ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Stefan Drašković

Functional Simulation of a Flight Management System

Semester Thesis SA-2013-63
October 2013 to January 2014

Tutor: Prof. Dr. Lothar Thiele
Supervisors: Pengcheng Huang, Georgia Giannopoulou, Lars Schor

Abstract

This thesis describes the development of a functional simulator used for the Flight Management System (FMS) application, a case-study of a mixed-critical real-time application developed by Thales [11] and targeted in the Certainty project [1]. FMS, and similar applications, consist of real-time tasks with various criticality levels and activation patterns. Tasks communicate through either mailbox (FIFO) or blackboard (shared variable) data channels. The language used to describe the FMS for simulation is the DOL-C language [5]. DOL-C consists of application code and a XML application specification. The simulator developed in this thesis is based on the existing DAL tool-chain [7, 6], in such a way that input applications are translated into a form appropriate for the existing simulator. The way applications in DOL-C are simulated, and the way the existing DAL simulator is re-used, is described and shown on benchmark applications that resemble the FMS.

Contents

List of Figures

Chapter 1

Introduction

As a reference application in this thesis, we consider the Flight Management System (FMS), which is an industrial application developed by Thales [11] and is considered in the Certainty project [1]. Its purpose is to provide embedded control systems for aircraft navigation sensors, computer-based flight planning, fuel management, radio navigation management and geographical information [4]. During operation, the FMS needs to monitor the environment and itself, and trigger certain functionalities accordingly, while fulfilling all real-time requirements of its tasks.

1.1 Motivation

The motivation behind this thesis is to design, implement and evaluate a functional simulator for mixed-critical real-time applications, like the FMS. A functional simulator is necessary, since having feedback about application design early, before deployment on hardware, reduces overall design time and cost.

The FMS contains various applications, which further contain various tasks. Each of these tasks has an activation pattern, depending on its function. Five different activation patterns exist in the FMS: periodic, periodic with mode, aperiodic, pseudo-periodic and restartable. Periodic tasks are activated with a predefined period. Periodic with mode tasks are also periodic, but may be activated in different execution modes, depending on the availability of input data. Pseudo-periodic tasks are activated a fixed time interval after the completion of their last execution. Restartable tasks are restarted if they are executed for more than a given time threshold, and aperiodic tasks are activated asynchronously. Communication between tasks is possible via data and control channels. Control channels are used for example to signal that a task starts execution, while data channels transfer data between tasks using different types of buffers. For details, please refer to [4, 3, 5].

Simulating at a functional level, while respecting all activation and communication patterns, the simulator validates the design with respect to its intended functionality. Comparing different design solutions through simulation enables designers to make right choices. Potential behaviour bugs, discovered during simulation, can be localized for easier correction. As a result of this, a more efficient design of real-time applications is made possible.

The simulator of this thesis is based on the existing DAL tool-chain [7, 6], where a different simulator already exists. Knowledge in the TEC group on the modelling of real-time applications is also used.

For the evaluation of the functional simulator, case studies that resemble the FMS were used. These FMS-style applications are written in the DOL-C language, described in [5]. Written in this way, an application consists of a XML specification and an application code.

1.2 Contribution

The overall contribution is the design, implementation and evaluation of the functional simulator for FMS like applications.
Firstly, as the FMS application is real-time, with tasks being activated at different times, a scheduler is implemented in the simulator. The scheduler acts as a synchronising mechanism between tasks, so tasks with various activation patterns are executed in a correct order.

The second contribution of this thesis is an extension of the DAL tool by adding a new layer to it, which translates an application written in DOL-C to the DAL language. Both the DOL-C XML specification and application code of an application are translated, the first into DAL XML specification and the second into DAL application code.

Finally, three case-studies were used to evaluate the correctness and determinism of the functional simulator. The corresponding applications exhibit all possible activation and communication patterns that appear in the FMS application.

1.3 Organization

Chapter 2 describes the background of this work and related work. The FMS application is described, with key aspects important for simulation highlighted. The language in which applications are written, DOL-C, is described, as well as the DAL tool and language to which these applications are translated.

The design of the functional simulator is presented in chapter 3. An overview of the design of the functional simulator is given first, before elements of the extension of the DAL tool are explained in detail. Included is the way applications written in DOL-C are translated to DAL.

In chapter 4, details about the implementation of the DOL-C framework, the addition to the existing DAL simulator that enables functional simulation, are described.

Finally chapter 5 concludes with the evaluation of the simulator. Characteristic aspects are presented through simulation results. Three example applications demonstrate behaviour of mailbox and blackboard type data channels, non-determinism created because of tasks activated at the same time, and changing modes of activation for periodic with mode tasks. In these examples, all activation patterns and communication types are present.

Chapter 2

Background and Related Work

This chapter describes the background of this thesis and related work on the subject. In particular, first introduced is the FMS application [3], which is used in this thesis as a case-study for mixed-critical real-time applications. Its elements, i.e. tasks and communication channels, are introduced and key attributes such as activation patterns and criticality levels are explained. Then presented is the modelling language used in this thesis for FMS like applications, DOL-C [5], and it is demonstrated with an example. Finally, the Distributed Application Layer (DAL) platform [10], on top of which the designed simulator is integrated, is introduced. In particular, the specification language used by DAL is presented, and the challenges to adopt DAL to simulate FMS like applications.

2.1 Flight Management System

The functional simulator developed in this thesis is designed for the Flight Management System application. The Flight Management System, or FMS, is developed by Thales [11] and is targeted by the Certainty project [1]. The FMS application is mixed-critical and real-time. The FMS is used in modern avionics to relieve workload off the crew by providing centralized control for systems like the aircraft navigation sensors, computer based flight planning, fuel management, radio navigation management and geographical situation information [4]. Figure 2.1, taken from [1], shows the FMS application's sensor and localization group. The sensor group is responsible for collecting data from sensor systems, such as anemometer and barometer measurements or data from GPS. The localization group uses data from the sensor group to compute the location of the aircraft.

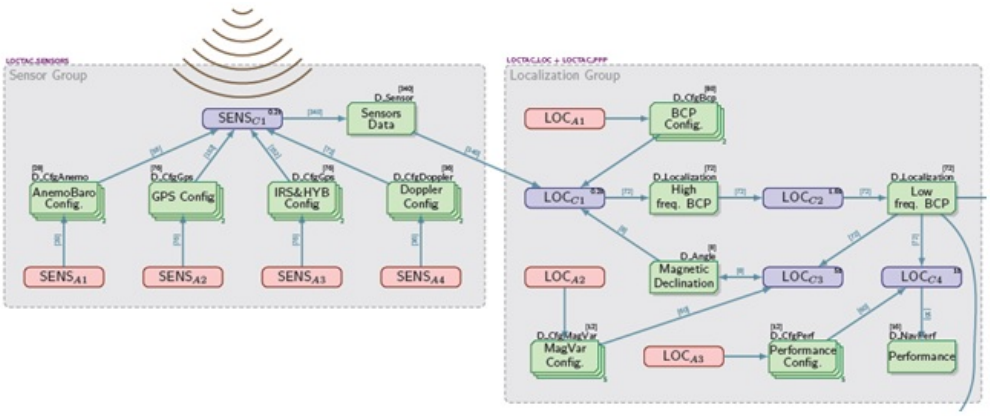


Figure 2.1: Subset of the Flight Management System

The FMS application contains one or more tasks, marked with red and blue on Figure 2.1. The tasks have different criticality levels and different activation patterns. These characteristics

make FMS applications not compliant with the Kahn Process Network (KPN) model (see Section 2.3 and [13]), where activations are data-driven, and there is one criticality level. Criticality levels express how important a task is in terms of safety, and it is expressed with design assurance levels A to E, as specified in the DO-178B protocol [12], with A being the highest criticality level. Tasks of a lower criticality level can only be executed if tasks of higher levels have their execution guaranteed. Even though an important aspect in the FMS, criticality levels do not impact execution in the functional simulator of this thesis.

Activation patterns, on the other hand, are simulated. An activation pattern describes when a task is executed, and five patterns are observed in the FMS: periodic, periodic with mode, aperiodic, pseudo-periodic and restartable. Periodic, periodic with mode and restartable tasks are described with a period. Every time a period passes, the corresponding task is activated. The difference is that, for periodic with mode, there is also a mode in which the task is started in, and for restartable tasks there is a time when the task is restarted, i.e. if its execution is not finished. Pseudo-periodic tasks are characterized by a parameter delta, which denotes the time interval between the completion of a task execution and the next activation. Aperiodic tasks are characterized by the maximum number of task activations within a time interval defined.

For the functional simulation, all tasks are assumed to execute in zero time. This makes the pseudo-periodic tasks behave like they are periodic. Restartable tasks can not be simulated, as their execution time influences the execution result. For more details, see Chapter 3.

In the FMS, tasks communicate through data and control channels. The use of control channels is tightly connected to activation patterns, as they are used to notify a task that it should start execution, or to signal in which mode a task should execute. Data channels are depicted in green on Figure 2.1. They transfer data using one of two mechanisms: mailbox or blackboard. Mailboxes are FIFO buffers, and have one reading and one writing task. Blackboard type channels are similar to shared variables, in the way that the same data can be read multiple times, once set. Blackboards are implemented as double buffers, meaning that data is written to one buffer and read from the other, and these buffers swap after every data write. A blackboard connects one writing process to one or more reading processes [5].

Reading from data channels is always non-blocking, i.e. a task is not stopped when reading an empty data channel. For mailboxes, as they are FIFO buffers, reading is destructive, while for blackboards reading is non-destructive. Writing to data channels is also non-blocking. Writing to a mailbox is non-overwriting, while writing to blackboards is overwriting.

2.2 DOL-C

DOL-C is a specification language used to model applications like the FMS. DOL-C is an extension of the DOL language, which is used to model streaming applications.

DOL-C is made out of application code and an application specification. Application code is used for the functional description of tasks, while the application specification contains tasks and network descriptions. Application code is written in C or C++ with DOL-C coding rules, while the application specification is described using XML.

Functional behaviour of a task, described by the application code, is split into three phases [14]: initialization, execution and termination, and these are defined in three corresponding functions: *init()*, *fire()* and *terminate()*.

- The function *init()* of a task is called once only, to initialize it before running.
- The *fire(int mode))* function is called repeatedly, every time the task is activated according to its activation pattern.
- The function *terminate()* performs clean-up when the task is not to be activated again.

Reading and writing to data channels uses the structure *DOLCData*. Only data of this type can be transferred via a data channel. This data structure has a validity bit, a pointer to the actual data, and the size of the data. The macro function *DOLCDataVar(type, name)* is used to initialize a *DOLCData* variable. Reading and writing of data is realized using functions *DOLC_read* and *DOLC_write*, as described in [14].

An example of DOL-C application code is given in Figure 2.2. Here, the three functions for a task named *aperiodic1* are presented: *aperiodic1_init()*, *aperiodic1_fire()* and *aperiodic1_terminate()*.

```
void aperiodic1_init(DOLCProcess *p) { ... }
void aperiodic1_fire(DOLCProcess *p, int mode) {
    switch (mode) {
    case MODE_NORMAL: {
        ...
        // Writing to a data channel
        DOLCDataVar(int, writeData);
        writeData.valid = true;
        *((int *)writeData.ptr) = p->local->index;
        int r = DOLC_write((void*) PORT_OUT, &(writeData), 1, p);

        ...
        break;
    }
    default: break;
    }
}
void aperiodic1_terminate(DOLCProcess *p) { ... }
```

Figure 2.2: DOL-C C application code

DOL-C specification contains the network description and describes fully each task, i.e., its activation pattern, criticality level, best-case and worst-case execution time, etc. A task inside an application specification in DOL-C is represented as two elements, a process and a controller. The process and controller are connected with two control channels. This is consistent with the Certainty Application Model [8], where a task is represented with two distinct components: a functional and a state component.

An example application with two tasks is presented in Figure 2.3. It has one aperiodic task, *Aperiodic_1*, marked red, that sends data to a periodic task *Periodic_1*, marked green. They communicate through a data channel *A_to_P*, marked yellow. Control channels in this figure are marked with an arrow with a circle on it, and they connect processes to their controllers. Connections, auxiliary elements that connect data channels and processes, are marked with an arrow with a triangle. When the aperiodic task activates at a random time, it writes a single piece of data to the data channel. Similarly, every time the periodic task activates, it reads a piece of data. If the data channel is empty, it reads data that is labelled as invalid. If not empty, the oldest data is read.

The XML specification of a process is demonstrated in Figure 2.4, where the *Aperiodic_1* process is described. XML specification of both controllers for *Aperiodic_1* and *Periodic_1* is shown in Figure 2.5, and different specifications for activation patterns can be observed. Finally, XML specification of the data channel and a control channel is given in Figure 2.6.

As seen in Figures 2.4 and 2.5, the process part contains information like the criticality, and execution and the minimal and maximal memory access times, and has a reference to the application code. The controller part has information about activation patterns. The *process* is the functional component of a task. It has the *source file*, *superblock* and *port* sub-elements.

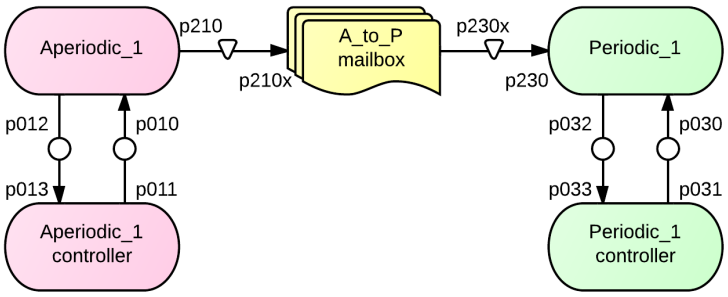


Figure 2.3: Example application with two tasks

```
<process name="aperiodic1" criticality="B">
    <superblock minRep="1" maxRep="1">
        <phase name="mode_normal">
            <info level="B" minAccess="0"
               maxAccess="0" minExecution="0" maxExecution="0"/>
            <info level="C" minAccess="0"
               maxAccess="0" minExecution="0" maxExecution="0"/>
            <info level="D" minAccess="0"
               maxAccess="0" minExecution="0" maxExecution="0"/>
        </phase>
    </superblock>
    <source location="aperiodic1.c" />
    <port type="in_event" name="p010">
        <event name="start" />
    </port>
    <port type="out_event" name="p012">
        <event name="finish" />
    </port>
    <port type="out_data" name="p210" />
</process>
```

Figure 2.4: DOL-C XML specification of a process

A source file is the application code. Ports connect the process to data and control channels. Superblocks inside a process describe the properties of a process - the ranges for execution and memory access times, the criticality, and the range for activation repetition. Since not all of these are used for the functional simulation, some rules here exist.

- The minimum and maximum times of repetition have to be set to one. These are the maximum and minimum number of consecutive task activations in case of restartable tasks. This is a simplification that is made as this attribute is not needed for the functional simulation, since restartable tasks are not simulated. The XML parameters for these are `maxRep` and `minRep`.
- The minimum and maximum access and execution times, given in infos inside phases inside the superblock, are considered zero by the functional simulator regardless of what is specified. These XML parameters are `minAccess`, `maxAccess`, `minExecution` and `maxExecution`.

The *controller* is the state component of a task and it holds information about activation of its process. It has ports and activation information as elements. Ports are used as with the process, to connect the controller to channels. The activation sub-element holds information about the task activation pattern. Parameters of activation patterns are specified here, and they depend on the activation pattern. For example, a periodic process has one parameter,

```
<controller name="Ctrl_aperiodic1" deadline="0.1">
    <activation type="aperiodic">
        <parameter name="m_max" value="2"/>
        <parameter name="interval" value="0.2"/>
    </activation>
    <port type="out_event" name="p011">
        <event name="start"/>
    </port>
    <port type="in_event" name="p013">
        <event name="finish"/>
    </port>
</controller>

<controller name="Ctrl_periodic1" deadline="0.1">
    <activation type="periodic">
        <parameter name="period" value="0.2"/>
    </activation>
    <port type="out_event" name="p031">
        <event name="start"/>
    </port>
    <port type="in_event" name="p033">
        <event name="finish"/>
    </port>
</controller>
```

Figure 2.5: DOL-C XML specification of controllers

the period, and an aperiodic has two, the minimum interval in which a task is executed at most "m" times. The deadline, meaning the deadline for the task execution, is given here, but it is not used in this simulation. All tasks are presumed to fulfil their deadline, since for the functional simulation, the execution times of tasks are assumed to be zero. All times in the XML specification are in milliseconds.

```
<control_channel name="Ctrl_periodic1_to_periodic1">
    <port name="p032"/>
    <port name="p033"/>
</control_channel>

<data_channel name="a_to_p_channel" type="mailbox" size="8" length="1">
    <port name="p210x" type="in_data"/>
    <port name="p230x" type="out_data"/>
</data_channel>

<connection name="a_to_p_channel_In">
    <port name="p210"/>
    <port name="p210x"/>
</connection>
```

Figure 2.6: DOL-C XML specification of communication channels

Every control channel has two ports as sub-elements, and these are the ports it connects, as seen in Figure 2.6. An event inside a port tells what signal the control channel can send. This events can be start, finish, restart, or the integer identifiers of the modes in which it can be executed. The process and its controller are connected via two control channels. An event can be "out" or "in", and this defines the direction of the control channel.

Every process is connected to its controller by two control channels, with opposite directions.

- The control channel from the process to the controller signals the finish event, meaning the task has finished execution. The arrow going from *Aperiodic_1* to *Aperiodic_1 controller* in Figure 2.3 is an example representation of this type of control channel.
- The control channel from the controller to the process, signals the start of execution of a task, or that the task should be restarted. This channel supports all events, except the finish one. The arrow going from *Aperiodic_1 controller* to *Aperiodic_1* in Figure 2.3 is an example representation of this type of control channel.
- As all tasks are considered to fulfil deadline requirements, the restart event is never sent during simulation, and restartable activation patterns are not simulated.
- To change the mode of activation of one task, a control channel connects one task's process to another task's controller. When the first task sends a mode identifier through the control channel, the controller of the second task activates the task according to the corresponding mode from then onwards.

A data channel has its own ports, and connections specify the direction of data flow and to which process the data channel is connected. Data channels always have one port for incoming data. Mailbox-type channels have one port for outgoing data, while blackboard channels can have more ports for outgoing data.

2.3 DAL

The Distributed Application Layer tool, or DAL [10], is developed by ETH Zurich Computer Engineering and Network Laboratory (TIK), and can be used to simulate the behaviour of a Kahn process network (KPN) [13]. Originally developed for streaming applications, writing FMS applications directly in DAL would be impossible.

A KPN is made of processes that are executed independently, and transfer data through FIFO buffers. Processes are data-driven, and the process is blocked when it reads an empty buffer or attempts to write to a full buffer. All processes have the same criticality level.

```
void process1_init(DALProcess *p) { ... }
int process1_fire(DALProcess *p) {
    ...
    int i;
    ...
    DAL_write((void*) PORT_OUT,&(i), sizeof(int), p);
    ...
}
void process1_finish(DALProcess *p) { ... }
```

Figure 2.7: DAL application code in C

Similar to DOL-C, the DAL requires as input a separate description of a XML specification and application code. DAL application code is written in C or C++ following DAL coding rules. An example is given in Figure 2.7, where three important functions are shown: *init*, *finish* and *fire*. These functions, analogous to DOL-C, represent three phases of behaviour: initialization, termination and execution.

In Figure 2.7, sending data to another process is shown. Transferring data between DAL processes is realized using functions *DAL_read* and *DAL_write*. These functions are blocking, i.e. processes are blocked when reading an empty channel, or writing to a full channel.

The application specification contains information on how processes are mutually connected. Buffers connecting processes in DAL are called *software channels*, and *connections* specify which port of a process is connected to which port of a software channel. An example showing DAL XML elements is shown in Figure 2.8. These elements are taken from Figure 2.3.

```
<process name="aperiodic1Cntr" id="5" type="io">
    <port name="p210" type="output" />
    <port name="sched_in" type="input" />
    <port name="sched_out" type="output" />
    <source type="c" location="aperiodic1Cntr.c" />
</process>

<sw_channel name="a1p1_channel" type="fifo" size="1" tokensize="8">
    <port name="p210x" type="input" />
    <port name="p230x" type="output" />
</sw_channel>

<connection name="a1p1_channel-periodic1Cntr">
    <origin name="a1p1_channel">
        <port name="p230x"/>
    </origin>
    <target name="periodic1Cntr">
        <port name="p230"/>
    </target>
</connection>
```

Figure 2.8: DAL XML specification

A KPN described in DAL has some similarities to the FMS application described in DOL-C. However, translating FMS tasks directly to processes and their communication to software channels is impossible. DAL specification is fundamentally different from DOL-C in several aspects.

First of all, DAL does not support the activation patterns present in the FMS application. The DAL simulator, as working with KPNs, assumes that tasks are activated upon availability of input data. There is no consistent way to model all activation patterns directly in DAL. To implement FMS tasks, it would require the user to know too many implementation details. Thus, it would be more practical to specify all tasks in the same manner as it is done in DOL-C, leaving the user to concentrate on the task design.

Secondly, the original DAL supports only blocking and destructive reading from and writing to channels, found in KPNs. For blackboard data channels non-destructive reading is needed together with overwriting write, while for mailbox data channels both non-blocking read and write has to be present. Implementing non-blocking read and write, overwriting write and non-destructive read in DAL is needed in order to simulate the FMS application.

These two reasons show why it is impossible to write the FMS application directly in the DAL language.

2.4 Summary

In this chapter, the case-study for mixed-critical real-time applications, FMS, was explained, together with the activation patterns and communication that are to be simulated with the functional simulator. The DOL-C language, in which the FMS is specified, was introduced, and its XML specification and application coding style were explained. The language for the existing simulator, DAL, was introduced with key features. DAL is used for the specification and

functional simulation of KPNs. The reasons why the FMS can not be modelled in DAL directly were also discussed.

Chapter 3

Simulator Design

This chapter describes the design of the functional simulator. First, Section 3.1 explains the overall design, with an overview of the auxiliary elements needed for correct simulation using the existing DAL tool, and the main mechanism to translate applications written in DOL-C to DAL. Section 3.2 goes into details for every element of the DAL tool extension. Here, newly generated processes in DAL are explained, and how they are generated from DOL-C is shown.

3.1 Design Overview

In this thesis, the functional simulator for applications written in DOL-C is based on the existing DAL simulator for Kahn Process Networks. The functional simulator without the existing DAL tool is a contribution of this thesis, and this can roughly be split into two parts. The first part is the generation of a time keeping element in DAL, and the second is the translation of tasks in DOL-C to DAL. The time keeping element is the scheduler process, and the DOL-C to DAL translation is done by wrapping appropriate processes and controllers into one, called DAL wrapped process. This is illustrated in Figure 3.1.

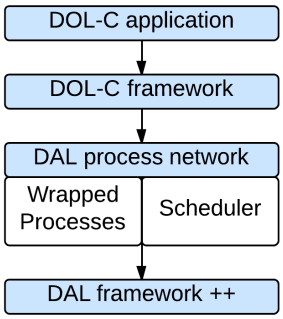


Figure 3.1: Functional simulator design overview

3.1.1 Time keeping

The need for a time keeping element for DAL is described in Chapter 2, as the first reason why FMS application can not be simulated in DAL directly. This is because the FMS tasks are activated according to given patterns, while task activation in KPNs is data-driven. Thus, a mechanism is needed to transform time-driven activation to data-driven.

The mechanism is introducing the scheduler and wrapped processes, as shown in Figure 3.2. The scheduler and wrapped processes are processes in a KPN, that is written in DAL. For every task a corresponding wrapped process is generated, and one scheduler is generated to

keep track of the time. The scheduler is connected to wrapped processes bidirectionally, using DAL software channels. Writing to these software channels is non-blocking, except when the channel is full, and reading from empty software channels is blocking. The scheduler activates a blocked wrapped process by sending to it a data token through their software channel. When activated by the scheduler, a wrapped process executes its corresponding task, and returns another data token to the scheduler.

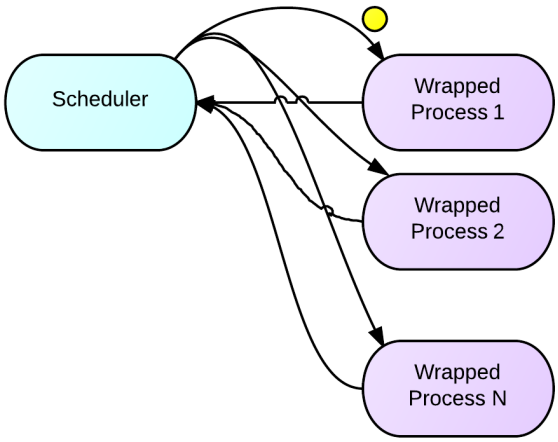


Figure 3.2: The scheduler and wrapped processes

The scheduler keeps information about activation times of tasks, and communicates with wrapped processes using data tokens. The structure of a token returned to the scheduler is given in Figure 3.3, and it consists of two integers - the token sender ID and the next event time. Tokens sent out by the scheduler consist only of the receiver ID. During a simulation step, the scheduler and the process controllers go through the following states:

- First, the scheduler is running. Wrapped process are blocked, as they are reading an empty buffer connecting them to the scheduler.
- The scheduler determines the current simulation time, and sends out tokens to wrapped processes whose tasks are to be activated at that time.
- The scheduler then waits for all the tokens to return, and until they do the scheduler is in a blocking state.
- Once a wrapped process gets a token, it first executes its task. Then, it sends a data token back to the scheduler indicating its next activation time. Finally, the wrapped process blocks while waiting for a new token.
- When all tokens are received by the scheduler, it unblocks and increases the simulation time. The simulation time is increased to the next task activation time. Everything starts over.

The time diagram in Figure 3.4 illustrates this by showing the scheduler and one wrapped process communicating using tokens.

Figure 3.2 demonstrates one token being sent to wrapped process 1.

```
typedef struct _dol_cntr_event {
    int sender;
    int nextEventTime;
} DOL_cntr_event;
```

Figure 3.3: The token sent to the scheduler

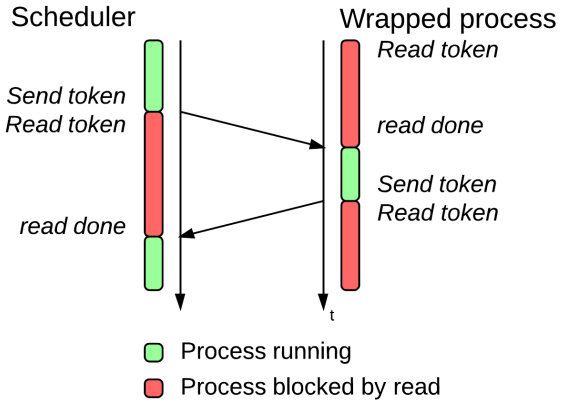


Figure 3.4: Time diagram of token use

3.1.2 Translation from DOL-C

Equally important, translation of tasks from DOL-C to DAL is needed. A task in DOL-C is described by a process and controller, and in DAL, every task has its wrapped process. The design used in this thesis minimizes the number of DAL processes needed, so everything described by a DOL-C process and controller of a task is to be embedded inside the DAL wrapped process.

Since DOL-C application code is written in C/C++, the same as DAL application code, the wrapped process starts the execution of a task by calling the appropriate DOL-C C/C++ function within its own application code. The function called containing DOL-C application code is the *fire* function. After execution, the DOL-C fire function calls the *yield* function, implemented in DAL application code. Yield acts as feedback to the wrapped process. This way, the DOL-C application code acts as the functional component of the task, while the generated wrapped process code acts as the state component, managing activation. This is shown in Figure 3.5.

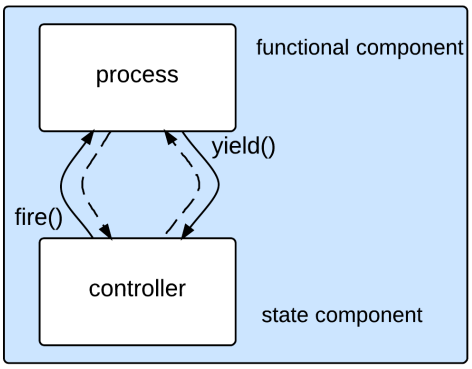


Figure 3.5: Wrapped process application code

The DOL-C XML is translated into DAL XML the following way.

- As DOL-C processes and controllers are embedded together inside a wrapped process, DOL-C control channels connecting them are discarded.
- Remaining DOL-C control channels and all data channels are translated into DAL software channels between wrapped processes.

Every time a task written in DOL-C writes to or reads from a DOL-C channel, an appropriate function is called which does the same but using the DAL channels.

The second reason why FMS applications can not be realized directly in DAL, the absence of non-blocking read, is resolved by implementing additional functions is DAL. These functions include non-blocking destructive read, non-blocking non-destructive read, non-blocking non-overwrite write and non-blocking overwrite write. The first and third functions are used for mailbox data channels, while the second and fourth for the blackboard data channels.

3.2 DAL Extension

This section focuses on details of the extension of the DAL tool. An overview of the extension is presented on Figure 3.6. The DOL-C framework generates DAL XML specification and DAL application code, using as input DOL-C XML specification and DOL-C application code. The generated DAL code includes the modified DOL-C application, and the DOL-C library. The generated DAL code serves as input to the existing DAL framework which is used for the actual simulation.

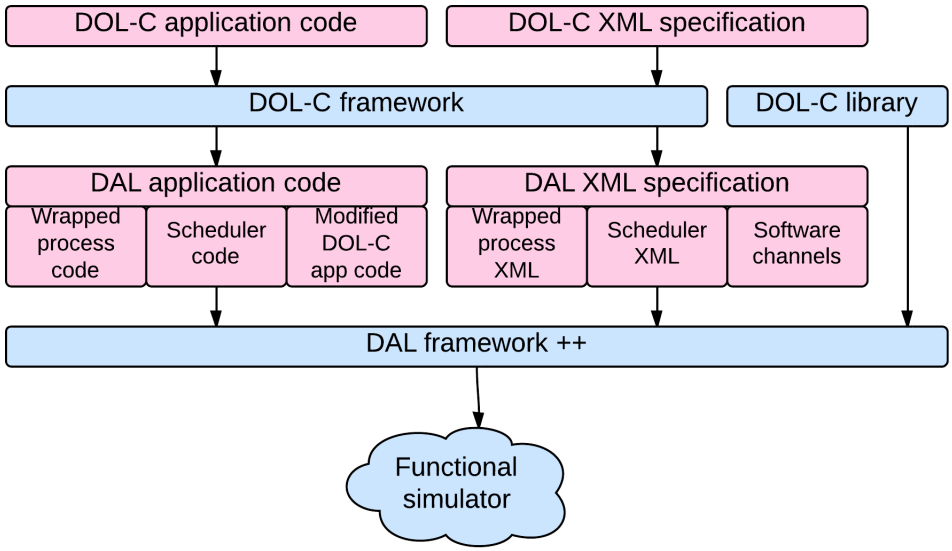


Figure 3.6: DOL-C tool flow

In the following subsections presented are first the wrapped processes and then the scheduler. Their XML specification and application code is explained together. Finally, software channels in the DAL XML specification are presented along with how they are generated and which processes they connect.

3.2.1 Wrapped processes

Wrapped processes are DAL processes. They are a result of DOL-C to DAL translation. The XML specification of a sample wrapped process is given in Figure 3.7.

Ports are either inherited from DOL-C processes or controllers (i.e. port *p210* in Figure 3.7), or ones generated for communicating with the scheduler (i.e. ports *sched_in* and *sched_out* in Figure 3.7).

- Ports in a DOL-C process that are used to transfer data to other DOL-C processes via data channels are translated to ports in the corresponding DAL wrapped process.
- Ports used to send events via control channels from one DOL-C process to a non-corresponding DOL-C controller are also translated to ports in the corresponding DAL wrapped process. These are only found with control channels used when changing a mode of operation of a periodic with mode task.
- Two ports, for communication with the scheduler, are always present. This is port *sched_in* used for receiving data tokens from the scheduler, and *sched_out* used for sending data tokens to the same.

```
<process name="aperiodic1Cntr" id="5" type="io">
    <port name="p210" type="output" />
    <port name="sched_in" type="input" />
    <port name="sched_out" type="output" />
    <source type="c" location="aperiodic1Cntr.c" />
</process>
```

Figure 3.7: DAL XML specification of a wrapped process

The DAL application code of the wrapped process is responsible for activating DOL-C tasks and sending data to the scheduler specifying their next activation. It is generated differently for different activation patterns. For periodic tasks, wrapped processes activate the corresponding DOL-C process and send to the scheduler the current time increased with the period. Periodic with mode wrapped processes are similar, with the difference that they have to activate their DOL-C process in the right mode. The information about modes is read from input ports. Wrapped processes for tasks with aperiodic activation are a bit different, as the next activation time is not known. In this case, just an empty token signalising the end of a task execution is sent to the scheduler.

3.2.2 Scheduler

The scheduler is a DAL process, generated as an auxiliary element. It communicates exclusively with wrapped processes. For sending tokens for activation to wrapped processes, each wrapped process has its own port in the scheduler XML specification. For receiving tokens, a single port is sufficient, as tokens contain information on their sending process. A sample scheduler XML is given in Figure 3.8.

```
<process name="dolcsched" id="-1" type="io">
    ...
    <port name="aperiodic1" type="output" />
    <port name="inport" type="input" />
    <source type="c" location="dolcsched.c" />
</process>
```

Figure 3.8: DAL XML specification of the scheduler

The scheduler is responsible for keeping track of time. The timing mechanism of the functional simulator is event driven, which means that the scheduler keeps track of all future activation times. When all tasks finish execution in a given simulation step, the scheduler jumps to the

next time an activation is scheduled.

All data the scheduler keeps track of is stored in the state variables, seen in Figure 3.9. This data is the simulation time (*time*), a list of future activations, collected from received data tokens (*events*), and a list that indicates what processes is active (*processActive*) at any given time.

```
typedef struct _local_states {
    int time;
    std::multimap<int, int> events;
    int processActive[TOTAL_NUMBER_PROCESSES];
} Dolcsched_State;
```

Figure 3.9: The scheduler state variables

3.2.3 Software channels

DAL processes communicate only using software channels, which are connected to processes via connections. Software channels in the DAL XML are either a result of DOL-C to DAL translation, or generated for communication between the scheduler and wrapped processes.

Data channels in the DOL-C XML specification, both mailbox and blackboard, are translated to DAL software channels in the same manner. DOL-C XML parameters *size* and *length* indicate the size of a single element inside the data channel, and the maximum number of data elements the data channel can store. These are translated to DAL XML parameters *tokensize* and *size*, respectively. Inside the DOL-C application code, there are functions for reading to and writing from data channels. For simulation, the DOL-C library converts these into reading from and writing to DAL software channels. Reading from software channels generated from data channels is always non-blocking. Non-blocking reading is an extension to the original DAL, since it is not present in KPNs. Depending on the type of data channel, reading from software channels is destructive (mailbox) or non-destructive (blackboard). Writing to data channels is always non-blocking. Writing to mailboxes writes the data to the last free space in the buffer, while writing to blackboards overwrites the data stored there before.

Connections are also translated from DOL-C XML to DAL XML. In DAL XML, connections contain information on what is the origin and what is the target element. This information is not present in DOL-C XML, but is found implicitly as the parameter *type* of a port inside a process. Connection examples can be seen in Figure 2.6 (DOL-C) and Figure 2.8 (DAL).

Control channels, used for changing mode of activation for periodic with mode tasks, are translated to software channels in the same manner as data channels. Connections are automatically added, since DOL-C control channels do not have them.

Additional software channels are generated connecting the scheduler with each of the wrapped processes. These channels have the scheduler as the writing process and wrapped processes as the reading ones, and are used for sending tokens for activation. For returning tokens to the scheduler, a single software channel is used. It has all the wrapped process as writing processes, and the scheduler as the reading process.

3.3 Summary

In this chapter, the design of the functional simulator was presented. The event-driven scheduler is described, and how it send tokens to simulate activation patterns. Wrapped processes are introduced, together with the way they communicate with the scheduler and how they execute the actual tasks.

Chapter 4

Implementation

This chapter focuses on the implementation of the DOL-C framework, that generates the input for the DAL simulator. The DOL-C framework is a collection of tools that together implement the design described in Chapter 3. The DOL-C framework is written in the Java programming language.

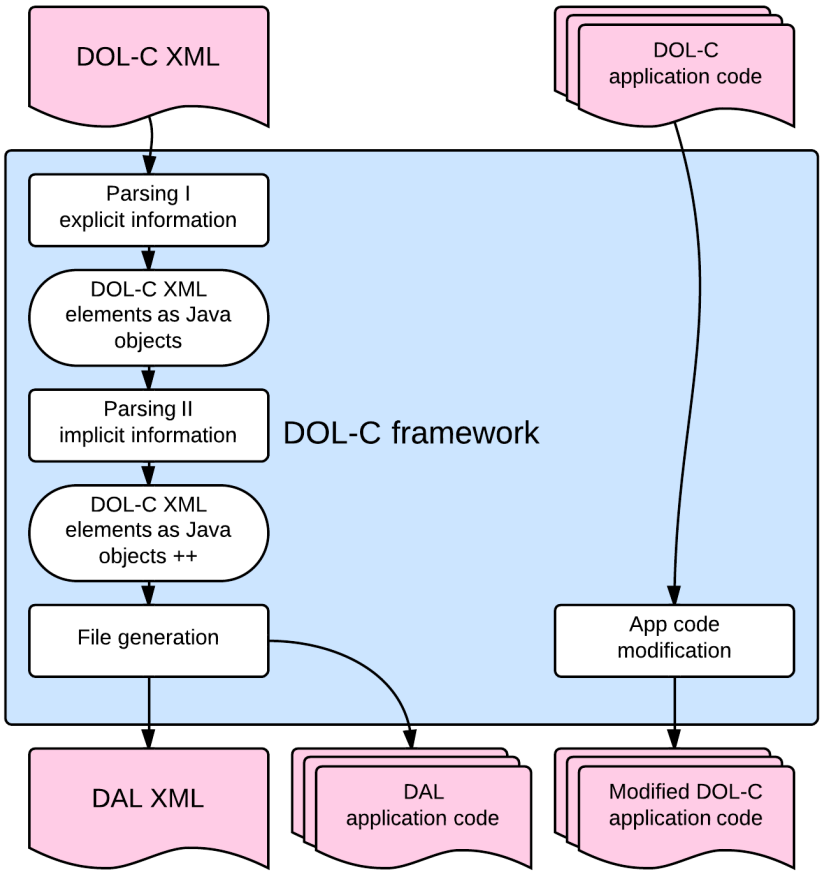


Figure 4.1: The DOL-C framework

Figure 4.1 shows the three main steps executed within the DOL-C framework using the DOL-C XML as input, i.e., the parsing of explicit and implicit information, and file generation.

Parsing means using the DOL-C XML as input to generate Java objects with corresponding data. Parsing of explicit information means reading data that is stated directly in the DOL-C XML. Parsing of implicit information means adding data that is inferred but not directly stated

in the DOL-C XML. One example of implicit information is finding the corresponding controller of a process, by analysing the control channels that connect them. Section 4.1 gives more information on how data passes through parsing.

After parsing, the DAL XML and application code are generated. With all data available, file generation is reduced to choosing the appropriate template and adding parsed data where needed. Every process in the DAL XML is given a ID. The ID of the scheduler is -1, and the wrapped process are given IDs from 0 upwards. Section 4.2 describes how scheduling is implemented in these generated files.

Figure 4.1 also shows modification of application code done in the DOL-C framework, using the DOL-C application code as input. This modification is minor.

- Calling the *yield()* function is added in the application *fire()* function, just before the *return* statement.
- The functions for reading from and writing from data channels are replaced by substitute functions for blackboards. This way, there are separate functions for mailboxes and blackboards.

4.1 Parsing

The first part of parsing, described in Figure 4.1 as *Parsing I*, uses the DOL-C XML file as input, and for every XML element found, it generates an object in Java. The Java object contains:

- All attributes of XML elements.
- All child elements (or sub-elements) of a XML element.

The second part of parsing, or *Parsing II* in Figure 4.1, finds implicitly described information and adds it to Java objects. Implicit information is found searching in the existing Java objects. The following data is searched for and added:

- For a control channel, the process and the controller it connects. This is found by finding matching port names inside control channels on one side, and processes and controllers on the other side.
- For a controller, its corresponding process. This is found by finding the two control channels that connect them.
- For a data channel, mailbox, the reading process and its corresponding port, and the writing process and its corresponding port. This is found by finding matching port names inside the data channel on one side, and process on the other side.
- For a data channel, blackboard, the reading process and its corresponding port, and the writing process or processes and their corresponding ports. This is found in the same manner as for the mailbox data channel.
- For a process, its corresponding controller, and lists of ports that connect the process to control channels, separated into input and output ports. The controller is found by finding the control channels that connect it to the process, and the input and output port list are found by analysing information contained in ports.

After parsing, file generation is done using the Java objects as input.

4.2 Scheduling

All processes in the DAL XML, both the scheduler and wrapped processes, are responsible for correct scheduling and task activation, but time keeping is implemented exclusively in the

scheduler.

During initialization of the scheduler, the global time is set to zero and all tasks are activated. There is no information on the first activation of tasks, so it is assumed that they all execute once at zero simulation time.

The scheduler activates a task by sending an empty data token to the appropriate wrapped process via their software channel. The wrapped process activates the task and returns a time token to the scheduler, with information on its ID and the next activation time. Aperiodic processes send the next activation time -1, which means that the next time is unknown. As soon as the scheduler receives a token with -1 as the next activation time, it replaces the time with a random time value, which is after the current simulation time.

Whenever there are two or more tasks scheduled at the same time, data tokens for activation are sent in a determinate sequence:

- For the first activation, which is done at simulation time 0, the task whose process is described last in the DOL-C XML specification is activated first.
- After the first activation, if there are multiple tasks scheduled at the same time, the task whose last activation is the earliest will be activated first.

The scheduler keeps all task activation times in a list sorted in increasing order. When all tasks finish execution on a given simulation time, the scheduler sets the new time as the time of activation of the first element of the list. It checks the list for how many tasks are there scheduled at that time, activates them, and removes them from the list. This repeats until every task has its next activation time greater than the maximum simulation time.

4.3 Summary

This chapter explained the main implementation steps of the functional simulator. Explicit and implicit parsing is introduced, as a two-step way to extract all information from the DOL-C XML. The scheduling mechanism is described, and how the scheduler keeps activation times.

Chapter 5

Evaluation

This chapter presents the evaluation of simple case-studies, which validate the functional simulator. Presented are three examples, via which we demonstrate all activation patterns and communication mechanisms. In particular:

- The first example validates blackboard and mailbox data channels, as well as aperiodic and periodic activation of tasks.
- The second example demonstrates the resulting non-determinism of the simulation results when race conditions exist between two or more periodic tasks which read/write from/to the same blackboard.
- The third example validates the periodic with mode activation pattern, and the change of the activation mode.

5.1 Data Channels

This basic example consists of two tasks connected with one data channel, as illustrated in Figure 5.1. The task writing to the data channel, with its process labelled as *A*, has aperiodic execution. The task reading from the channel, with its process labelled *P*, is periodic. Elements *AC* and *PC* are controllers for processes *A* and *P*, and *A2P* is the data channel. For simplicity, control channels and connections are shown as arrows.

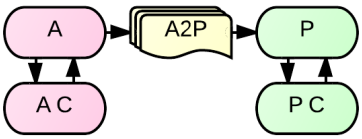


Figure 5.1: Example application 1

The simulation time is set as $1200\ ms$. Task *P* has periodic activation, with a period of $200\ ms$. Task *A* has aperiodic activation, and it is executed in the following times: $0\ ms$, $433\ ms$, $876\ ms$ and $1138\ ms$. This is depicted in Figure 5.2.

Two simulations have been done to show characteristics of both mailbox and blackboard buffers. Writing to these buffers will produce same results, but reading from them will not. Table 5.1 displays the simulation results. The first two columns show when activations happen and what tasks have been activated. The rest columns show data that has been written to or read from the data channel. When no data is present in the data channel, "none" is stated. When a task has not been activated for read or write, a slash "/" is written in the corresponding field. The same random seed has been used to generate aperiodic task activation time for both

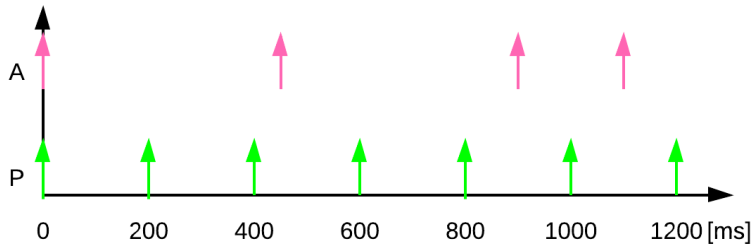


Figure 5.2: Task schedule for example 1

Table 5.1: Simulation results for example 1

Time [ms]	Active task	Data written	Data read - for mailbox	Data read - for blackboard
0	A, P	0	none	none
200	P	/	0	0
400	P	/	none	0
433	A	1	/	/
600	P	/	1	1
800	P	/	none	1
876	A	2	/	/
1000	P	/	2	2
1138	A	3	/	/
1200	P	/	3	3

simulating with the blackboard and mailbox.

Table 5.1 shows differences between mailboxes and blackboards. Mailboxes behave like FIFO buffers, and become empty when all data is read. This can be seen at time $800\ ms$. Data in blackboards does not expire, and the last value written is always read. This can also be seen at time $800\ ms$.

Table 5.1 and Figure 5.2 also validate that periodic tasks are activated after a fixed time period, while aperiodic tasks are activated randomly.

5.2 Non-determinism: Race Condition

This example shows a source of non-determinisms in the simulation - race conditions. Race conditions occur when two processes activate at the same time. It is not known which task is to be activated first. In the case that more tasks are connected via a data channel and are activated at the same time, different results may happen. This means that, when a race condition is present, running the simulator multiple times may produce different output.

This is demonstrated in this example. Two periodic processes, connected with a mailbox type buffer in the same manner as in Figure 5.1, are activated with the same period. The simulation lasts for $400\ ms$. Figure 5.3 shows the timing diagram with three task activations of both tasks. Table 5.2 and Table 5.3 show two of the possible simulation results. Table 5.2 is for when the reading process is scheduled always first, while Table 5.3 is for when the writing process is scheduled always first.

Observing the simulation results, some points can be highlighted:

1. Both results are legitimate.

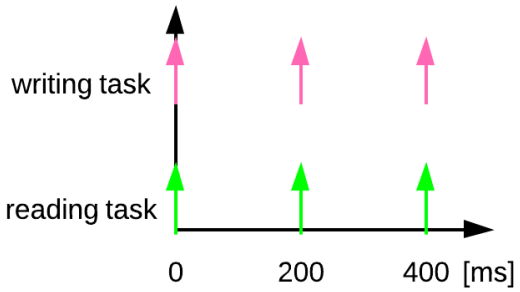


Figure 5.3: Task schedule for example 2

Table 5.2: Simulation results for application 2 - with the reading task activated first

Time [ms]	Active task	Data written	Data read
0	writing, reading	0	none
200	writing, reading	1	0
400	writing, reading	2	1

Table 5.3: Simulation results for application 2 - with the writing task activated first

Time [ms]	Active task	Data written	Data read
0	writing, reading	0	0
200	writing, reading	1	1
400	writing, reading	2	2

2. The final results of task execution are the same, even though results in Table 5.3 arrive one period later than the results in Table 5.2.
3. A situation can be simulated where the reading process is activated first on one time and then second on another time.
4. Both simulated cases show that a mailbox of length 1 is sufficient. However, for the situation described in the last point a mailbox of length 2 is needed to avoid loss of data.

5.3 Modes of Operation

The final example shows how one task changes the mode of a periodic with mode task.

The example application is shown in Figure 5.4, and it consists of two tasks: *P* and *M*. Task *P* is periodic, and is represented with its process *P* and controller *PC*. Task *M* is periodic with mode, and is represented with its process *M* and controller *MC*. These tasks are connected with a data channel *P2M*, which is a blackboard, and with a control channel *control*. This control channel is used to send the new mode of activation from *P* to *M*. Other control channels and connections are represented as arrows, for simplicity.

The simulation lasts for $1200\ ms$, and the periods are $100\ ms$ for task *P* and $200\ ms$ for task *M*. At 0 simulation time, task *P* executes first. The simulation time diagram is shown in Figure 5.5, and simulation results in Table 5.4.

Task *P* sends mode 0 during its first activation, and then changes the mode between 0 and 1 every third activation.

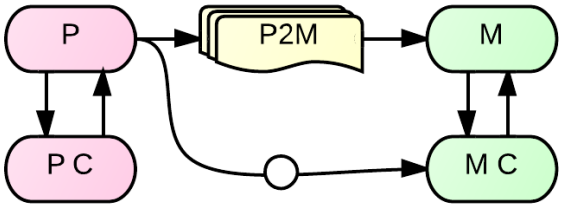


Figure 5.4: Example application 3

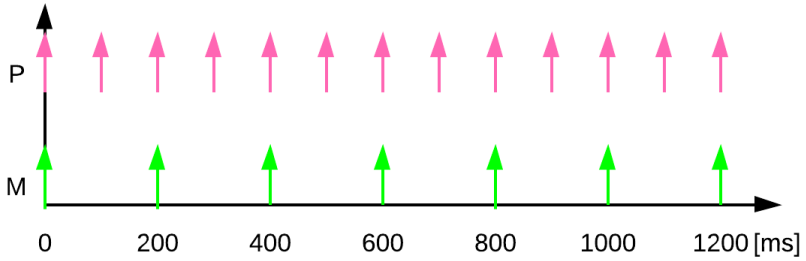


Figure 5.5: Task schedule for example 3

Table 5.4: Simulation results for application 3

Time [ms]	Active task	P writes	P sets activation mode	M reads	M activated in mode
0	P, M	0	0	0	0
100	P	1	none	/	/
200	P, M	2	none	1	0
300	P	3	1	/	/
400	P, M	4	none	3	1
500	P	5	none	/	/
600	P, M	6	0	5	1
700	P	7	none	/	/
800	P, M	8	none	7	0
900	P	9	1	/	/
1000	P, M	10	none	9	1
1100	P	11	none	/	/
1200	P, M	12	0	11	1

During simulation, at time 0 task *P* is executed first, while at times $200\ ms$, $400\ ms$, $600\ ms$, $800\ ms$, $1000\ ms$ and $1200\ ms$ task *M* is executed first. Table 5.4 shows how task *M* is always activated in the last mode sent by task *P*.

The simulation also shows the way blackboards only keep the last data written to them, so because of the shorter period of task *P*, half of the data written is not read.

5.4 Summary

This chapter validated, through three FMS like examples, the designed functional simulator. All examples were consisting of two tasks. All activation patterns are validated, including the change of mode for periodic with mode tasks. Transporting data via blackboards and mailboxes has also been validated and compared. Finally, the race condition example explained that when two tasks are scheduled at the same time, it is not determined which task will be executed first.

Chapter 6

Conclusion

In this thesis a functional simulator for the Flight Management System application is designed, implemented and evaluated. The FMS application is a case study of a mixed-critical and real-time application, and it consists of tasks that have various criticality levels and one of five possible activation patterns.

For the functional simulator, the existing DAL simulator is re-used. Reasons why DAL could not be used in the first place arise from the fact that DAL is used to simulate Kahn Process Networks, so instead of time-driven activation, DAL process activation is data-driven. Besides this, reading from and writing to communication channels in DAL is blocking when these channels are empty and full, respectively, while in DOL-C read and write functions are always non-blocking.

The solution to these limitations is the development of the DOL-C framework, which acts as a pre-processing tool, as it translates DOL-C input files into DAL code for simulation. The DOL-C framework uses the DOL-C language, which describes applications using a XML specification and application code.

The design of the functional simulator assumes that all tasks are executed in zero time. This makes the number of simulated activation patterns reduced to three: periodic, periodic with mode and aperiodic.

Communication between tasks is implemented using different mechanisms. Data channels are presented, and the difference between its two types are explained. Mailboxes are FIFO buffers, while blackboards behave like shared variables. Control channels are also presented, how they are used within a task to send activation signals, and how they are used between two tasks to send the mode of activation from one task to the other. A limitation that has to be kept in mind during the use or the simulator is the race condition, when two tasks share a communication channel are scheduled to activate in the same time.

The implementation of the functional simulator has been done in Java. The main steps are parsing the DOL-C XML, finding both explicitly written and implicitly stated information, and file generation done by using templates.

Finally, three example applications with two tasks each demonstrate activation patterns, data channels and race conditions between tasks. Simulation results are presented and discussed.

Appendix A

Presentation Slides

ETH zürich

Functional Simulation of a Flight Management System

Stefan Draskovic

Tutor: Prof. Dr. Lothar Thiele
Advisors: Pengcheng Huang, Georgia Giannopoulou, Lars Schor

Computer Engineering and Networks Laboratory

ETH zürich

Overview

- ***Introduction***
- Design
- Implementation
- Evaluation

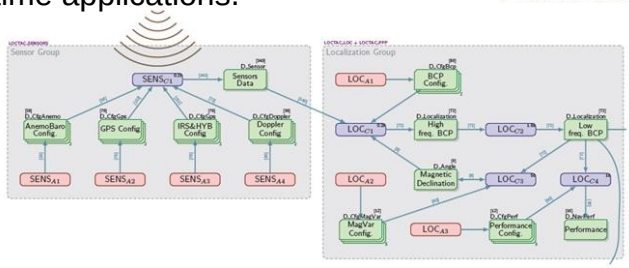
Computer Engineering and Networks Laboratory

ETH zürich

Motivation

- Flight Management System – a case-study of mixed-critical real-time applications.

THALES



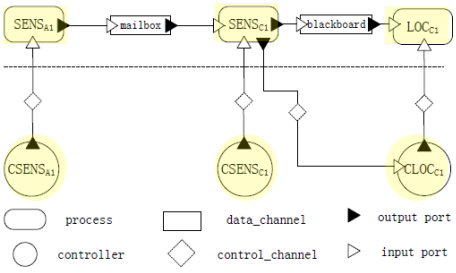
- Simulation gives feedback early in the design process

Computer Engineering and Networks Laboratory

ETH zürich

Distributed Operation Layer – Critical (DOL-C)

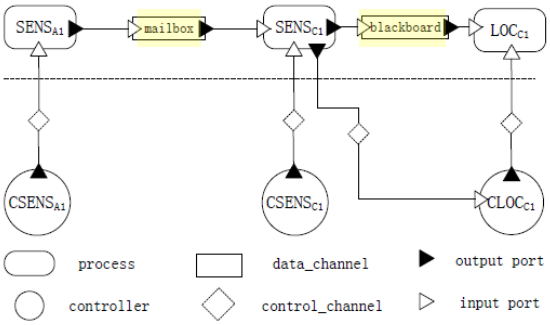
- DOL-C C/C++ app code, XML app specification
- Simulate different activation patterns
  - Aperiodic, periodic, periodic with mode, pseudo-periodic



ETH zürich

Distributed Operation Layer – Critical (DOL-C)

- Respect data communication channels
- Mailbox – “non-blocking” FIFO buffers
- Blackboard – shared variables

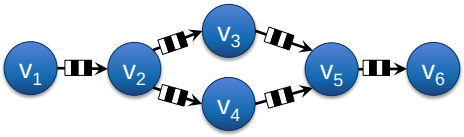


ETH zürich

Approach

1. Translate DOL-C app into DAL app
2. Simulate DAL app using existing simulator

- DAL can be used to simulate the behavior of a Kahn process network (KPNs)
  - DAL C/C++ application code, XML process network specification



Computer Engineering and Networks Laboratory

ETH zürich

Challenges

- DAL KPNs cannot be used to model the FMS
- Data-driven vs. time-driven activation
- Controllers in addition to functional processes
- Different communication patterns

Computer Engineering and Networks Laboratory

ETH zürich

Contribution

- Designed and implemented framework that translates DOL-C into DAL
- Developed mechanism to simulate time-driven activation
- Designed various benchmark apps for DOL-C

ETH zürich

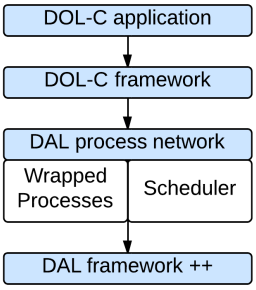
Overview

- Introduction
- ***Design***
- Implementation
- Evaluation

ETH zürich

Simulator Design

- Wrap DOL-C processes and controllers into DAL processes – called process controllers
- Add scheduler process for global time keeping and process controller synchronization
- Simulate on top of DAL
- Assume all tasks executed in zero time

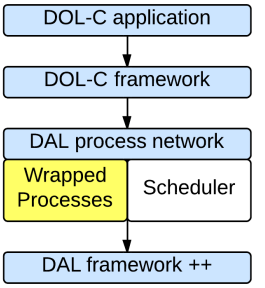


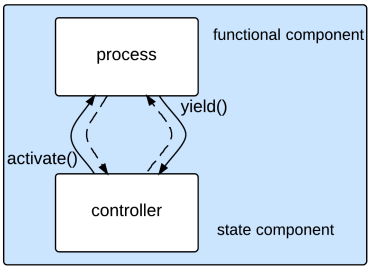
Computer Engineering and Networks Laboratory

ETH zürich

Wrapped DAL Process

- DOL-C processes and controllers wrapped together into DAL processes
- The state and functional component are in one DAL process
- Communication between the components realized through activate() and yield()



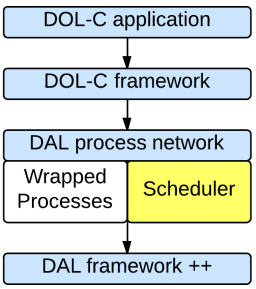


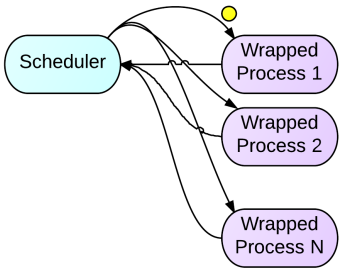
Computer Engineering and Networks Laboratory

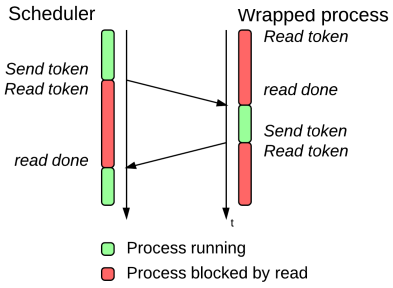
ETH zürich

Scheduler

- DAL process that manages global tim
  - Uses data tokens for process activation and communication







ETH zürich

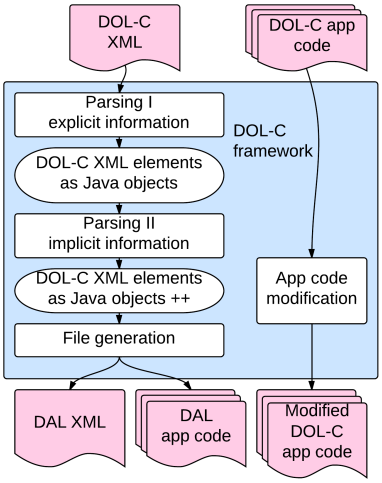
Overview

- Introduction
- Design
- ***Implementation***
- Evaluation

ETH zürich

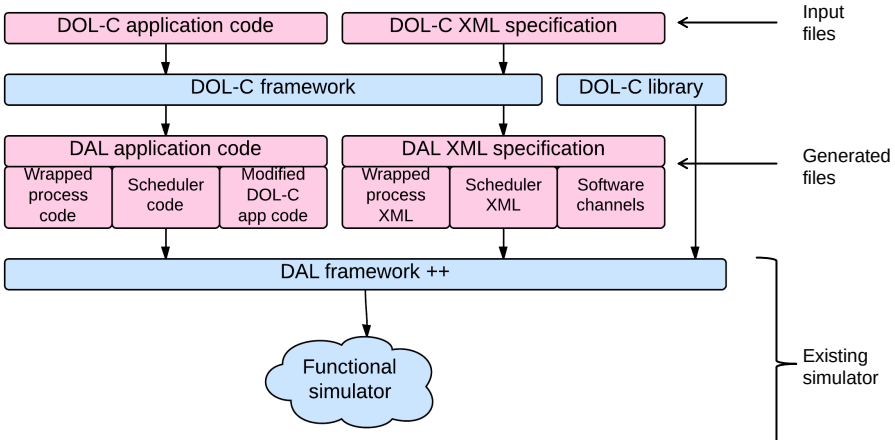
Implementation of the DOL-C framework

- Implemented in Java
- Parses information given in the DOL-C XML and generates Java objects
- Generates all files using these objects



ETH zürich

Complete DOL-C Tool-Chain



ETH zürich

Overview

- Introduction
- Design
- Implementation
- ***Evaluation***

ETH zürich

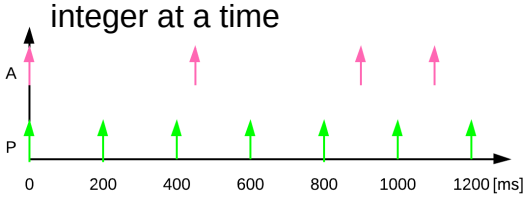
Evaluation goals

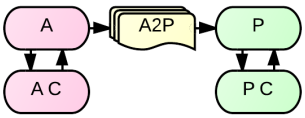
- Example 1
  - Aperiodic and periodic activation
  - Blackboard vs. mailbox demonstration
- Example 2
  - Periodic with mode activation and mode change

ETH zürich

Example application – Mailbox and Blackboard

- Aperiodic and periodic task writing and reading one integer at a time



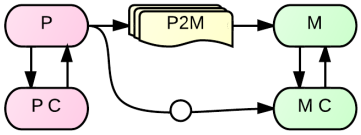


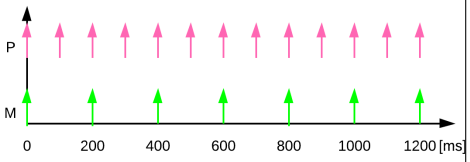
Time [ms]	Action, when mailbox	Action, when blackboard
433	A writes “1”	
600	P reads “1”	
800	P reads “no data”	P reads “1”

ETH zürich

Example application – Mode of Activation

- Periodic tasks sends data every activation, changes mode of other task every third activation

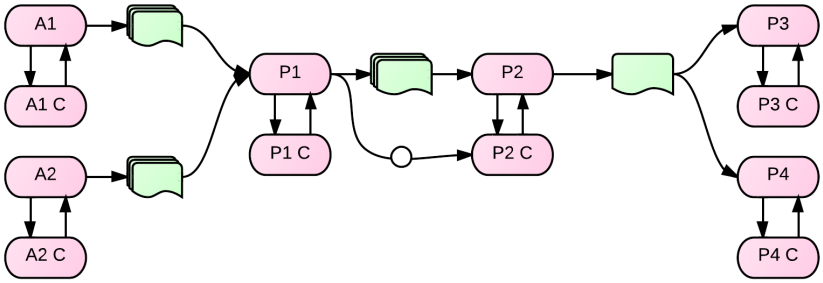




Time [ms]	Action	
200	M reads “1” in mode 0	P writes “2”
300	P writes “3”, changes mode of activation to 1	
400	M reads “3” in mode 1	

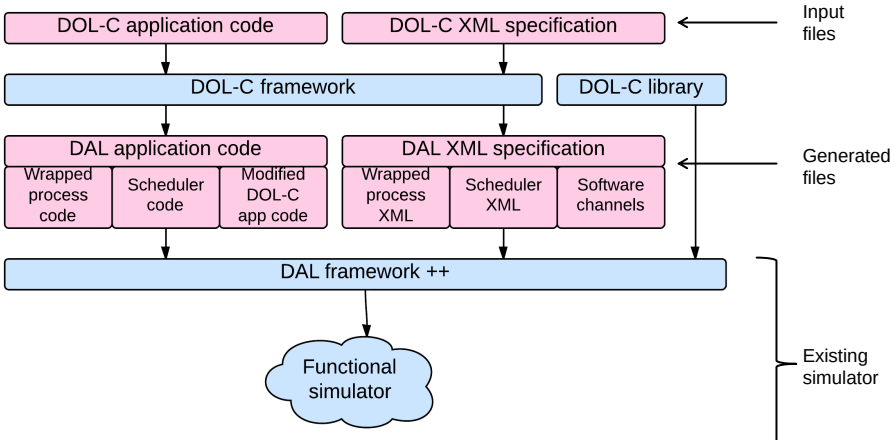
ETH zürich

Demonstration



ETH zürich

Complete DOL-C Tool-Chain



ETH zürich

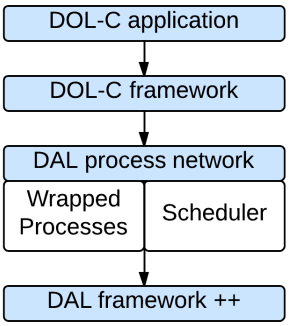
Demo results

Time [ms]	Action
300	P2 activated in mode 1, reads “30”, writes “30”
400	P4 reads “30”
400	P1 reads “no data” and “no data”
400	P2 activated in mode 1, reads “no data”
433	A2 writes “31”
433	A1 writes “1”
500	P3 reads data “30”
500	P2 activated in mode 1, reads “no data”
600	P1 reads “1” and “31”, writes “1” and “31”

ETH zürich

Conclusion

- Functional simulator designed, implemented and evaluated
- Using simulator enables the designer to check their design



- Future work: Translate FMS to DOL-C and simulate

Appendix B

Code Structure and Use

This chapter explains how to use the functional simulator, and provides details on the Java implementation of the DOL-C framework, used to translate DOL-C applications into DAL for simulation.

B.1 Simulator Use

As the functional simulator is an extension of the DAL tool, it is used in the same way. Below are steps needed for the simulator set-up, and the actual simulation.

1. First, make sure that the corresponding code is copied on your local machine. Start your favourite shell and navigate to the *dal* folder.
2. Place the input files in the below stated folder. Replace X with any other string, this way more examples can be prepared for simulation.

   ```
   dal/examples/exampleX/
   ```

   Inside the folder, there should be the DOL-C XML, which follows the shown naming convection, and a folder for the application source code.

   ```
   exampleX-DOLC.xml
   dolc-src
   ```

   Inside $dolc - src$ should be the source codes, as specified by the XML.
3. Next, build the source code of the simulator. Navigate back to the *dal* folder, and run the below given command.

   ```
   $ ant -f build.xml
   ```

4. Finally, navigate to the newly created folder to execute the functional simulation using the below given command. Replace the value of $-Dnumber$, written as X below, to match the name of the application.

   ```
   $ cd build/bin/main/
   $ ant -f runexample.xml -Dnumber=X
   ```

B.2 Code Structure

As the DOL-C framework is developed in Java, data is realized as Java objects. The objects contain all information needed for functional simulation. The location of the objects is given below.

```
dal/src/dal/datamodel/dolc
```

For every XML element, there is a corresponding Java object.

The parsing of explicit data, i.e. transferring all data given in the DOL-C XML to Java objects, is done using the following parser object.

```
dal/src/dal/parser/xml/dolcschema/DOLCHandler.java
```

The parser returns an object stack with one object which represents the application, and contains pointers to all child elements.

The parsing of implicit data is done using the Java objects as inputs, and is done using the below stated visitor object.

```
dal/src/dal/visitor/dolc/DOLCObjectConnect.java
```

Besides this, three other visitor objects exist.

```
dal/src/dal/visitor/dolc/DOLCGenerateXml.java
dal/src/dal/visitor/dolc/DOLCGenerateCntr.java
dal/src/dal/visitor/dolc/DOLCSourceHandler.java
```

The first two use the Java objects as input. The first generates the DAL XML, and the second generates DAL wrapped process application code. The last visitor is used to modify the DOL-C application code.

Files referred to as the DOL-C library, needed for correct simulation, are located in the following folder.

```
dal/src/dal/visitor/dolc/lib/
```

Appendix C

Project Assignment

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Semester Thesis at the
Department of Information Technology and
Electrical Engineering

for

Stefan Draskovic

Functional Simulation of a Flight Management System

Advisors: Pengcheng Huang
Georgia Giannopoulou
Lars Schor

Professor: Prof. Dr. Lothar Thiele

Handout Date: 11.10.2013
Due Date: 24.01.2014

1 Project Description

Embedded control systems usually feature complex task activation patterns, with real time requirements associated with them. One of the main reasons is that control systems need to closely monitor the environment which includes the operations of the systems themselves. Corresponding functionalities are then triggered according to the dynamic changes in the environment. As an example, the Flight Management System (FMS) application [4] considered in the Certainty project [1] has 5 different activation patterns: periodic, periodic with mode, aperiodic, pseudo-periodic and restartable. For the periodic activation pattern, the tasks need to be activated periodically while for the periodic with mode activation pattern, tasks may be activated with different periods in different modes. Details can be found in [4, 3, 5]. One of the most important issues for the design of such systems is to simulate at an early stage the functionalities of the applications, without deploying them on real hardware platforms. The simulation can help to gain early knowledge of the behaviors of the applications, and if possible, to detect the potential design bugs and to find the right design choices. As a result, simulation can usually lead to more efficient design of embedded control systems.
This thesis will explore the functional simulation of the Flight Management System. The task will be based on existing knowledge in the TEC group on the modeling of the FMS application. The implementation of the simulator will be based on the existing DAL tool [7, 6], where a simulator in a different context already exists.

2 Project Goals

The main goal of this thesis is to design and implement the core structures that are required for the functional simulation of the FMS application based on an existing simulation framework. The student needs to understand how different activation patterns work in the FMS application, design the simulator component for each activation pattern. And then run them on top of the existing DAL framework.
Specifically, some new features that need to be supported by DAL for the functional simulation of FMS include:

- wrapping pairs of FMS controller and process into corresponding DAL processes (FMS processes associated with different activation patterns need to be wrapped up differently with their controllers);
- a global scheduler process which is responsible for the activations of the wrapped DAL processes;
- an integration of the wrapped DAL processes and the global scheduler process into the DAL simulator, which leads to the final simulation of the FMS.

Depending on the progress of this project, the goals of this project may further include:

- a complete simulation of the entire FMS;

- the evaluation of the original FMS based on its simulation and suggestions on the design of FMS.

3 Tasks

The project will be split up into several subtasks, as described below:

3.1 Familiarization with FMS and DAL

In the beginning of the project, the focus is firstly on getting acquainted with the FMS application: 1) how the different activation patterns work; 2) how the activations are modeled with the controller and process language; 3) how the FMS application is specified in the XML format. In addition, the student should learn the basic mechanisms to structure the simulator [8]. Furthermore, the student needs to get himself familiar with the existing DAL framework, in particular how the existing simulator in DAL works. Detailed descriptions of the FMS application and the DAL tool can be found in [4, 3, 5, 7, 6, 2]. All the necessary resources for this part of the project will be made available as soon as the project starts. At the end of this first project phase, it should be clear how FMS processes with different activation patterns can be wrapped together with their controllers into DAL processes. A suggested step at this phase is to try a simple producer and consumer example and let it run in the DAL simulator with the simple periodic activation pattern.

3.2 Implementation of Simulator

In this second phase of the project, the implementation of the simulator needs to be performed. The student needs to implement the basic simulator components for different activation patterns, as well as the scheduler component for the new simulator. Those simulator components need to be then tested with the DAL tool. Depending on the progress, the mentioned optional tasks can be also conducted at this phase.

3.3 Thesis Report and Final Presentation

Finally, a thesis report is written that covers all aspects of the project. In addition, the final presentation has to be prepared.

4 Project Organization

4.1 Weekly Meeting

There will be a weekly meeting to discuss the project's progress based on a schedule defined at the beginning of the project. A revision of the working document should be provided at least 24 hours before the meeting.

4.2 Semester Thesis Report

Two hard-copies of the report are to be turned in. All copies remain the property of the Computer Engineering and Networks Laboratory. A copy of the developed software needs to be handed in on CD or DVD at the end of the project.

4.3 Initial and Final Presentation

In the first month of the project, the topic of the thesis will be presented in a short presentation during the group meeting of the Computer Engineering Lab. The duration of the talk is limited to three minutes. At the end of the project, the outcome of the thesis will be presented in a 15 minutes presentation, again during the group meeting of the Computer Engineering Lab.

4.4 Work Environment

The work will be carried out in the framework of the European CERTAINTY [1] project to which the Computer Engineering Lab is contributing in terms of scheduling techniques, performance analysis, and mapping optimization. Concretely, this means that the results of your work will be used by the involved project partners if the project goals are met.

References

[1] Certification of Real Time Applications designed for mixed criticality (Certainty). http://www.certainty-project.eu/.

[2] Semantics of the dal xml schemata. 2012.

[3] Certainty. Flight management system addendum to use-case specifications (upper fms part), 2012.

[4] Certainty. Fms usecase requirements, 2012.

[5] Certainty. Dol-critical, 2013.

[6] L. Schor, I. Bacivarov, D. Rai, H. Yang, S.-H. Kang, and L. Thiele. Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. CASES '12, pages 71–80, New York, NY, USA, 2012. ACM.

[7] L. Schor, D. Rai, H. Yang, I. Bacivarov, and L. Thiele. Reliable and efficient execution of multiple streaming applications on intels scc processor. In *Proc. Workshop on Runtime and Operating Systems for the Many-core Era (ROME)*, Aachen, Germany, Aug 2013. Springer.

[8] L. Thiele. Certainty application model. 2013.

Zurich, February, 2013

Bibliography

[1] Certification of Real Time Applications designed for mixed criticality (Certainty). http://www.certainty-project.eu/.

[2] Semantics of the dal xml schemata. 2012.

[3] Certainty. Flight management system addendum to use-case specifications (upper fms part), 2012.

[4] Certainty. Fms usecase requirements, 2012.

[5] Certainty. Dol-critical, 2013.

[6] L. Schor, I. Bacivarov, D. Rai, H. Yang, S.-H. Kang, and L. Thiele. Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. CASES '12, pages 71-80, New York, NY, USA, 2012. ACM.

[7] L. Schor, D. Rai, H. Yang, I. Bacivarov, and L. Thiele. Reliable and efficient execution of multiple streaming applications on intels scc processor. In *Proc. Workshop on Runtime and Operating Systems for the Many-core Era (ROME)*, Aachen, Germany, Aug 2013. Springer.

[8] L. Thiele. Certainty application model. 2013.

[9] Distributed Operation Layer. http://www.tik.ee.ethz.ch/ shapes/dol.html

[10] Distributed Application Layer. http://www.tik.ee.ethz.ch/ euretile.php

[11] Thales. http://www.thalesgroup.com/

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

[13] G. Kahn. The Semantics of a Simple Language for Parallel Programming. Information Processing 74, Proceedings of IFIP Congress 74, Stockholm, Sweden, August 5-10, 1974. North Holland, 1974.

[14] ICT Collaborative Project, D06.1 - Formal Composition Language and Validation, 2011.