot be executed after the second, etc., within a common period of theirs. Of course, other processes may be executed in between the dependent processes, but it must be enforced that every process in the chain is triggered after its predecessor has finished execution. Precedence constraints can be defined only for task chains with non-increasing criticality levels. Namely, a process can have a predecessor of higher or equal criticality, but not of lower criticality. Additionally, each process of an application can be part of at maximum one precedence chain.

The <global> element may contain not more than one child element <cycle>. This child element has one compulsory attribute `value`, which specifies the global cycle constraint in Hertz. It may also contain zero or more <delay> elements. Each of them has three compulsory attributes: `name`, `chain`, and `value`. The `chain` attribute gives an ordered list of names of processes that are affected by this delay constraint, and the `value` attribute specifies the value of the delay constraint in seconds. Finally, the <global> element may contain zero or more <precedence> elements. Each of them has two compulsory attributes: `name` and `chain`. The `chain` attributes defines the dependent tasks in their required relative order of execution.

Listing 6 shows an example XML specification of global constraints.

```
01 <global name="g1">
02   <cycle value="15"/>
03   <delay name="vert_guide" chain="FPLN_A6,TRAJ_R1,GUID_C2" value="15"/>
04   <precedence name="prec" chain="FPLN_A6,TRAJ_R1"/>
05 </global>
```

Listing 6: XML specification of one global element

2.1.6 <port>

As already discussed in Sections 2.1.1 and 2.1.2, we distinguish between data input and data output ports as well as between event input and event output ports. The element <port> can have two attributes, `name` and `type`. The `name` attribute is always compulsory and it must be **unique** within an <app> element. The values of the `type` attribute can be: `in_data` for data input ports, `out_data` for data output ports, `in_event` for event input ports, and `out_event` for event output ports. If the port is an event port, it has at least one child element <event> with attribute `name` that specifies the name of the event that is supported by this port. When ports are used in the <control_channel> or <connection> elements, the `type` attribute and the child elements <event> are omitted to avoid redundancy.

Listing 7 shows an example XML specification of a port.

```
01 <port type="in_event" name="p5">
02   <event name="start" />
```

```
03 </port>
```

Listing 7: XML specification of one port element

2.1.7 <connection>

The <connection> element is used to establish the connection between a process and a data channel. Each <connection> element has one attribute `name` and contains exactly two ports from the two elements it connects. One port is a data input port, and the other port is a data output port.

Listing 8 shows an example XML specification of a connection.

```
01 <connection name="c1">
02   <port name="p1" />
03   <port name="p16" />
04 </connection>
```

Listing 8: XML description of one connection element

3 Architecture specification

For the specification of a multicore hardware platform, we maintain a high-level abstraction of the platform architecture, where the computing resources (processors) and communication resources (shared bus, network on chip) are described. Inter-core communication is assumed to be achieved through shared buses within a cluster or a network-on-chip (NoC) among different clusters. A simple example of an architecture that can be modeled in DOL is depicted in Figure 4. There, Processor 1 and Processor 2 belong to the same cluster, whereas Processor 3 belongs to a different cluster.

The architecture specification consists of the following basic elements: <port>, <processor>, <shared>, <noc>, and <link>, which are described in the following.

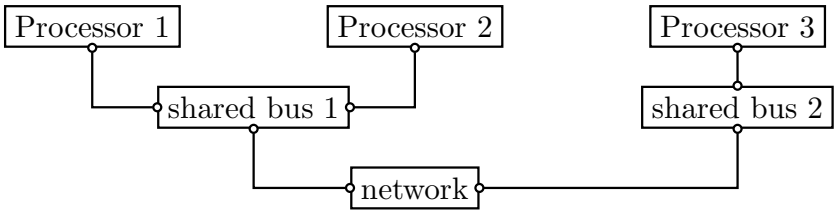


Figure 4: Example Architecture

3.1 Elements

3.1.1 <port>

The <port> element is used to specify a port of a resource (<processor>, <shared> or <noc>). It has the mandatory attribute `name` which specifies the name of the port.

The port `name` needs to be **unique** within <processor>, <shared> or <noc>, but not within the high-level <architecture> element. All ports are bidirectional.

Listing 9 shows an example XML specification of a platform resource port.

```
01 <port name="port_1"/>
```

Listing 9: XML description of one resource port element

3.1.2 <processor>

The <processor> element is used to specify a computing resource. Its attribute `name` defines the name of the processor. A <processor> element may contain one or more <port> elements to specify its connection with other components of the architecture, i.e., with <shared> elements. It contains exactly one element <frequency> with an attribute `value` which specifies the speed of this processor in Hz.

Listing 10 shows an example XML specification of a processor resource at 1 GHz.

```
01 <processor name="processor_1">
02   <port name="port_1"/>
03   <frequency value="1000000000"/>
04 </processor>
```

Listing 10: XML description of one processor element

3.1.3 <shared>

The <shared> element specifies a local shared bus, in particular the interconnection inside a cluster of the platform. It has one attribute, `name`, which defines the name of the shared bus. The <shared> element may contain a list of <port> elements to specify its connection with other components of the architecture, i.e., with <processor> or <noc> elements. Furthermore, it may contain a list of <configuration> elements. A <configuration> element has two attributes, `name` and `value`, defining the name of a bus parameter and its corresponding value. For example, one or more <configuration> elements can be used to specify supported arbitration policies of the specified bus. In this case, the `name` of the <configuration> is assigned to `arbitration` and `value` can be assigned to any of the strings `fifo`, `roundrobin`, `fixedpriority`, `tdma`. Moreover, <shared> can contain one child element <bandwidth> which has a compulsory attribute `value` that specifies the bandwidth of the bus in terms of number of memory accesses that can be served per second by the bus. <shared> contains also exactly one child element <latency> with compulsory attribute `value` which specifies the latency (bound) in seconds for serving a single memory access by the bus.

Listing 11 shows an example XML specification of a shared bus resource.

```
01 <shared name="bus_1">
02   <port name="port_4"/>
03   <port name="port_5"/>
```

```
04   <port name="port_6"/>
05   <bandwidth value="1000000"/>
06   <latency value="0.00001"/>
07   <configuration name="arbitration" value="roundrobin" />
08 </shared>
```

Listing 11: XML description of one shared bus

3.1.4 <noc>

The <noc> element specifies a network on chip and has one attribute, `name`, which defines the name of the network. The <noc> element may contain a list of <port> elements to specify its connection with local shared buses, i.e., with <shared> elements. Similarly with the <shared> element, the <noc> contains the elements: <bandwidth> and <latency>.

Listing 12 shows an example XML specification of a network-on-chip resource.

```
01 <noc name="network_1" >
02   <port name="port_9"/>
03   <port name="port_10"/>
04   <bandwidth value="500000"/>
05   <latency value="0.02"/>
06 </noc>
```

Listing 12: XML description of one network-on-chip element

3.1.5 <link>

The <link> element is used to specify the connection between two elements through their respective ports. In particular, it can either connect a <processor> element with a <shared> element or a <shared> element with a <noc> element. The two elements which are connected by the link are specified using a pair of <end_point> elements. The <end_point> specifies the name of component and has a child element <port> to indicate the port name to which the link is connected. Note that links are bidirectional and the direct link between two <processor> elements is not allowed.

Listing 13 shows an example XML specification of a link.

```
01 <link name="link_1">
02   <end_point_1="processor_1">
03     <port name="port_1"/>
04   </end_point_1>
05   <end_point_2="bus_1">
06     <port name="port_4"/>
07   </end_point_2>
08 </link>
```

Listing 13: XML description of one link element

3.2 Example

To illustrate the use of the above elements, we provide the complete XML specification of the example architecture of Figure 4 in Listing 14.

```
01 <architecture name="example architecture">
02
03   <!-- Processors -->
04   <processor name="processor_1">
05     <port name="port1" />
06     <frequency value="1000000000"/>
07   </processor>
08   <processor name="processor_2">
09     <port name="port2" />
10     <frequency value="1500000000"/>
11   </processor>
12   <processor name="processor_3">
13     <port name="port3" />
14     <frequency value="2000000000"/>
15   </processor>
16
17   <!-- Shared buses-->
18   <shared name="shared_bus_1">
19     <port name="port4" />
20     <port name="port5" />
21     <port name="port6" />
22     <bandwidth value="1000000"/>
23     <latency value="0.002"/>
24     <configuration name="arbitration" value="roundrobin"/>
25   </shared>
26   <shared name="shared_bus_2">
27     <port name="port7" />
28     <port name="port8" />
29     <bandwidth value="800000"/>
30     <latency value="0.005"/>
31     <configuration name="arbitration" value="roundrobin"/>
32   </shared>
33
34   <!-- NoC-->
35   <noc name="network">
36     <port name="port9" />
37     <port name="port10" />
38     <bandwidth value="500000"/>
39     <latency value="0.02"/>
40   </noc>
41
42   <!-- Links -->
43   <link name="link_1">
44     <end_point_1 name="processor_1">
45       <port name="port1" />
46     </end_point_1>
47     <end_point_2 name="shared_bus_1">
48       <port name="port4" />
49     </end_point_2>
50   </link>
51   <link name="link_2">
52     <end_point_1 name="processor_2">
53       <port name="port2" />
54     </end_point_1>
55     <end_point_2 name="shared_bus_1">
```

```
56        <port name="port5" />
57      </end_point_2>
58    </link>
59    <link name="link_3">
60      <end_point_1 name="processor_3">
61        <port name="port3" />
62      </end_point_1>
63      <end_point_2 name="shared_bus_2">
64        <port name="port7" />
65      </end_point_2>
66    </link>
67    <link name="link_4">
68      <end_point_1 name="shared_bus_1">
69        <port name="port6" />
70      </end_point_1>
71      <end_point_2 name="network">
72        <port name="port9" />
73      </end_point_2>
74    </link>
75    <link name="link_5">
76      <end_point_1 name="shared_bus_2">
77        <port name="port8" />
78      </end_point_1>
79      <end_point_2 name="network">
80        <port name="port10" />
81      </end_point_2>
82    </link>
83  </architecture>
```

Listing 14: XML specification of architecture example

4 Mapping specification

The mapping defines where and how the tasks of the FMS application are executed on a multicore platform. Mapping needs to be done in the spatial domain, which is referred to as binding, and in the temporal domain, which is referred to as scheduling. In DOL-Critical, the binding specification defines a mapping of application elements on the platform resources. The scheduling specification defines the scheduling policy on each resource and the corresponding parameters, e.g., a time-triggered scheme and the associated slot lengths, fixed priority scheduling and the associated priorities or static scheduling and the associated ordering.

4.1 Elements

4.1.1 <binding>

The <binding> element serves for the definition of bindings. It has two attributes: `name`, which is used to specify an identifier (name) for the binding and `type`, which in DOL-Critical is always set to `computation`. The <binding> element has two child elements, <process> and <processor>, specifying which process (identifier as defined in the application specification) is bound to which processor (identifier as defined in

the architecture specification). The listing below shows a <binding> of `processA` to `processorA` to illustrate the usage of these elements.

```
01 <binding name="process_binding" type="computation">
02   <process name="processA"/>
03   <processor name="processorA"/>
04 </binding/>
```

4.1.2 <schedule>

The <schedule> element is used to define the schedule on the processors of the multicore platform. This element has two attributes, `name` and `type`. The `name` attribute is used to specify an identifier for the schedule. The `type` attribute which can for example be one of `tts`, `fixedpriority`, `fifo`, `roundrobin`, `static` or `tdma` defines the scheduling policy. The <schedule> element has different child elements depending on the scheduling policy.

Hereafter, we focus on the case of `tts` scheduling, namely Time-Triggered scheduling with Synchronisation points (TTS) for mixed-criticality applications, and we present the corresponding child elements.

TTS is defined by a scheduling cycle with a given period and static time frames of given lengths. The frames of the scheduling cycle start and stop synchronously on all cores. Each time frame is divided into sub-frames, where only tasks of the same criticality level are allowed to be executed. The sub-frames in each frame correspond to criticality levels of decreasing order, namely the first sub-frame corresponds to the highest criticality level and the last one corresponds to the lowest criticality level. The sub-frames, also, start and stop synchronously on all cores. Their lengths are dynamic, not fixed. Synchronisation is achieved through a barrier mechanism (*synchronisation points*). Particularly, each sub-frame in a frame starts when all tasks of the previous sub-frame have completed their execution on all cores. The first sub-frame starts upon start of the frame. The tasks in each sub-frame are executed according to a static schedule. An example of a TTS schedule is depicted in Figure 5. The tasks of higher criticality level (HI, DAL-B) are marked with orange color and the tasks of lower criticality level (LO, DAL-C) are marked with green color. The binding and schedule specification for the example of Figure 5 is given in Listing 15.

The <schedule> element for TTS has the following child elements:

- <cycle> defines the period of the TTS scheduling cycle through attribute `length`.
- <frame> defines a static frame of the TTS scheduling cycle. It has two attributes, `name` and `length`, which define an identifier for the frame and its fixed length in seconds, respectively. A <frame> element contains one or more <barrier> child elements, which correspond to its sub-frames and define the worst-case lengths of them under different criticality level assumptions. A <barrier> element contains the attributes `criticality`, specifying the criticality of the tasks that are scheduled in the corresponding TTS sub-frame, `scenario`, specifying the criticality level assumption (relevant for the assumed worst-case execution times and memory accesses of the tasks),

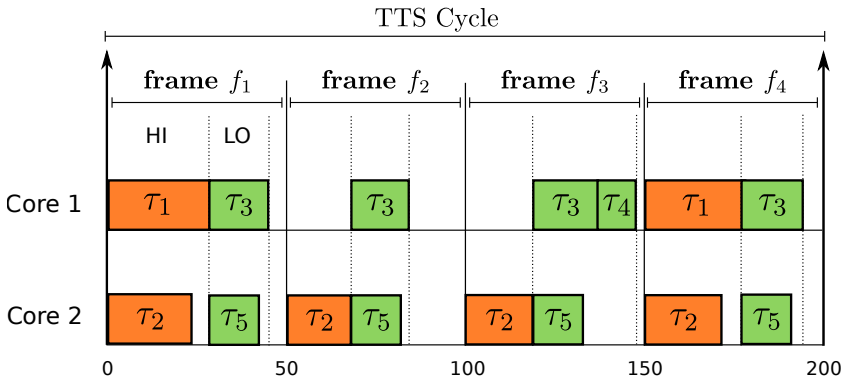


Figure 5: Example Specification

and `time`, specifying the worst-case sub-frame length (in seconds) under the above assumption. The last parameter is estimated during mapping optimization as a product of timing/interference analysis.

- <processor> defines the schedule on a particular processor (core). The `name` attribute specifies which <processor> (identifier as defined in the architecture specification) is addressed. The <processor> element has the child element <container>. The <container> element specifies a sub-frame, namely a non-fixed time interval where tasks of the same criticality level are executed in parallel.

  The identifier of the corresponding sub-frame is defined by the `name` attribute. The <container> element has, furthermore, one child element, <configuration> with the attributes `name`, always assigned to `frame` for TTS, `value`, which is assigned to the identifier of the containing TTS frame as defined earlier in the respective <frame> element, and `criticality`, which indicates the criticality level of the tasks that are scheduled in the sub-frame. <container> may also contain zero or more <process> child elements. These define which tasks are scheduled within the sub-frame. The `name` attribute specifies the corresponding task identifier (as defined in the application specification). If more than one <process> elements exist within a <container>, their order of appearance corresponds to their relative order of execution in the sub-frame. E.g., in the `f32_c1` container of Listing 15, task 4 will be triggered after task 3 completes its execution. If no tasks are scheduled in a sub-frame (e.g., cases `f21_c1, f31_c1` in Listing 15), then no instances of the <process> element exist in the corresponding <container>.

```
01 <mapping  name="Example TTS mapping">
02   <!-- task bindings -->
03   <binding name="task1" type="computation">
04     <process name="task1" />
05     <processor name="core1" />
06   </binding>
07   <binding name="task2" type="computation">
08     <process name="task2" />
09     <processor name="core2" />
```

```
10  </binding>
11  <binding name="task3" type="computation">
12    <process name="task3" />
13    <processor name="core1" />
14  </binding>
15  <binding name="task4" type="computation">
16    <process name="task4" />
17    <processor name="core1" />
18  </binding>
19  <binding name="task5" type="computation">
20    <process name="task5" />
21    <processor name="core2" />
22  </binding>
23
24   <!-- schedule tables -->
25  <schedule name="example-tts" type="tts">
26    <cycle length="0.2"/>
27    <frame name="f1" length="0.05">
28       <barrier criticality="B" scenario="B" time="0.04"/>
29       <barrier criticality="B" scenario="C" time="0.03"/>
30       <barrier criticality="C" scenario="B" time="0.00"/>
31       <barrier criticality="C" scenario="C" time="0.01"/>
32    </frame>
33    <frame name="f2" length="0.05">
34       <barrier criticality="B" scenario="B" time="0.01"/>
35       <barrier criticality="B" scenario="C" time="0.005"/>
36       <barrier criticality="C" scenario="B" time="0.00"/>
37       <barrier criticality="C" scenario="C" time="0.01"/>
38    </frame>
39    <frame name="f3" length="0.05">
40       <barrier criticality="B" scenario="B" time="0.01"/>
41       <barrier criticality="B" scenario="C" time="0.005"/>
42       <barrier criticality="C" scenario="B" time="0.00"/>
43       <barrier criticality="C" scenario="C" time="0.035"/>
44    </frame>
45    <frame name="f4" length="0.05">
46       <barrier criticality="B" scenario="B" time="0.04"/>
47       <barrier criticality="B" scenario="C" time="0.03"/>
48       <barrier criticality="C" scenario="B" time="0.00"/>
49       <barrier criticality="C" scenario="C" time="0.01"/>
50    </frame>
51
52    <processor name="core_1">
53      <container name="f11_c1">
54        <configuration name="frame" value="f1" criticality="B"/>
55        <process name="task1"/>
56      </container>
57      <container name="f12_c1">
58        <configuration name="frame" value="f1" criticality="C"/>
59        <process name="task3"/>
60      </container>
61      <container name="f21_c1">
62        <configuration name="frame" value="f2" criticality="B"/>
63      </container>
64      <container name="f22_c1">
65        <configuration name="frame" value="f2" criticality="C"/>
66        <process name="task3"/>
67      </container>
68      <container name="f31_c1">
69        <configuration name="frame" value="f3" criticality="B"/>
70      </container>
71      <container name="f32_c1">
```

```
72          <configuration name="frame" value="f3" criticality="C"/>
73          <process name="task3"/>
74          <process name="task4"/>
75        </container>
76        <container name="f41_c1">
77          <configuration name="frame" value="f4" criticality="B"/>
78          <process name="task1"/>
79        </container>
80        <container name="f42_c1">
81          <configuration name="frame" value="f4" criticality="C"/>
82          <process name="task3"/>
83        </container>
84      </processor>
85
86      <processor name="core_2">
87        <container name="f11_c2">
88          <configuration name="frame" value="f1" criticality="B"/>
89          <process name="task2"/>
90        </container>
91        <container name="f12_c2">
92          <configuration name="frame" value="f1" criticality="C"/>
93          <process name="task5"/>
94        </container>
95        ...
96      </processor>
97    </schedule>
98 </mapping>
```

Listing 15: XML description of the mapping of Figure 5.

The child elements of the <schedule> element are defined accordingly for other scheduling policies. For instance, the <cycle> and <frame> elements can be skipped for non time-triggered policies. The <container> element can be adapted for event-driven scheduling policies, e.g., to specify the task subsets with the same priority in a priority-driven scheme. To illustrate this case, Listing 16 shows a possible schedule specification of the tasks of Figure 5 on `core_1` and `core_2` with given execution priorities. The <configuration> child element defines now the priority for the corresponding <container>. If more than one instances of the <process> child element exist in the same <container>, the corresponding tasks are assumed to be of equal priority.

```
01 <mapping  name="Example FP mapping">
02  <schedule name="example-fp" type="fixedpriority">
03      <processor name="core_1">
04          <container name="prio1">
05             <configuration name="priority" value="1"/>
06             <process name="task1"/>
07          </container>
08          <container name="prio2">
09             <configuration name="priority" value="2"/>
10             <process name="task3"/>
11          </container>
12          <container name="prio3">
13             <configuration name="priority" value="3"/>
14             <process name="task4"/>
15          </container>
16       </processor>
17      <processor name="core_2">
18          <container name="prio1">
19             <configuration name="priority" value="1"/>
```

```
20            <process name="task5"/>
21          </container>
22          <container name="prio2">
23            <configuration name="priority" value="2"/>
24            <process name="task2"/>
25          </container>
26        </processor>
27      </schedule>
```

Listing 16: XML description of a mapping with fixed-priority scheduling

References

[1] RTCA/DO-178B, Software Considerations in Airborne Systems and Equipment Certification, 1992.

[2] G. Behrmann, A. David, K. G. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks. Uppaal 4.0. In *Proceedings of the 3rd international conference on the Quantitative Evaluation of Systems*, QEST '06, pages 125–126, Washington, DC, USA, 2006. IEEE Computer Society.

[3] R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha. Coscheduling of cpu and i/o transactions in cots-based embedded systems. In *Proceedings of the 2008 Real-Time Systems Symposium*, RTSS '08, pages 221–231, Washington, DC, USA, 2008. IEEE Computer Society.