Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

Modelling of the real-time control system for a nuclear fusion experiment using Uppaal

Student
Emanuele Zampieri

Advisor
Prof. Michele Moro

Co-Advisor
Dr. Gabriele Manduchi

Academic Year 2016/2017

Table of Contents

List of Figures

List of Tables

Abstract

Recent nuclear fusion experiments require a real-time control system to improve plasma confinement and suppress its magneto hydrodynamic (MHD) instabilities. Referring to the RFX experiment (Padua, Italy), we want to model its real-time control system with the tool Uppaal. The main objective of this thesis is to analyse how the system's behavior changes according to the different schedulers and their configurations. Two categories of scheduler for real-time threads are considered: scheduler with fixed assignment and Linux 2.6 free scheduler. The results obtained show that fixed assignment with a one-to-one thread-core relationship guarantees better performance for the RFX control system. Moreover, the simulations performed show the modelling limitations due to the performance of the Uppaal verification engine.

Abstract

I recenti esperimenti di fusione nucleare necessitano di un sistema di controllo real-time per migliorare il confinamento del plasma e per la soppressione delle sue instabilità magneto-idrodinamiche (MHD). Facendo riferimento all'esperimento RFX (Padova, Italia), si vuole modellizzare il suo sistema di controllo real-time utilizzando il tool Uppaal. L'obiettivo principale di questa tesi consiste nell'analizzare come cambia il comportamento del sistema al variare degli scheduler utilizzati e delle loro configurazioni. Sono state considerate due categorie di scheduler per processi real-time: scheduler con assegnazione fissa e scheduler libero di Linux 2.6. I risultati ottenuti mostrano che l'assegnazione fissa con relazione uno-a-uno tra processi e core garantisce migliori performance per il sistema di controllo di RFX. Inoltre, le simulazioni eseguite mostrano le limitazioni di modellazione derivanti dalle prestazioni del motore di verifica usato da Uppaal.

Chapter 1

Introduction

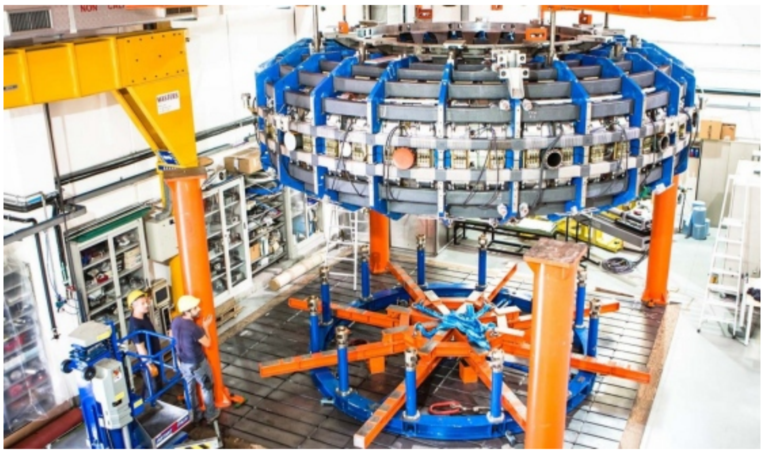


Figure 1.1: Conductive shell of the RFX experiment.

In recent years, the need to have a source of sustainable energy is becoming increasingly evident. One possibility is offered by nuclear fusion, but it needs further research and experimentation before it can be available for energy production. Today, there are several nuclear fusion experiments around the world.
Recent nuclear fusion experiments use real-time control systems to improve plasma confinement. Plasma is created inside a shell, and it's important to control its position and suppress its magneto hydrodynamic (MHD) instabilities, which can lead to an interaction between plasma and shell. The conductive shell can only partially and temporarily counteract plasma instability. Without an active control system, interactions between the plasma and the container occur, and this can lead to a disruption [1].
This thesis refers to the RFX nuclear fusion experiment, supported by the homonymous consortium based in Padua (Italy) and inaugurated in 1992. The RFX machine is a magnetic confinement experiment for study plasma confinement techniques and for reduce the MHD instabilities.
Around the container of this machine several coils are positioned that generate different magnetic fields. Some of these coils are used to perform global placement corrections of the plasma column inside the container. Instead, the remaining coils perform local corrections to limit MHD instabilities.

The RFX machine has been improved over the years by introducing a new real-time control system in 2012. This new control system is based on a multi-core server capable of processing all the necessary control algorithms fulfilling the implied time constraints. The control system is able to acquire the measures of some physical quantities through electromagnetic probes and coils placed around the container. Acquired data are encapsulated in UDP packets and sent to the server via a Gigabit Ethernet network. These data must be pre-processed by the server to provide information on the plasma position and its currents and to provide the requested control signals.
The control system of this experiment uses Scientific Linux distribution with the PREEMPT_RT patch as operating system (soft real-time) and the MARTe framework [2]. This distribution of Scientific Linux is released by Fermilab, CERN, and is used by several universities and laboratories around the world [3]. Its primary purpose is to reduce duplicate effort and to have a common install base for the various experimenters [3].
As shown in [1], the performance of the real-time control system depends on various parameters such as the type of scheduler, the number of CPUs available, the interactions between the real-time threads that compose the system, and so on.

To study the system performance, we want to model the real-time system by taking into consideration its main aspects and used features of the operating system. To do this, the Uppaal tool, developed by the universities of Uppsala and Aalborg, was chosen [4]. This tool is used for modelling, simulation, and verification of real-time systems. Uppaal being able to describe the system with a network of timed automata [4], each one may interact with another. When the system has been modelled, it's possible to verify some properties for validate the implemented model. To verify properties, queries are implemented using a specific language. Queries are used to analyse the system performance depending on the different schedulers and their configuration. In addition, we want to find the limits of Uppaal in system modelling and see if it's possible to represent the main features of the real-time system and operating system in more general terms.

1.1 Related works

During the development of this thesis, there weren't Uppaal applications for modelling real-time control systems applied to nuclear fusion experiments. At the same time Uppaal has already been successfully applied for the study of various real-time systems and communication protocols, as reported in [4].
One of the studies that is closer to this thesis is described in [5]. The authors of this article want to determine schedulability of a real-time taskset on a multicore processor with global scheduling policies such as global fixed-priority (FP) or earliest deadline first (EDF) algorithms.

1.2 Thesis structure

This thesis is organised as follows. Chapter 2 describes the Uppaal tool in detail: formal definition of timed automata, its extensions, the model definition language and the model property verification. Chapter 3 presents the RFX nuclear experiment: characteristics of the experiment, the system hardware components and the software organisation. Chapter 4 explains how the Linux scheduler works, referring to kernel 2.6. Chapter 5 shows the model implementation: description of the various functional components of the system and explanation of the implementation choices. Chapter 6 discusses model validation: description of the queries used for validating the model and analysis of the results obtained with the different schedulers. Chapter 7 discusses the Uppaal verification engine performance. Chapter 8 presents the conclusions and possible future developments of this thesis.

Chapter 2

UPPAAL

Figure 2.1: Uppaal official logo.

UPPAAL is a tool environment for modelling, simulation and verification of real-time systems, jointly developed by Aalborg University and Uppsala University [6]. Typical application areas are real-time controllers and communication protocols, or more in general all areas where timing aspects are critical. This tool has been applied successfully in several case studies as reported in [4].
The main idea is to model a system using *networks of timed automata* extended with user defined functions, integer variables, structured data types and channel synchronisation. Timed automata are finite state machines with clocks and all these automata can communicate with each other through shared variables or channels.

UPPAAL has three main parts [7]: a description language, a simulator and a model checker. The description language serves as a design language to define system behavior using clocks, user functions and simple data types (e.g. arrays, integers, bounded integers, etc.). The simulator is a validation tool which allows examination of possible dynamic executions of a system [8]. This tool provides an inexpensive mean of fault detection prior to verification by the model checker which is computationally heavy because it covers the exhaustive dynamic behavior of the system [8]. The model checker is an exhaustive search that checks reachability, safety and liveness properties by exploring the state-space of a system.

The two main design criteria for Uppaal are efficiency and ease of usage [7]. The model checker efficiency is achieved with the application of on-the-fly searching and a symbolic technique that reduce verification problems to that of solving simple constraint systems [9, 10].

Since its first release in 1995, UPPAAL has been constantly updated in order to meet requirements arising from the various case studies. Uppaal has been developed with client-server architecture, allowing it to be split into two components: a graphical user interface (GUI) and a model checking engine. The client is developed in Java and

enables the user to design and validate the system through a user-friendly graphical user interface. Instead, the server (model checking engine) is compiled specifically for some operating systems such as Windows, Linux, Solaris and OS X [6].
The subdivision into client and server allows execution on different machines that communicate via TCP/IP. This is very useful when the model checking engine runs complex models that require high computing and memory capability. Thanks to the rich offer of cloud services available today, it's possible to use resources needed only for the time required to validate the model.

2.1 Timed Automata

A timed automaton is a finite state machine extended with clock variables. All the clocks progress synchronously with the same rate and they are initialized with zero when the system is started. In Uppaal a system is modelled as a network of timed automata in parallel. The model can be extended with user-defined variables that are defined and used as in common programming languages. A *state* of the system is defined by the locations of all automata, the values of the variables and the clock values [4]. Every automata may have edges where each of them represents a transition that can be taken, separately or synchronise with other automaton when the guard on the edge is satisfied. The guard allows to define the conditions for which the transition is enabled and can be taken. When a transition is taken, the corresponding edge leads to a new state. Every edge has a source location from which it starts, and a destination location that is reached when the respective transition is taken.
It's also possible to define local timing constraints for the locations called *location invariants* [11]. When the location invariants are defined, an automaton may remain in a location as long as the invariant condition of the location is satisfied.

Assume the following notations [4]: C is a finite set of clocks and *B(C)* is a set of clock constraints compose by the conjunctions of conditions of the form $x \bullet c$ or $x - y \bullet c$, where $x, y \in C$, $c \in \mathbb{N}$ and $\bullet \in \{<, \leq, =, \geq, >\}$.

Definition 2.1 (Timed Automaton) [4]: A timed automaton A is a sextuple (L, I_0, C, A, E, I) where

- L is a finite set of locations (or node),
- $I_0 \in L$ is the initial location,
- C is the finite set of clock variables,
- A is a set of actions, co-actions and the internal τ-action,
- $E \subseteq L \times A \times B(C) \times 2^{\mathrm{C}} \times L$ is a set of edges between locations with an action, a guard and a set of clocks to be reset, and
- I: $L \longrightarrow$ *B(C)* assigns invariants to locations

To keep track of the changes of clock values, we use the functions $u : C \to \mathbb{R}_+$ called clock assignments [11]. Let $u_0(x) = 0$ for all $x \in C$ and $\mathbb{R}^{\mathrm{C}}$ be the set of clock assignments. Let u denote clock assignments function, and use $u \in g$ to indicate that the clock values denoted by u satisfy the guard g.

Definition 2.2 (Semantics of Timed Automaton) [4]: Let (L, I_0, C, A, E, I) be a timed automaton. The semantics is defined as a labelled transition system $\langle S, s_0, \rightarrow \rangle$, where $S \subseteq L \times \mathbb{R}^{\mathrm{C}}$ is the set of states, $s_0 = (l_0, u_0)$ is the initial state, and $\rightarrow \subseteq S \times (\mathbb{R}_+ \cup A) \times S$ is the transition relation such that:

- $(l, u) \xrightarrow{d} (l, u + d)$ if $\forall d' : 0 \leq d' \leq d \Longrightarrow u + d' \in I(l)$, and
- $(l, u) \xrightarrow{a} (l', u')$ if there exists $e = (l, a, g, r, l') \in E$ s.t. $u \in g$, $u' = [r \mapsto 0]u$, and $u' \in I(l')$

where for $d \in \mathbb{R}_+$, $u + d$ maps each clock x in C to the value $u(x) + d$, and $[r \mapsto 0]u$ denotes the clock valuation which maps each clock in r to 0 and agrees with u over $C \backslash r$.

Figure 2.2 illustrates an example of a timed automaton semantics. From the initial location A, we can choose to take an action or delay the transition. This Figure shows how a delay transition can lead to an invalid state due to the violation of the invariant for the location B.

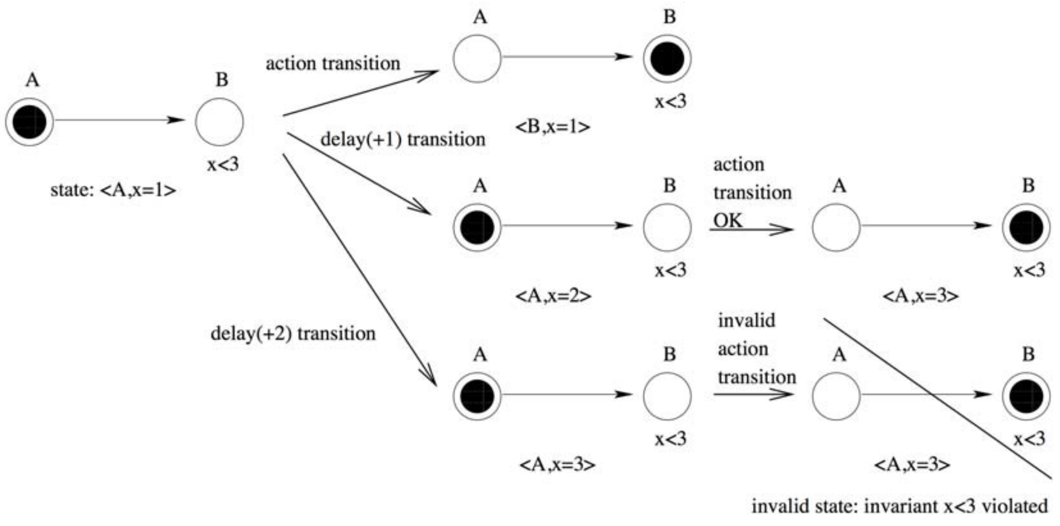


Figure 2.2: Semantics of timed automata [4].

In agreement with the definition there are two types of transitions between states: *delay transition* where the automaton may delay for some time and *action transition* when the automaton follows an enabled edge [11].

A model of a system is often composed into a *network of timed automata* over a set of clocks and actions. The invariant functions are composed into a common function over location vectors $I(\bar{l}) = \wedge_{\mathrm{i}} I_{\mathrm{i}}(l_{\mathrm{i}})$, where $\bar{l} = (l_1, ..., l_{\mathrm{n}})$ is a location vector. As indicated in [4], $\bar{l}[l'_{\mathrm{i}}/l_{\mathrm{i}}]$ denotes the vector where the ith element l_{i} of $\bar{l}$ is replaced by l'_{i}.

Definition 2.3 (Semantics of a network of Timed Automata) [4]: Let $A_{\mathrm{i}} = (L_{\mathrm{i}}, {l_{\mathrm{i}}}^0, C, A, E_{\mathrm{i}}, I_{\mathrm{i}})$ be a network of n timed automata. Let $\bar{l}_0 = ({l_1}^0, ..., {l_{\mathrm{n}}}^0)$ be the initial location vector. The semantics is defined as a transition system $\langle S, s_0, \rightarrow \rangle$, where $S = (L_1 \times \cdots \times L_{\mathrm{n}}) \times \mathbb{R}^{\mathrm{C}}$ is the set of states, $s_0 = (\bar{l}_0, u_0)$ is the initial state, and $\rightarrow \subseteq S \times S$ is the transition relation defined by:

- $(\bar{l}, u) \xrightarrow{d} (\bar{l}, u + d)$ *if* $\forall d' : 0 \leq d' \leq d \Longrightarrow u + d' \in I(\bar{l})$.
- $(\bar{l}, u) \xrightarrow{a} (\bar{l}[l'_{\mathrm{i}}/l_{\mathrm{i}}], u')$ *if there exists* $l_{\mathrm{i}} \xrightarrow{\tau g r} l'_{\mathrm{i}}$ *s.t.* $u \in g$, $u' = [r \mapsto 0]u$ *and* $u' \in I(\bar{l}[l'_{\mathrm{i}}/l_{\mathrm{i}}])$.
- $(\bar{l}, u) \xrightarrow{a} (\bar{l}[l'_{\mathrm{j}}/l_{\mathrm{j}}, l'_{\mathrm{i}}/l_{\mathrm{i}}], u')$ *if there exist* $l_{\mathrm{i}} \xrightarrow{c?g_{\mathrm{i}}r_{\mathrm{i}}} l_{\mathrm{i}}$ *and* $l_{\mathrm{j}} \xrightarrow{c!g_{\mathrm{j}}r_{\mathrm{j}}} l'_{\mathrm{j}}$ *s.t.* $u \in (g_{\mathrm{i}} \wedge g_{\mathrm{j}}), u' = [r_{\mathrm{i}} \cup r_{\mathrm{j}} \mapsto 0]u$ *and* $u' \in I(\bar{l}[l'_{\mathrm{j}}/l_{\mathrm{j}}, l'_{\mathrm{i}}/l_{\mathrm{i}}])$.

where *c!* and *c?* represent the synchronisation between two or more timed automata.

2.1.1 Timed Automata Extensions

The Uppaal description language extends the definition of timed automata with the following main features:

Templates automata are defined with a set of parameters [4], e.g. int or synchronisation channel, that are replaced by a given argument in the process declaration.

Binary synchronisation channels are declared as *chan name*, e.g. *chan c;*. An edge labelled with *c!* represents the sender that synchronises with the receiver represented by an edge labelled with *c?*. A synchronisation pair is chosen non-deterministically if several combinations are enabled [4] and not necessarily belong to the same template.

Broadcast channels are declared as *broadcast chan name*, e.g. *broadcast chan c;*. In this type of synchronisation one sender *c!* can synchronise with an arbitrary number of receivers *c?*. When some receiver can synchronise with the sender, they must do so. If there are no receiver, then the sender can still execute the *c!* action [4] (never blocking).

Urgent synchronisation channels are declared by adding the keyword *urgent* as a prefix of the channel declaration. The urgent synchronisation transitions should be taken as soon as they are enabled. If an edge uses urgent channels for synchronisation it can not have time constraints, i.e. clock guards are not allowed.

Urgent locations are a particular locations where time can not pass. These are semantically equivalent to adding an extra clock x, that is reset on all incoming edges, and having an invariant $x \leq 0$ on the location [4].

Committed locations are used to model atomic sequences of actions [11] and are more restrictive on the execution than urgent locations. If any process is in a committed location then only transitions (synchronisation or not) starting from a committed location are allowed [11]. This implies that process in committed locations may be interleaved only with other processes in committed locations.

User functions has C-like syntax but without pointers. User functions can be defined global or local to templates and only local functions can access template parameters.

Moreover, the extension provides the possibility of declaring constants, integer and bounded integer variables, arrays, record types and custom types.

2.1.2 System behaviour definition in Uppaal

In Uppaal the edges of the automata can be enriched with four types of optional labels: select, guard, synchronisation and update. In addition, the locations may be extended with invariants labels, which are conditions expressing constraints on the clock values. The exhaustive BNF is in the appendix A.

Select label defines variables accessible only to the associated edge and they take a nondeterministic value in the range of their respective types [4]. This label contains a comma-separated list of expressions of the form *name : type*, which express name and type of the variables, respectivelly.

Guard label allows to define conditions on the values of integer variables, clocks and constants (or arrays of these types) that must be satisfied in order to enable the associated transition [8]. A guard may call a user functions (side effect free) that return a boolean value. These conditions must be side effect free, evaluate to a boolean and clocks (or clock differences) are only compared to integer expressions [4].

Synchronisation label allows to define roles (sender or receiver) for the synchronisation between two or more automata. This label is on the form *Expression!* or *Expression?*. The expression must be side effect free, evaluate to a channel, and only refer to integers, constants and channels [4].

Update label contains a list of side effect expressions, each separated from the other by a comma. In this case an expression must only refer to clocks, integer variables and constants and only assign integer values to clocks [4]. Moreover, it may call user functions.

Invariant label is a conjunction of conditions of the form $x \bullet i$ where $\bullet \in \{<, \leq\}$, x is a clock and i evaluates to an integers. The invariant may call a side effect free user function (global or local) that returns a boolean value.
An invariant must be side effect free and only refer to clocks, integer variables and constants [4].
An example is shown in Figure 2.2, where the location B has the invariant $x < 3$. This example shows that the invariant violation leads to an invalid state.

Figure 2.3 illustrates an example of timed automata definitions. The automaton on the left has a loop with guard $x >= 2$, which activates the transition when the value of the clock x is greater than 2. When this loop is taken, the synchronisation between the two automata is performed through the binary channel *reset*. An example of update label is visible in the edge that goes from location *taken* to *idle* in the right automaton.

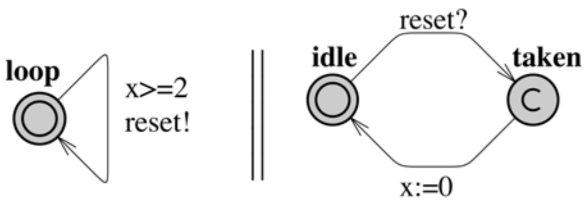


Figure 2.3: Example of timed automata definitions with Uppaal [4].

2.2 Verifying properties with Uppaal

The Uppaal's model checker engine is designed to verify system requirements with a subset of TCTL formula [12] for networks of timed automata. TCTL allows to express the requirement specification in a machine readable and formally well-defined language. This language consists of *path formulae* and *state formulae*, and should be one of the following forms [11]:

- $A[\,]\phi$ - for all paths ϕ always holds.
- $E <> \phi$ - there exists a paths where ϕ eventually holds.
- $A <> \phi$ - for all paths ϕ will eventually hold.
- $E[\,]\phi$ - there exists a path where ϕ always holds.
- $\phi \leadsto \psi$ - whenever ϕ holds ψ will eventually hold.

where the local properties ϕ and ψ can be checked locally on a state.

A state formula is an expression (see Appendix A) that can be evaluated on a state without considering the behaviour of the system [4], these expressions must be side effect free. Also a state formula allows to verify whether a process is in a specific location using an expression on the form *P1.l0*, where *P1* is a process and *l0* is a location.
Uppaal uses the keyword *deadlock* as a special state formula for the deadlock. This formula is satisfied for all deadlock states. When a state has no outgoing transitions from itself and any of its delay successors, then it is a deadlock state.

Uppaal supports different path formulae: reachability, safety and liveness. These formulae are shown in Figure 2.4.

Reachability property verifies whether a state formula ϕ *possibly* can be satisfied by any reachable state. Reachability property is satisfied if there is a path starting at the initial state where ϕ is eventually holds.
In Uppaal, these properties are expressed using the syntax $E <> \phi$.

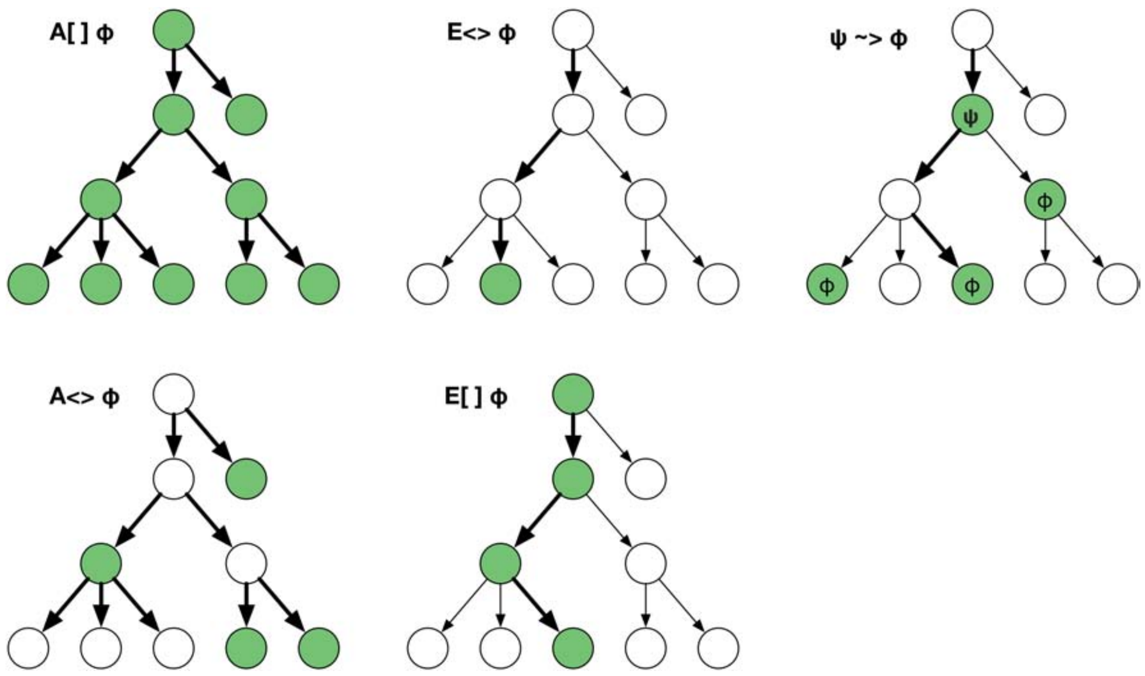


Figure 2.4: The Uppaal's path formulae. Bold edges denote the paths on which the formulae are evaluated. The filled states are those for which a given state formulae ϕ holds.

Safety properties are on the form "something good will always happen", or negatively as "something bad will never happen".
Let ϕ be a state formula. The $A[]\phi$ path formulae expresses that ϕ should be true in *all* reachable states, whereas $E[]\phi$ expresses that there should exist a maximal path such that ϕ il always true [4]. A maximal path is a particular path that is infinite or where the last state has no outgoing transitions.
Uppaal uses the positive formulation of safety property with the syntax $A[]\phi$ and $E[]\phi$, respectively.
Note that $A[\,]\phi = \bar{E} <> \bar{\phi}$.

Liveness properties are on the form "something will eventually happen" [4] and are expressed with the path formulae $A <> \phi$. An alternative formulation is the response property which is interpreted as whenever ϕ is satisfied, then eventually ψ will be satisfied. The response property is defined with the path formula $\phi \leadsto \psi$. In Uppaal, these properties are expressed using the syntax $A <> \phi$ and $\phi -- > \psi$, respectively.
Note that $A <> \phi = \bar{E}[\,]\bar{\phi}$.

Chapter 3

The Nuclear Experiment

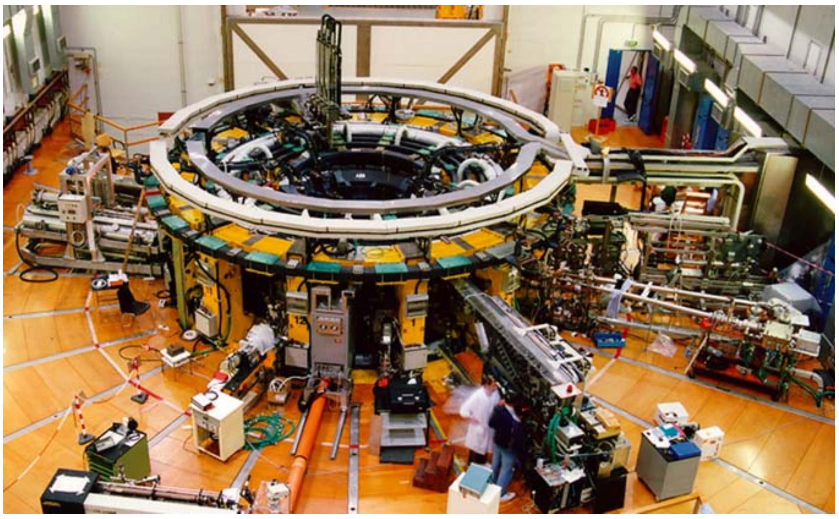


Figure 3.1: The RFX machine for a nuclear fusion experiment.

The RFX (Reversed Field eXperiment) machine is a medium-size magnetic confinement experiment designed to study this phenomenon and to reduce the mechanisms that destabilise plasma in high current regimens. Experiments are performed using plasmas of hydrogen, deuterium or helium. RFX can operate in two configurations: Reversed Field Pinch and Tokamak.

The RFX experiment was inaugurated in 1992. The knowledge gained in the past years led to a deep modification of the reactor between 1999 and 2004. The various optimisations introduced over these years allow to execute experiments with high plasma current, up to 2 million ampere (2MA). One of the most innovative changes is the introduction of an active control system for magneto hydrodynamic (MHD) instabilities. This new feature contributed to several important results such as the experimental evidence of a new helical equilibrium, published in Nature Physics in August 2009 [13]. The control system was further upgraded in 2012. It is today the most advanced plasma stability control system for this type of nuclear fusion experiments, allowing it to run more sophisticated control algorithms than the previous version.
The main technical features of the RFX machine are briefly listed in the Table 3.1.

Vacuum chamber material	Inconel 625
First wall material (2016 tile)	graphite
Shell material (3mm)	copper
Conductive material of the coils	copper
Insulating material of the coils	fiberglass and kapton
Greater radius of the toroid	2 m
Smaller radius of the toroid	0,5 m
Vessel volume	8,31 m^3
Maximum plasma current	2 MA
Maximum toroidal magnetic field	0,7 T
Vessel vacuum level	10-12 bar
Pulse duration	$>$0.5 s
Electron Temperature	$>$1.2 keV

Table 3.1: Main technical characteristics of the RFX machine [14].

3.1 The real-time control system

In recent years, nuclear fusion experiments use real-time control systems to make the plasma confinement more effective. Plasma position control is possible by coils that generate different magnetic fields: toroidal, poloidal, vertical and radial. A vertical field is used to shape the plasma column, otherwise the radial field is used for the control of the plasma MHD instabilities.

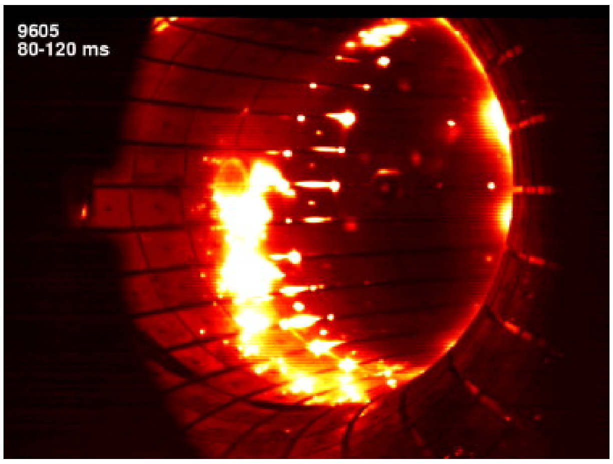


Figure 3.2: Interaction between plasma and wall due to plasma instabilities [15].

However, the conducting shell surrounding the plasma container performs corrective actions. This happens because the plasma instabilities generate counterbalancing currents inside the conducting shell [1]. The advantage is that this effect is immediate and with a response time lower than a digital control system, but this corrective effect ends soon, due to the finite resistance of the shell and penetration time of the magnetic field [1]. The conductive shell without the real-time control system is unable to fully suppress plasma instabilities that increase the interaction between plasma and the container (Figure 3.2), eventually leading to a disruption.

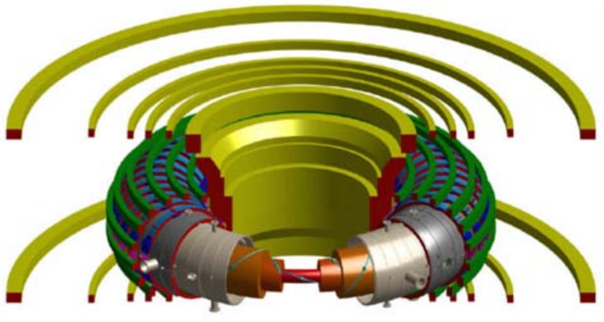


Figure 3.3: Magnetic fields in the RFX experiment. The toroidal coils are shown in blue and the field shaping coils are shown in green. The yellow coils are used to induce plasma current but not for control [1].

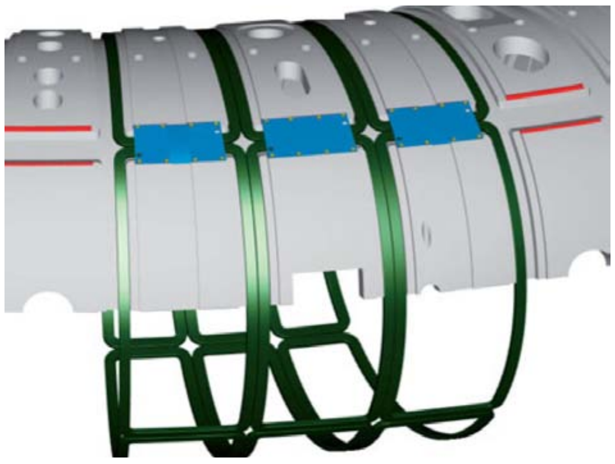


Figure 3.4: The saddle coils mounted around a torus section [15].

The RFX experiment performs two control tasks: axisymmetric control and MHD stability control [16]. The axisymmetric control allows to control the position and shape of the plasma column and other parameters such as magnetic field reversal and plasma current. This is possible by acquiring information from a set of 128 electromagnetic probes and by driving the current flowing into toroidal and field shaping coils (see Figure 3.3), that generate toroidal field component and vertical field components, respectively. The MHD control task is designed to limit the plasma instabilities. Its inputs derive from a set of 192 saddle coils located around the torus as a 48x4 array, as shown in the Figure 3.4. The saddle coils allow to acquire the radial magnetic field. The 2D Fouries transform is used to compute the spatial modes (m,n) that are the input for the control

algorithm, which computes 192 reference signals for the power supply units feeding the actuator coils located in the same array [15]. Any physical quantities used for control computation are not directly derived from measurements, therefore a pre-elaboration of the signals acquired by the probes is performed.
In practice, the saddle coils are used to provide localised corrections, whereas the toroidal and field shaping coils are used for correcting global parameters which are the same along the poloidal direction.

An important factor for a control system is the maximum allowable latency. In the RFX experiment, the plasma instabilities that are too fast to be handled by the digital control system are compensated by the conducting shell. This compensation ends after a few tens of microseconds and the intervention of the control system is then required [1]. For this experiment, the MHD control system must provide a response time less than 2ms [15] which is faster than the axisymmetric control system ($<$10ms).

3.1.1 Hardware organisation

The first version of the RFX experiment control system was replaced by a new one in 2012, described in more detail in [3]. The new hardware organisation is shown in Figure 3.5 and is based on a multi-core server (HP ProLiant DL370) hosting 12 2.8Ghz cores with 12GB RAM. The data acquisition is supervised by four VME chassis, each hosting a Motorola MVME 5500 single board computer with 1Ghz clock speed and an analog to digital converter (ADC). All VMEs are interconnected with the multi-core server via Gigabit Ethernet. In this new hardware organisation, the VME processors only acquire raw data and send it to the multi-core server via UDP packets [17]. The raw data packets transmission over the network causes communication latency that affects the total latency of the system.
The server executes the control algorithms and is connected to four PXI racks via fiber-optics based bus extenders where each bus has a dedicated PCIe port. Each PXI rack generates the reference waveforms and hosting three National Instruments NI6723 digital to analog converter (DAC) with 32 channels [17]. As shown in the Figure 3.5, two sets of DAC are used to generate the 192 reference waveforms for the 192 saddle coils that are required for MHD control and the correction of the radial field. This configuration with two different PXI racks allows to parallelize the generation of the reference waveforms and reduces the latency due to the conversion time.
Other set of DAC device, hosted in two different PXI racks are used for the generation of the reference waveforms for toroidal and axisymmetric control.

3.1.2 Software organisation

Before 2012, the software architecture of the control system was based on VxWorks, a commercial real-time operating system (OS). A completely different approach was adopted for the new system and this was possible by the alternatives offered by Linux and the new framework MARTe [2]. A key element for adopting Linux instead VxWorks is the introduction of the PREEMPT_RT Linux Patch (kernel version 2.6). This patch provides preemptible critical sections, priority inheritance, in-kernel semaphores and spinlocks and preemptible interrupt handlers [17].

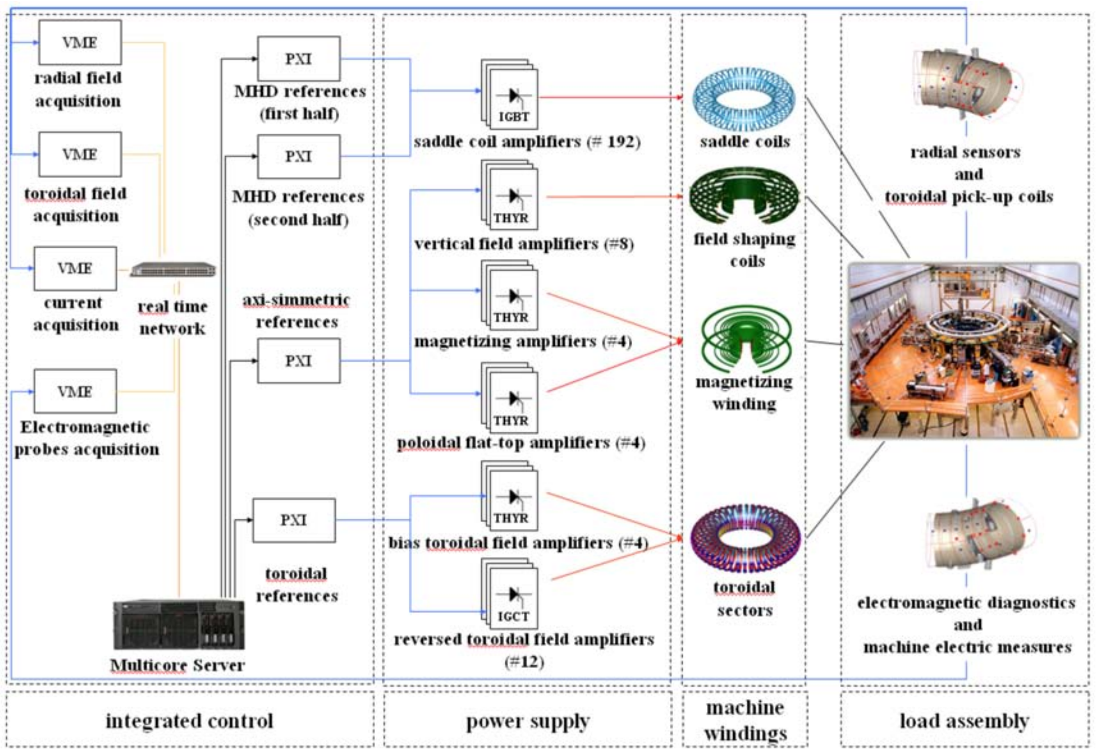
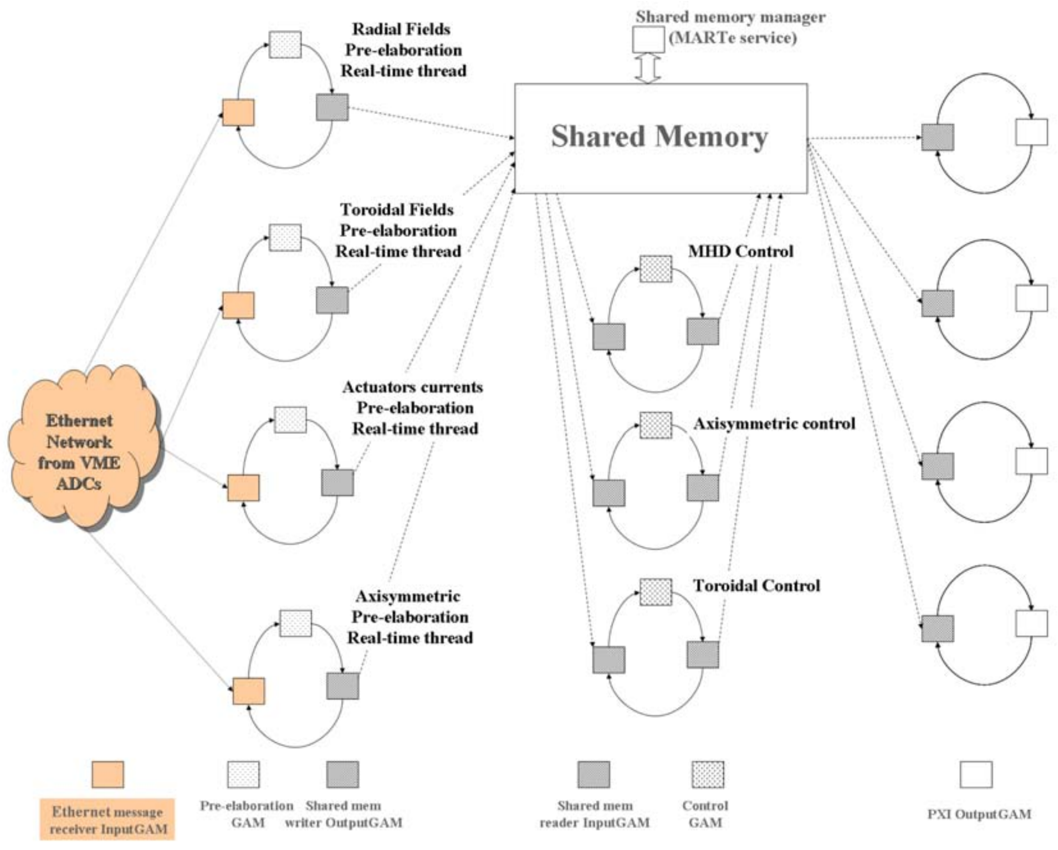

VME
radial field acquisition
VME
toroidal field acquisition
VME
current acquisition
real time network
VME
Electromagnetic probes acquisition
Multicore Server
PXI
MHD references (first half)
PXI
MHD references (second half)
axi-simmetric references
PXI
PXI
toroidal references
IGBT
saddle coil amplifiers (# 192)
THYR
vertical field amplifiers (#8)
THYR
magnetizing amplifiers (#4)
THYR
poloidal flat-top amplifiers (#4)
THYR
bias toroidal field amplifiers (#4)
IGCT
reversed toroidal field amplifiers (#12)
saddle coils
field shaping coils
magnetizing winding
toroidal sectors
radial sensors and toroidal pick-up coils
electromagnetic diagnostics and machine electric measures
integrated control
power supply
machine windings
load assembly

Figure 3.5: The hardware organisation of the new real-time control system [17].

The software environment chosen for the new system is MARTe [3], a framework for real-time control used in several fusion experiments [2]. This framework is written in C++ and developed at JET [18] with the collaboration of RFX [3]. MARTe is a flexible and generic engine for the implementation of multithreaded control and provides the supervision of the data movement and of the real-time threads. Every real-time thread runs cyclically a set of Generic Application Modules (GAMs). The GAMs allow to integrate in MARTe some specific components for data acquisition and control algorithms. All the configurations concerning MARTe are defined in a user-friendly text file, thus allowing an easy reconfiguration of the system.
MARTe has an interface called IOGAM that specifies the data flow from/to hardware devices and provides a customisation of read, write and synchronisation operations for a specific hardware. It also provides InputGAM and OutputGAM to perform generic input and output operations. Every GAMs is associated with a specific IOGAM to use a particular hardware. In the RFX experiment, two IOGAM classes have been developed for receiving the raw data (UDP packets) over Ethernet and for generating the reference waveforms via DAC devices. Moreover, GAMs have been developed for the derivation of the bi-dimensional Fourier transform for the raw data and for MHD, toroidal and axisymmetric control [17].
In MARTe, all threads share the same address space because they belong to a single process. This allows the inter-thread communication directly in memory and the data consistency is ensures through locking and synchronisation mechanisms between threads. This two mechanisms are implemented by a service component and the basic functions for writing and reading shared data are implemented with an IOGAM class.

As shown in Figure 3.6, the MARTe configuration of the RFX control system defines 11 real-time threads. Four real-time threads acquire raw data packets sent from the VME system through a specific InputGAM. The InputGAM of these threads are connected to another GAM which performs pre-elaboration of input data. There are two type of pre-elaboration: spatial bi-dimensional Fourier (first three threads) and the derivation of several plasma parameters (last thread). The MHD, toroidal and axisymmetric control are performed by the three real-time threads in the middle. These threads receive data from the pre-elaboration threads and produce the output values for the reference wave-forms [17]. The last four real-time threads supervise the DAC conversion, performed by four PXI racks using a specific OutputGAM.



Radial Fields Pre-elaboration Real-time thread
Shared memory manager (MARTe service)
Shared Memory
Toroidal Fields Pre-elaboration Real-time thread
MHD Control
Actuators currents Pre-elaboration Real-time thread
Axisymmetric control
Ethernet Network from VME ADCs
Toroidal Control
Axisymmetric Pre-elaboration Real-time thread
Ethernet message receiver InputGAM
Pre-elaboration GAM
Shared mem writer OutputGAM
Shared mem reader InputGAM
Control GAM
PXI OutputGAM

Figure 3.6: Real-time threads organisation of the RFX experiment [1].

The software organisation represents a pipeline: the Figure 3.7 illustrates the data flow for the MHD control. At every acquisition clock edge (Acq clock i in the figure) data are sampled and read by the VMEs (VME Acquisition), then the data are sent via UDP packets (Data communication). Three real-time threads read the incoming raw data packets and perform the pre-elaboration on these data (CPU core [0, 2] Pre Elaboration). As shown in the figure, all of these operations are performed in parallel. The next acquisition clock edge may occur before pre-elaborated data are ready [15]. When data pre-elaboration are completed, a specific real-time thread will carry out

MHD control (CPU core3 Elaboration). The output data of the MHD control are exchanged with two real-time threads that generate in parallel the output reference waveforms.
In this system, the latency is defined as the time between the acquisition clock edge and the generation of the last DAC output for that cycle [3].

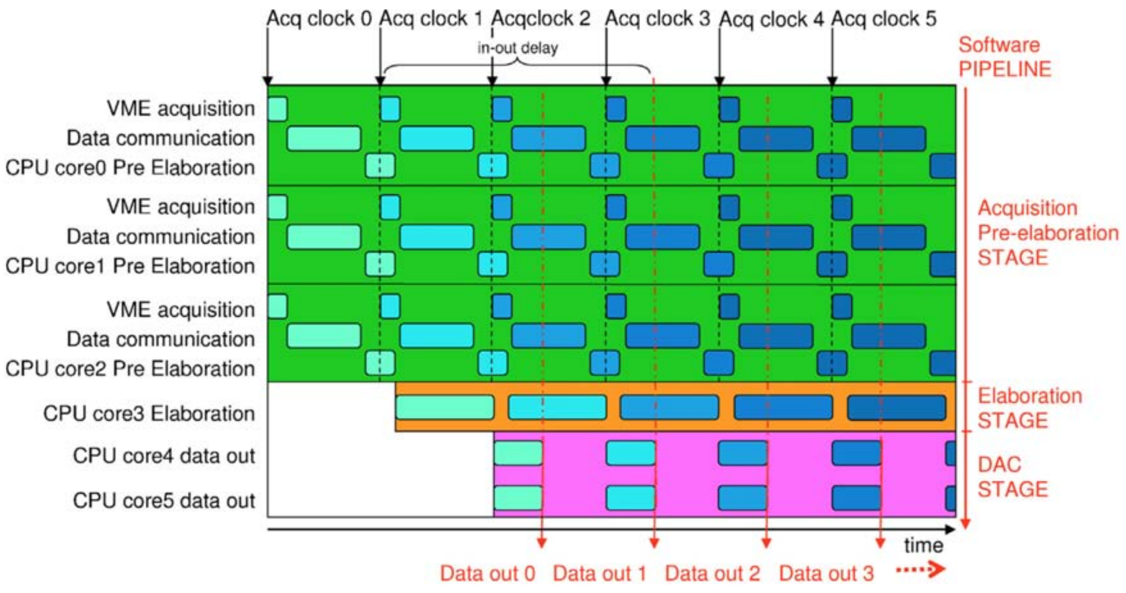


Figure 3.7: Pipelined organisation of MHD control task [15].

3.1.3 Core assignment

The multi-core server used for the RFX experiment has 12 core, this allows a fixed core assignment: 11 cores are dedicated for the 11 real-time threads and the left core is used for the system activity. Fixed core assignments is realised by the ISOLCPU Linux boot parameter and with the *cpu_setaffinity()* system call. ISOLCPU boot parameter forces the Linux scheduler to use only specific cores for system activity. The system call *cpu_setaffinity()* is useful for explicitly assign threads to cores. Fixed assignment removes most jitter in latency due to non determinism in core assignment which may occur if the Linux scheduler is configured to use all cores [1]. Additionally, with a exclusive fixed core assignment, where each core is assigned to a single real-time thread, there are no context switches among the threads.
In large systems, the number of processes may be higher than the number of cores and in this case it is required that multiple threads share the same core. Two approaches are possible:

- Assign more than one thread to a given core. The freedom of the Linux scheduler is limited but now it handles the required context switches for the threads assigned to the core.
- Enable processor hyperthreading in order to double the number of cores usable by the OS. This mode exploits the replicated components of the cores, instead non-duplicate hardware resources are contended by the threads.

The performance obtained with the different configurations can be observed in [1].

Chapter 4

The Linux Scheduler

Every operating system (OS) has a scheduler that assigns cores to various ready-to-run processes in the system. Available processors can be considered as shared resources that are contended among the ready processes. The scheduler manages these shared resources and allocates them temporarily to a task (for a time period called time slice). The time slice length estimation is critical for balancing system performance vs process responsiveness. In addition, the scheduler must minimise the response time for the most critical tasks (real-time).

The scheduler is one of the kernel components that selects the next process to be execute. With the introduction of the kernel preemtability in Linux kernel 2.6, the Linux operating system responsiveness has been increased and now is considered a soft real-time OS [19]. In this type of real-time systems, the system response times to external events can be considered almost always bounded and more deterministic than the previous version, but this can not be guaranteed for all cases. The PREEMPT_RT Linux patch further improves system responsiveness with the following novelties:

- preemptible critical section;
- preemptible interrupt handlers;
- priority inheritance for kernel semaphores.

In Linux kernel 2.6, critical sections in the kernel are protected by a mechanism implemented as spinlocks, in PREEMPT_RT this mechanism is implemented with a particular semaphore called rt-semaphore. Real-time performance can be reduced due to priority inversion that can indefinitely delay a high-priority task. This problem is avoided with the priority inheritance that allows the scheduler to boost the priority of a specific task. Another enhancement introduced by the patch concerns the length of the interrupt service routines (ISR). In practice, preemptible interrupt handlers are implemented by performing very short ISR and shifting the remaining driver actions into real-time threads which are managed by the Linux scheduler according to their priority [19].

Before the 2.6 kernel, the scheduler used algorithms with complexity O(n) which represents a large limitation when many tasks were active. With this type of scheduler when the system is very loaded, the processor can be used predominantly by scheduling

algorithm, leaving little CPU time to tasks.
The early 2.6 scheduler (called *O(1) scheduler*), developed and implemented by Ingo Molnár, was designed to solve the scalability problem of the previous scheduler version. The O(1) scheduler manages tasks with *run-queues*; there are two queues (one for active task and one for expired task) for each CPU. This task organisation allows to identify the task to execute next simply by dequeuing the first task from the specific per-priority run queue.
CFS[20] scheduler replaces the previous O(1) scheduler for user tasks, and introduces task organisation through a time-ordered red-black-tree instead of the run/expired queues [20]. In addition, CFS introduces the scheduler classes where each task belongs to a specific class, which defines the scheduling policy for a task.

4.1 2.6 scheduling structure

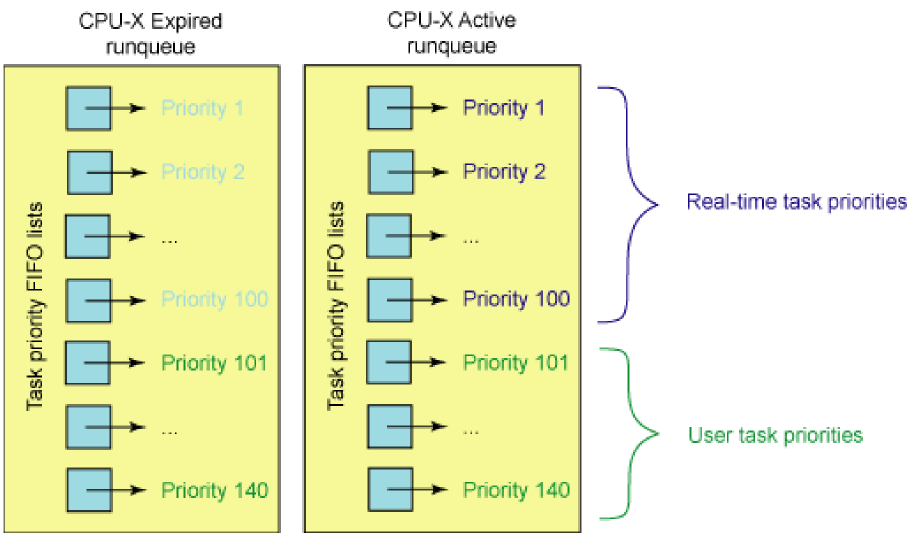


Figure 4.1: 2.6 scheduler run-queue structure [21].

In the 2.6 scheduler each CPU has two run-queues, one *active run-queue* and one *expired run-queue*. These two queues are made up of 140 priority list that are served with FIFO policy, as shown in the Figure 4.1. The first 100 priority list (with the highest priority) are reserved for real-time tasks, and the remaining 40 are used for user tasks [21]. The executable tasks are added at the end of their respective active run-queue's priority list. For each task is defined a time slice and when the task on the active run-queue uses all of its time slice, it's moved into the expired run-queue. Whit this structure, the scheduler selects the task to be executed on the highest priority list. Note that the time it takes to select the task to execute depends only on the number of priorities, so the time to schedule is fixed and deterministic. For this reason the scheduler is O(1) respect to the number of active task.
Linux 2.6 scheduler allows preemption, this means that a low priority task will not run until there are high priority task ready to run. The scheduler can preempt the lower priority task, when this happens the task is inserted back on its priority list and rescheduled.

This scheduler version also offers dynamic task prioritisation and symmetric multiprocessing (SMP) [22] load balancing.

4.2 SMP load balancing

The symmetric multiprocessing (SMP) architecture is composed by two or more identical CPUs connected to each other through a shared memory. When a task is created in this type of systems, it's inserted on a CPU's run-queue. This assignment is not based on the computing time required to complete the task, and therefore the initial allocation of tasks to CPUs is almost always suboptimal. It becomes important to keep the workload balanced between the CPUs, moving a task from an overloaded CPU to an underloaded one. Generally, load balancing is performed every 200ms [21], but this value can be changed as needed.
A task migration has a disadvantage that data for a task is not placed in the new CPU's local cache. This requires to pull its data into the new cache space.

Chapter 5

Model implementation

The model implementation in Uppaal requires the creation of templates that represent the different components of the real-time system.
This model declares some global variables which allow it to be versatile and configurable. The main variables used by the templates are the following:

- const int N : is the number of VME machines, pre-processing threads and DAC supervisory threads
- const int M : is the number of control algorithm threads
- const int *num_ cycle* : is the number of cycles, where a cycle begins with the generation of raw data and ends with the generation of the reference waveforms
- const int *num_ core* : is the number of core available for real-time threads
- const int *num_ threads* : is the number of real-time threads
- clock *master_ clock* : is the main clock used to calculate the time between the start of the simulation and the deadlock

The model during the simulation performs a number of cycles equal to *num_ cycle*. Every component of the real-time system completes a cycle when the associated automaton returns to its initial location. When all the components of the model complete the cycles, the deadlock occurs. During dynamic execution, the model may have different evolutions due to the possible different interlaces possibilities between transitions. Each of these evolutions is characterised by a sequence of transitions that define the sequence of crossed states. By imposing a maximum number of cycles, the number of possible transitions and the size of the states sequence can be limited. This limitation reduces the calculation time required to validate model properties, because Uppaal has to check conditions on fewer evolutions and relative states.

5.1 RAW_Data template

The *RAW_Data* template models raw data acquisition by VMEs and the subsequent transmission of UDP packets over the Gigabit Ethernet. VMEs acquire data at a fixed frequency and incapsulate these data in an UDP packet before sending them. These operations are performed sequentially.
This template has the following parameters:

- urgent chan &*notify_end* : is the binary synchronisation channel to signal completion of RAW data transmission
- const int *t_pck* : is the sampling period and is calculated as $t\_pck = \frac{1}{sampling\_frequency}$
- const int *t_acq* : is the acquisition time of the VME machine
- const int *t_delay* : is the average transmission delay
- const int *t_jitter* : is the transmission jitter

In addition, an integer variable *count* and three clock x, y and t are declared. The *count* variable is required to count the number of acquisition cycles performed.

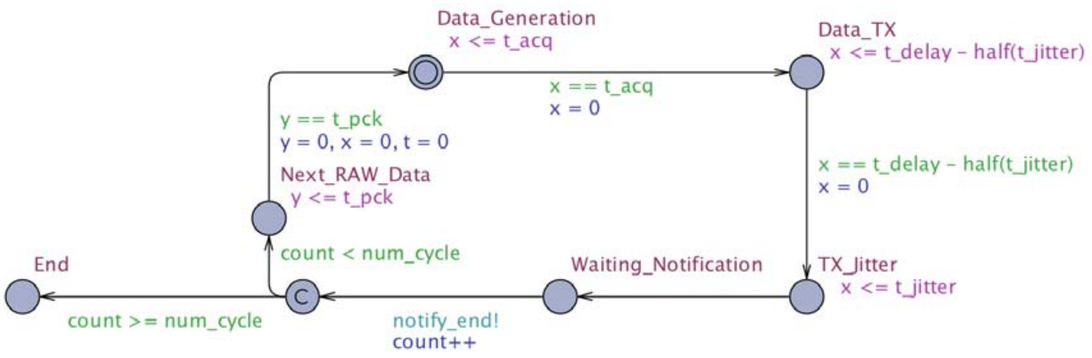


Figure 5.1: Automaton for raw data acquisition thread.

The Figure 5.1 shows the timed automaton for raw data acquisition, where the initial location is represented with two concentric circles. Starting from the initial location, the wait for the data acquisition is modelled simply by imposing the activation of the outgoing edge from that state only at a specific instant of time (*t_acq*). Moreover, the invariant defined for this location prevents to stay here longer than *t_acq*. Subsequently, there is the transmission of data that requires an average transmission time *t_delay* with jitter *t_jitter*. To simulate the presence of jitter is defined an invariant for *Data_TX* location, forcing the automaton to take the outgoing edge of that location after a time equal to $t\_delay - half(t\_jitter)$, where *half(int x)* is a function that returns the largest integer value less or equal to $x/2$. After this, the automaton is in the *TX_Jitter* location where it can wait for a maximum time *t_jitter* before taking the outgoing edge leading to the *Waiting_Notification* location.
After this phases it's necessary to notify the availability of data to the pre-processing thread for the preliminary processing. Notification is possible through the binary synchronisation channel *notify_end*, where the RAW_Data's edge has the role of sender,

while the receiver is represented by a pre-processing thread. The channel is defined as urgent, this ensures that the edge is taken immediately when its receiver is awaiting synchronisation. Note that the *Waiting_Notification* location can not be removed because the *TX_Jitter* location has an invariant that can be violated if its outgoing edge requires a not available synchronisation (with a specific pre-processing automaton) when the clock x is equal to *t_jitter*. This leads the systems into a illegal state. When synchronisation is complete, if the automaton has run a number of cycles equal to *num_cycle*, then the new data generation is interrupted by going in the *End* location. If the number of cycles executed is less than *num_cycle*, then the automaton waits for the expiration of time *t_pck* (counted by the clock y, initialised to 0 in the initial state) and repeats the execution from the initial location.

5.2 Pre_Proc template

The *Pre_Proc* template models the pre-processing of raw data and storage operation into shared memory.
This template has the following parameters:

- urgent chan &*start* : is the binary synchronisation channel to signal completion of RAW data transmission

- urgent chan &*notify_end[num_cycle + 1]* : is the binary synchronisation channels array to signal completion of the pre-processing operation

- urgent chan &*lock* : is the binary synchronisation channel for mutually exclusive access to shared memory

- urgent chan &*unlock* : is the binary synchronisation channel for memory lock release

- chan &*lock_request* : is the binary synchronisation channel to record the request for shared memory access

- urgent chan &*wait_lock* : is the binary synchronisation channel that signals the unavailability of shared memory

- urgent chan &*core_request* : is the binary synchronisation channel to require a core

- urgent chan &*core_assigned* : is the binary synchronisation channel to assign a core

- urgent chan &*core_release* : is the binary synchronisation channel to release a core

- const int *t_comp* : is the average calculation time for pre-processing

- const int *t_save* : is the average time needed for saving in shared memory

In addition, a clock y and an integer variable *count* are declared.

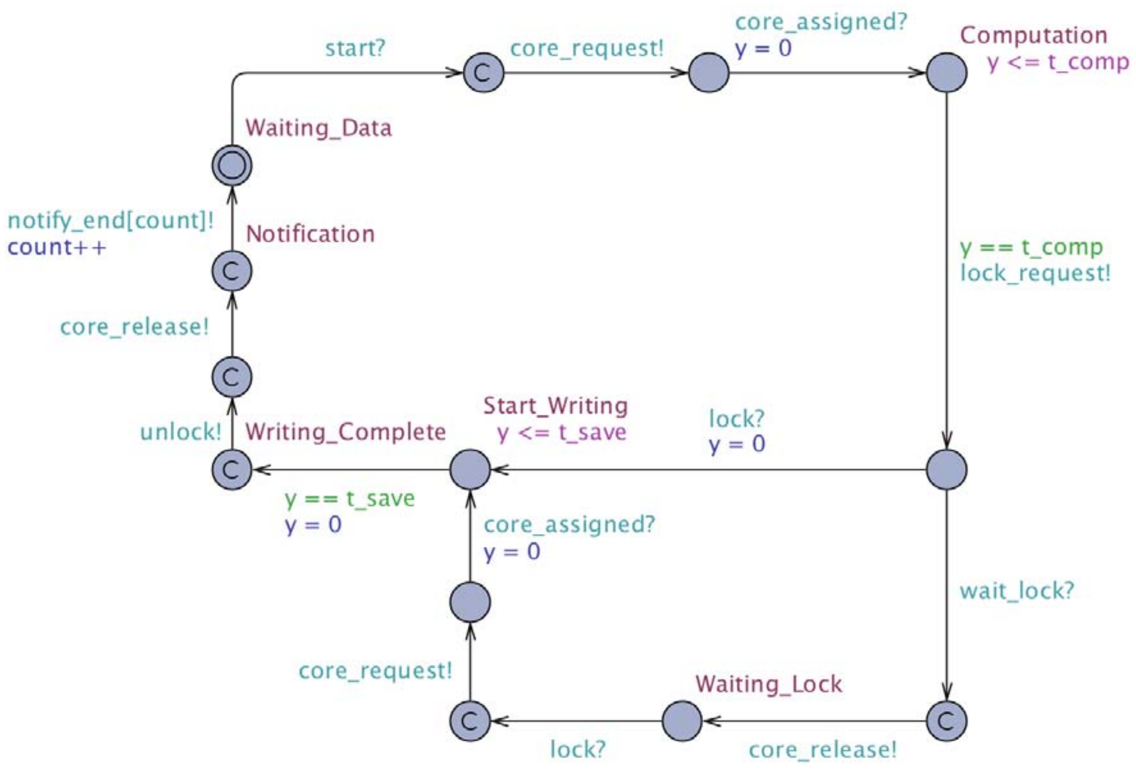


Figure 5.2: Automaton for pre-processing thread.

The Figure 5.2 shows the timed automaton for pre-processing computation, where *Waiting_Data* is the initial location. The outgoing edge for the initial location becomes active when the corresponding *RAW_Data* automaton is in the *Waiting_Notification* location. The next step modelling the request and assignment of a core for executing the pre-processing algorithm. The *notify_end* array has *num_cycle + 1* elements because at the end of the last cycle the automaton is in the *Waiting_Data* location and waits for synchronisation on the channel with index *num_cycle + 1*. So if the array contained only *num_cycle* channels the simulation would return an error.
The *core_request* channel allows to register the core request, otherwise *core_assigned* allows to active the corresponding edge when the scheduler assigns a core to that automaton. When the core is assigned, pre-processing of the data can start and this operation requires a fixed time equal to *t_comp*. When outgoing edge for the *Computation* location is taken, with the channel *lock_request* is possible to request the access to the shared memory (served with FIFO policy). Subsequently, the pre-processing thread saves the data in shared memory with mutually exclusive access. The automaton can take one of two edges that synchronise on the *lock* channel or on the *wait_lock* channel. Synchronisation via the *lock* channel is possible only if the memory space is not used in write mode by another pre-processing thread. If the shared memory is unavailable, synchronisation on channel *core_release* occurs. Note that both channels are urgent, the automaton can't take the outgoing transition after an arbitrary time. In the latter case, the automaton releases the core and waits for the memory lock. When the lock is

available, the core request is repeated and it waits the core assignment. In both cases the automaton arrives in the *Start_ Writing* location where it waits for *t_save* time units. Once the writing is completed, the memory lock and core are released through the *unlock* and *core_release* channels, respectively.
Now the automaton needs to report the availability of data to control algorithms. To do this, the automaton uses the channel with index *count* in the *notify_end* array. The array contains a channel for each pre-processing cycle, this avoids possible false data availability notifications due to the execution of different cycles for pre-processing and control algorithms threads.
Finally, the automaton returns to its initial location and waits for new raw data.
In this template some locations are committed, but this is not mandatory for the correct operation of the automaton. The automaton also works correctly without the committed property because all outgoing edge for the locations with this property require synchronisation on an urgent channel. The committed property is used to limit the possible interlacing of transactions that may occur during model simulation and verification. This eliminates some possible system evolutions that are not relevant, and results in a reduction in property verification times. Exclusion of some evolution is possible because interlacing between transitions involves only transitions that don't increase the clock values (transactions are instantaneous).

5.3 Shared_Memory template

The *Shared_Memory* template simulates the management of shared memory written by some threads. Write access to shared memory must be mutually exclusive and access list must be managed with FIFO policy.
This template has the following parameters:

- const int *A* : is the number of threads that want to use the shared memory
- urgent chan &*lock[N]* : is the binary synchronisation channels array to assign the memory lock to a specific thread
- urgent chan &*unlock[N]* : is the binary synchronisation channels array to release the memory lock
- chan &*request[N]* : is the binary synchronisation channels array to request access to shared memory
- urgent chan &*wait* : is the binary synchronisation channel to signal the lock unavailability

In addition, a *int[0, A-1] id_t* custom type and an integer variable *length* are declared. Every shared memory user has an integer identifier *i* in the range [0, A] and two channels *lock[i]* and *unlock[i]*. Instead, the *wait* channel is the same for all threads. The memory manager uses three functions: *enqueue()* to register the memory access request into the access list; *dequeue()* to remove the thread id from the list; and *next()* which returns the next thread identifier for the lock assignment.

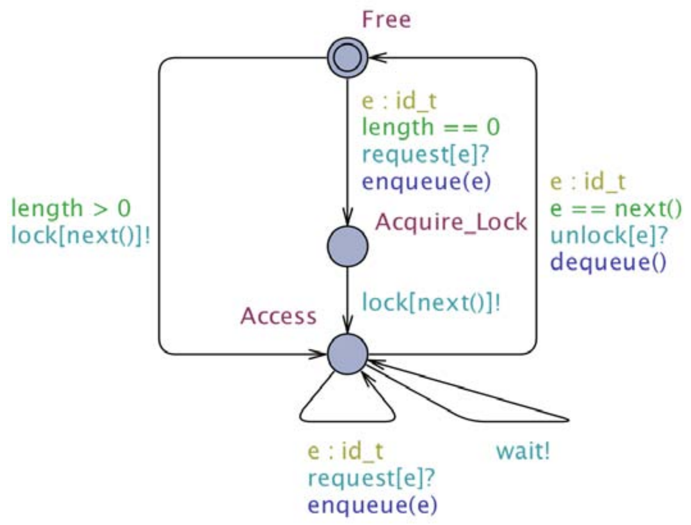


Figure 5.3: Shared memory implementation.

As shown in the Figure 5.3, the initial location, called *Free*, indicates the availability of write access to memory and has two outgoing edges. The edge leading to the *Acquire_Look* location is active only if the access list is empty ($length = 0$) and an automaton is awaiting synchronisation on the i-th channel of the *request* array. When the transition is taken, the thread id i is inserted into the access queue (in the first position because it is empty). In this case, the memory lock is assigned through the transition leading to the *Access* location. This transition allows the memory management to synchronise only with the automaton waiting on the i-th *lock* channel, where i is the first thread id in the access list returned by the *next()* function.
If $length > 0$ in the *Free* location, then the lock is assigned to the first thread in the access list.
Location *Access* represents occupied (write mode) memory status. This location has two loops that are required to register new memory access requests (through the channel *request[e]*) and to indicate that memory is used in write mode by another process (through the channel *wait*).
When a thread wants to release the memory lock, the transition outgoing from *Access* location is enabled. The transition removes the thread from the access list (with *dequeue()* function) and the memory comes back available and accessible for the next thread. The ability to release the memory lock is only available for the thread that owns the lock, the control is executed by setting a guard on the transition using the *next()* function.

The full implementation of the template functions is shown in appendix B (Listing 9.1).

5.4 Threads_Notification template

The *Threads_Notification* template handles input data availability notifications. It's used by pre-processing and control algorithms to signal completion of their execution and the output data availability in shared memory. At the same time, it's used by control algorithms and reference signal generators to know when their input data are available for processing. This synchronisation mechanism requires the configuration of dependencies between threads that generate output data and threads involved in their processing. The configuration is defined by an array of array with the following organisation:
$\{\{id_{0,0}, id_{0,1}, ..., id_{0,n-1}\}, \{id_{1,0}, id_{1,1}, ..., id_{1,n-1}\}, ..., \{id_{m-1,0}, id_{m-1,1}, ..., id_{m-1,n-1}\}\}$
where n is the number of threads that produce output data, m is the number of threads that await the input data and $id_{j,k} \in \{0, ..., m-1\} \cup -1 \ \forall j \in \{0, ..., m-1\}, \forall k \in \{0, ..., n-1\}$. In practice, each subarray is associated with a thread and contains the ids (in increasing order of value) of the threads that generate the data that it needs for computing. Often a thread only needs the data provided by a subset of the m generating threads. In this case the associated subarray contains an amount of thread id less than m, the remaining positions will be filled with the value *-1*. The Listing 5.1 shows an example where a thread waits for data generation by threads 0,1,2 and 3, a thread waits for data generated by threads 0 and 2, and a thread that only waits for data generated by the thread 3.

```
{{0, 1, 2, 3}, {0, 2, -1, -1}, {3, -1, -1, -1}}
```

Listing 5.1: Example of dependency configuration array for 4 data generation threads.

This template has the following parameters:

- const int *num_sender* : is the number of threads that notify data availability (n)
- const int *num_receiver* : is the number of threads awaiting receipt of the notification (m)
- urgent chan &*notify[N][num_cycle + 1]* : are the binary synchronisation channels array used to notify the data availability
- urgent chan &*start[N][num_cycle + 1]* : are the binary synchronisation channels array used by data generating threads to notify data availability
- const int *sources[N][N]* : is the input dependency configuration array described above

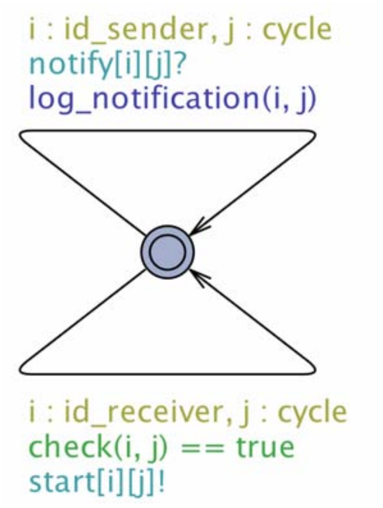


Figure 5.4: Automaton for synchronisation between threads.

This template has a single location with two loops needed to record data availability and to notify this availability to the waiting threads. As shown in Figure 5.4, the upper loop registers the data availability thanks to the *log_notification()* function. This allows to track which thread generated the data and in which cycle, ensuring that there are no overlapping notifications in different execution cycles. Threads notify the data availability using the appropriate *notify[i][j]* channels, where *j* indicates the cycle in which the thread operates and *i* indicates the thread id.
Threads that require input data remain awaiting synchronisation on the *start[i][j]* channel, where *i* indicates the thread id and *j* indicates the cycle in which the thread operates. As shown in the figure, the lower loop can only be executed if the edge guard is satisfied, ie when the function *check()* return *true*. This function verifies whether all generating threads, whose id is present in the *sources* subarray, for a specific waiting thread have notified the data availability. If all dependencies are satisfied for a certain thread, then the function returns *true*, otherwise *false*.

The full implementation of the template functions is shown in appendix C (Listing 9.2).

5.5 Control_Alg template

The *Control_Alg* template models the execution of control algorithms and storage operation into shared memory. This template has the same *Pre_Proc* parameters except for the following:

- urgent chan *&start[num_cycle + 1]* : are the binary synchronisation channels array to signal data availability
- const int *t_synch* : is the time to receive the data availability notification

As shown in the Figure 5.5, the implementation of this template is similar to *Pre_Proc*. The template simulates the time between the generation of the data availability notification (by the last pre-processing) and the activation of this control algorithm. This is

possible by adding the location *Waiting_Synch*, with $y <= t\_synch$ invariant, before the core request through the channel *core_request*.
In the model implementation, the clock *time_partial* are defined. This clock is used to measure the time that the algorithm needs to execute a cycle, which includes the time for core assignment, the time for access to shared memory and its writing.

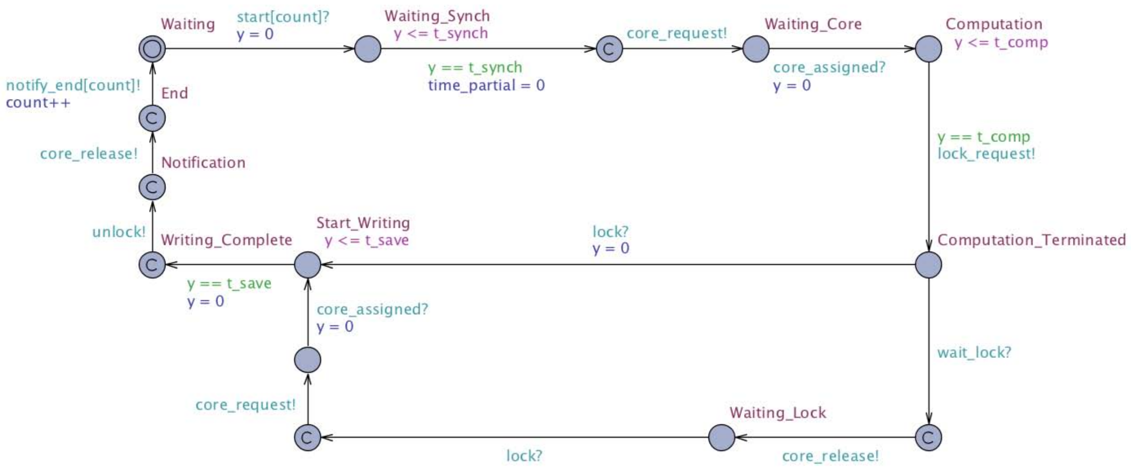


Figure 5.5: Automaton for control algorithms thread.

5.6 Output template

The *Output* template modelling the execution of the algorithms, called *output algorithm*, that generate the reference waveforms. These algorithms receive input data processed by control algorithms through shared memory.
The template has the following parameters:

- urgent chan &*start[num_cycle + 1]* : are the binary synchronisation channels array for signalling data availability
- urgent chan &*core_request* : is the binary synchronisation channel to require a core
- urgent chan &*core_assigned* : is the binary synchronisation channel to assign a core
- urgent chan &*core_release* : is the binary synchronisation channel to release a core
- const int *t_synch* : is the time to receive the data availability notification
- const int *t_comp* : is the average calculation time for processing

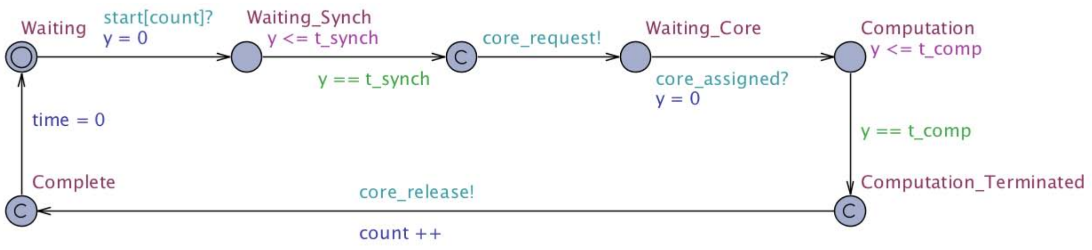


Figure 5.6: Templates for DAC supervisory threads.

The implementation of this template is visible in the Figure 5.6, where *Waiting* is the initial location. When the automaton is in the initial location, it waits for a data notification through the channel *start* (for a specific cycle). After synchronisation, the automaton goes to the *Waiting_ Synch* location and waits for a time equal to *t_ synch*. This simulates the time needed for the system to send the notification of data availability to this thread. The next step modelling the request and assignment of a core for executing the output algorithm. The automaton reaches the *Computation* location and waits for *t_ comp* time unit, simulating the execution of the output algorithm. After the computation, the core is released and the automaton returns to the initial location. Location *End* is used only to facilitate the definition of some verification queries and is not required for proper model operation. Even in this template, the committed property is not essential for proper model operation but reduces possible interlacing between the transitions. As in other cases, reducing interlacing of transitions leads to a reduction in query verification time.

5.7 Templates for scheduling

The type of scheduler chosen is decisive for system performance and for this reason is important to analyse the behavior of the system according to the different cores assignment policies for the real-time threads. Two types of schedulers have been implemented in this model: *Scheduling_ Group* and *Scheduling_ Free*.
Scheduling_ Group provides initial core assignment to individual threads and then the core can be used by individual threads in order of request according to FIFO policy. Otherwise, *Scheduling_ Free* does not require an initial core assignment and this is done automatically by the scheduler. The core assignment depends on the cores workload when the scheduler receives the first core request from a specific thread.

Many schedulers define a time slice, usually 100 milliseconds. In the particular case of this model, threads use cores for a period of time less than 100 milliseconds, making unnecessary the time slice simulation.

Another feature of modern schedulers that work on SMP systems is the ability to balance the workloads among CPUs. For this model, balancing is not considered because during the simulation a limited number of cycles are performed, and the virtual simulation time is less than the time after which this balancing is performed (if necessary). Balancing generally occurs every 200 milliseconds. As shown in the next section, the model is generally executed for a maximum of three cycles, with a virtual simulation time less than 10 milliseconds. Then the CPU workloads balance is not implemented because it would never executes.

In scheduler implementation, we have assumed that all real-time threads have the same level of priority.

5.7.1 Scheduler_Group template

The first scheduler implemented is the version with predefined core allocation, whose definition is contained in the *Scheduler_Group* template. This scheduler can be used to simulate the 1-1 assignment where each core is assigned uniquely to a single thread, or is possible to assign multiple threads to a single core. The ability to configure the scheduler allows the simulation of all possible assignment combinations. This scheduler allocates a queue of execution requests for each core, managed with FIFO policy. Each thread must send a request before it can use the core.
Assignment configurability is possible by defining two-dimensional array with *num_core* x *num_thread* channels, in particular for *core_request*, *core_assignment* and *core_release* channels. The main idea is to assign an array of *num_thread* channels to each core, where each array's channel is associated with a single thread of the system. To facilitate configuration, the channel index in the subarray associated with the core coincides with the thread id. For example, to assign the thread with id 4 to core 1 it is sufficient to force the automaton that modelling this process to operate on channels *core_request[1][4]*, *core_assignment[1][4]* and *core_release[1][4]*. The choice to define an array of *num_threads* channels for each core is due to the need to know with which thread the scheduler is operating. Every time there is a synchronisation on the index channel *[i][j]*, the scheduler knows that the operation involves the core i and thread with id j. This need is more evident for the core assignment to a thread, where the scheduler informs the thread of the assignment via the *core_assignment* channel. If each core has only one of this channel, there may be more processes waiting for synchronisation on the specific *core_assignment[i]* channel for core i. Since the channel is binary, when the scheduler activates the synchronisation a waiting thread is randomly selected, violating FIFO policy. This does not occur if threads waiting for synchronisation use different channels. In practice, the fixed assignment takes place through the correct configuration of the channels needed to use the scheduler.

The template of this scheduler has the following parameters:

- urgent chan &*request[num_core][num_thread]* : are the binary synchronisation channels to require a core
- urgent chan &*assigned[num_core][num_thread]* : are the binary synchronisation channels to assign a core
- urgent chan &*release[num_core][num_thread]* : are the binary synchronisation channels release a core

In addition, a *int[0, num_core - 1] id_core* and *int[0, num_thread - 1] id_t* custom types are declared.

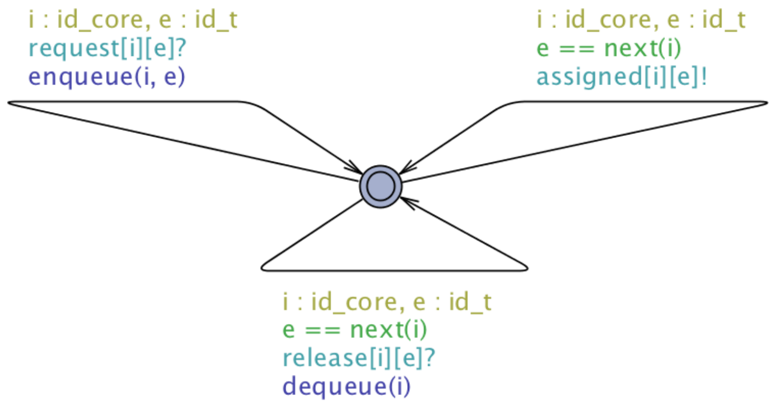


Figure 5.7: Implementation of the *Scheduler_Group* template.

The template implementation is shown in the Figure 5.7. Scheduler implementation only has one location and three loops needed to perform the request, assignment, and release operations for cores. The first upper left loop shown in the figure is taken when a thread with id e requires a core i, ie it is awaiting synchronisation on the *core_request[i][e]* channel that coincides with the channel *request[i][e]* of this template. When this edge is taken, the function *enqueue()* that stores the request in the appropriate queue according to the required core is executed. Requests are always inserted at the end of the queue.
The upper right loop allows the scheduler to report the core assignment to a specific thread. This edge syncs only with the channel of the specific threads id that is in the first position in the core's queue. The correct channel synchronisation occurs thanks to the *next(i)* function, that returns the thread id in the head position in the core i queue.
The last loop allows a thread to release the core, assuming that all threads use the cores for a time ever lower than the time slice. When this edge is taken, the function *dequeue()* removes the thread id from the core queue. In addition, the edge is only active for those threads whose id is in the first queue position for a specific core. This control is not strictly necessary but avoids errors in queue management due to errors in the cores requests and releases.

The full implementation of the template functions is shown in appendix D (Listing 9.3).

5.7.2 Scheduler_Free template

The second type of scheduler implemented simulates the linux 2.6 scheduler for real-time threads without fixed core assignments. This scheduler provides two queues for each core, one for active threads and one for expired threads. When a thread requires core usage, it is placed in the correct queue of active threads. Subsequently, when it releases the core, it is placed in the expired threads queue. The active threads queue is handled with FIFO policy.
The organisation of the channels for requesting, assigning and releasing the core is different from the previously described scheduler version. In this case, it's enough to associate a single channel with the threads for each operation (request, assign and

release).
In these schedulers, thread allocation to a core occurs during the first core request for it. The core is chosen by searching the first core that has an active queue with the smallest number of threads. When the core is found, the thread id is inserted into its active queue. Subsequently, all requests don't require the search of a free core, but only the insertion of the thread id into the active queue of the core already identified in the first request.
The implementation of this scheduler is in the *Scheduler_Free* template and has the following parameters:

- urgent chan &*request[num_thread]* : are the binary synchronisation channels array to require a core
- urgent chan &*assigned[num_thread]* : are the binary synchronisation channels array to assign a core
- urgent chan &*release[num_thread]* : are the binary synchronisation channels array to release a core

As for the *Scheduler_Group* template, a *int[0, num_core - 1] id_core* and *int[0, num_thread - 1] id_t* custom types are declared.

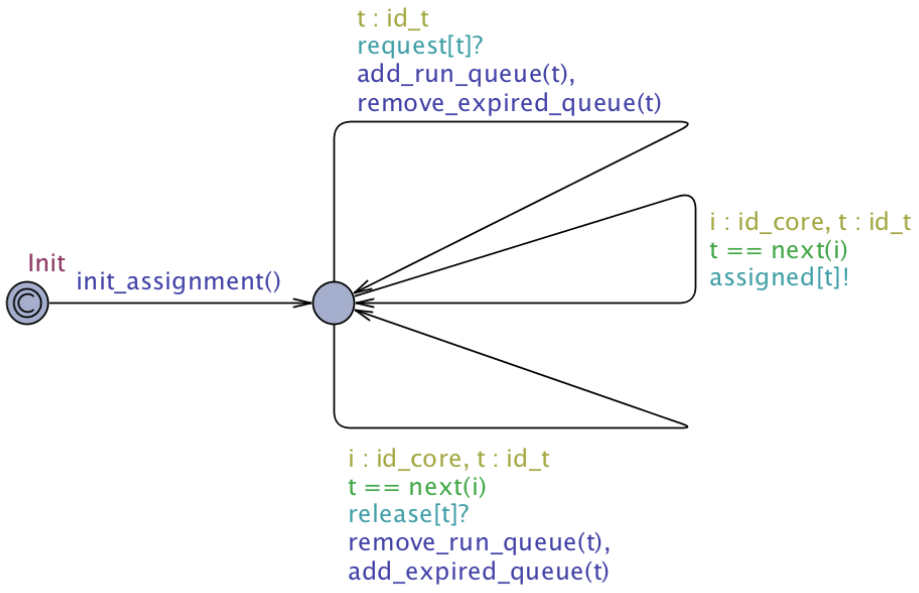


Figure 5.8: Implementation of the *Scheduler_Free* template.

The scheduler implementation is visible in the Figure 5.8. The initial location *Init* is committed to ensure that the outgoing edge is taken at the beginning of system simulation. The outgoing edge from the initial location executes the function *init_assignment()*, which initialises to *-1* all the elements of an array. This array is the assignment array, where each element with index i containing the id of the core assigned to the i-th thread. This edge leads to the second location where the automaton will remain for the entire execution.

The second location has three loops that perform similar functions to those already seen for the previous scheduler. The higher loop is taken when a thread sends a core request. In this case, the id of the requesting thread is inserted into the active threads queue of the assigned core, possibly removed from the expired threads queue. If the thread has not been assigned to a core (value *-1* in the assignment array) then the first core with the smallest number of threads in the active queue is selected.
The central loop is similar to the same edge in the *Scheduler_ Group* template, where synchronisation on the core assignment channel is only enabled for the thread that is in the top position in the active threads queue for a specific core *i*.
The last loop allows a thread to release the core, even in this case the time slice is not considered because it is greater than the running time of the threads. When this edge is taken, the synchronised thread is removed from the active threads queue (by *remove_ run_ queue()* function) and inserted into the expired thread queue (by *add_ expired_ queue()* function). As with the previous scheduler, it imposes that this edge is only active for the threads that are actually running on a core.

The full implementation of the template functions is shown in appendix E (Listing 9.4).

5.8 System definition

The templates described above implements the basic component that define the real-time system, and make the model easily configurable and totally modular. All templates are parameterised, allowing the declaration of components that share the same template but with different configuration parameters.
This model defines 4 components for raw data acquisition, 4 pre-processing units, 3 control algorithms and 4 components for supervising the generation of reference waveforms. The 4 pre-processing units process different types of raw data and take the following names (the value in brackets is the id): Radial_P (0), Toroidal_P (1), Actuators_Currents_P (2) and Axisymmetric_P (3). The units that perform the control algorithms are named MHD_C (0), Toroidal_C (1) and Axisymmetric_C (2). The last four components for the reference waveform generation are called MHD_First_Half_O (0), MHD_Second_Half_O (1), Axisymmetric_O (2) and Toroidal_O (3).
One of the configuration parameters is the communication time over Gigabit network and the calculation times for preprocessing algorithms, control algorithms, and final algorithms for generating reference waveforms (also called *output alghoritms*). In this model, an average transmission time of 100 μs with a jitter of 10 μs is considered. For the output algorithms, an average computing time of 30 μs is considered. In addition, the memory writing time *t_ save* is equal to 1 μs, the time *t_ synch* needed for synchronisation between threads is equal to 10 μs and the VMEs acquisition time is equal to 10 μs. Processing times for other algorithms are given in the Table 5.1 and 5.2. The mean times given in the tables are obtained by measuring the actual system during its normal operation.

As already announced, control algorithms need some input data that is produced by pre-processing threads. In particular the MHD control (MHD_C) algorithm requires all the data provided by the 4 pre-processing, otherwise the Toroidal_C and Axisymmet-

Radial_P	60 μs
Toroidal_P	14 μs
Actuators_Currents_P	15 μs
Axisymmetric_P	80 μs

Table 5.1: Average times measured for pre-processing algorithms.

MHD_C	60 μs
Toroidal_C	34 μs
Axisymmetric_C	20 μs

Table 5.2: Average times measured for control algorithms.

ric_C algorithms only need data processed by the Axisymmetric_P pre-processing. Instead, for the final output algorithms, MHD_First_Half_O and MHD_Second_Half_O need the data inputs provided by MHD_C, otherwise Axisymmetric_O requires data from Axisymmetric_C and Toroidal_O needs Toroidal_C data. The two configuration arrays are visible in the Listing 5.2. In the listing there is a subarray for control algorithms with only *-1* values, this is because the fourth element is not used but is required because the template *Threads_Notification* expects an array with NxN dimensions.

```
const int sources_control_alg[N][N] = {{0, 1, 2, 3}, {3, -1, -1, -1}, {3,
    -1, -1, -1}, {-1, -1, -1, -1}}; //control_alg

const int sources_output[N][N] = {{0, -1, -1, -1}, {0, -1, -1, -1}, {2,
    -1, -1, -1}, {1, -1, -1, -1}};
```

Listing 5.2: Arrays for configuration of input sources for control algorithms and for output algorithms.

Pre-processing and control threads need a shared memory space for storing their output data. The two process groups must store the data on two different memory spaces, so the model allocates two different instances of the *Shared_Memory* template.

The last configuration concerns the type of scheduler to be used in the simulation. In the case of fixed core assignment, we must associate the correct synchronisation channels to the threads in accordance with the assignment we want to obtain, as explained at the beginning of this section. Instead, the totally free scheduler only requires that each timed automata has its own synchronisation channel (for request, assign and release operation).

Chapter 6

Model validation

The Uppaal main feature allows to define and verify the properties that the model has to satisfy. Properties are implemented with a specific language and are called *query*, as described in Chapter 2. Some queries have been defined to validate the correctness of the implemented model and to determine the execution times of the various system threads. Each query is evaluated for all model configurations, in particular for the different types and configurations of the schedulers considered.
Simulated scenarios are the following:

1. static thread allocation in a one-to-one thread-core relationship by using the *Scheduler_ Group* template.
2. static thread allocation of 6 cores, as shown in the Table 6.1, by using the *Scheduler_ Group* template.
3. free allocation of 11 core by using the *Scheduler_ Free* template.

Core	0	1	2
Threads	Radial_P and Toroidal_P	Actuators_Currents_P and Axisymmetric_P	MHD_C

Core	3	4	5
Threads	Toroidal_C and Axisymmetric_C	MHD_First_Half_O and MHD_Second_Half_O	Axisymmetric_O and Toroidal_O

Table 6.1: Core allocation for the second scenario.

As already mentioned, each model component performs up to 3 execution cycles. We chose to limit the number of cycles to 3 because it represents the minimum number of iterations for analyse the effect of the pipeline (see Figure 3.7) on the system behavior. If the number of cycles increases, then the possible dynamic behaviors of the model would increase enormously, making property verification difficult.

All queries have been validated using a Uppaal remote server installed on an Amazon AWS EC2 instance [23]. The instance used is the *R4.large*, which has 2 virtual CPUs, 15.25 GiB of RAM, and uses Intel Xeon E5-2686 v4 processors.

6.1 Verify the correct execution of the model

The first implemented query allows to verify if the model runs correctly and executes all the required cycles. Each modelled thread must run a number of cycles equal to *num_cycle*. The model has some threads that depend on others because they need their output to execute the algorithm. For this reason, if a pre-processing or control threads doesn't run all the cycles, then even the threads awaiting their output will not be able to execute all the cycles. More specifically, considering a pre-processing algorithm that only executes c cycles (with $c < num\_cycle$), then the control algorithm will only execute c cycles because it doesn't have input data for the $c + 1$ cycle.
The query implementation is visible in the Listing 6.1.

```
A[] deadlock imply
output0.count == num_cycle and
output1.count == num_cycle and
output2.count == num_cycle and
output3.count == num_cycle
```

Listing 6.1: Query for check the correct execution of the model.

As shown in the listing, the query only verifies if output algorithms execute a number of cycles equal to *num_cycle*. Is possible to restrict the control to output algorithms because if any other threads did not complete all the cycles, then even the output algorithms wouldn't complete all the cycles.

In all scenarios considered, this query has obtained positive validation, confirming the correctness of the model.

6.2 Model execution time

In order to be able to analyse the behavior of the model for different scheduler types and configurations, is necessary to measure the model execution times. Execution time is the time interval that begins with the first raw data acquisition and ends with the generation of the last reference waveform in the last cycle for the output threads. Uppaal does not allow the definition of a query that can provide the maximum or minimum value of a clocks in a specific system state. For this reason, it's necessary to implement a query to verify whether the value of a specific clock takes a certain value or an inequality relation is satisfied. The two queries implemented are visible in the Listing 6.2 and 6.3, and allow to measure the maximum and minimum execution time, respectively. These queries use the *master_clock* that is initialised to 0 at the model execution startup.
The maximum execution time is detected by attempts, by changing the x value for the $master\_clock <= x$ inequality. If for a given value of x the query is not satisfied, then

the maximum value is greater than x, otherwise the maximum is equal or less than x. Then the maximum execution time coincides with the smallest value of x for which the query is satisfied. The minimum execution time is determined in a similar way, but in this case the minimum time coincides with the highest value of x for which the query in the Listing 6.3 is satisfied.

```
A[] (output0.Complete and output0.count == num_cycle) or
(output1.Complete and output1.count == num_cycle) or
(output2.Complete and output2.count == num_cycle) or
(output3.Complete and output3.count == num_cycle)
imply master_clock <= 769
```

Listing 6.2: Query to determine the maximum execution time.

```
A[] (output0.Complete and output0.count == num_cycle) or
(output1.Complete and output1.count == num_cycle) or
(output2.Complete and output2.count == num_cycle) or
(output3.Complete and output3.count == num_cycle)
imply master_clock >= 637
```

Listing 6.3: Query to determine the minimum execution time.

The times obtained in the different scenarios for one cycle or three cycles are shown in the Table 6.2. The first observation concerns the minimum execution times, for both cycle numbers the minimum times of the three scenarios coincide. This does not imply that in the best case the three scheduling configurations are equivalent, because in scenarios 2 and 3 the scheduler performs context switches (the cores are shared with two or more threads). The context switch time is not considered in the model, and for this reason the minimum times coincide with the case of one-to-one assignment of scenario 1. Scenario 1 has a maximum execution times lower than the other two cases for both cycle numbers. This shows how the fixed thread allocation with a one-to-one thread-core relationship has the best execution times, as stated in [1].

	one cycle		3 cycles	
Scenario	Minimum Time	Maximum Time	Minimum Time	Maximum Time
1	237	290	637	690
2	237	379	637	779
3	237	324	637	769

Table 6.2: Execution times [μs] obtained with different schedulers.

From the results obtained, it's possible to observe the benefits of the software pipeline organisation (see Figure 3.7). Table 6.3 shows the execution times in the absence of software pipeline organisation, these values are obtained by multiplying by 3 the maximum and minimum times for a single cycle in Table 6.2. In the case with pipelines, times are lower than the case without it for all three scenarios, as expected. This demonstrates that the model correctly simulates the presence of the pipeline organisation.

	3 cycles without pipeline	
Scenario	Minimum Time	Maximum Time
1	711	870
2	711	1137
3	711	972

Table 6.3: Execution times [μs] for different schedulers without pipeline.

During the different executions of the model is possible to noticed that the free scheduler (scenario 3) assigns the core when a thread executing his first cycle. This assignment only involves the first 4 core of 11 available, due to the core choice approach used by this scheduler for assign a core to a thread. To prove what's being observed, a new query has been implemented (Listing 6.4). This query verifies if there is a path where the cores with id from 4 to 10 are assigned. To do this check, the query uses the array *threads_ on_ core* (defined in the *Scheduler_ Free* template) which indicates how many threads are assigned to each core of the system. For example, *threads_ on_ core[i]* indicates how many threads use the core with id i.

```
E<> (scheduler.threads_on_core[4] != 0) or
(scheduler.threads_on_core[5] != 0) or
(scheduler.threads_on_core[6] != 0) or
(scheduler.threads_on_core[7] != 0) or
(scheduler.threads_on_core[8] != 0) or
(scheduler.threads_on_core[9] != 0) or
(scheduler.threads_on_core[10] != 0)
```

Listing 6.4: Query to check if only the first four core (with id from 0 to 3) are assigned.

If the property expressed by this query is verified, then at least one of the cores with id between 4 and 10 is assigned to one or more threads.
From the tests performed this property is not verified, showing that the observation made is true for scenario 3. This result indicates that the same execution times for scenario 3 are obtained for any number of available cores greater or equal to 4.

The time determined with queries 6.2 and 6.3 when $num\_cycle = 1$ coincides with the latency of the system in case of a single cycle, it measures the time between the start of the raw data acquisition and the generation of the last reference waveforms for the cycle 1. To understand how the scheduler affects the behavior of the control system, it's also necessary to determine the maximum and minimum latency times for cycles 2 and 3. To do this, the queries in the listing 6.5 and 6.6 have been implemented. These queries test a condition on the clock t, which is initialised to zero each time the raw data acquisition begins. In order to determine the latency for the i-th cycle, it's necessary to set $num\_cycle = i$.
Note that these two queries are a special case of those in the Listing 6.2 and 6.3.

```
A[] (output0.Complete and output0.count == 2) or
(output1.Complete and output1.count == 2) or
(output2.Complete and output2.count == 2) or
(output3.Complete and output3.count == 2)
imply t <= 290

A[] (output0.Complete and output0.count == 2) or
(output1.Complete and output1.count == 2) or
(output2.Complete and output2.count == 2) or
(output3.Complete and output3.count == 2)
imply t >= 237
```

Listing 6.5: Queries to find the maximum and minimum latency times for cycle 2.

```
A[] (output0.Complete and output0.count == 3) or
(output1.Complete and output1.count == 3) or
(output2.Complete and output2.count == 3) or
(output3.Complete and output3.count == 3)
imply t <= 290

A[] (output0.Complete and output0.count == 3) or
(output1.Complete and output1.count == 3) or
(output2.Complete and output2.count == 3) or
(output3.Complete and output3.count == 3)
imply t >= 237
```

Listing 6.6: Queries to find the maximum and minimum latency times for cycle 3.

	Latency for cycle 1		Latency for cycle 2	
Scenario	Minimum Time	Maximum Time	Minimum Time	Maximum Time
1	237	290	237	290
2	237	379	237	379
3	237	324	237	349

	Latency for cycle 3	
Scenario	Minimum Time	Maximum Time
1	237	290
2	237	379
3	237	369

Table 6.4: Latency times [μs] for cycles 1, 2 and 3. The latency times for cycle 1 are those of the "one cycle" column in Table 6.2.

The latencies obtained for cycles 1, 2 and 3 are shown in Table 6.4. From these results it's noted that in scenarios 1 and 2 the maximum and minimum latency times coincide

in each cycle. This indicates that for both schedulers, the latency varies over a constant time interval for each execution cycle.
Instead in scenario 3 with free scheduler, the maximum latency time increases in cycles 2 and 3 (compared to the first). This shows how with this type of scheduler the latency can increase from a cycle to the next, which proves that the best control system configuration doesn't require the use of the free scheduler.

6.3 Times between an output generation and the subsequent

System analysis requires the study of the time variation between the i-th output generation and the subsequent generation for different types of schedulers and their settings. In practice, for each of the four output algorithms, the maximum and minimum time between an output generation and the subsequent is measured for each scenario. To determine these times, the two queries in the Listing 6.7 and 6.8 are implemented for each output thread. In both queries, a condition is imposed on the value of the clock *time* when the automaton is in the *Complete* location (see Figure 5.6). In this case, the value of the clock is considered only when the control algorithms have execute at least one cycle. The time measured in the first cycle is discarded because it coincides with the time between the start of the simulation and the generation of the last output (is the latency of the system).

```
A[] output0.Complete and output0.count > 1 imply output0.time <= 240
```

Listing 6.7: Query to determine the maximum time between an output generation and the subsequent for the *MHD_First_Half_O* thread (with id *0*).

```
A[] output0.Complete and output0.count > 1 imply output0.time >= 160
```

Listing 6.8: Query to determine the minimum time between an output generation and the subsequent for the *MHD_First_Half_O* thread (with id *0*).

Even in this case, the maximum and minimum times are determined by attempts.
Table 6.5 shows the results obtained by performing three cycles ($num\_cycle = 3$). Scenarios 2 and 3 have a greater difference between maximum and minimum times than scenario 1. At the same time, scenario 1 has a minimum times greater than the other two cases. This does not imply that the type of scheduler and its configuration used in scenarios 2 and 3 provide better results than the first case. In fact, in the first case, maximum times recorded are lower respect maximum times in the other two scenarios. For the latter reason and for the smallest variation between the maximum and minimum times, the first scenario with fixed and one-to-one thread-core assignment is preferable. Also in the first scenario, times are closer to the sampling period of 200 μs than the other cases.

In scenario 1, the maximum and minimum times of *MHD_First_Half_O* and *MHD_Second_Half_O* threads are the same, this observation is also valid for *Axisymmetric_O* and *Toroidal_O*. This is because the two thread groups have the same

	MHD_First_Half_O		MHD_Second_Half_O	
Scenario	Minimum Time	Maximum Time	Minimum Time	Maximum Time
1	190	210	190	210
2	160	240	160	240
3	91	326	91	326

	Axisymmetric_O		Toroidal_O	
Scenario	Minimum Time	Maximum Time	Minimum Time	Maximum Time
1	187	213	187	213
2	84	316	105	293
3	117	285	119	281

Table 6.5: Maximum and minimum times [μs] between an output generation and the subsequent.

dependencies for input data and each of them has the exclusive use of a core (they can start at the same instant).
Similarly, times for *MHD_First_Half_O* and *MHD_Second_Half_O* are the same in scenario 2 and 3. Regarding scenario 2, the times coincide because the two threads share the same core and have the same input data dependencies. In this case, only one thread at a time can be executed, and therefore the dynamic evolution of the system considers the case where *MHD_First_Half_O* is executed before *MHD_Second_Half_O* and vice versa. In scenario 3, we can't make assumptions about the cores assignments to these threads. However, it's possible to check if the two threads are always assigned to the same core. This check is done by the query in the Listing 6.9, where the thread *MHD_First_Half_O* has id *7* and *MHD_Second_Half_O* has id *8*. The query uses the *core_assignment* array, declared in the *Scheduler_Free* template, that associates at each thread the assigned core id. This property is verified and therefore the assignment to the same core isn't guaranteed.
Note that the implemented query returns true when these threads are assigned to different cores.

```
E<> ((output0.Complete and output0.count == 1) or
(output1.Complete and output1.count == 1)) and
(scheduler.core_assignment[7] != scheduler.core_assignment[8])
```

Listing 6.9: Query to check if there is a case where *MHD_First_Half_O* and *MHD_Second_Half_O* are assigned to two different cores.

6.4 Execution times of the MHD control algorithm

The most critical algorithm is the MHD control that allows to limit plasma instabilities. For this reason, it's important to evaluate how the type of scheduler used can affect the execution times of this algorithm. The measured time considers the core assignment time, the calculation time, and the time to save the data in shared memory.
To determine the maximum and minimum executing times, several attempts are made using the queries visible in the Listing 6.10 and 6.11, respectively. Both queries use the clock *time_partial* defined in the *Control_Alg* template, as shown in the Figure 5.6.

```
A[] control_alg0.End imply control_alg0.time_partial <= 106
```

Listing 6.10: Query to determine the maximum execution time of the MHD control.

```
A[] control_alg0.End imply control_alg0.time_partial >= 71
```

Listing 6.11: Query to determine the minimum execution time of the MHD control.

In practice, the two queries determine if the *time_partial* clock in the location *End*, for the instance representing the MHD control of the *Control_Alg* template, satisfies a certain inequality $time\_partial \bullet x$, where $\bullet \in \{\leq, \geq\}$.
Also in this case, the maximum execution time coincides with the smallest value of x that satisfies the property in the Listing 6.10. Otherwise, the minimum execution time coincides with the highest value of x that satisfies the property in the Listing 6.11.

Scenario	Minimum Time	Maximum Time
1	61	61
2	61	61
3	61	96

Table 6.6: Execution time [μs] of the MHD control algorithm.

The measured times by executing three cycles ($num\_cycle = 3$) are visible in Table 6.6. The results obtained show that recorded times in scenario 1 and 2 are identical, and coincides with the running time of the MHD algorithm that is equal to 61 μs (see section 5.8). This indicates that in both cases the MHD control algorithm doesn't have to wait for the core assignment, and doesn't even wait to acquire the lock required for write in shared memory. The core acquisition time is zero since in both scenarios the MHD control thread has the exclusive use of the core (no context switch). Otherwise, the lock acquisition time is zero because all control algorithms finish their execution before the access request to shared memory by MHD algorithm. The MHD control immediately acquires the lock because it's not owned by any other threads. These observations are valid for each cycle of the MHD algorithm execution, because the maximum and minimum time coincide for each cycle.
In scenario 3, the MHD control thread can be assigned to a core in a non-exclusive way and then compete with other threads for its use. For this reason, the core acquisition

time is not always zero and the algorithm may have to wait for the write-lock availability. What this entails is visible in the Table 6.6, where the maximum execution time of the MHD algorithm is greater than the other scenarios.
This analysis also concludes that the best execution times are obtained when the algorithm uses a core assigned exclusively to him.

The results provided by the queries considered in this section show the limit of the simulated model. Figure 6.1 shows the real execution time distribution for the MHD control if the scheduler is free to assign the core to the threads, as in the scenario 3. This execution time does not take into account the time for core acquisition, but instead considers the lock acquisition time and the time for memory writing [1]. The execution time shown in the figure has two peaks that are not detected in the simulations of this model. Non-detection is due to the inability to simulate all aspects of a Linux operating system. For example, the model does not consider the CPU cache management, Translation Lookaside Buffer (TLB) management, and other aspects that can affect the execution time. The model can't be extended to consider all the major aspects of the operating system that may affect execution times, because the model would become too complex and computationally heavy.

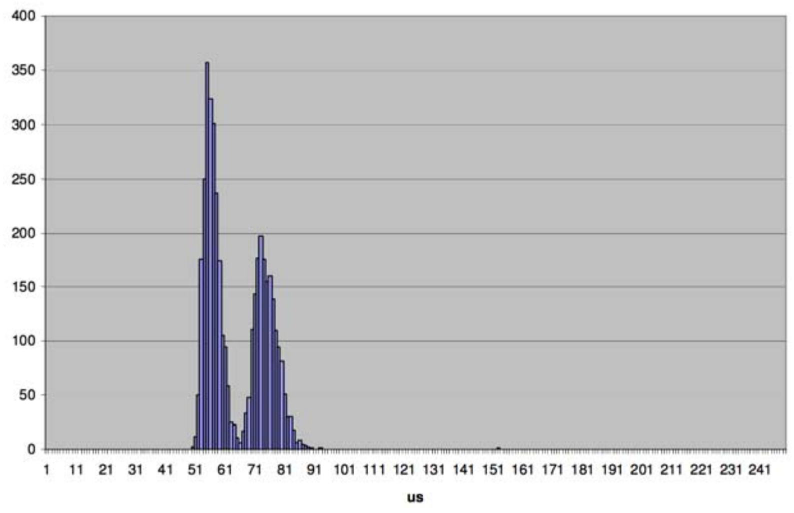


Figure 6.1: Real execution time of MHD control in a configuration in which the Linux scheduler is free to assign threads to any core [1].

Chapter 7

Query verification times

Queries are a fundamental tool to verify the model properties. Every query requires a verification time that depends on the number of states on which the conditions are evaluated. For this reason, if the possible dynamic evolution of the model increases then the number of states to be analysed also increases.
Table 7.1 shows the verification time of the main queries used in Chapter 6. Except for column "One Cycle", all the times in the table refers to queries verified for the model that executes 3 cycles. These times are obtained by averaging the times of five independent verifications, without the *reuse* option of Uppaal. This option allows to reuse part of the previous verification processing for the same model configurations. The main benefit is to reduce the time of property analysis.

	Execution Time			
	One Cycle		3 Cycles	
Scenario	Minimum Time	Maximum Time	Minimum Time	Maximum Time
1	0,8	1,3	4,3	4,9
2	0,4	0,5	941	914
3	8	16	3300	1601

	Time between an output and the subsequent		MHD Control Execution Time	
Scenario	Minimum Time	Maximum Time	Minimum Time	Maximum Time
1	20,9	4,8	4,4	4,3
2	1896	1635	977	934
3	6365	4013	3492	3401

Table 7.1: Query verification times [s] when the condition is verified.

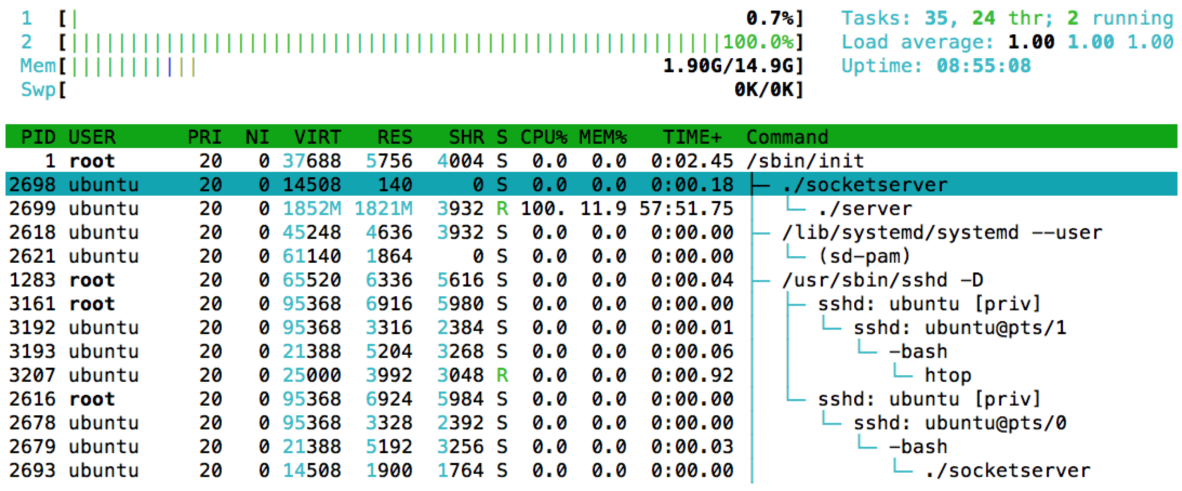
The table shows that in the scenario 2 and 3 the verification times are higher than scenario 1. This is due to a greater dynamism in the evolution of the model in the two scenarios, thanks to the greater possibility of interlacing between transitions respect to scenario 1.

As highlighted in Chapter 6, queries allow to determine execution times by attempts. The approach to determine the maximum and minimum times it's based on the query test with a value close to the maximum or minimum that we want to determine. Based on the query result, the test is performed on a value greater or less than the previous one. This requires performing a large number of tests, so we must try to reduce the time that Uppaal takes for verification. In particular, it's important to know in a short time whether a certain value doesn't satisfy the relationships expressed by the query. Queries to determine execution times impose a condition that must be verified for all model states. When Uppaal checks these query, return false statements as soon as it finds a state that violates the condition, avoiding the complete analysis of all state sequences. If a failure in query execution requires a similar verification time to those in the table, then the determination of the maximum and minimum times would be difficult to accomplish.
In general, Uppaal adopts a *breadth first* exploration approach, where the analysis of the tree, that represents the evolutions of the model, starts from the root node and explores the neighbor nodes first, before moving to the next level of neighbors. Most verified queries require condition analysis of system states that are in the final part of the sequence of states of any possible dynamic evolution of the model. This would require the analysis of a long sequence of states before it could get a false result for the query. So a *depth first* approach is preferable to *breadth first*, where condition analysis starts from the root node of the tree and explores all branches before backtracking.
Uppaal allows the choice of the space exploration strategy and also implements the *depth first* technique. In this way, in the worst case and for heavyer computational queries we can get a false result in a maximum of 10 minutes (a query verification may take more than one hour).

To speed up the property validation, Uppaal provides the *reuse* option. When *reuse* is active, Uppaal reuse portions of the state space generated during the analysis of previous queries for the same model configurations. The verification time reduction obtainable with this option is mainly observed for queries that determine the model execution time, where the verification time is reduced by approximately 40%.

In addition to the verification time, we must also consider the amount of RAM needed to run the verification process and to store the evolution tree of the model on which the property analysis takes place. The RAM consumption during the verification performed is never more than 4GB. Due to the availability of primary memory provided by the EC2 instance, this is not considered a critical point.

Another factors affect the property verification time. The Uppaal server process, called *socketserver*, is not multithreaded and this is observable by analysing it behavior. Figure 7.1 shows the screen obtained through the Htop software [24], where is possible to see that the *socketserver* process (highlighted in blue) has only one thread child called *server*. This doesn't allow the Uppaal server to exploit the various vCPUs that are made available by the Amazon AWS instance or more in general by the recent computers.



```
  1  [|                                                     0.7%]   Tasks: 35, 24 thr; 2 running
  2  [|||||||||||||||||||||||||||||||||||||||||||||||||||||100.0%]   Load average: 1.00 1.00 1.00
  Mem[||||||||||                                   1.90G/14.9G]   Uptime: 08:55:08
  Swp[                                                  0K/0K]

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
    1 root       20   0 37688  5756  4004 S  0.0  0.0  0:02.45 /sbin/init
 2698 ubuntu     20   0 14508   140     0 S  0.0  0.0  0:00.18 ├─ ./socketserver
 2699 ubuntu     20   0 1852M 1821M  3932 R 100.  11.9 57:51.75 │  └─ ./server
 2618 ubuntu     20   0 45248  4636  3932 S  0.0  0.0  0:00.00 ├─ /lib/systemd/systemd --user
 2621 ubuntu     20   0 61140  1864     0 S  0.0  0.0  0:00.00 │  └─ (sd-pam)
 1283 root       20   0 65520  6336  5616 S  0.0  0.0  0:00.04 ├─ /usr/sbin/sshd -D
 3161 root       20   0 95368  6916  5980 S  0.0  0.0  0:00.00 │  ├─ sshd: ubuntu [priv]
 3192 ubuntu     20   0 95368  3316  2384 S  0.0  0.0  0:00.01 │  │  └─ sshd: ubuntu@pts/1
 3193 ubuntu     20   0 21388  5204  3268 S  0.0  0.0  0:00.06 │  │     └─ -bash
 3207 ubuntu     20   0 25000  3992  3048 R  0.0  0.0  0:00.92 │  │        └─ htop
 2616 root       20   0 95368  6924  5984 S  0.0  0.0  0:00.00 │  └─ sshd: ubuntu [priv]
 2678 ubuntu     20   0 95368  3328  2392 S  0.0  0.0  0:00.00 │     └─ sshd: ubuntu@pts/0
 2679 ubuntu     20   0 21388  5192  3256 S  0.0  0.0  0:00.03 │        └─ -bash
 2693 ubuntu     20   0 14508  1900  1764 S  0.0  0.0  0:00.00 │           └─ ./socketserver
```

Figure 7.1: Htop screenshot with threads running on Amazon EC2 instance.

As shown in this chapter, the time to verify a property can be very high due to the multiple dynamic evolution of the model. This doesn't allow the creation of models that simulate every aspect of the system, because some queries may require very high verification times. In the specific case of the RFX system, it isn't possible to model some aspects of the operating system and of the real-time algorithms, and at the same time keeping the property verification time low. For example, adding a jitter to the computing time of real-time algorithms the model becomes heavy for the simulation, that may requires up to 3 hours for a single verification. For these reasons, it's important to model only the most significant aspects of the system.

Chapter 8

Conclusions and future works

Uppaal is the fundamental tool for developing this thesis. Its versatility has allowed to model the most significant aspects of the real-time control system used in the RFX experiment. Query definition and verification is a very powerful and generic tool that offers the ability to test whether the model satisfies certain properties, allowing to determine the execution times of system components. The limits of modelling are evident when queries are being verified. In fact, the verification time depends on the complexity of the model and its possible dynamic evolution. Whenever simulation considers new aspects of the real-time system or operating system, such as jitter time for algorithms execution, the verification time may increase until it doesn't allow the analysis in an acceptable time. For this reason, it's necessary to model only the aspects of the system and operating system that most affect the behavior of the control system.
Even in the presence of some limitations, Uppaal has allowed to develop a correct model that simulates the behavior of the real-time control system and operating system. The most important aspects of the control system are the synchronisation mechanisms between thread, the simulation of the algorithm execution, shared memory management with mutual exclusion, and subsequent writing. For the operating system, it was possible to develop models describing the execution of two schedulers types for real-time tasks, in particular fixed core allocation and free assignment.
From queries validation, we can confirm what is shown in [1]. In fact, thanks to the results obtained, it's possible to state that the best configuration of the control system involves the use of fixed assignment with one-to-one thread-core relationship (scenario 1). In this case, the latency times, the execution times of the MHD control algorithm and the times between output generation and the subsequent, guarantee better performance for the control system than the other two scenarios.

A possible future development for this thesis is to implement a simulation where the physical cores are virtually doubled by hyperthreading. This would allow a more in-depth comparison with the measurements of the real control systems reported in [1]. A second development considers the refinement of the model, introducing new aspects of the operating system and control system. An aspect of the control system not considered in the current model is the presence of jitter in the execution time of algorithms for the real-time threads. Currently the model considers a constant execution time for algorithms but in reality this time varies within a range. For the operating system, we can model the time it takes to perform a context switch or the CPU cache management. These features are not currently considered because the template becomes too complex for queries verification, requiring in the worst case up to 5-6 hours for a single verification. Future Uppaal optimisations could make possible the model improvements.

Chapter 9

Appendix

9.1 Appendix A

Syntax of expressions in BNF is the following:

$$
\begin{aligned}
Expression \rightarrow\ & ID \mid NAT \\
& \mid Expression\ '['\ Expression\ ']' \\
& \mid '('\ Expression\ ')' \\
& \mid Expression\ '++' \mid '++'\ Expression \\
& \mid Expression\ '--' \mid '--'\ Expression \\
& \mid Expression\ AssignOp\ Expression \\
& \mid UnaryOp\ Expression \\
& \mid Expression\ BinaryOp\ Expression \\
& \mid Expression\ '?'\ Expression\ ':'\ Expression \\
& \mid Expression\ '.'ID
\end{aligned} \tag{9.1}
$$

$$
UnaryOP \rightarrow\ '-' \mid '!' \mid 'not' \tag{9.2}
$$

$$
\begin{aligned}
BinaryOp \rightarrow\ & '<' \mid '<=' \mid '==' \mid '!=' \mid '>=' \mid '>' \\
& \mid '+' \mid '-' \mid '*' \mid '/' \mid '\%' \mid '\&' \\
& \mid '|' \mid '\hat{}' \mid '<<' \mid '>>' \mid '\&\&' \mid '||' \\
& \mid '<?' \mid '>?' \mid 'and' \mid 'or' \mid 'imply'
\end{aligned} \tag{9.3}
$$

$$
\begin{aligned}
AssignOp \rightarrow\ & ':=' \mid '+=' \mid '-=' \mid '*=' \mid '/=' \mid '\%=' \\
& \mid '|=' \mid '\&=' \mid '\hat{}=' \mid '<<=' \mid '>>='
\end{aligned} \tag{9.4}
$$

9.2 Appendix B

```
typedef int[0, A-1] id_t;
id_t queue[A];
int[0, A] length;

// Puts an element at the end of the queue
void enqueue(id_t e){
        queue[length++] = e;
}
// Removes the first element of the queue
void dequeue(){
        int i = 0;
        length -= 1;
        while (i < length)
        {
                queue[i] = queue[i + 1];
                i++;
        }
        queue[i] = 0;
}
// Returns the first element of the queue
id_t next(){
   return queue[0];
}
```

Listing 9.1: Functions implementation for the Shared_Memory template.

9.3 Appendix C

```
typedef int[0, num_sender - 1] id_sender;
typedef int[0, num_receiver - 1] id_receiver;
typedef int[0, num_cycle] cycle;
bool logbook[num_sender][num_cycle + 1];

//Registration of the notification
void log_notification(int i, int j){
 logbook[i][j] = true;
}
//Check if all input data are available
bool check(int i, int j){
 int k;
 for(k = 0; k < num_sender; k++){
  if(sources[i][k] < 0)
   return true; //All inputs are available
  if(logbook[sources[i][k]][j] == false)
   return false;
 }
 return true;

}
```

Listing 9.2: Functions implementation for the Threads_Notification template.

9.4 Appendix D

```
typedef int[0, num_core - 1] id_core;
typedef int[0, num_thread - 1] id_t;

id_t queue[num_core][num_thread];
int [0, num_thread - 1] length[num_core];


// Puts an element at the end of the queue for core "i"
void enqueue(id_core i, id_t e){
        queue[i][length[i]++] = e;
}
// Removes the first element of the queue for core "i"
void dequeue(id_core i){
        int k = 0;
        length[i] -= 1;
        while (k < length[i])
        {
                queue[i][k] = queue[i][k + 1];
                k++;
        }
        queue[i][k] = 0;

// Returns the first element of the queue for core "i"
id_t next(id_core i){
   return queue[i][0];
}
```

Listing 9.3: Functions implementation for the Scheduler_Group template.

9.5 Appendix E

```
typedef int[0, num_core - 1] id_core;
typedef int[0, num_thread - 1] id_t;

int[-1, num_core - 1] core_assignment[num_thread];
int [0, num_thread] threads_on_core[num_core];

id_t run_queue[num_core][num_thread];
id_t expired_queue[num_core][num_thread];

int [0, num_thread - 1] length_run[num_core];
int [0, num_thread - 1] length_expired[num_core];

// Init
void init_assignment(){
 int i;
 for(i = 0; i < num_thread; i++)
  core_assignment[i] = -1;
}
```

```
// Assigns the thread to the core if it's the first request and return the
     core_id
id_core core_assign(id_t t){
 if(core_assignment[t] != -1)
  return core_assignment[t]; //Core already assigned
 else{
  int min = num_thread;
  id_core core;

  // Select the core with less workload on run_queue
  int i;
  for(i = 0; i &lt; num_core; i++){
   if(length_run[i] < min){
    min = length_run[i];
    core = i;
   }
  }

  core_assignment[t] = core;
  threads_on_core[core]++;

  return core;
 }
}

// Puts an element at the end of the run_queue for the correct core
void add_run_queue(id_t t){
 id_core core;
 core = core_assign(t);
 run_queue[core][length_run[core]++] = t;
}

// Puts an element at the end of the expired_queue for the correct core
void add_expired_queue(id_t t){
 id_core core = 0;
 core = core_assign(t);
 expired_queue[core][length_expired[core]++] = t;
}

// Removes the element "t" of the exprired_queue for the correct core
void remove_expired_queue(id_t t){
 int k = -1, i;

 id_core core = 0;
 core = core_assign(t);

 for(i = 0; i < length_expired[core]; i++){
  if(expired_queue[core][i] == t){
   k = i;
   i = length_expired[core];
  }
 }

 if(k >= 0){
        length_expired[core] -= 1;
        while (k < length_expired[core])
        {
                expired_queue[core][k] = expired_queue[core][k + 1];
                k++;
        }
        expired_queue[core][k] = 0;
 }
}

// Removes the first element of the run_queue for the correct core
void remove_run_queue(id_t t){
 int i, k = 0;

 i = core_assign(t);

        length_run[i] -= 1;
        while (k < length_run[i])
        {
                run_queue[i][k] = run_queue[i][k + 1];
                k++;
        }
        run_queue[i][k] = 0;
}

// Returns the first element of the run_queue for specific core
int next(id_core i){
 if(length_run[i] > 0)
     return run_queue[i][0];
 else
  return -1;
}
```

Listing 9.4: Functions implementation for the Scheduler_Free template.

Bibliography

[1] G. Manduchi, A. Luchetta, and C. Taliercio, "The new multicore real-time control system of the rfx-mod experiment," 2013. Consorzio RFX, Padova Italy.

[2] A. C. Neto, F. Sartori, F. Piccolo, R. Vitelli, G. D. Tommasi, L. Zabeo, A. Barbalace, H. Fernandes, D. F. Valcarcel, and A. J. N. Batista, "Marte: A multiplatform real-time framework," *IEEE Transactions on Nuclear Science*, vol. 57(2), pp. 479–486, April 2010.

[3] G. Manduchi, A. Barbalace, A. Luchetta, A. Soppelsa, and E. Zampiva, "Upgrade of the rfx-mod real time control system," *Fusion Engineering and Design*, vol. 87(12), pp. 1907–1911, June 2012.

[4] G. Behrmann, A. David, and K. G. Larsen, *A Tutorial on UPPAAL 4.0*. http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf. 28 Nov 2006.

[5] Z. Gu, Z. Wang, and H. Chen, "A model-checking approach to schedulability analysis of global multiprocessor scheduling with fixed offsets," *Int. J. Embedded System*, vol. 6, pp. 176–187, 2014.

[6] *Uppaal official site*. http://www.uppaal.org. Online in date 10/05/2017.

[7] *Tool Environment for Validation and Verification of Real-Time Systems*. https://www.it.uu.se/research/group/darts/papers/texts/uppaal-pamphlet.pdf.

[8] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell."

[9] *UPPAAL 4.0: Small Tutorial*. 16 Nov 2009.

[10] K. G. Larsen, P. Pettersson, and W. Yi, "Compositional and symbolic model-checking of real-time systems," *Proceedings, 16th IEEE Real-Time Systems Symposium*, pp. 76–87, December 1995. IEEE Computer Society Press.

[11] J. Bengtsson and W. Yi, "Timed automata: Semantics, algorithms and tools."

[12] R. Alur, C. Courcoubetis, and D. L. Dill, "Model-checking for real-time systems," *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pp. 414–425, June 1990.

[13] *Nature Physics*, vol. 5. August 2009.

[14] *Technical features RFX machine - Official Site*. https://www.igi.cnr.it/www/?q=content/machine. Online in date 10/05/2017.

[15] G. Manduchi, A. Luchetta, A. Soppelsa, and C. Taliercio, "From distributed to multicore architecture in the rfx-mod real time control system," *Fusion Engineering and Design*, vol. 89(3), pp. 224–232, 2014.

[16] M. Cavinato, A. Luchetta, G. Manduchi, G. Marchiori, and C. Taliercio, "First operation of rfx-mod realtime control system," *Fusion Engineering and Design*, vol. 81(15-17), pp. 1765—1770, July 2006.

[17] G. Manduchi, A. Luchetta, A. Soppelsa, and C. Taliercio, "The new feedback control system of rfx-mod based on the marte real-time framework," *IEEE Transactions on Nuclear Science*, vol. 61(3), pp. 1216–1221, June 2014.

[18] *JET official site.* https://www.euro-fusion.org/jet/. Online in date 10/05/2017.

[19] I. C. Bertolotti and G. Manduchi, *Real-Time Embedded Systems - Open-Source Operating Systems Perspective.* CRC Press, 2012.

[20] M. T. Jones, *Inside the Linux 2.6 Completely Fair Scheduler.* https://www.ibm.com/developerworks/library/l-completely-fair-scheduler/, December 2009. Online in date 10/05/2017.

[21] M. T. Jones, *Inside the Linux Scheduler.* https://www.ibm.com/developerworks/linux/library/l-scheduler/, June 2006. Online in date 10/05/2017.

[22] M. T. Jones, *Linux and symmetric multiprocessing.* https://www.ibm.com/developerworks/library/l-linux-smp/, March 2007. Online in date 10/05/2017.

[23] *Amazon AWS EC2 official site.* https://aws.amazon.com/ec2/. Online in date 10/05/2017.

[24] *Htop official site.* http://hisham.hm/htop/. Online in date 10/05/2017.