Thesis for the Degree of "Magistère" of Computer Science[†]

System-Level Modeling of Embedded Control Systems:

a case study from the Automated Transfer Vehicle

by

Nicolas Berthier

Supervised by

Florence Maraninchi and Christophe Rippert

Verimag Laboratory

September 2008

[†]This is an improved version of a thesis for the Degree of Master of Computer Science presented in June 2008.

Abstract

Model-driven approaches are more and more used in order to develop embedded control applications. Indeed, high-level models ease the design and the reasoning of these complex systems. Moreover, models with well-defined semantics allow formal analysis and validations of the system. In addition, when this semantics is executable, then they can be used to build a virtual prototype of a platform. This enables starting the development of the software as well as tests and evaluations of the system earlier and before the physical existence of the hardware. These models can also serve to guide the target software code generation, this last operation being sometimes automated, therefore making safer or even eliminating the error-prone manual implementation of the application.

However, these approaches usually involve a remaining manual development stage, consisting in implementing the code assuring the integration of the multiple application parts on the target platform (also called system-level code). In addition, past experiences have shown that the most successful approaches aiming at this issue consist in the characterization of the target platform by identifying the related constraints and behaviors, and their integration in the high-level models.

We try to address this problem in the same manner, but in the domain of distributed embedded control applications involving fault tolerance and using point-to-point communication protocols. We notably studied the existing Proximity Flight Safety (PFS) control system which entailed these characteristics.

After having manually integrated a control application onto a virtual prototype of the PFS implemented in SystemC, we identified the needed information to build a synchronous model of the platform including the system-level code. Several kinds of synchronous models were then designed with the Lustre language. Then, we were able to point out the basis for a model-driven design flow taking this integration code into account.

Keywords: embedded control application, system-level modeling, model-based engineering, virtual prototyping, synchronous programing.

Résumé

Les approches dirigées par les modèles sont de plus en plus utilisées pour le développement d'applications de contrôle embarqué. En effet, l'emploi de modèles de abstraits facilite la conception et l'appréhension de ces systèmes complexes. De plus, lorsqu'ils ont une sémantique bien définie, ces modèles permettent des analyses formelles de propriétés de ces systèmes. Lorsqu'ils sont exécutables, alors ils peuvent être utilisés afin de construire un prototype virtuel d'une plate-forme. Cela permet de commencer le développement du logiciel, ainsi que sa validation et son évaluation plus tôt et parfois avant même la réalisation du matériel. Ces modèles peuvent également servir à guider la génération du code de l'application, cette opération étant parfois automatisée, diminuant alors fortement les risques liés à une implantation totalement manuelle.

Cependant, ces approches nécessitent quelquefois une étape de développement supplémentaire, consistant à l'implantation du code assurant l'intégration des différentes parties de l'application sur la plate-forme cible. De plus, les expériences passées ont montré que les approches résolvant le mieux ce problème ont consisté en la caractérisation des plates-formes cibles par l'identification des contraintes et comportements qu'elles impliquent, puis en la prise en compte de celles-ci dans des modèles de haut niveau.

Nous essayons de résoudre ce problème dans la même optique, mais dans le cadre des applications de contrôle embarqué distribuées, tolérantes aux fautes et utilisant des protocoles de communication point-à-point. Notamment, nous étudions le PFS (Proximity Flight Safety), un système de contrôle existant.

Après avoir implanté manuellement une application de contrôle sur un prototype virtuel de cette plate-forme réalisé avec SystemC, nous avons identifié les informations nécessaires à la conception d'un modèle synchrone du système complet. Plusieurs représentations de haut niveau ont ensuite été créées avec Lustre. Nous pouvons alors avancer les bases d'une méthode de conception dirigée par des modèles prenant en compte les caractéristiques de la plate-forme cible influant sur le logiciel complet.

Mots-clés : Application de contrôle embarqué, modélisation système, conception dirigée par les modèles, prototypage virtuel, programmation synchrone.

Contents

Chapter 1

Introduction

Nowadays, hardware and software systems are becoming more and mode complex. Likewise, manual implementation of such systems is very error-prone, and it is also too complex to check the correctness of the obtained programs. As a consequence, systematic approaches allowing the development of complex systems are needed.

Besides, an analysis of the evolution of programming languages tends to reveal the growing needs for high-level languages. Therefore, the arising demands concern new techniques supporting system design at higher abstraction levels. Model-driven approaches are an attempt to propose the usage of high level concepts.

1.1 Model-driven approaches

The main principle of the model-driven approaches is to use models all along the development cycle.

Since modeling means to specify a system or subsystem in a formalism with well-understood syntax and semantics, using models eases the design and the reasoning of complex systems. It also facilitates the collaborations between different engineering perspectives.

When models have well-defined semantics, then they allow formal analysis and simulations, possibly through transformations.

They also enable starting the development of the application software earlier and before the physical existence of the hardware platform. Indeed, when they have an executable semantic, they can be used to build a virtual prototype of a platform: this allows early tests and evaluations of the software. This may dramatically reduce the time and cost of designs.

Besides, model-driven implementation (also called model-based development) approaches use software models to guide the development of the final application code. Several formalisms allow an automatic partial generation of this code, such as the class diagrams of the *Unified Modeling Language* (UML). Other models aim at generating all the target software code. For instance, we can cite the Lustre extensions proposed by Adrian Curic [Cur05], dedicated to the automatic deployment of control applications on *Time-Triggered Architectures* (TTA). These development methods may reduce the risks related to the manual implementation of the application.

At last, they can also be used for documentation purposes.

Moreover, embedded control application design flows usually employ a model-based development method and use high-level models allowing formal verifications to check their correctness. The development steps consist in the transformation of these models into refined ones, and eventually into the target software code.

1.2 Objectives

After a model-driven design process, a manual development step usually remains to be done after the software code implementation: multiple application parts, sometimes developed independently, have to be integrated into the target platform with additional code (that we will

call *system-level code* thereafter). This remaining code highly depends on the target platform. For instance, two different *Real-Time Operating Systems* (RTOSs) may not provide the same services, and the system-level code will not use their *Applications Programming Interface* (API) in the same manner.

In addition, past experiences have shown that the most successful approaches addressing the model-based development consist in using models taking the characteristics of the target platform into account. For instance, Caspi et *al.* [CCM+03] proposed to take advantage of the TTA properties so as to translate Simulink models into application code targeting these architectures. They used an extended Lustre as intermediate language to perform formal analysis of the models.

Following the same idea, we try to use high-level models including characteristics of the platform to guide the development steps of embedded control applications.

1.3 Our approach

In order to achieve the aforementioned objectives, we have studied the Proximity Flight Safety (PFS) chain. The PFS is an existing control system aiming at ensuring the safety of the maneuvers of the Automated Transfer Vehicle (ATV). The ATV, by its turn, is an unmanned spacecraft used for the periodic resupplying of the International Space Station.

Next, we have built a virtual prototype of a simplified version of the PFS system in SystemC along with a simple control application implemented in Lustre, in order to implement manually the needed system-level code. At last, we have proposed two synchronous models of the whole system (*i.e.* including the system-level code and the software tasks), the latter being a refinement of the former, thus allowing more precise simulations and behavior representations. Then, we were able to propose some solutions aiming at integrating this supplementary code into a model-driven design flow.

1.4 Outline

In the first chapter, we introduce the prerequisites that are needed to understand this work. Next, we depict the characteristics of our case study: the real PFS system. We then present the model-driven approaches in general to identify some weak points and assets of the existing solutions. The last chapter concentrates on our approach and results we were able to obtain with the PFS models.

Chapter 2

Background

This chapter presents the prerequisites that are needed to understand our work. It starts with an introduction to the synchronous paradigms and a detailed presentation of the Lustre language, followed by a presentation of the SystemC library and an overview of the Architecture Analysis and Design Language.

2.1 The synchronous languages

In opposition to a *transformational system* that computes results from initial data in finite time (*e.g.* a compiler), a *reactive system* takes a sequence of events or *stimuli*, and computes a sequence of reactions. Again, a reactive system differs from an *interactive* one because it cannot impose its own rhythm to its outputs: the reactions must be computed at a speed determined by constraints depending on the execution environment. Finally, most of these systems are *critical* because they must fulfil safety constraints. Example of such systems are control/command appliances like nuclear plant controllers or flight control software.

To program such systems, *synchronous languages* have been introduced in the 80s.

2.1.1 The synchronous paradigms

Synchronous languages are high level languages with a rigorous mathematical semantics which allow the programmers to develop critical software [BB91, Hal93].

A synchronous program is a composition of *synchronous components*, the latter behaving as a generalization of the Mealy machines with arbitrary data types. Several assumptions and properties characterize this programs. Firstly, atomic execution time, or zero-time reaction (also called *synchronous hypothesis*) is presumed: the computation time of the outputs from the corresponding inputs is assumed to be null. Hence, the sequence of atomic reactions introduces a notion of *execution step*. Secondly, the synchronous programs are generally described as concurrent processes communicating through *signals* (namely, instantaneous communications).

The synchronous hypothesis and the instantaneous communications imply the concept of *intrinsic parallelism* of a synchronous program: one step of the composition of several components consists of a "simultaneous" step of each component. Nevertheless, even though we can see the processes of this program as running concurrently, in effect the compilation of such a program into a classical language like C results in a sequential code.

Among the existing synchronous languages we can cite Lustre [HCRP91], Signal [LGLL91], Esterel [BG92] and SCADE. The former will be presented thereafter.

2.1.2 The Lustre language

Lustre is a formally defined language used to describe any synchronous system proposed in 1984 by the Synchronous team of the Verimag research laboratory. It was transfered to *Esterel Technologies*, and later became part of the core of their industrial tools which are used by Airbus and Schneider Electric among others.

It is a declarative language based on the data-flow model. It has simple semantics and is strongly typed. A LUSTRE program has a cyclic behavior and is structured into *nodes*. Each node takes at least one input, produces one or more output, and contains a set of equations.

Each variable or expression of an equation denotes a *flow* (also called *stream*): a variable x references a sequence of data $X = \langle x_1, x_2, \cdots, x_n, \cdots \rangle$, where x_n is the value of the variable x at the instant n. Likewise, a constant 2 denotes the sequence of values $\langle 2, 2, 2, \cdots \rangle$. Furthermore, a stream, in addition to the possibly infinite sequence of values, contains an associated *clock* denoting the activity of the stream at each *logical instant*. If a stream is inactive at a given moment, *i.e.* its clock is "$false$" [1], then its *current value* does not exist. In addition, a program including equations that use streams with different clocks will not even compile. Further, each node possesses a *basic clock*, expressing the logical time at which the node is running (hence, the basic clock of a whole LUSTRE program is always "*true*").

The language supplies a set of classical operators for arithmetic (+, -, *, etc.) and booleans (not, =, <> meaning $\neq$, and, xor, $\Rightarrow$, etc.), plus a "control flow" operator "if b then x else y".

Furthermore, it provides two sequence operators:

- the "previous" operator pre is a memorization operator: if x references the sequence $\langle x_1, x_2, x_3, \cdots \rangle$, then the stream associated with pre x is $\langle nil, x_1, x_2, x_3, \cdots \rangle$. The first element exists (the clock associated to the resulting stream is the same as those of x), but owns an undefined value (written *nil* in the sequel);
- the operator $\rightarrow$ is an initialization operator: if x and y reference the sequences $\langle x_1, x_2, x_3, \cdots \rangle$ and $\langle y_1, y_2, y_3, \cdots \rangle$ respectively, then the stream associated with $x \rightarrow y$ is $\langle x_1, y_2, y_3, \cdots \rangle$.

The listing 2.1 presents a basic LUSTRE node, illustrating the syntax of the language and the usage of the previously described operators. The figure 2.1 shows a representation of the related network of operators.

```
-- detects a positive or a negative edge on x,
-- the initial output value is init.
node EDGE (init: bool; x: bool) returns (y: bool);
let
  y = init -> (x <> pre x);
tel
```

Listing 2.1: An example of LUSTRE *node returning true if it detects a modification of the values of its input stream* x.

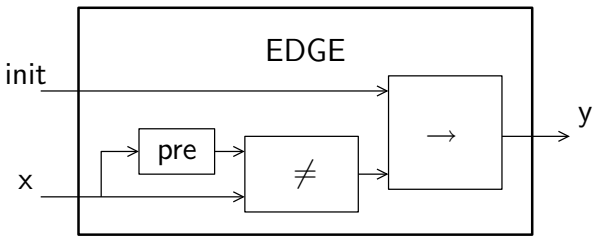

Figure 2.1: A data-flow representation of the EDGE *node.*

Besides, two other operators serve to manipulate the clock of a stream:

[1] One can notice that the clocks are most often described with boolean expressions, even if there exist some subtle differences.

- the when operator "samples" a stream with a given clock in order to get another stream with a "slower" clock. If x denotes an arbitrary stream and c a stream of booleans with the same clock as x, then "x when c" is the sequence of values referenced by x when c is *true*.

- the current operator "interpolates" a stream: if c is the boolean stream representing the clock of x, then "current x" represents the stream "holding" the current value of x when c was *true*. Initially, its value remains undefined while c does not get *true*.

The table 2.1 sums up this two concepts.

c	*false*	*false*	*true*	*false*	*false*	*true*	*true*	*false*
x	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
x when c			x_3			x_6	x_7	
current (x when c)	*nil*	*nil*	x_3	x_3	x_3	x_6	x_7	x_7

Table 2.1: Sampling and interpolating a stream.

A variant of the basic control flow operator, with b then x else y, allows the creation of "recursive" nodes as it is expanded during the compilation. It thus requires that the boolean expression b is statically computable. Other possible constructs concern the manipulation of *arrays*. Their size has to be known at compile time because they are expanded statically too. Also, one can manipulate these structures with *slices*. For instance, "a[0 .. n-1] = b[0 .. n-2] | [1]" denotes the aggregation of an array of n streams (n must refer to a strictly positive constant integer stream): $a[0 \,..\, n-2] = b[0 \,..\, n-2]$ and $a[n-1]$ is affected to the constant integer 1.

Finally, it is also possible to define *assertions* to express known (or expected) properties on the streams. Their primary use is to give to the compiler supplementary information in order to allow better optimizations. These constructs can also help program verification (*cf.* section 2.1.3).

Since a LUSTRE program is a composition of nodes, we can observe that this modularity involves major advantages. In fact, one can develop, test and maintain each node of an application separately.

2.1.3 Tools related to the Lustre language

Thanks to its properties, a LUSTRE program supports many static checks. Furthermore, it exists a variety of tools designed to check properties of LUSTRE programs. For instance, Lesar [HLR92] is a symbolic model checker, and NBac [Jea03] is a tool for abstract interpretation. Lucky, Lutin [RR02] and Lurette [RWNH98], are tools dedicated to automatic testing. At last, Lutess [DORZ99] is a testing environment for synchronous software based on LUSTRE.

2.2 SystemC

The need for techniques allowing a relatively complete description of hardware components at high level of abstraction led to the definition of new languages and frameworks. HANDELC [APP+98] and SPECC [FN01] are examples of *Domain Specific Languages* resulting from attempts to extend the C language as to allow the description of hardware systems. In the same way, SYSTEMC is the current commonly accepted solution for modeling such systems.

2.2.1 The SystemC library

The SystemC library was born in 1999, and is developed by the *Open SystemC Initiative*[2] since then. It has been standardized by the Institute of Electrical and Electronics Engineers (IEEE) in 2005 [Ope05].

This library, including a simulation core and a set of classes, introduces a notion of parallelism essential for hardware description, on top of the C++ language. The main advantages of using this standard and widely accepted language are notably the ability to use well-spread and efficient tools (C++ compilers and debuggers), as well as a faster learning of the related concepts.

A SystemC model defines a (possibly hierarchic) set of *modules*, each one able to contain many associated *threads*. The SystemC execution core is a discrete-event based simulator, composed of a non deterministic and non preemptive scheduler managing the threads defined in the model. It also provides some mechanisms of communication between these threads. They can communicate with each other through two types of communication mediums: *events*, direct function calls and *channels*. Notifying an event wakes up all the threads waiting for its occurrence. In addition, channels are mechanisms contributing to the encapsulation and reuse of communication protocols. They shall be used through interfaces, also called *ports*, to facilitate the reuse of the modules and channels through different platforms.

One can eventually generate an executable program containing both the simulation core and a representation of the model, by compiling the set of modules along with the library.

A common use of this library is the *Register Transfer Level* (RTL) modeling of circuits, describing the flows of signals between registers and the associated combinational logic operations. A real-life sized system model generally leads to a huge amount of SystemC code and slow simulation. Dedicated tools allow its automatic synthesis, but this usage is quite limited in practice.

2.3 The Architecture Analysis & Design Language

The *Architecture Analysis & Design Language* (AADL) [FGHL04, SAE04] has been introduced to model hardware and software architectures in avionics as it inherits from MetaH [Ves00], which is dedicated to real-time avionics control software. It has later been used to describe distributed real-time embedded systems more generally. This standard defines a textual and a graphical syntax. Also, its semantics allow for static analysis and verification over the models.

An AADL description consists of a set of *components* with *interfaces*, *properties* and *connections* between them.

2.3.1 Components

A component declaration primarily consists of the definition of an interface and one or more possible implementations (or instantiations). Those possibly encompass a composition of *subcomponents* as well as associated connections.

All the components can be classified into three classes: *software*, *hardware* and *composite*.

Software components. *Processes*, *threads*, and *thread groups* represent software computation elements and may be bound to a processor (see below). A process must contain at least one thread (but a thread may not be a subcomponent of a process). A thread group is just a facility to represent multiple threads sharing properties.

A *subprogram* abstracts a procedure (in an imperative style): it may be implemented in any language or with at least one *call sequence*. In the same way a thread can comprise either a subprogram or multiple call sequences. A call sequence is a composition of one or more

[2] http://www.systemc.org

subprograms and serves to describe a nondeterministic software behavior: when a thread or a subprogram contains multiple call sequences, then this represents a nondeterministic choice of one of these sequences.

At last, *data* components are akin data types.

Hardware components. The hardware components are *processors*, *memories*, *buses* or *devices*. Buses exchange data and events between the three other hardware components. Devices are models of the hardware interfaces with the environment. Finally the processors abstract the mechanisms responsible for executing and scheduling the processes and threads.

Composite component. The only hybrid component type is the *system* one: it encapsulates hardware and software components so as to allow the description of a full system.

Besides, a component interface consists of provided *features*.

2.3.2 Component features

Features enable the connection of components and allow them to communicate. *Data ports* are typed and directional (in, out or both) logical connections (data flow). Specifically, data ports applied to subprograms are called *parameters*. Moreover, an *event port*, which is also directional, represents a pure control flow: a subprogram can notify or wait for an *event* through this kind of port. Incoming event ports are queued and *event data ports* depict event ports transporting extra information.

Finally, one can model shared data among components with dedicated features: a component may *require* or *provide* an *access* to a data component. This sharing possibility also exists for the buses.

2.3.3 Component properties

The language permits the specification of properties about the components. For a processor, these properties can specialize its scheduling protocol or its clock rate for instance. Similarly, the dispatch_protocol property of a thread depicts its activation policy (*e.g.* periodic or aperiodic). Deadlines and periods are also thread attributes, and shared data or buses can specify a concurrent control protocol. Moreover, some other properties allow to abstract information like the redundancy of components. Projects or tool-specific property sets can be defined.

At last, a concept of *operational modes* and associated *mode transitions* and *events* have also been introduced in the language to capture dynamic aspects of system architectures. An operational mode represents a non-functional state of a component, and specifies an implementation (*i.e.* subcomponents, connections, bindings and properties). Dynamically going from a mode to another, according to a mode transition, means to modify non-functional properties of the component.

2.3.4 Related tools

There exist standard annexes to the language: ARINC 653, Behavior [BDF+06], Reliability modeling, Error model, etc.

Finally, we can cite some tools based on this modeling language: for example, Ocarina [Ver06] can perform code generation from AADL descriptions, Cheddar [SLNM04] carries out some schedulability and performance analysis of AADL models, Sokolsky et *al.* [SLC06] also studied the same problem. At last, ADeS simulates the behavior of system architectures described in the AADL language.

Chapter 3

Presentation of the case study

In this chapter, we present our case study: the Proximity Flight Safety system, a controller which is part of the Automated Transfer Vehicle. Most of these information have been provided by David Lesens, a member of the team from EADS Astrium Space Transportation which worked on this system.

3.1 The Automated Transfer Vehicle

The Automated Transfer Vehicle (ATV) is an unmanned spacecraft used for the periodic re-supplying of the International Space Station (ISS). EADS Astrium Space Transportation was in charge of the development of this module: they designed the overall management and verification systems of the vehicle as well as the flight software and its tests.

Mission overview. The vehicle, after having been launched by Ariane 5, is expected to flight so as to reach the ISS orbit and then perform rendezvous and docking maneuver. After that it remains attached to the station several weeks and finally carries out the separation and departure from the station, before the atmospheric re-entry.

Since there is no human presence in the ATV, all these operations have to be done in a totally automated manner by the ATV management system.

The ATV management system. The ATV management system is located in a non-pressurized compartment. It contains several equipments such as rechargeable batteries, a *Fault Tolerant Computer Pool* (FTCP — sometimes called FTC), Guidance, Navigation and Control sensors (*e.g.* accelerometers) and communications and thermal control units. Four serial STD-MIL-1553B data buses [Mil78], named "system buses", each coupled with its own power distribution system, form four electrically independent hardware chains (or "lanes").

The FTCP, composed of three *Data Processing Units* (DPUs), comprises the overall architecture management software. These DPUs exchange and vote their data, and are capable of managing the four system buses. Initially, each DPU manages at least one system bus and can later take over this role during the flight in case of a DPU failure. Besides, they control the ATV physical attitude by sending commands through analog links to a set of four redundant *Propulsion Drive Electronic* (PDE) managing the thrusters. A voting mechanism oversees the latter commands.

As a last resort, the safety of the ATV operations on the proximity of the station is ensured by an independent Proximity Flight Safety chain (*cf.* section 3.2 on the following page).

In order to master the complexity of this system, the designers have designed several (dedicated) development facilities.

Development facilities. In 1998, Lacan [Lac98] outlined an overview of the data processing architecture at that time and developed facilities aimed to validate the system and software. Notably, a specific software development environment had been designed from the hardware

specification of the global system in order to ease the software integration and validation. It included the real target computer pool and a simulated physical environment encompassing the physical 1553 data bus level.

3.2 The Proximity Flight Safety chain

The *Proximity Flight Safety* (PFS) system is a segregated chain dedicated to ensure the safety of the ATV operations in the ISS proximity. Indeed, since the station is inhabited, any failure of the ATV flight in its neighborhood could cause loss of human life.

Therefore, this chain consists of dedicated power sources and independent sensors. It observes the attitude of the ATV as well as the behavior of the FTCP in order to detect any possible failure. If anything goes wrong, then its goal is to prevent a disaster by performing a *Collision Avoidance Maneuver* (CAM) consisting in safely driving the ATV away from the ISS, then heading it toward the Sun to wait for instructions and reload batteries. In addition, the CAM can be triggered from a "red button" command from inside the ISS or even from the Earth: a human operator or an ISS inhabitant can decide to initiate a CAM. In fact (as the first ATV has reached successfully the ISS on April 3, 2008), two CAM tests have been performed during the approach phase in the space, the first from the Earth, triggered by a human operator, and the second by an ISS inhabitant.

3.2.1 The Proximity Flight Safety Architecture

The figure 3.1 on page 12 depicts the global architecture of this chain.

One should notice that the commands sent to the set of PDE by the MSUs have a higher priority than the commands sent by the DPUs: a hardware mechanism manages this choice. Again, all command and health status transmitted through analog links are akin a boolean information, except for data received by the MSUs from the DTGs and the SSUs, and commands sent to the PDEs which use Analog-Digital or Digital-Analog converters.

Besides, the transmitted data related to this system can be qualified as *high level* or not. The high level data are commands and information compatible with the "A Category" requirements [BBD06] of the MSU software: these are only booleans.

3.2.2 Functional behavior

There are two MSUs for fault tolerance purposes and both run the same software. Nonetheless, they behave in an asymmetric manner: one of them is said to be the *master*, while the other is the *slave* (the default status of the MSUs is hardware encoded). In case of a failure of the former, the latter takes the mastership. So, this system can not put up with two MSU failures.

During the critical phase of the mission (*i.e.* in the neighborhood of the ISS), the MSUs listen to all system buses so as to deduce the status of the ATV equipment and receive health status reports from each DPU through the dedicated discrete links. Once the master has detected two anomalies (failure of a DPU, erroneous physical state of the ATV, etc) or receives a "red button" HLTC (coming from the Earth or the ISS and relayed by the CPFs), then it begins a CAM: it resets the FTCP and continuously send inhibit commands to its DPUs in order to prevent interactions between the main computer and the master MSU. It afterwards controls the ATV to take it away from the ISS.

The real MSU software has been devised with a data-flow driven design flow in order to perform complete validations, and has been implemented with the ADA language. As it scans periodically the received information it can thus be considered as a reactive software. This software, combined with an ADA runtime, runs natively onto the MSUs.

3.3 Observations

The first observation we can make concerns the health status transmission: the boolean information transiting through hardware links are related to *General Purpose Input/Output* (GPIO) communication mechanisms: a GPIO can be configured as an input or an output link. Basically, GPIO controllers can easily be configured to detect positive (low to high) or negative (high to low) edges from connected discrete links. Depending on the usage, one can associate an interrupt on signal edges or poll its state.

In addition, the other analog links serve to transmit non-boolean data through pulse modulations (Pulse Width Modulation, Pulse Code Modulation, etc.). Hence, they employ dedicated Digital-to-Analog and Analog-to-Digital converters.

We can also notice the exclusive usage of *point-to-point communication protocols*: *i.e.* for each medium, only a unique initiator component can send requests on it. In other words, there is only one sender per hardware link (obviously) and serial bus.

At last, the internal processor's behavior does not directly influences the global behavior of the system, so we can ignore it. Indeed, the considered communications are related to terrain buses, highly differing from internal core components such as internal buses, caches, etc.

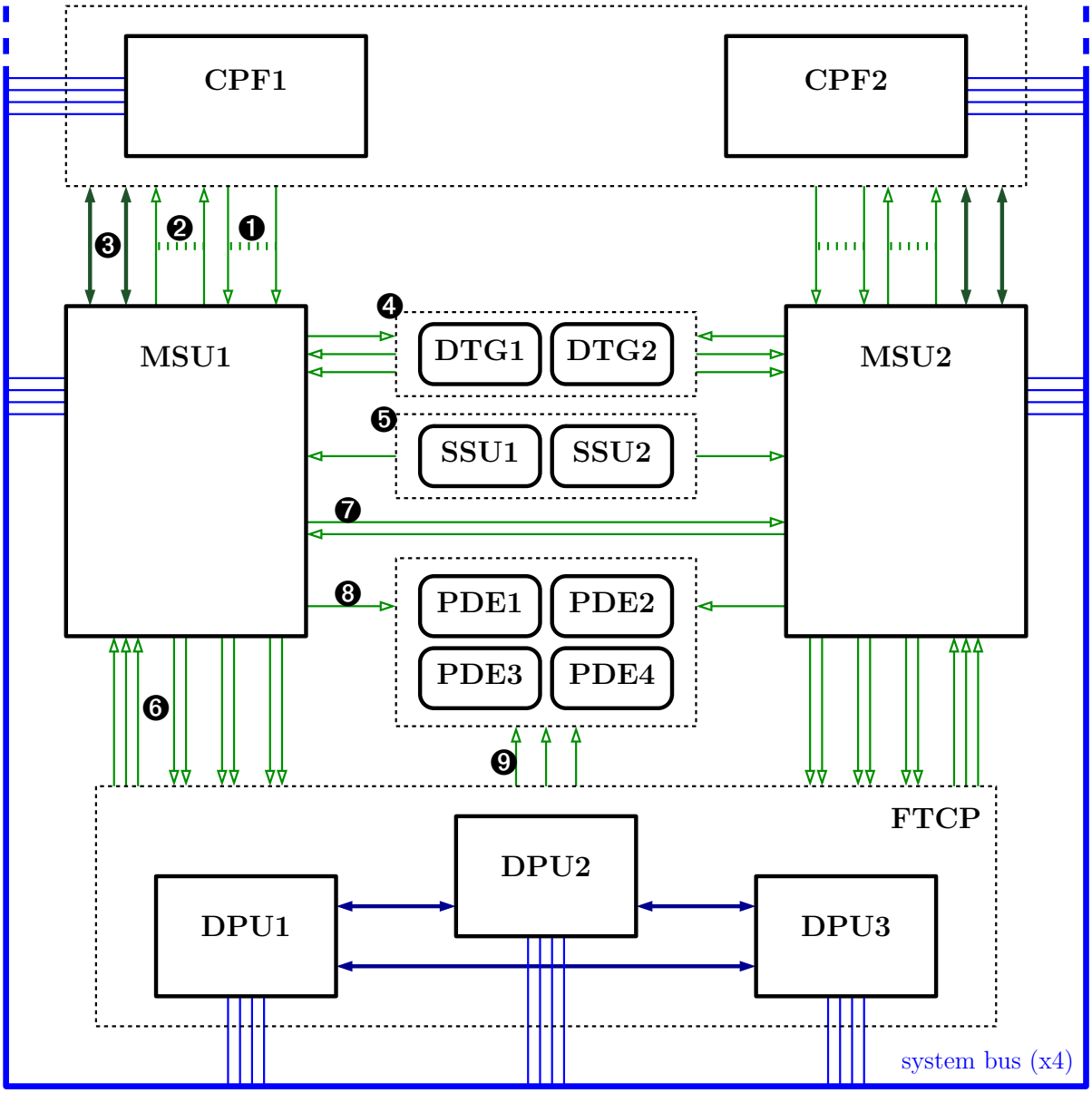

Figure 3.1: Global overview of the PFS hardware architecture. The heart of this system encompasses two Monitoring and Safety Units *(MSUs), each one bound to the four system buses in order to spy all the communications between the DPUs and the overall equipment. Again, they receive* High Level Telecommands *(HLTCs) from the Earth and the ISS from two redundant* Communication Process Formatters *(CPFs) through analog links* ❶*. These units also relays* High Level Telemetries *(HLTMs) and* Telemetries *(TMs) sent by each MSU, through a set of analog links* ❷ *and two point-to-point serial buses* ❸ *respectively, to fetch their status to the Earth and the ISS. These serial buses also serve to sent* Telecommands *to the MSUs. Furthermore, a set of two* Dynamically Tuned Gyroscopes *(DTGs)* ❹ *and two* Sun Sensor Units *(SSUs)* ❺*, communicate the ATV physical attitude and relative sun position information respectively (through analog links too). Next, other analog links also let them to receive the health status of each DPU and return them inhibit and reset commands at the beginning of a CAM* ❻*. The MSUs exchange their own state* ❼*, send mode controls (fine or coarse) to the DTGs, and send data to the set of PDE in order to manage the ATV attitude during a CAM* ❽*. Finally, as stated in section 3.1 on page 9, the three DPUs send navigation commands to the PDEs* ❾*.*

Chapter 4

Model-driven approaches to embedded system design

This chapter gives an overview of existing model-driven approaches. We then identify their key concepts as well as their advantages and drawbacks.

4.1 Model-driven approaches

Nowadays, because of the growing complexity of hardware and software systems, systematic approaches for the development of these systems are needed. Indeed, after the introduction of structured programming with the Fortran programming language, the appearance of higher-level control structures (*e.g.* "while" loops) in the 60's and the emergence of Object-Oriented Programming with the Simula language, the arising demands concern new techniques supporting system design at higher abstraction levels: even with "recent" languages like Java, hand-written code remains rather error-prone, and the belated discovery of high-level errors is very expensive and time-consuming. It is also too complex to check the correctness of systems developed with these languages.

Model-driven approaches are an attempt to solve these problems.

4.1.1 Using models

Modeling means to specify and describe (parts of) a system in a formalism with well-understood syntax and semantics, in order to abstract unneeded details such as implementation. Models can differ on their level of abstraction or intended use, and can also be designated for documentation purposes. Hence, using them is presumed to ease the design process and the reasoning of complex systems.

Besides, *model-checkers* are tools that can help the designers to prove certain properties from a model (*i.e.* state reachability in an automaton behavior representation, etc). These properties can be derived from *requirements*. Thus, taking advantage of models offering possibilities to use such tool (directly or through model transformations) also permits some early validation of the system under design, to check its correctness.

In this context, we notice a tendency to model asynchronous behaviors into synchronous models. Indeed, due to the assumptions and characteristics of these formalisms, more properties can be checked from them.

Moreover, models can serve either to guide the target software code generation, to build a virtual prototype or both.

4.1.2 Model-Driven Engineering

Model-Driven Engineering (at times called *Model-Based Engineering* or *Development*) describes a software development method based on high-level models. Its goal is to integrate such models throughout the development process so as to reduce its costs and time-to-market.

Schätz et *al.* [SPHP02] defined the model-driven development as the usage of "domain-specific abstractions". They pointed out an analogy between the translation of C code and the model-driven development: likewise one can compile a classic C program to many possible assembler languages, a model can be translated into a number of high-level target languages. Since models hide implementation details, they also serve as a way to restrict the "degree of freedom", impose constraints, thus limit the error possibilities.

Some kinds of models with executable formalisms supports *simulations* of the designs: finite-state-automatons or synchronous data-flow models are examples of such formalisms.

Once a software model has been designed, further development steps consist in transforming it into a refined model. Next, the target application software may be obtained. This phases may be (partially) performed automatically, depending on the synthesis support of the models employed to describe the application. For instance, although using UML class diagrams to describe the interface and data structures of an application can not enable the execution of the model, it allows a partial code generation (*e.g.* Java, C++, etc) from it.

Finally, comparisons between a model and the resulting software are usually made through executions and trace analysis.

An example of model-driven engineering approach is the *Model Driven Architecture* (MDA) development method.

The Model Driven Architecture

This software design approach has been launched by the Object Management Group[1] in 2001 [MDA]. It is based on the separation of the design from the platform, hence lets the technologies employed to realize the architectures evolving separately from the application specifications: decoupling both domain should allow independent improvements.

Specifications are described with a *Platform-Independent Model* (PIM) focusing primarily on the functionality and behavior: it does not care about the implementation details. Tools can then generate one or more *Platform-Specific Models* (PSM) and interface definition sets from the PIM. A PSM would be used to guide the code implementation. These models are related to multiple standards, such as the UML and the Meta-Object Facility (MOF) among others, each one including itself many kind of models and languages.

This approach mostly implies the usage of non-executable models. Nevertheless it exists a suggestion bound to execute the models used in MDA, consisting in adding extensions to standard UML models, then testing and compiling it into an abstract programming language [MB02]. Formal verification of these models can be performed but it lacks in accuracy and the simulation possibilities are quite limited.

4.1.3 Virtual Prototyping

When a platform model has an execution semantic then it permits to carry out simulations by using it as a *virtual prototype* executing the real software. Moreover, if it contains non-functional aspects such as energy or timing consumption then it enables the designers to accomplish multiple property and performance estimations.

These models also allow an early and cheap architectural exploration and optimization with respect to the manufacture of physical solutions. At last, a virtual prototype may contribute to reduce the time needed to start the real software implementation because it can serve as a development platform.

In the following sections, I will present two endeavors to model asynchrony with synchronous languages followed by an overview of multiple attempts to define global modeling frameworks and some model-driven implementation approaches.

[1] http://www.omg.org

4.2 Synchronous modeling of asynchronous systems

4.2.1 Using the Signal language

The POLYCHRONY workbench [LTL03], improving the SACRES project [GLG99], is a unified model-driven environment dedicated to perform embedded system design exploration. It integrates a SIGNAL compiler along with a visual editor and an associated model-checker, among other tools.

On the other hand, the ARINC 653 specification [Avi97] is based on the *Integrated Modular Avionics* (IMA) approach, in which several applications can run on a single shared computer system. It defines the *APplication EXecutive* (APEX): the interface between the application software and the system. Also, the isolation ensuring that shared resources are safely allocated among the applications is achieved by creating *partitions* containing one or more *processes* and a dedicated resource allocator. A partition must be located on a single processor and its processes can communicate asynchronously with the others (either in and out of its partition) through various mechanisms provided by the APEX interface.

In this context, Gamatié et *al.* [GGB03] proposed a method in order to use the SIGNAL language to model avionics applications. The aim of this study was to model ARINC *partitions* with the SIGNAL language so as to perform formal verifications and analysis on the system under design using the performance evaluation technique proposed in POLYCHRONY. They defined a library of components including basic communication mechanisms and ARINC component models. Likewise, they modeled APEX services (*e.g.* the partition-level operating system and the resource management) with SIGNAL components. Each process has been modeled with a control-flow/data-path pattern: the control part is a SIGNAL implementation of an automaton specifying the execution flow of the task and the computation part, divided into blocks, is executed accordingly to the state of the former.

After having modeled a partition, they performed non-functional interpretations through SIGNAL program transformations (called "morphism") preserving semantic properties [KL96]. So they could analyze the temporal behavior of sample SIGNAL programs, and carry out execution time estimations. However, they were not able to apply this evaluation technique on the ARINC model without simplifications because of limitations of the tools impacting the analysis of programs managing complex data structures.

4.2.2 AADL model execution in Lustre

Jahier et *al.* [JHR+07] have proposed to translate a subset of AADL into a nondeterministic synchronous model. Their solution consists in the synchronous modeling of the architecture in LUSTRE or SCADE in order to use the associated tools to simulate and check properties efficiently.

They have proposed to model the nondeterminism with *oracles*, which are additional inputs added to a deterministic model so as to control nondeterministic choices. This technique has strong advantages because the simulations of the obtained models are then reproducible and the suppositions over the nondeterminism can be explicitly expressed through associated constraints.

Sporadic activation of synchronous components

The asynchronous behaviors have been modeled with *sporadic activations*[2], exploiting the clock mechanisms provided by most synchronous languages: the activation of a node (its basic clock) is then constrained by a condition. For instance, the listing 4.1 and the figure 4.1 on the following page illustrate this concept.

[2]This construct is referred to as "*condact*" in the SCADE language.

```
-- y equals the value of the EDGE node when b is true .
-- it is initially false while b remains false .
  y = if b then
        current(EDGE((init , x) when b))
      else
        false → pre y;
```

Listing 4.1: Conditional activation of the EDGE *node (cf. listing 2.1 on page 4): the node is only executed when* b *is true.*

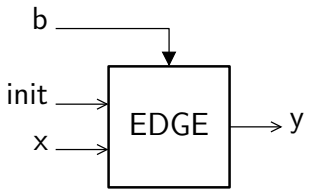

Figure 4.1: Representation of the conditional activation construct.

Jahier et *al.* have used this mechanism to model the real-time scheduler behavior: they have constrained the parallelism of the various processes sharing a single processor with conditional activations, such that only a single process can own a CPU at any time.

Using quasi-synchronous clocks

In order to restrain the processor clock synchronization nondeterminism, they have made a hypothesis about their timing behavior: these are assumed to satisfy the quasi-synchronous property proposed by Caspi et *al.* [CMR01]: "A set of clocks is said to be *quasi-synchronous* if, between two successive activations of any clock, each other clock is activated at most twice." This assumption is helpful to represent a realistic behavior of a set of processing elements, because it expresses that each clock does not drift "too much" from each other. Moreover, in a set of periodic reactive processes whose activation clocks satisfy this property, each process is guaranteed to miss at most one sample of the output of another in a row. An architecture whose processors related clock satisfy this property is qualified as *Globally Asynchronous Locally Synchronous* (GALS). Several inter-task communication mechanisms have been advised to ensure the semantic equality between purely synchronous application models and their deployed counterpart over GALS architectures: for example, Scaife and Caspi suggested the double buffer technique [SC04].

Assuming that the modeled architecture satisfies the GALS property, Jahier et *al.* used a Lustre implementation of a node generating a set of quasi-synchronous clocks.

Finally, non-instantaneous tasks have been modeled with delay intervals, expressed as numbers of logical steps. Yet the distant communications have been abstracted with simple synchronous signals.

Their prototype tool, AADL2sync, allows the virtual prototyping of an architecture description including the mapped target software, by producing a synchronous model of the overall system. Some difficulties have been encountered to "execute" the resulting synchronous program because the studied application (supplied from a real-world software) involved a complex initialization sequence. Also, the usage of the AADL language implied some strong abstractions impacting the communication mediums behaviors. Indeed, as stated in section 2.3 on page 6, the AADL language supplies only the ability to depict simple buses with a kind of latency and does not

provide any support for the realistic description of special communication links and protocols (*e.g.* analog link or serial buses with advanced protocols like a STD-MIL-1553B bus).

We are now able to notice that these two modeling methods, based on the usage of synchronous languages to describe asynchronous behaviors, have revealed once more the strength of the synchronous paradigms in terms of representation of asynchronous systems.

4.3 Modeling frameworks

A set of modeling frameworks has been studied at the same time. Each follows various concepts and techniques so as to put forward new modeling techniques.

4.3.1 The Rosetta framework

The primary idea of the ROSETTA framework proposed by Kong and Alexander [KA03] is the separation of concerns. In the purpose of easing the composition of heterogeneous specifications coming from various domains, it defines specific vocabulary and semantics for each one, then facilitates the gathering and specification of a system from different "engineering perspectives". Hence, it eases the collaborations, taking the cross-disciplinary effects into account.

A ROSETTA specification uses a collection of *domains* to describe system models (called *facets*). As said before, domains are vocabulary and semantics for defining the facets. A facet is built from the requirements, behaviors, constraints or functions over a specified domain. For example, a domain can be the functionalities, the constraints on time, the constraints on power, etc. and an associated facet is a description using the associated semantics and vocabulary to express functionalities, constraints on time and power respectively.

Furthermore, a domain interacts with each other and then associates constraints to the interconnections between the facets. Since there does not exist a common semantics between the domains, the facet interactions depend on their related domains.

Finally, a system description is a composition of facets, domains and interactions represented by *facet algebra* expressions.

This approach has the advantage of allowing the collaboration of various designers in order to build complex systems. But a major drawback is the difficulty to simulate such system because of the heterogeneity of employed semantics. At last, the need to learn these concepts and languages can drastically limit the usage and efficiency of this modeling framework.

4.3.2 The Metropolis framework

The METROPOLIS framework has been proposed by Balarin et *al.* [BWH+03] to target heterogeneous design models.

Primarily, as ROSETTA, the main ideology of the associated modeling method is the separation of concerns. In fact, it considers two initial spaces: the *application space* and the *architecture space*. The goal of the design flow is to bridge the gap between application space and architecture space. Thus, the development process covers the three following aspects: a top down application development, a bottom up *design space exploration*, and eventually a platform mapping.

The core of METROPOLIS is the *Metropolis Meta Modeling* language (MMM) and a dedicated compiler transforming an MMM description into an internal representation, along with multiple back-ends performing simulations, verifications and synthesis from the models.

Writing system models with the MMM language consists in specifying *computing processes*, *medias* and *quantity managers*. A computing process, also called *active object*, encapsulates a thread and represents the computation activities; a media (*passive object*) describes a service implementation; quantity managers contribute to schedule access to resources and quantities. Next, a model is a network of active objects using the passive ones through *ports* characterized

with an *interface*. This division of computations and communications facilitates the modularity, thus the reuse of subparts of models for other designs. A process execution is a sequence of *events*, representing an action of the encapsulated thread (*e.g.* starting or terminating to read or write a media). Finally, *constraints* over these events, written with logic formulas, restrict the set of legal executions of a process: they can represent coordinations of behaviors through synchronizations of events (*e.g.* mutual exclusion). Hence, the meta-model supplies a non-deterministic behavior modeling capability because two events can happen in any order if there is not any constraint between them.

The proposed design flows are based on several models, each represented with the previously described formalism.

Functional model. A functional model uses a network of processes and medias representing unbounded FIFOs. Refining a functional model consists in adding constraints over the processes events.

Architecture model. The architecture model describes a set of services provided to the functional model. For instance, a computation service supplied by a CPU can be "execute", "read" or "write"; communication mediums can also implement services like "transfer", "request" or "acknowledge". Just like the processes of a functional model, a service implementation is a sequence of events. The difference comes from the associated costs (*e.g.* energy or time), measured by quantity managers. These can serve to model shared resources (*e.g.* CPU) too. In addition, there exists a standard library of quantity managers because the models usually manage common quantities.

Top layer model. A manual mapping transformation links the functional and the architecture models without any modification of both networks. It relates the two initial descriptions by synchronizing events between them with new constraints. This produces a new encapsulating top layer network, whose legal executions become the intersection of the execution sets satisfying the new constraints.

One can perform many operations on these models by using several front-ends:

Synthesizing. Automatic code generation is only available on a subset of what the MMM language can represent. So as to schedule a concurrent specification on computational resources with limited support for concurrency, this front-end uses the concept of *quasi-static scheduling synthesis*: the tool analyzes the application and tries to transform it into a statically scheduled set of tasks [CKL+02]. When only some dynamic behaviors remain, it then employes a dedicated library to manage the communications. Since it handles all the final tasks, it can perform global optimizations on the software.

Simulating. Simulations can be carried out from a model using another tool translating the specification into an executable SystemC model [YSWB04]. It also generates C++ *monitors* analyzing the extracted traces and reporting violations of specified formulas written with the *logic of constraints* (LOC) proposed by Chen et *al.* [CHBW04].

Verifying. As the MMM has a well-defined formal semantics, some tools can perform formal property verifications. For example, a back-end interfaces the model to the Spin model-checker [Hol97], and another checks LOC properties [YHC+06].

To sum up, we can observe that the Metropolis framework has been designed to model heterogeneous systems, and primarily follows a concept of separation of concerns. Indeed, the proposed modeling technique involves independence between the computations and the communications, as well as between the functional model and the architecture one. Moreover,

the synthesis possibilities are relatively marginal owing to the limited support of the meta-model by the synthesis tool.

Whereas this is clearly an ambitious and promising approach, it has not been extensively used in the industry.

We think that, despite this solution may fit with the needs for system design when the choices for hardware or software implementation of certain functionalities remains to be done very late in the development flow, it does not address the issue of abstraction levels. Besides, the need to learn a new and complete language can limit its impact.

An enhanced version of this framework is currently under study in order to address some practical problems encountered during the case studies the designers performed [DDM+07].

4.3.3 Transaction-Level Modeling with SystemC

The Transaction-Level Modeling concepts

The main goal of the *Transaction-Level Modeling* is to permit an early creation of virtual platforms, on which the real embedded software can run: it allows early development and evaluation of the software. Thus, it helps to drastically reduce the design costs and the time-to-market.

The abstraction levels of Transaction-Level (TL) models are intermediates between purely functional descriptions and cycle accurate models. Building such models involves a component-based approach so it eases the reuse of previously designed subsystems. Furthermore, they are simpler to design than cycle accurate ones, and run really much faster.

In a Transaction-Level model, *modules* represent hardware blocks and high-level communications are called *transactions*. On the other hand, a transaction is a data exchange between an *initiator* and a *target*, through a connected *master port* and *slave port* respectively. Contents of transactions depend on the associated protocol: for instance, it generally includes the address of the target and exchanged data. In addition, whereas Donlin [Don04] suggested to represent interrupts (unidirectional data exchange which does not require any additional protocol) with synchronous *signals*, others propose the usage of specific transactions.

Donlin also defined four levels of abstraction to characterize the various TL models:

- The *Communicating Processes* (CP) and *Communicating Processes plus Timing* (CP+T) classes of models are architecture independent and correspond to a functional view of the modeled system, considering parallel processes with parallel point-to-point communications. Contrary to the CP models, the CP+T ones contain timing information. In opposition to Donlin, we don't regard these abstraction levels as transactional because they just reflect the functional behavior and do not necessarily need the transactional concepts.

- The *Programmer's View* (PV) and the *Programmer's View plus Timing* (PV+T) classes of models are intended to represent a complete platform with all the necessary information needed to run a real embedded software: a set of components linked together with communication channels. In a PV model, the components are assumed to have a reactive behavior, *i.e.* the communications and computations take zero time. A TL model at the PV+T level of abstraction is useful for preliminary performance analysis of the modeled systems, and is generally built from a "functionally equivalent" model described at the PV level.

The comparison of models at these different levels of abstraction and at the corresponding representations at the RTL one is still an open problem, as stated by Giovanni Funchal [Fun07]. Additionally, Cornet et *al.* [CMM08] have studied techniques to add non-functional aspects to TL models, *e.g.* to add timing annotations to a model at the PV level of abstraction to get a corresponding PV+T model with the same functional behavior.

Using SystemC to build Transaction-Level Models

Among others, Bart Vanthournout [Van04] has suggested to use the SystemC library to implement Transaction-Level models.

Firstly, this allows an easy reuse of the SystemC components and concepts (modules, ports, signals, etc — *cf.* section 2.2.1 on page 6). Also, the underlying C++ constructs provide encapsulation mechanisms and already expendable library utilities: for instance, one can easily abstract a hardware component decoding images with a dedicated library therefore speeding up the simulations.

At last, it exists an already available library and an upcoming standard [Ros05] introducing several key concepts: standard interfaces describe the services implemented by communication channels carrying the transactions. Also, they can be blocking or non-blocking, *i.e.* assumed to be able to have a reactive behavior or not. Besides, interfaces can be bidirectional or unidirectional, *e.g.* a request-response sequence can either be modeled with a single bidirectional transaction or two unidirectional ones: the first solution improves the simulation speed, but cannot allow the precise timing analysis that the second does while permitting an interlacing of different requests on a single channel.

Formalization of Transaction-Level models written in SystemC has been studied by Matthieu Moy [MMM06], and Claude Helmstetter [Hel07] has worked on the validation of such models, but these aspects remain the major issues of this approach. However, the primary goal of SystemC is the simulation of complex embedded systems, and it had not originally been thought for formalization purposes.

After this overview of various attempts to create global approaches using models, we can now present several model-driven implementation solutions.

4.4 Model-driven implementation

4.4.1 Using a resource-oriented model

Kim and Choi [KC07] have proposed a technique making an extensive use of the state-chart formalism. In this approach, the whole embedded software is considered to be divided into an application and a system part. The latter serves to interpret hardware information as an interface between the application and the hardware: it often contains semaphores, message-queues, etc, and controls the software execution. Hence, the system part represents the operating system as well as the device drivers, and is considered here as a set of methods provided to the software in order to control all the resources.

Furthermore, they divide these resources into hardware-oriented and software-oriented classes. The former class encompasses the CPUs, memories, digital I/Os, serial ports, AD/DA converters, sensors and actuators, etc, whereas the later class represents the system software resources, thus the semaphores and the scheduler for instance.

They have defined a Resource-Oriented Model (ROM), between a Resource Independent Model (RIM) and an Implementation Specific Model (ISM):

Resource-Independent Model. The RIM describes only the software behavior. It corresponds to the logical architecture of the system, with abstract synchronization and communication mechanisms.

Resource-Oriented Model. It is the RIM plus a *resource model* including the constraints and properties of the resources. This resource model captures hardware related information in terms of constraints concerning its behavior and timing to restrain the software. It also includes the concept of availability of resources in order to describe the behavior of

shared memory components for instance: schedulers, semaphores, etc, are described using state-charts.

Implementation Specific Model. This implementation model is divided into two parts: the *dynamic* and the *static* ISMs. The first is the detailed behavioral description of the code of the software functions. It represents the whole software model, along with the software-oriented behaviors included in the resource model (scheduler, semaphore, device drivers, etc). To the contrary, the static ISM includes information about interfaces provided by the resource model (thus the hardware and the system part of the software) to the application model. This results in a specification of the needed implementation details, such as data structures and type.

This is an interesting approach in our point of view: they propose to identify and describe properties and constraints from the implementation platform, notably the timing and availability related to the resources, and to integrate it into the application model to build the ROM. Moreover, all the models are executable, thus the ROM can be simulated and serve to execute the software model combined with the representation of the resources provided by the target platform.

Nevertheless, this approach necessitates the modeling of the precise behavior related to each resource supplied by the platform. It hence does not address the abstraction issue and leads to the design of models containing too much details to be really usable in practice.

Another negative aspect comes from the formalism used to represent the models. In effect, the state-chart set is too rich to allow a simple formalization, and possible constructs give non predictable behaviors. There also exist a lot of semantics related to this formalism, but no real verification tools related to it.

4.4.2 Lustre extensions for Time-Triggered Architectures

The Time-Triggered Architectures

The *Time-Triggered Architecture* (TTA) [KB03] designed by the TTA-Group[3] has been proposed to target safety critical applications. This kind of architecture is most widely used and popular in the automotive industry.

Firstly, it employs a division of the software into multiple parts, allowing a modular design and verification of the systems. The architectures are a composition of subsystems (called *nodes*) using asynchronous (but *a priori* lossless) communications, ensured by using the *Time-Triggered transmission Protocol* (TTP) based on *Time Division Multiple Access* (TDMA) bus access strategy. It also uses synchronous computations taking advantage of fully specified communication interfaces between the subsystems, as the nodes run a *Time-Triggered Operating System with TTP support* including a transparent fault tolerance layer.

Besides, since the TDMA protocol employed by TTP is based on a sparse time model, each subsystem can use a global notion of time. Indeed, this time management technique divides the time-line into activity and silence intervals. Thus, it allows a static scheduling of the application software, consequently leading to an easier implementation and verification of the systems.

Programming Time-Triggered Architectures with an extended Lustre

This proposition has been made by Adrian Curic [Cur05]. The goal of his work was to design a method to specify applications targeting Time-Triggered Architectures in Lustre allowing automated code generation. Employing this language also benefits from the available related tool-set.

[3] http://www.ttagroup.org

He proposed an extension of LUSTRE comprising (quite "generic") real-time and distribution directives. For instance, he has improved the definition of the streams with *availability dates*, and added a notion of *task* into the language. A method to perform an automatic mapping of these tasks using static analysis techniques has been presented too. He also suggested some verification techniques over this kind of models, such as specific scheduling verifications according to the TTA architectures. He also put forward to transform the model of the distributed implementation into an equivalent sequential one (described with classic LUSTRE) so as to test and check some implementation properties.

To sum up, we can observe that this solution aims well-defined architectures with related interfaces and behaviors. This led to implicit abstractions which could be introduced as extensions into the LUSTRE language. But this is a very restrictive and specific approach, as it implies the usage of the TTA architecture and its constraints, along with the usage of LUSTRE in an exclusive way.

4.4.3 Using SystemC for automatic generation of software

Krause et *al.* [KBR05] proposed a general method consisting in target software generation. They designed a tool taking a SYSTEMC model of the software and a specification of a *Real-Time Operating System* (RTOS) API, and then building the real target software.

This suggestion is included in the proposition of a more general framework based on the refinement of SYSTEMC/TLM models from the CP to the PV+T level of abstraction (*cf.* section 4.3.3 on page 19). The final steps of this design flow are based on the PV+T representation of the system, annotated with few specific *macros* to specialize the mapping of the software modules encapsulating a thread in one hardware processor component. So, a direct transformation process configured with a given target RTOS API characterization is able to generate the real application software. The same method applied with a SYSTEMC RTOS model configuration provides a refined virtual prototype including system calls to a SYSTEMC operating system model (an abstract scheduler and a resource manager), in order to simulate and verify more accurately the designed system.

Considering the communications, they used a dedicated refinement tool to bind the calls to channel methods on characterized primitives (*e.g.* device drivers or memory-mapped I/O).

Instead of starting from a very high level of abstraction, they proposed to transform a detailed model (since their tools take a PV+T model as input). Hence, the manual implementation work has to be done on the SYSTEMC model of the platform rather than on the real target. The main advantage of this approach is that the related design flow eases and speeds up the simulation of the applications mapped on the RTOS model. For all that, due to the limitations of SYSTEMC, the timing abstractions of the modeled RTOS tasks could introduce some side effects over the simulated system behavior.

4.4.4 Other suggestions

Considering other software code generation suggestions, several other solutions have been brought up.

The GIOTTO language, proposed by Benjamin Horowitz [Hor03], targets embedded systems with a periodical control such as TTA architectures, with a strict separation of timing and functionality. XGIOTTO [SGH03] follows the same way but introduces a notion of *logical lime* to make the timing behavior of a program deterministic. The associated compiler will then map the program on the given platform only if the logical execution time can be guaranteed.

OCARINA [Ver06] is a tool devoted to the generation of application software from AADL models. It can also generate *Petri-nets* in order to allow some formal analysis of this kind of models. Besides, the Ravenscar profile [Bur99] defines a restricted set of ADA in order to

ensure that produced programs support the usage of a deterministic scheduling as well as static analysis checking the absence of deadlocks or priority inversions for instance. Moreover, the annex D of the AADL specification [SAE04] describes some coding guidelines to translate the AADL software components into ADA or C source code. Based on this tools and considerations, another proposition aiming at synthesizing a target software from an AADL model has been suggested by Zalila et *al.* [ZHHP07]: their tool can generate ADA source code satisfying the Ravenscar profile.

Also, Gauthier et *al.* [GYJ01] proposed a method for automatic generation of application specific operating systems and software code. They considered a small but flexible kernel, providing generic services such as communication, I/O or memory management, but does not support neither analysis nor simulation of the models. Also, the descriptions given to the proposed tools are specific to this approach and does not address the abstraction issues.

4.5 Discussion

To make a long story short, we can notice multiple aspects.

On one hand, solutions based on modeling techniques and languages targeting a wide class of platforms entailed the appearance of complex and relatively uncommon tools. On the other hand, some presented model-based implementation approaches have been proven efficient if the characteristics and constraints of the target architectures are first identified and exploited. For example, we have seen some language extensions so as to propose solutions targeting well-defined classes of platforms integrating specific real-time operating systems or communication mediums. Moreover, we could observe that using synchronous languages to design system-level models have multiple advantages: the formal semantics and intrinsic parallelism of this paradigm lets use the resulting models to perform automatic model transformations along with verifications and validations.

At last, approaches focusing on model-driven implementation usually follow two main ideas: either they involve too much details in the system-level models (*e.g.* the solution using SystemC to target specific real-time operating system interfaces or the one based on the resource oriented model does not address the abstraction issue), or they are specific to a restrained class of target architecture (*e.g.* Lustre extensions targeting TTA systems).

Chapter 5

Contribution

5.1 Our approach

The long-dated goal of our work is to propose a model-based development solution for embedded control applications taking the system-level code into account.

However, past experiences have shown that finding a general approach is too difficult in this context and thus generally leads to complex, inefficient and little-used solutions. Yet starting off with the identification of the characteristics and the constraints imposed by a class of execution platforms and integrating them in the system-level models is somehow more effective.

In order to gather these information, we have chosen to analyze the Proximity Flight Safety control system (*cf.* chapter 3 on page 9), an existing platform which had been previously studied by some members of the team. Nonetheless, since it was obviously not possible to work with the real system and as we have deemed that the complexity of the real software is too high, we have decided to carry out some abstractions and simplifications over the system, as well as to design a virtual prototype of the platform in SystemC and to manually deploy the application on it. As stated in the previous chapter, using Transaction-Level models described with SystemC allows the design of more precise descriptions than using AADL ; it also produces executable models well-tailored to permit a good understanding of the system under study.

Afterwards, we have tried to perform some abstractions on this system, while designing synchronous models in Lustre.The goal of this last step was to identify the needed information from the detailed virtual prototype designed in SystemC in order to characterize an abstraction level accurate enough to represent the behaviors and constraints of this class of system.

Using Lustre to create a system-level model of the PFS has multiple advantages. Firstly, the formal characteristics of this languages would let use the designed models to perform some automatic property verifications along with simulations. This aspect also allows a possible exploitation of a model written in Lustre to carry out (partially or not) automatic system-level code generation from it. Finally, we have seen in section 4.2 on page 15 that using the synchronous paradigms to design system-level models have already revealed its efficiency.

At last, generalizing these aspects would allow us to propose a modeling method suitable for integrating the system-level code into the design flow of the targeted class of systems.

5.2 Prototyping the PFS

5.2.1 Abstractions and simplifications of the system

As a first step, we have simplified the PFS system in order to get a realistic but workable case study. We focused on the communications between the MSUs and the rest of the PFS system, because it involves multiple interactions. Moreover, we reduced the physical complexity of the environment, by performing the navigation in an unidimensional space and suppressing the sun pointing phase of the CAM. This involved sharp software simplifications and easier validation since there is no need to model a complex physical behavior, but no loss of generality. We put two communicating tasks on each MSU however, to avoid handling a degenerated case.

The modeled PFS

Since we have chosen to model an unidimensional physical environment, the information provided by the sensors have been simplified: the DTGs now send acceleration information and the PDE (an abstraction of the four initial PDEs) receives discrete acceleration commands.

Moreover, the three DPUs have been joined into a single FTCP component so that we can ignore the complex inter-DPU communications not necessary to model the PFS. The associated software, a single cyclic reactive task with a period of 50ms, performs the navigation: its goal is to reach a given point by means of sending commands to the PDE.

Concerning the MSUs, we have considered two communicating tasks, each with a reactive behavior but different activation periods: the Controller runs each 50ms and the Navigation task managing the ATV during a CAM, each 100ms. We chose *harmonic*[1] periods to allow the *static scheduling* of the tasks, thus simple inter-task communication mechanisms. Otherwise, we would had to treat an unrealistic case because the constraints involved by dynamic scheduling bring about too much validation difficulties in order to be accepted with respect to the required safety of the system. Also, in this context there are often means to circumvent creating non-harmonic tasks. So as to enforce the interactions between the MSUs, we also added two hardware links to let them exchange their respective state (in CAM or not) as well as their health status.

Next, we did not consider the CPFs, because they do not use unrepresented communication mediums (since the system bus is a serial link) and mainly serve as remote logging system and relay to commands arriving from the Earth or the International Space Station. Indeed, in the functional point of view the impact of the associated serial buses is limited because the telemetries and telecommands (the low-level ones) are not vital to the functioning of the PFS system.

Finally, The FTCP now sends start commands to the MSUs through a new hardware link (GPIOs) when the ATV reaches a specified position. Whereas the default role of each MSU is hardware encoded in the real system, the FTCP now sends this default status to the MSUs through the spied system bus to replace the telecommands normally received from the CPFs.

5.2.2 The fine-grain model

First, we implemented a transactional level model of the whole system in SystemC to ease the development of the software and allow fine-grain simulations. This led to a faithful virtual prototype, on which we could design a simple synchronous reactive application in Lustre.

The virtual platform

The figure 5.1 on the facing page presents an overview of the TL model of the considered platform. We represented the GPIO related communications with synchronous signals for simplicity, but we would have modeled them with transactions through the router.

We also used a *loose-timing* mechanism to represent the clock drifts: randomly shifting the simulated execution time associated to each computing element lets simulate a realistic Globally Asynchronous Locally Synchronous system behavior. It also reduces the impact of the "non-specified but deterministic" side effects entailed by the SystemC library simulation core. Indeed, even though a SystemC model integrates an implicit non-deterministic behavior (*i.e.* it can represent a set of concurrent threads possibly leading to multiple legal schedules), the utilized execution library always involves the same execution sequence. Adding randomly chosen timing behavior hence allows a better exploration of the possible model executions.

Furthermore, the physical environment has been simulated by a separated module including a thread: this technique allows a readily adaptable physical simulation time resolution and thus permits a trade-off between the simulation speed and its accuracy.

[1] A periodic task set is said *harmonic* if every period evenly divides all larger periods.

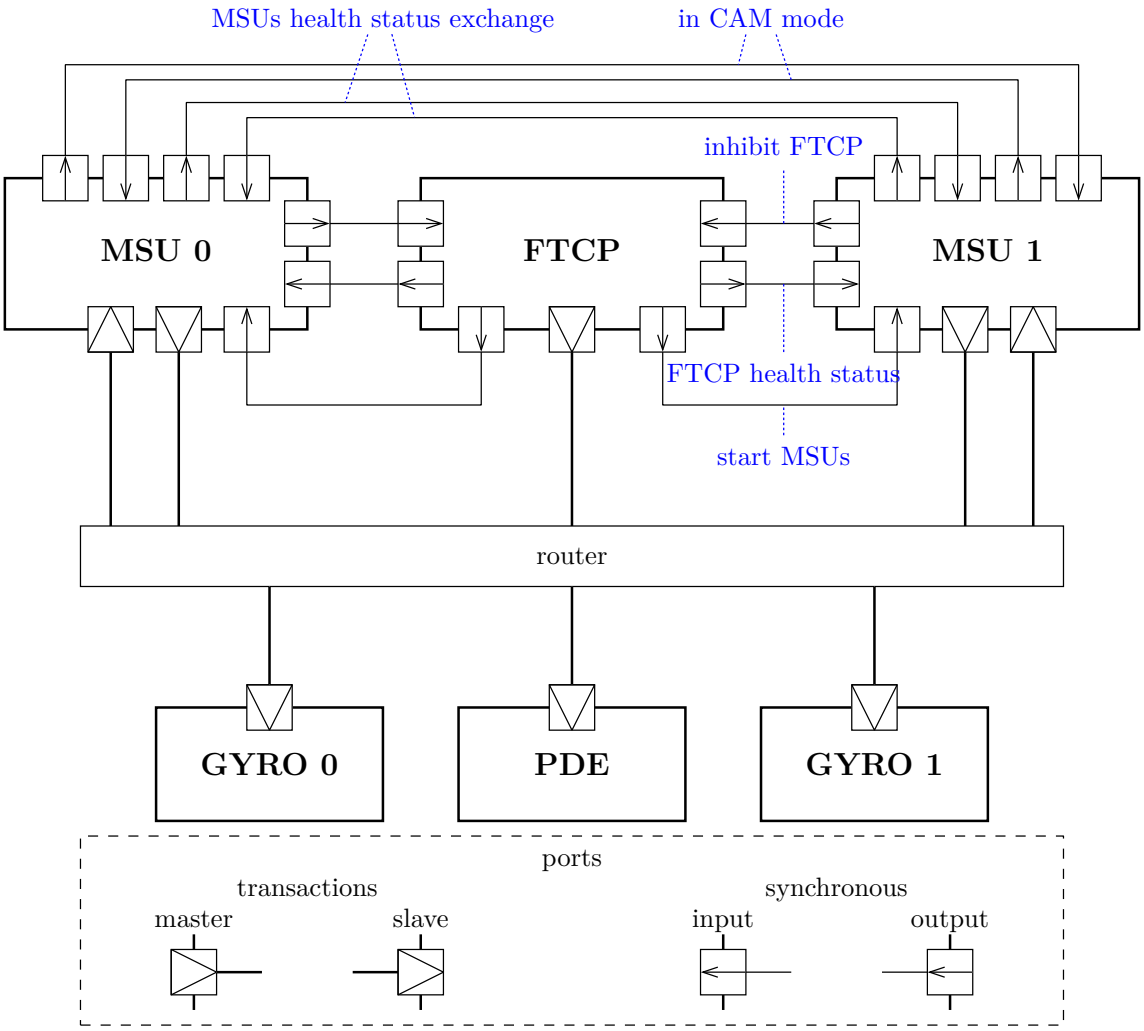

Figure 5.1: Transaction-level modeling of the PFS architecture. Both MSUs and FTCP modules include a thread, executing the statically scheduled application tasks and the system-level code. As its name indicates, the router only carries transactions (e.g. system bus communications and analog pulses, etc.), without any other behavior: we did not timed the communications because of the scale difference between their latencies and the software tasks execution time.

The application deployment

As said before, we have implemented the three tasks of the MSUs and FTCP software in Lustre. A Lustre node associated to an application task can be considered to be a memory and a set of functions : one to set the input values, another to get the outputs and two others to initialize and compute a basic step.

Integrating the synchronous application onto the previously described virtual platform involved the development of system-level code encompassing the static scheduler behavior and the control of the input and output communication mediums. The algorithm 5.1 on the next page presents the behavior of the system-level code needed to execute the two periodic reactive synchronous tasks we deployed onto each MSU. Hence, the goal of this software part is to interface the real application tasks and the services provided by the platform. For instance, we have considered the timers, serial bus and GPIO management primitives as such services. The figure 5.2 on the following page illustrates the implementation concept of the two application tasks onto an MSU processor model.

```
initialize(Navigation)
initialize(Controller)
turn ← Even                          // turn can be either Even or Odd
loop
    set_inputs(Controller)           // retrieve and set the input data of the Controller task
    step(Controller)                 // run one step of the Controller
    get_outputs(Controller)          // get (and possibly send) the computed output values
    if turn = Odd then               // shall we run the Navigation task?
        set_inputs(Navigation)       // retrieve and set the input data of the Navigation task
        step(Navigation)             // run one step of the Navigation
        get_outputs(Navigation)      // get (and possibly send) the computed output values
        turn ← Even
    else
        turn ← Odd
    wait_for_next_period             // wait until the end of the current period
```

Algorithm 5.1: Basic algorithm triggering the execution of the Controller task twice as often as the Navigation task. The considered period is thus the one of the Controller.

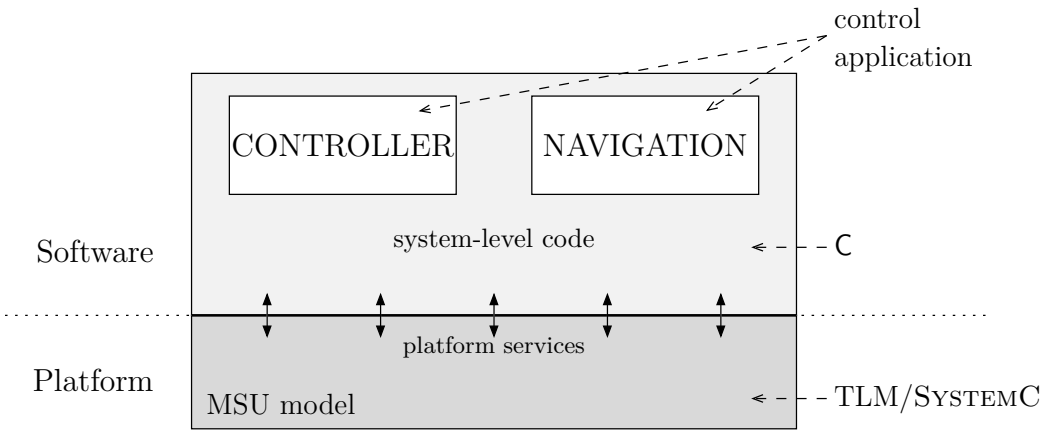

Figure 5.2: Block diagram depicting the software integration related to an MSU model. The system-level code manages the two application tasks and utilizes the services provided by the SystemC *platform.*

5.2.3 Validation

We have validated the obtained system prototype by carrying out many simulations. Indeed, such transactional-level models written in SystemC does not currently support formal validations as it is not the primary goal of these models (*cf.* section 4.3.3 on page 20). However, the modeling possibilities of this library allowed us to produce a virtual system using faithful communication mechanisms with respect to the real PFS chain (*cf.* appendix A on page 45 for an example of produced trace).

5.3 Synchronous modeling of the PFS

Further on, we tried to design a synchronous model of this system since we had well understood its behavior and the related characteristics and constraints. First, we designed a model mainly inspired from the work done by Jahier et *al.* hence following a discrete time basis.

5.3.1 Towards an "AADL2sync-like" synchronous model

In spite of highly inspired from the technique of Jahier et *al.* (*cf.* section 4.2.2 on page 15 — [JHR+07]), the synchronous model of the PFS has been designed in a more precise manner. In fact, so as to precisely represent the system-level code characteristics, one of the weak point of their approach concerns the lack of details available about the communication protocols. Modeling the notion of availability of data would be useful in order to represent the mediums behaviors.

Modeling the application tasks

Firstly, we used almost the same technique as the AADL2sync tool uses to represent the execution time of the AADL threads: the application processes are wrapped along with a node signaling the end of the process computation after a number of logical steps non-deterministically chosen into an interval. The figure 5.3 illustrates the representation of a single application task.

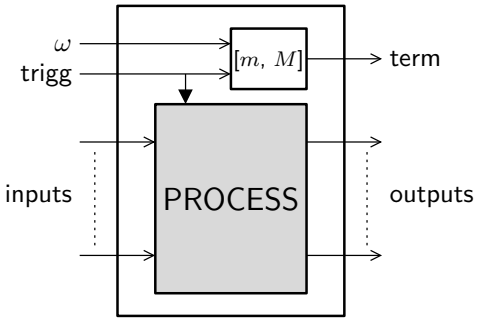

*Figure 5.3: Data-flow diagram describing the wrapper of an application process (*PROCESS*). "$[m, M]$" denotes a node whose output* term *becomes true only x logical steps after its input* trigg*, where $x \in [m, M]$. The non-deterministic choice is made using an oracle (ω).*

Next, the GPIO and serial communications have to be modeled in order to describe precisely the communication protocols used in the PFS system.

Modeling the GPIO mediums

A GPIO link can transmit boolean information: the associated hardware controller allows to read its value and detect any state change. The common implementations of this mechanism also permits to automatically reset its value when it is read.

In our context, modeling this behavior consists in representing the consumption of the boolean data: the pending value becomes *true* when a state change occurs on the signal representing the "hardware link"; reading it resets this awaiting value. On the other hand, writing a *true* value generates a state change on the associated boolean signal. The figure 5.4 on the next page represents the two related nodes, one for writing and the other for reading such GPIO links.

At last, we can notice that this modeling technique enables the representation of hardware links with multiple readers, since the boolean signal can be treated by any number of reading node[2].

Modeling the serial mediums

The other needed communication model is the serial link. The possibility to consume all received values in their arrival order matters in this case. Indeed, a real serial controller fills a bounded

[2]For practical purposes, due to physical reasons targeting several GPIO input pins would necessitate the usage of multiple output pins.

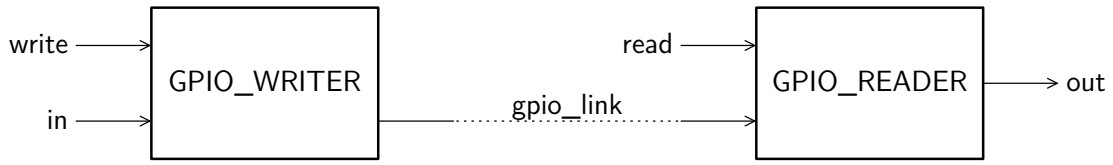

Figure 5.4: Data-flow diagram depicting the modeling of GPIO communications: the information transits among two computing elements (not represented here) through a state change on the gpio_link *signal, and the received value is consumed when* read *becomes true. As in the real mechanisms, a second state change before the consumption of the value will be lost: it is not memorized.*

buffer of received messages, and trying to consume data while this buffer is empty should be announced with an error report by the associated driver (or at least, it should be possible to know this vacuity).

We have chosen to use the same separation between the sender and receiver models utilized in the GPIO link modeling method so as to allow the representation of serial links with one initiator sending to multiple recipients (recall that we only consider point-to-point communication protocols). Also, the concept of availability of data has been expressed with a dedicated boolean signal whose state change indicates a new value sent on the medium during the previous execution step. The figure 5.5 depicts the nodes employed to model these behaviors.

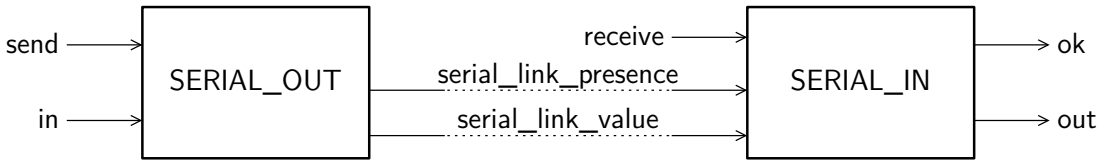

Figure 5.5: Data-flow diagram describing the modeling of serial mediums: a state change of the serial_link_presence *signal represents the presence of a new value on the* serial_link_value *link. This value is then memorized into a bounded FIFO by the* SERIAL_IN *node until its consumption by a* receive *command also positioning the* ok *output to true; if there does not exist any non-consumed value, then this output remains false.*

At last, modeling the whole PFS system consisted in creating three processor models and the associated communication mediums.

Modeling the whole system

Modeling a processor then consists in adding a static scheduler node controlling the execution of the tasks associated to a computing element: this can be the implementation of a simple automaton periodically triggering a sequence of application tasks execution, thus a model of the static scheduler.

The complete system model now consists of a set of processors (in fact merely representing the associated software) whose activation clocks satisfy the quasi-synchronous property because of the assumptions we have made on the relative clock drifts. The hardware controller models run concurrently with the software part, thus their basic clock is the one of the whole system-level model. Indeed, we can consider that the asynchronous timing only concerns the triggering of the application tasks and the software managing the platform services because the other nodes represent purely parallel behaviors.

The figure 5.6 on the next page depicts the modeling of two processors running two statically scheduled tasks each, and exchanging data through two GPIO communication links. The

QUASI-SYNCHRONOUS CLOCKS GENERATOR is a node generating, as its name says, a set of quasi-synchronous clocks. It is the same as used by Jahier et *al.*

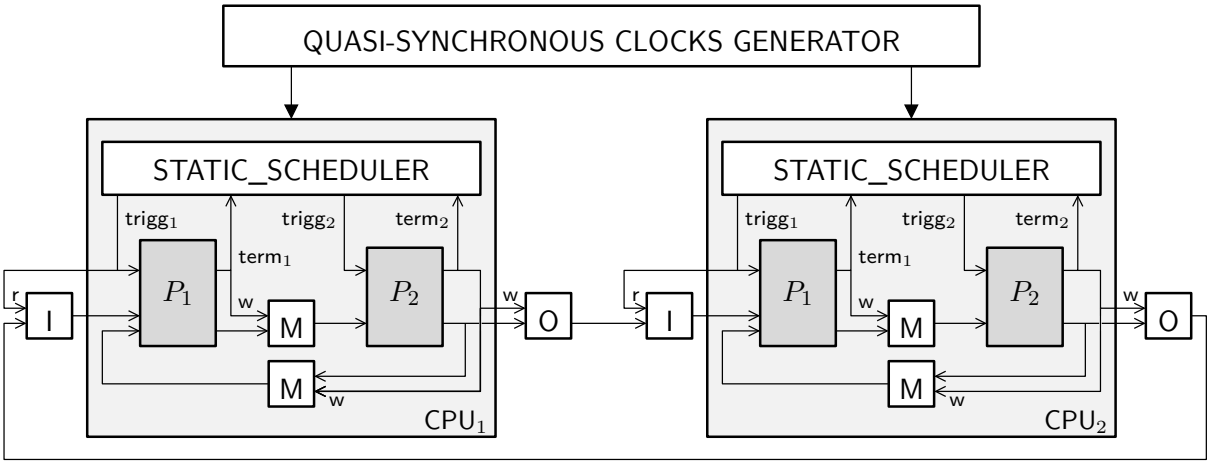

Figure 5.6: The modeling of two CPUs running the same software consisting of two wrapped processes (P_1 and P_2). For each CPU, the STATIC_SCHEDULER *manages the two tasks, triggering the execution of P_i with the boolean* trigg$_i$ *and receiving a termination signal* term$_i$ *when the task ends its computation. Moreover, these signals trigger the reading or writing of values from or to the GPIO links (through the nodes* I *and* O *running concurrently with the software), along with the writing of intermediate memory nodes* M*.*

Modeling the environment

In order to simulate the physical behavior of the PFS system, one need to represent the real time. At the software level (the processor model), the simulated time advance together with the basic clock of the processor node. However, the simulated real time of the whole system does not even exist. A practical solution has been to regard the real time clock as represented by one of the output of the quasi-synchronous clock set generator, or to add a new one in this set.

This synchronous modeling method produces a virtual prototype more accurate that what Jahier et *al.* obtained from AADL, since we have modeled the serial communications and GPIOs, notably the notion of *availability* of data. Yet the scale difference between the latency of the mediums (micro-second order) and the periods of the tasks (ten-millisecond order) implied (and allowed) some abstractions. But it would be easy to refine this medium modeling with communication timings by adding delay nodes on the related presence and value signals, and by using a fine-grain time resolution. This would imply much longer simulation times yet.

Nevertheless we could observe a lack of precision by executing the produced model, notably concerning the behavior of the system-level code. Indeed, in case of multiple receptions through a single serial link and within one simulation step, one cannot process all these data between two consecutive executions of a task needing them without any dedicated platform service representation. This solution also entails a timing accuracy issue deriving from the time resolution choice.

In order to address these problems, we have refined the resulting model so as to represent more precisely the behavior of the real system.

5.3.2 Towards a more precise synchronous model

The goal of this new modeling technique is to explicitly model the platform services management so as to represent the system-level code behavior more accurately than in the previously described method. Also, the main idea is to precisely represent the behavior of the Transactional-Level model described in the section 5.2.2 on page 26. Along these lines the refinement mainly concerns the concurrency and the improvement of the separation of the control and the computation parts of the software models.

In fact, giving details to the implementation model boils down to refine the control part of the software. Hence, this consists in elaborating the automaton represented by the static scheduler in the previously described modeling method. Gamatié et *al.* [GGB03] have used the same concept to model avionics applications, and we can also pick out the similarity in-between this software modeling technique and the separation of control-flow and data-path usually employed in the classic hardware context (purely synchronous in essence) in order to simplify the circuit designs.

However, this implies to refine the timing information of the model so as to be able to represent actions occurring during relatively different durations: *e.g.* executing a task step can take much more time than consuming data from a GPIO link.

Modeling the time

Detailing the timing precision could be achieved by modeling the SystemC scheduler behavior without the event management, thereby improving the dependability of the synchronous model with respect to the virtual prototype we designed in the section 5.2.2 on page 26. Indeed we could observe a main property of this TL model: it does not need any SystemC event because the scheduling of the threads is only performed with suspension times, and the software code running on the CPU modules does not need to be triggered asynchronously as it just periodically polls the state of the received signals (GPIOs) and reads the data spied on the system bus in a buffer automatically filled by a component of the platform.

Given these observations, we could design a synchronous pseudo-scheduler model inspired from the SystemC scheduler and managing a fixed set of software threads (in the sense of non-preemptible execution flow). The figure 5.7 represents the resulting node. Basically, it manages an always sorted array of threads with corresponding waiting times, and an increasing global time.

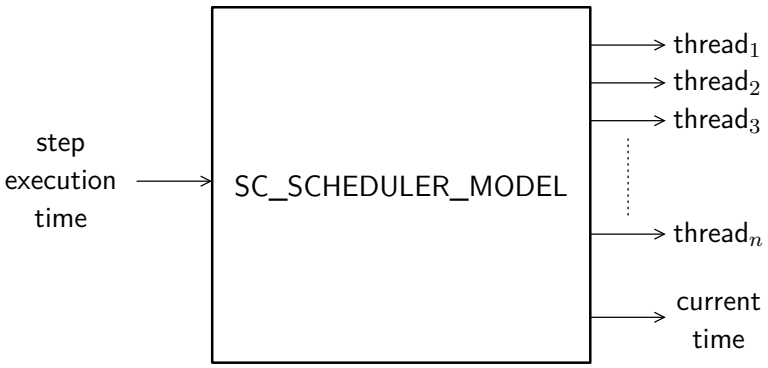

Figure 5.7: Representation of the node modeling the SystemC *scheduler behavior. At each step,* one and at most one *of the boolean output* thread_i *is true, nominating a unique thread to run. This thread will then output an execution time, assigned to the input of the scheduler. This model also outputs the current simulation time (cf. appendix B on page 47 for more details).*

We could then use this node so as to represent the time and manage the software components

of the modeled system. This has thus resulted in the integration of the time into the system-level model: the global time is then managed with discrete steps of variable lengths (whereas the steps was fixed in the previous method).

Refining the software controller model

The refined software controller model does not only trigger the application tasks execution anymore, but it also controls more precisely the services provided by the platform. The new controller automaton now contains more states, each one representing an action performed by the software layer (reading inputs, triggering a task step, writing outputs, etc.). At last, the software controller now outputs the simulated time related to its current state (thus the duration of the current action execution) in order to allow the global time management. For example, when its state represents the execution of a task, then it outputs the expected execution time, yet varying non-deterministically in a given time interval.

In this context, there is no need to wrap the application tasks anymore: their execution steps is now triggered by an output of the software controller, and the latter also knows their expected duration intervals.

Moreover, modeling a processor remains almost the same as before except that the related node now outputs the real time it has to wait before its next execution step. The latter information is now signaled by a new boolean input. Also, its basic clock is the one of the whole system model because a processor node can now encapsulate both hardware and software behaviors: the hardware parts behaves in parallel whereas the software ones are executed asynchronously.

Likewise, a notion of *feedback* has been introduced so that the software controller manages the platform services. For instance, a feedback can be the report of a read operation on a serial bus controller. Indeed, the system-level code may have to react differently in accordance with the presence or absence of data. For example, a serial input buffer may have to be fully emptied periodically, and the control automaton will then include a loop on a state whose meaning is to read a single data from this serial input.

The figure 5.8 represents the modeling of a single processor using this technique, and the figure 5.9 on the following page depicts an example of software controller behavior for this processor.

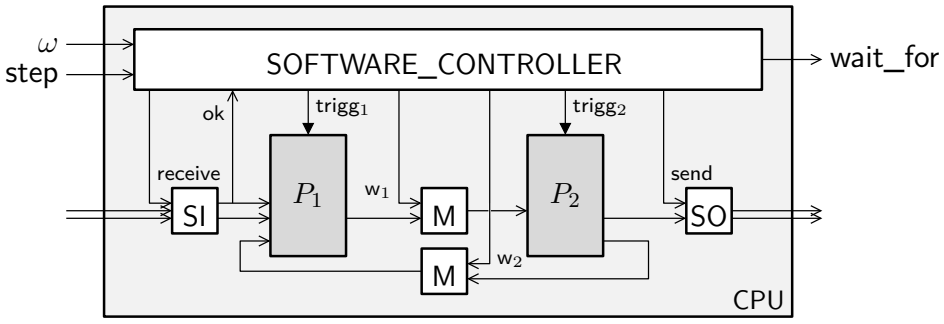

Figure 5.8: Representation of a processor running two periodic tasks. The two reactive application tasks (P_1 and P_2) are now conditionally executed and do not need a wrapper anymore. Moreover, the software controller manages the serial input and output nodes (SI and SO), as well as the memory nodes (M). Additionally, it outputs the real time (in fact, the simulated time) before its next execution step, varying non-deterministically according to an oracle (ω). Hence, the expected real execution time of the application processes are known by the software controller in this model.

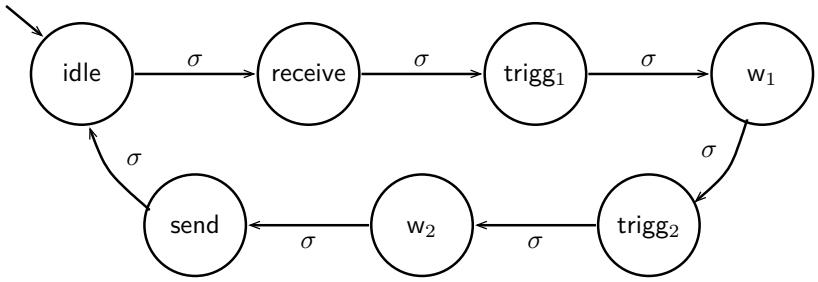

Figure 5.9: Example of finite state machine (a Moore machine) representing a possible behavior of the software controller of the figure 5.8 on the preceding page. The σ input corresponds to an execution step of the processor. In this case, the controller does not use any feedback information (e.g. ok*).*

The PFS model

The figure 5.10 represents the synchronous model of the PFS system.

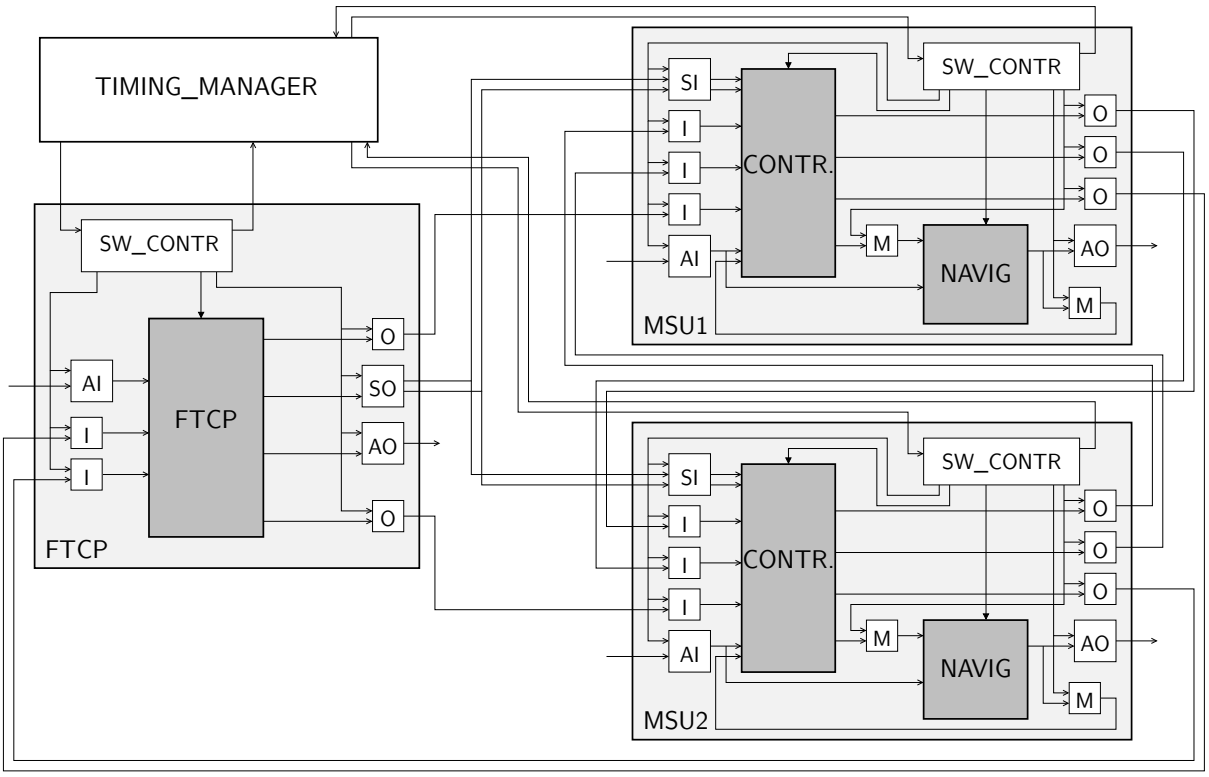

*Figure 5.10: Representation of the synchronous model of the PFS. We have represented the three processors (*MSU1*,* MSU2 *and* FTCP*) along with some GPIO links, serial mediums, and the timing manager. The* AI *and* AO *nodes are analog-to-digital and digital-to-analog converters models, almost identical to the GPIO models except that it manages always available data.*

A downside of this modeling method comes from the timing management: as the non-preemptive SystemC scheduler behaves, there is always one and at most one thread execution step at any time. Moreover, the resulting model does not benefit from the possibility offered by the synchronous languages to execute concurrently (within a single logical step) the application software running on different processor nodes (what the other technique does).

However, the attained timing accuracy leads to the opportunity to use this model as a virtual

platform precise enough to represent the behavior and timing of the system-level code and communication mediums.

5.3.3 Validation

The synchronous models of the PFS system have been simulated using Lurette, and we have also manually compared the traces we obtained between the three models (*cf.* appendix C on page 51 for an example of obtained trace).

However, concerning the formal verification of properties of these models, they use too much integers or complex constructs so the available tools could not be used easily. Since the SCADE tool-suite employs more used tools, we have in mind working on this remaining issue by converting the current Lustre programs to SCADE models. Hence, more time would have been needed in order to perform a better usage of these models.

5.4 Exploiting the models

5.4.1 Targeting model-driven development from the synchronous models

First, we can observe that the second modeling method allows an easy identification of the behavior of the system-level code. Indeed, for each processor, this is only represented by the software controller. Furthermore, in our context (management of statically scheduled application tasks with a reactive behavior), this code can be described with a simple Moore machine whose states represent the actions performed by the software controller (*e.g.* reading an input, triggering a task, etc.). Using the feedback information provided to the controller would boil down to add guards to the transitions. In addition, the time annotations, representing the expected minimal and largest times needed to perform each action correspond to additional outputs of this automaton merely used for simulation and verification purposes.

Moreover, the well-defined semantics of Lustre and the structure of the obtained models should also allow an automatic generation of the system-level code from the models. Indeed, the clear separation between the controller and the other software parts (*i.e.* the platform services along with the application tasks) can greatly ease this automation. Also, the outputs of the controller directly corresponds to simple actions: for instance, it can represent a call to a device-driver functionality, or directly the reading or writing of a memory-mapped I/O. Thus, using an annotated version of Lustre to describe the system-level models would permit an automatic generation of the code remaining to be manually written in the classic approaches.

5.4.2 Virtual prototyping

Another usage of these models would be to use it as a virtual prototype of the target platform including the behavior of the system-level code. In fact, the allowed timing precision involves a kind of model precise enough to be used in order to develop and test an application earlier and before the existence of the physical platform. Besides, since the whole system model has a well-defined semantics, then it should also allow formal analysis of the platform along with the real application.

5.4.3 Generation of the models

Finally, another possibility that we can point out is the generation of these synchronous models from a higher-level language or directly from a Transaction-Level model of the platform.

Indeed, since we used predefined communication medium representations (*i.e.* GPIO and serial buses) and we only considered statically scheduled application tasks, then we can propose the usage of a language (possibly dedicated) for the design of this kind of systems. Such language

should represent the constraints imposed by the platform and possibilities to define the behavior of the system-level code.

We can finally notice that considering the proposed synchronous modeling technique, representing a single processor can be regarded as translating a module containing a single thread from a Transaction-Level model of the platform. This points out a new solution to translate such TL models (described in TLM/SystemC for example) to Lustre, thus giving a clear semantics to a subclass of TL models.

Chapter 6

Conclusion

6.1 Summary

After the analysis of a real control application system, several communication mechanisms and the associated behaviors have been identified. Then, we manually designed a virtual prototype of a simplified version of this platform using these communication mediums in SystemC. We also created a dedicated control software in Lustre. Next, we implemented the supplementary code needed to integrate this application into the platform. Simplifications performed on this system allowed us to carry out simulations in order to validate the design.

The analysis of this virtual platform led to the identification of the behaviors needed to take the system-level code into account in a high-level model-driven approach to the system design. We could then design a synchronous high-level model of the whole system, including both software and hardware parts: communication mediums and system-level code behaviors were represented. The choice of using the synchronous paradigm to model this system came from its multiple advantages. Notably, many available tools would allow formal analysis of such programs, and it also have already proved its strengths in the modeling of systems.

In addition, we were able to propose a second synchronous model of this system, mostly inspired from the previous design and the SystemC simulation core behaviors. This model involved more precise timing and system-level code representation. Indeed, while the simulated time is roughly counted in terms of basic steps in the first synchronous model, the second uses the functional time representation built in SystemC. Thus, this eliminates the time resolution issue encountered in the first model at the expense of a loss of synchronous parallelism.

6.2 Observations

This work has allowed us to point out some observations.

First, the division of the target software into real application and control parts is a key concept in our synchronous modeling approach. Indeed, this helps to identify the part representing the system-level code using the services provided by the platform and managing the multiple application tasks.

Furthermore, a precise timing of the model is also needed to accurately represent the control code. In fact, the first synchronous model that we designed has shown that a too coarse time resolution leads to difficulties in the representation of advanced behaviors, such as the repeatedly processing of a serial input buffer just before activating an application task needing this data. Hence, an accurate description of the system-level code necessitates a precise representation of the services provided by the target platform.

At last, the work we have done also leads to a method allowing the formalization of a restricted set of timed transaction-level models. Indeed, the second synchronous model we have designed is a high-level description of a set of modules, each one encompassing a single thread and only communicating in a point-to-point manner. In other words, this transformation would be possible as long as the threads do not wait for asynchronous events, and when all the

data transfers made through transactions correspond to the direct mapping of functional-level communications on mediums using point-to-point protocols.

6.3 Perspectives

As a short-run perspective, we have in mind working on the formal verification of the synchronous models. Indeed, only manual comparisons and validations of the three models could be done through trace analysis. More time would have been needed in order to perform better comparisons of the simulation traces and formal verification of properties concerning the synchronous models. Since the global structure of the first model does not differ too much from the one proposed by Jahier et *al.* [JHR+07], we should be able to formally check several properties of this representation. Furthermore, our second model uses integers to represent the time, but it only performs additions and comparisons of such values. So, it may be possible to check properties about delays between two events by using abstract interpretation tools such as NBac. For example, it would be interesting to evaluate the maximum time necessary to take over the master role after an MSU failure, or the time elapsed between a failure detection triggering a CAM and the first emitted command to the thrusters.

Also, we can propose to study more embedded control applications in order to design new synchronous models of communication mediums.

We can also envisage to perform the automatic generation of the proposed models from a more appropriate formalism than AADL. Indeed, we have noticed that AADL models do not provide enough details about the communication mediums to allow a precise description of the platform. Additionally, in AADL there is a complete distinction between hardware and software parts of a system, whereas we need to precisely represent the behavior of the software supplying services to the applications. Alternatively, using the existing annexes of AADL to describe this behavior would not lead to models abstract enough.

On the other hand, the general framework *42* [MB07] makes a clear distinction between the control and the computation. Thus, regarding our observations and since *42* is a strictly hierarchic component-based formalism, we can consider the system-level code as a controller and the application tasks and platform services as managed subcomponents. Moreover, we can study the possibility to guide or automate the generation of the system-level from *42* high-level models or directly from an annotated version of a synchronous language such as Lustre.

Finally, the transformation of a subset of timed transaction-level models to synchronous ones should be more studied, since this could conduct to a clear formalization of such models.

References

[APP+98] Matt Aubury, Ian Page, Dominic Plunkett, Matthias Sauer, and Jonathan Saul. Advanced silicon prototyping in a reconfigurable environment. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Proceedings of WoTUG-21: Architectures, Languages and Patterns for Parallel and Distributed Applications*, pages 81–92, March 1998. (cited p. 5)

[Avi97] Avionics Application Software Standard Interface. ARINC Specification 653, January 1997. (cited p. 15)

[BB91] Albert Benveniste and Gérard Berry. Special section: Another look at real–time programming. *Proc. of the IEEE*, 79(9), September 1991. (cited p. 3)

[BBD06] Olivier Boudillet, David Berthelier, and Didier Dalemagne. Category A* Software Development for the ATV. In L. Ouwehand, editor, *Proceedings of the Data Systems in Aerospace conference (DASIA), May 22 – 25, 2006, Berlin, Germany*, volume 630 of *ESA Special Publication*, July 2006. (cited p. 10)

[BDF+06] Jean-Paul Bodeveix, Pierre Dissaux, Mamoun Filali, Pierre Gaufillet, and Fran cois Vernadat. AADL behavioural annex. In *Proceedings of the Data Systems in Aerospace conference (DASIA), May 22 – 25, 2006, Berlin, Germany*, http://www.esa.int/SPECIALS/ESA_Publications/index.html, 2006. European Space Agency (ESA Publications). (cited p. 7)

[BG92] Gérard Berry and Georges Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992. (cited p. 3)

[Bur99] Alan Burns. The Ravenscar Profile. *ACM SIGADA Ada Letters*, 19(4):49–52, December 1999. (cited p. 22)

[BWH+03] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *IEEE Computer*, 36(4):45–52, 2003. (cited p. 17)

[CCM+03] Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, Stavros Tripakis, and Peter Niebert. From Simulink to SCADE/LUSTRE to TTA: a layered approach for distributed embedded applications. *ACM SIGPLAN Notices*, 38(7):153–162, July 2003. (cited p. 2)

[CHBW04] Xi Chen, Harry Hsieh, Felice Balarin, and Yosinori Watanabe. Logic of constraints: a quantitative performance and functional constraint formalism. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 23(8):1243–1255, 2004. (cited p. 18)

[CKL+02] Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, Claudio Passerone, and Yosinori Watanabe. Quasi-static scheduling of independent tasks for reactive systems.

In Javier Esparza and Charles Lakos, editors, *Proceeding of the 23rd International Conference on Applications and Theory of Petri Nets (ICATPN'02), June 24-30, Adelaide, Australia*, volume 2360 of *Lecture Notes in Computer Science*, pages 80–100. Springer, 2002. (cited p. 18)

[CMM08] Jérôme Cornet, Florence Maraninchi, and Laurent Maillet-Contoz. A method for the efficient development of timed and untimed transaction-level models of systems-on-chip. In *Design Automation and Test in Europe (DATE)*, pages 9–14, Munich, Germany, March 2008. (cited p. 19)

[CMR01] Paul Caspi, Christine Mazuet, and Natacha Reynaud-Paligot. About the design of distributed control systems: The quasi-synchronous approach. *Lecture Notes in Computer Science*, 2187:215–226, 2001. (cited p. 16)

[Cur05] Adrian Curic. *Implementing* Lustre *Programs on Distributed Platforms with Real-Time Constraints*. PhD thesis, Université Joseph Fourier, Grenoble, France, July 2005. (cited p. 1 and 21)

[DDM+07] Abhijit Davare, Douglas Densmore, Trevor Meyerowitz, Alessandro Pinto, Alberto Sangiovanni-Vincentelli, Guang Yang, Haibo Zeng, and Qi Zhu. A next-generation design framework for platform-based design. In *DVCon 2007*, February 2007. (cited p. 19)

[Don04] Adam Donlin. Transaction level modeling: Flows and use models. In *CODES+ISSS '04: Proceedings of the international conference on Hardware/Software Codesign and System Synthesis*, pages 75–80, Washington, DC, USA, 2004. IEEE Computer Society. (cited p. 19)

[DORZ99] Lydie Du Bousquet, Farid Ouabdesselam, Jean-Luc Richier, and Nicolas Zuanon. Lutess: A specification-driven testing environment for synchronous software. In *International Conference on Software Engineering*, pages 267–276, 1999. (cited p. 5)

[FGHL04] Peter H. Feiler, David P. Gluch, John J. Hudak, and Bruce A. Lewis. Embedded system architecture analysis using SAE AADL. Technical note cmu/sei-2004-tn-005, Carnegie Mellon University, 2004. (cited p. 6)

[FN01] Masahiro Fujita and Hiroshi Nakamura. The standard SpecC language. In *ISSS 2001, September 30 - October 3, 2001, Montréal, Québec, Canada, Seventh ACM International Symposium on Systems Synthesis, Proceedings*, pages 81–86, 2001. (cited p. 5)

[Fun07] Giovanni Funchal. Comparison of embedded system models: from signals to transactions, 2007. (cited p. 19)

[GGB03] Abdoulaye Gamatié, Thierry Gautier, and Luic Besnard. Modeling of avionics applications and performance evaluation techniques using the synchronous language SIGNAL. In *Proceedings of Synchronous Languages, Applications, and Programming (SLAP'03)*, volume 88 of *Electrical Notes in Theoretical Computer Science (ENTCS)*, pages 87–103. Elsevier Science, Dordrecht, The Netherlands, 2003. (cited p. 15 and 32)

[GLG99] Thierry Gautier and Paul Le Guernic. Code generation in the SACRES project. In *Procedings of the Safety-critical Systems Symposium (SSS'99)*, pages 127–149. Springer Verlag, Huntingdon, UK, February 1999. (cited p. 15)

[GYJ01] Lovic Gauthier, Sungjoo Yoo, and Ahmed Amine Jerraya. Automatic generation and targeting of application specific operating systems and embedded systems software. In *DATE*, pages 679–685, 2001. (cited p. 23)

[Hal93] Nicolas Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993. (cited p. 3)

[HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991. (cited p. 3)

[Hel07] Claude Helmstetter. *Validation de modèles de systèmes sur puce en présence d'ordonnancements indéterministes et de temps imprécis*. PhD thesis, INPG, Grenoble, France, March 2007. (cited p. 20)

[HLR92] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Transactions on Software Engineering*, 18(9):785–793, September 1992. (cited p. 5)

[Hol97] Gerard J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. (cited p. 18)

[Hor03] Benjamin Horowitz. *Giotto: a time-triggered language for embedded programming*. PhD thesis, 2003. Chair-Thomas A. Henzinger. (cited p. 22)

[Jea03] Bertrand Jeannet. Dynamic partitioning in linear relation analysis: Application to the verification of reactive systems. *Formal Methods in System Design*, 23(1):5–37, 2003. (cited p. 5)

[JHR+07] Erwan Jahier, Nicolas Halbwachs, Pascal Raymond, Xavier Nicollin, and David Lesens. Virtual execution of AADL models via a translation into synchronous programs. In Christoph M. Kirsch and Reinhard Wilhelm, editors, *Proceedings of the 7th ACM & IEEE International conference on Embedded software, (EMSOFT'07), September 30 - October 3, 2007, Salzburg, Austria*, pages 134–143. Association for Computing Machinery, 2007. (cited p. 15, 29, and 38)

[KA03] Cindy Kong and Perry Alexander. The ROSETTA meta-model framework. In *ECBS*, pages 133–140. IEEE Computer Society, 2003. (cited p. 17)

[KB03] Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003. (cited p. 21)

[KBR05] Matthias Krause, Oliver Bringmann, and Wolfgang Rosenstiel. Target software generation: an approach for automatic mapping of systemC specifications onto real-time operating systems. 2005. (cited p. 22)

[KC07] Jin Hyun Kim and Jin-Young Choi. Embedded system modeling based on resource-oriented model. In *ECBS '07: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, pages 203–212, Washington, DC, USA, 2007. IEEE Computer Society. (cited p. 20)

[KL96] Apostolos A. Kountouris and Paul Le Guernic. Profiling of SIGNAL programs and its application in the timing evaluation of design implementations. *IEE Seminar Digests*, 1996(36):6–6, 1996. (cited p. 15)

[Lac98] Philippe Lacan. Automated Transfer Vehicle - Architecture and Development Facilities of the ATV Software. In B. Kaldeich-Schürmann, editor, *DASIA 98 - Data Systems in Aerospace*, volume 422 of *ESA Special Publication*, pages 233–+, July 1998. (cited p. 9)

[LGLL91] Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991. (cited p. 3)

[LTL03] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christphe Le Lann. Polychrony for system design. *Journal for Circuits, Systems and Computers*, April 2003. (cited p. 15)

[MB02] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. Foreword By-Ivar Jacoboson. (cited p. 14)

[MB07] Florence Maraninchi and Tayeb Bouhadiba. *42*: Programmable models of computation for a component-based approach to heterogeneous embedded systems. In *Sixth ACM International Conference on Generative Programming and Component Engineering (GPCE'07)*, Salzburg, Austria, October 2007. (cited p. 38)

[MDA] OMG model driven architecture. http://www.omg.org/mda/. (cited p. 14)

[Mil78] *MIL-STD-1553B, Aircraft Internal Time-Divison Multiplexing Data Bus*, 1978. Washington, D.C., Department of Defense. (cited p. 9)

[MMM06] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. LusSy: an open tool for the analysis of systems-on-a-chip at the transaction level. *Design Automation for Embedded Systems*, 2006. special issue on SystemC-based systems. (cited p. 20)

[Ope05] Open SYSTEMC Initiative, Language Working Group. SYSTEMC draft standard language reference manual 2.1, 2005. http://www.systemc.org/. (cited p. 6)

[Ros05] Rose, Adam and Swan, Stuart and Pierce, John and Fernandez, Jean-Michel. Transaction-Level Modeling in SYSTEMC, 2005. OSCI TLM Working Group. (cited p. 20)

[RR02] Pascal Raymond and Yvan Roux. Describing non-deterministic reactive systems by means of regular expressions. volume 65.5. Electronic Notes in Theoretical Computer Science, 2002. (cited p. 5)

[RWNH98] Pascal Raymond, Daniel Weber, Xavier Nicollin, and Nicolas Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998. (cited p. 5)

[SAE04] SAE. Architecture Analysis & Design Language (AADL). AS5506, Version 1.0, SAE Aerospace, November 2004. (cited p. 6 and 23)

[SC04] Norman Scaife and Paul Caspi. Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS'04)*, pages 119–126. IEEE Computer Society, 2004. (cited p. 16)

[SGH03] Marco Sanvido, Arkadeb Ghosal, and Thomas Henzinger. xGiotto language report. Technical Report UCB/CSD-03-1261, EECS Department, University of California, Berkeley, Jul 2003. (cited p. 22)

[SLC06] Oleg Sokolsky, Insup Lee, and Duncan Clarke. Schedulability analysis of AADL models. In *IPDPS*. IEEE, 2006. (cited p. 7)

[SLNM04] Frank Singhoff, Jérôme Legrand, Laurent Nana, and Lionel Marcé. Cheddar: a flexible real time scheduling framework. In John W. McCormick and Ricky E. Sward, editors, *Proceedings of the 2004 Annual ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-Time & Distributed Systems using Ada and Related Technologies, November 14-14, Atlanta, GA, USA*, pages 1–8. ACM, 2004. (cited p. 7)

[SPHP02] Bernhard Schätz, Alexander Pretschner, Franz Huber, and Jan Philipps. Model-based development of embedded systems. In Jean-Michel Bruel and Zohra Bellahsene, editors, *Advances in Object-Oriented Information Systems, OOIS 2002 Workshops, Montpellier, France, September 2, 2002, Proceedings*, volume 2426 of *Lecture Notes in Computer Science*, pages 298–312. Springer, 2002. (cited p. 14)

[Van04] Bart Vanthournout. *Developing Transaction-level Models in* SystemC. CoWare Inc., 2004. (cited p. 20)

[Ver06] Thomas Vergnaud. *Modélisation des systèmes temps-réel répartis embarqués pour la génération automatique d'applications formellement vérifiées.* PhD thesis, December 01 2006. (cited p. 7 and 22)

[Ves00] Steve Vestal. MetaH. 2000. (cited p. 6)

[YHC+06] Guang Yang, Harry Hsieh, Xi Chen, Felice Balarin, and Alberto Sangiovanni-Vincentelli. Constraints assisted modeling and validation in metropolis framework. *Fortieth Asilomar Conference on Signals, Systems and Computers (ACSSC'06)*, pages 1469–1474, Oct.-Nov. 2006. (cited p. 18)

[YSWB04] Guang Yang, Alberto Sangiovanni-Vincentelli, Yosinori Watanabe, and Felice Balarin. Separation of concerns: overhead in modeling and efficient simulation techniques. In Giorgio C. Buttazzo, editor, *EMSOFT 2004, September 27-29, 2004, Pisa, Italy, Fourth ACM International Conference On Embedded Software, Proceedings*, pages 44–53. ACM, 2004. (cited p. 18)

[ZHHP07] Bechir Zalila, Irfan Hamid, Jérôme Hugues, and Laurent Pautet. Generating distributed high integrity applications from their architectural description. In Nabil Abdennadher and Fabrice Kordon, editors, *Ada-Europe*, volume 4498 of *Lecture Notes in Computer Science*, pages 155–167. Springer, 2007. (cited p. 23)

Appendix A

Trace produced with the virtual prototype in SystemC

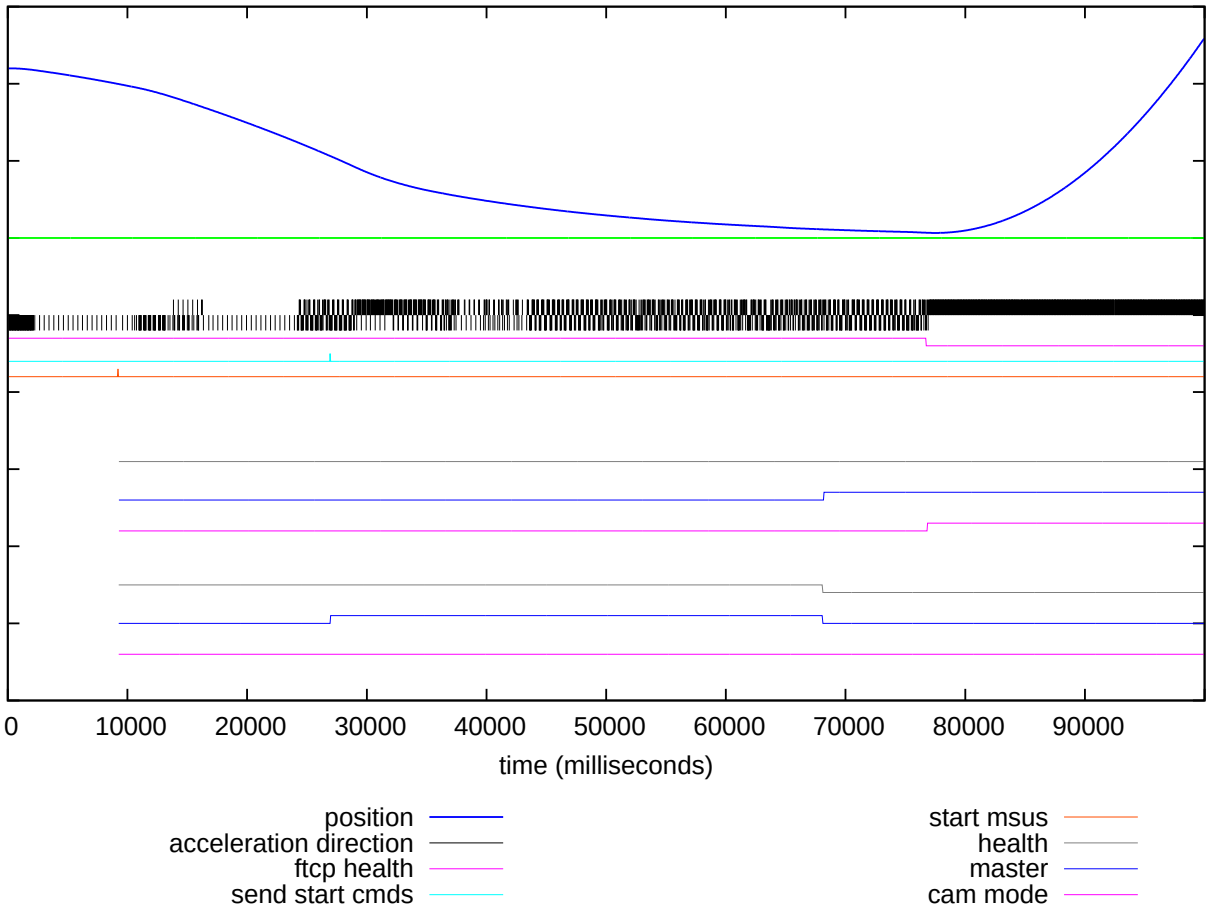

Figure A.1: A trace produced with the virtual prototype implemented in SystemC*. The "physical" goal and position of the system has been represented by the green and blue lines respectively. The bars around zero correspond to the acceleration commands sent to the thrusters. The other lines represent the boolean information. The* health*,* master *and* cam mode *lines are represented for each MSU. First, we observe the start sequence triggered by the FTCP: it initially starts the MSUs and send them dedicated commands through the system bus to name the initial mater. Next, an MSU failure triggers a role exchange, and an FTCP failure provokes a collision avoidance maneuver, consisting in taking the physical system away from the initial goal.*

Appendix B

The synchronous SystemC scheduler model

The listing B.1 on the next page presents the sorted list used by the scheduler model. The scheduler model, by its turn, is presented in the listing B.2 on page 49.

The Lustre node presented in the listing B.3 on page 50 has been used to produce the execution depicted figure B.1.

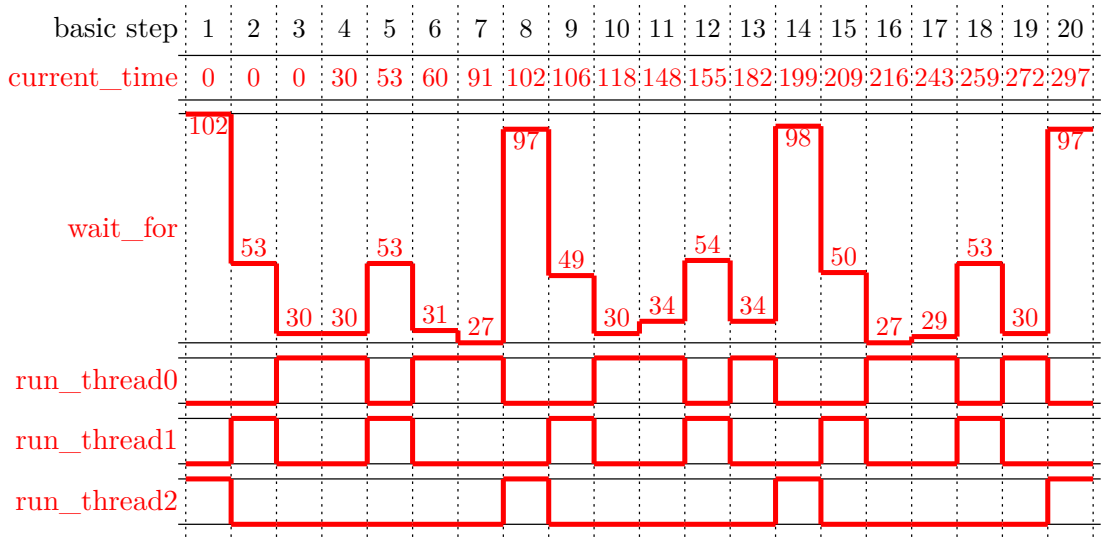

Figure B.1: A resulting scheduling of three concurrent threads.

```
-- We consider a thread to be a couple of a time and an identifier.
-- This node represents a list of n threads, ordered by
-- increasing time.
-- The initial condition is that, for each element of init_times,
-- init_times[i] >= init_times[i+1].
node SC_SCHEDULER_LIST
  (const n: int; const init_times, thread_ids: int^n;
   replace: bool; in_time, in_id: int)
returns
  (out_time, out_id: int);
var
  curval_time, pval_time, maxval_time, minval_time: int;
  curval_id, pval_id, maxval_id, minval_id: int;
let

  -- a facility to use the memorized values.
  (pval_time, pval_id) =
    ((init_times[n-1] -> pre curval_time),
     (thread_ids[n-1] -> pre curval_id));

  -- select the minimal and maximal times.
  (minval_time, minval_id, maxval_time, maxval_id) =
    if replace then
      if in_time < pval_time then
        (in_time, in_id, pval_time, pval_id)
      else
        (pval_time, pval_id, in_time, in_id)
    else
      (init_times[n-1] -> pre out_time,
       thread_ids[n-1] -> pre out_id, pval_time, pval_id);

  -- output the thread with a minimal time value.
  out_time = minval_time;
  out_id = minval_id;

  (curval_time, curval_id) =
    with n = 1 then
      -- The last element of the list keeps its current thread.
      (maxval_time, maxval_id)
    else
      -- insert the thread with the maximal value in the rest of
      -- the list if needed.
      SC_SCHEDULER_LIST(n-1,
        init_times[0 .. n-2], thread_ids[0 .. n-2],
        replace and in_time >= pval_time,
        maxval_time, maxval_id);

tel
```

Listing B.1: The SC_SCHEDULER_LIST node used by the scheduler model.

```
-- This node manages n threads, outputs a thread identifier
-- (an integer here) and the current simulated time.
node SC_SCHEDULER
  (const n: int; const init_times, thread_ids: int^n;
   time_to_wait_for_previous_id: int)
returns
  (elected_id, current_time: int);
var
  time_to_wait: int;
let

  assert time_to_wait_for_previous_id >= 0;
  assert current_time >= (0 -> pre current_time);

  -- Insert the thread which ran during the last basic step,
  -- and elect the thread with a minimal time to wait.
  -- The latter can be the former.
  (time_to_wait, elected_id) =
    with n = 1 then
      ((0 -> pre current_time) +
        (init_times[0] -> pre time_to_wait_for_previous_id),
       thread_ids[0] -> pre elected_id)
    else
      SC_SCHEDULER_LIST(n-1,
        init_times[1 .. n-1], thread_ids[1 .. n-1], true,
        (0 -> pre current_time) +
          (init_times[0] -> pre time_to_wait_for_previous_id),
        thread_ids[0] -> pre elected_id);

  -- The current simulated time is the time the elected thread
  -- had to wait.
  current_time = time_to_wait;

tel
```

Listing B.2: The simplified SystemC *scheduler model implemented in* Lustre.

```
-- This node uses the scheudler model to manage three periodic
-- threads.
-- It uses an oracle to shift the time.
node THREE_THREADS
  (oracle: int)
returns
  (current_time, wait_for: int;
   run_thread0, run_thread1, run_thread2: bool);
var
  id_to_run: int;
let

  -- The initial waiting time of the threads is null.
  (id_to_run, current_time) =
    SC_SCHEDULER(3, [0, 0, 0], [1111, 2222, 3333],
                 wait_for + oracle);

  -- a facility to represent the scheduling.
  run_thread0 = (id_to_run = 1111);
  run_thread1 = (id_to_run = 2222);
  run_thread2 = (id_to_run = 3333);

  -- This equation represents the computation of the time that the
  -- current thread has to wait before its next activation.
  wait_for = if run_thread0 then 30
        else if run_thread1 then 50
        else if run_thread2 then 100
        else 0;                    -- normally unused.

  -- Check an error case.
  assert wait_for <> 0;

  -- Allow a ``very loose-timing'' (with respect to the normal
  -- behavior of real physical clocks).
  assert oracle > -10 and oracle < 10;

tel
```

Listing B.3: The Lustre *node which served to produce the scheduling of the figure B.1 on page 47.*

Appendix C

Trace produced with the precise Lustre model

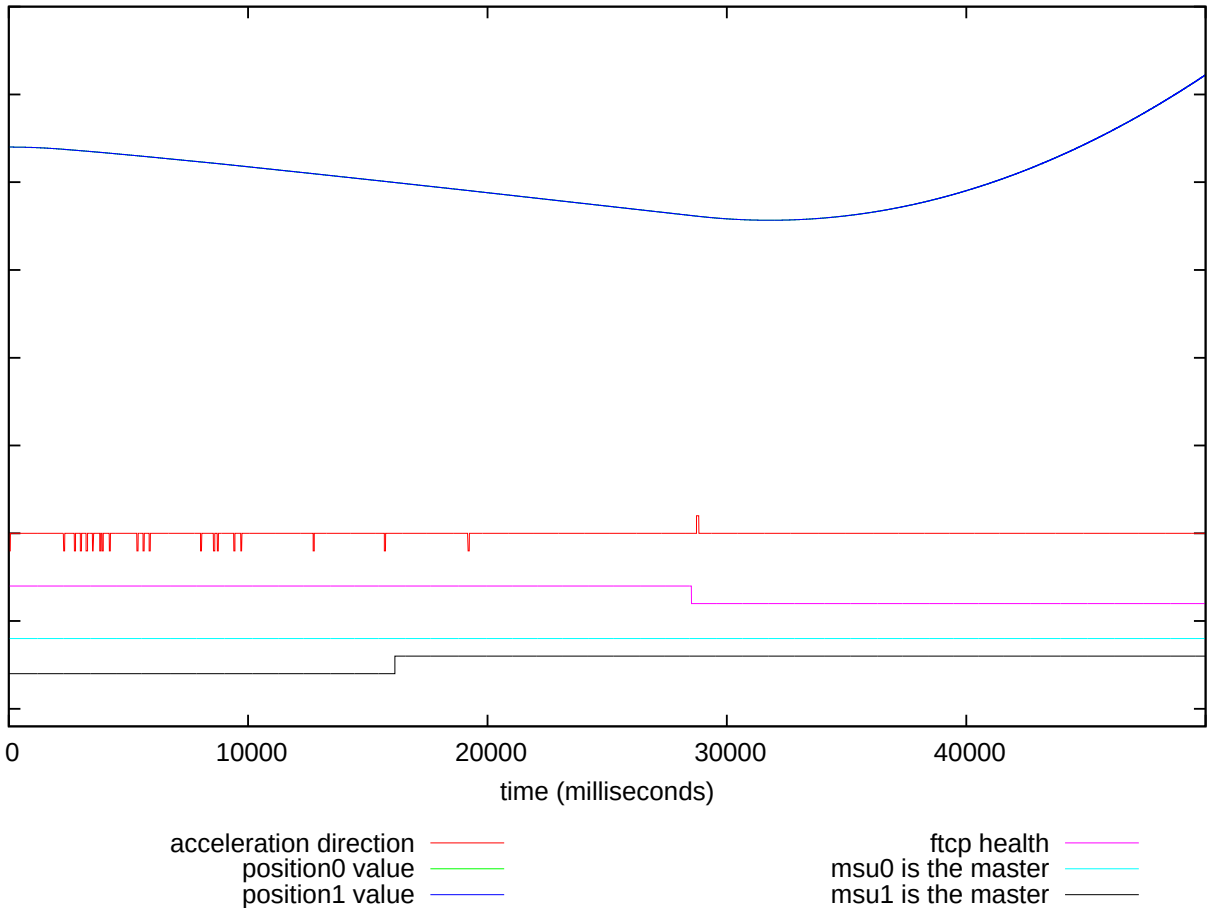

Figure C.1: A simplified trace produced with the second model implemented in Lustre. *The goal of the system remains the same as A on page 45, but the health of the MSUs has not been represented. We can observe an FTCP failure triggering a CAM after about 30 seconds.*