Linköping Electronic Articles in
Computer and Information Science
Vol. 4(1999): nr 20

SIGNAL-SIMULINK: Hybrid System Co-simulation

Stéphane Tudoret

Linköping University Electronic Press
Linköping, Sweden

http://www.ep.liu.se/ea/cis/1999/20/

Published on 21 February, 2000 by
Linköping University Electronic Press
581 83 Linköping, Sweden

Linköping Electronic Articles in
Computer and Information Science
ISSN 1401-9841
Series editor: Erik Sandewall

Typeset by the author using LaTeX
Formatted using étendu *style*

Recommended citation:
<Author>. <Title>. Linköping Electronic Articles in Computer and Information Science, Vol. 4(1999): nr 20. http://www.ep.liu.se/ea/cis/1999/20/. *21 February, 2000.*

This URL will also contain a link to the author's home page.

Abstract

This report presents an approach to simulating hybrid systems. We show how a discrete controller that controls a continuous environment can be co-simulated with the environment (plant) using C-code generated automatically from mathematical models. This approach uses SIGNAL with SIMULINK to model complex hybrid systems. The choices are motivated by the fact that SIGNAL is a powerful tool for modelling complex discrete behaviours and SIMULINK is well-suited to deal with continuous dynamics. We present various alternatives for implementing the communication between the plant and the controller, and how the MATLAB code generation mechanism in Real-time Workshop can be used for this purpose. Finally, we present interesting scenarios in the co-simulation of a discrete controller with its environment: a non-trivial siphon pump proposed by the Swedish engineer Christofer Polhem in 1697.

Acknowledgments

First of all I would like to thank my supervisor Simin Nadjm-Tehrani who made me very welcome in her team. She helped me when I needed and she introduced me to the good persons to help me to solve my problems. I am very glad to have been given the opportunity to work together with the persons in the Embedded Systems Laboratory (ESLab) at Linköping University and with the persons in the EP-ATR (Programming environment of real-time applications) project at Rennes University.

At Linköping, I would like to thank Peter Loborg (Real-Time Systems Laboratory (RTSLAB)) for the long and fruitful discussions about Simulink modeling, as well as Valure Einarsson (Division of Automatic Control) for his advice. I would like to thank Jan-Erik Strömberg who provided the main case study of this work and helped a lot. I would like also to thank all the members of the ESLab for their kindness.

At Rennes, I would like to thank Albert Benveniste for providing this subject and also for sending me several stimulating mails. I would like also to thank Sophie Pinchinat for introducing me to Simin allowing me to spent seven months in Sweden. I would like to thank Hervé Marchand for his help concerning Signal. He patiently read the report as well as Simin, and gave many useful comments which improved its content and appearance. Finally, I would like also to thank all the members of the project EP-ATR who help me somehow.

Contents

Introduction

The class of *reactive* systems have been proposed by David Harel and Amir Pnueli [HP85]. Reactive systems are characterized as computing systems which continuously interact with a given physical environment. These systems differ from *interactive* systems by the very fact that the physical environment is not synchronized logically with the reactive system, e.g., the environment can not wait for the system. Reactive systems can be such as real-time embedded system, control and communication systems, and interactive software or hardware. Generally, these systems share some important features[HCRP91]:

- Parallelism: the system and its environment must run in parallel.
- Time constraints: response times of the system should be in accordance with requirements induced by the environment.
- Dependability: Most of these systems are highly critical ones, e.g., nuclear plant control system or aircraft control system. Hence, this domain of application requires very careful design and verification methods such as formal methods.

The synchronous approach: In the middle of the eighties, the new class of *synchronous languages* dawned. This class constitute an answer for applying formal verification methods since these languages provide a precise semantic of time. The fundamental assumption characterizing this computational model is the *ideal synchronous hypothesis* [Ber89, Hal93]:

> "Ideal systems produce their outputs synchronously with their inputs."

Hence all computation and communication is assumed null. Of course, this assumption is just an approximation of the reality, however it is still valid so long as it is possible to assure that the program reacts faster than the environment. This assumption enables the temporal ordering of events to be determined more easily. Many synchronous languages have been efficiently implemented. The class of synchronous languages is usually split in two: on the one hand *imperative* and on the other hand *dataflow* synchronous languages.

- **Imperative synchronous languages:** These languages use a sequential approach and have classical control structure like iteration and condition. They include ESTEREL [BC85, BS91], STATECHARTS [Har87] and ARGOS [Mar90, MH96].

- **Synchronous dataflow languages:** Traditionally electrical and control engineers model their systems by means of networks of operators transforming data flows, and from a higher level, by means of block-diagram structures and by means of systems of dynamical equations which capture the behavior of the model. The dataflow languages are functional languages.Hence they are well adapted to formal verification. They have also parallelism, reusability and modularity properties.
  Synchronous dataflow languages are dataflow languages where synchronization constraints between the different events of the systems and of the environment are expressed. This class of languages include SIGNAL [GBBG85, AGMR95] and LUSTRE [HCRP91].

Hybrid systems: Real world engineering systems are characterized by having both continuous and discrete elements. Thus an extension in the direction of adding more realism to the formal modeling of real life reactive systems is the consideration of *hybrid* systems. The hybrid systems can be seen as a means to code the reactive systems. Synchronous languages and more generally transition systems provide a discrete approximation of models with discrete time, whereas hybrid systems provide model considering the time as continuous, which Synchronous languages can not do. Hybrid systems are systems consisting of a mixture of discrete components with continuous components which are traditionally described by continuous formalisms such as differential equations. The motivation for such an extension and some proposals for appropriately extended formalisms are discussed in [KP91, NSY91]. Complex hybrid dynamic systems are evident in myriad applications, such as manufacturing and production systems, intelligent robots, avionics, automotive control systems, railway systems, energy systems, and transportation networks.

Motivations and aims: Designing controllers for hybrid systems and verifying the resulting closed loop system is a difficult task. Moreover, the checking of each component of a hybrid system does not prove the good behavior of the whole hybrid system, this fact is discussed in [LGS94]. In Krister Edström [Eds99]'s opinion, the problem of formal analysis of hybrid systems is so difficult compared to, e.g., formal analysis of continuous systems, that the use of simulation is very important to get intuition for, and detect properties of, the systems. Of course, simulation results can not be taken as proof that a system works well in general but they can be taken as proof that it works in specific cases, or, more importantly, that it doesn't work in others.

The goal of this work is to develop a hybrid system simulator with the following architecture:

- a discrete controller specified with SIGNAL language interconnected with
- several continuous time dynamical systems, specified using SIMULINK.

As compared with SIMULINK/STATEFLOW, see chapter 4 of [Mat97b], the idea is to explore an alternative route, namely:

- SIMULINK/STATEFLOW views STATEFLOW as continuous time dynamical systems (with piecewise constant state trajectories), thus the hybrid system is lifted to continuous time, and delivered to the solver for its simulation.
- We view it differently. We want to keep the discrete controller DISCRETE during the simulation. Thus we will introduce "monitors" (A/D converters) linking the controller to the different continuous systems. We view the controller as SUPERVISING several Ordinary Differential Equation(ODE) solvers in a (quasi) distributed way.

Clearly, this is an alternative route, more suitable to the simulation of large systems composed of interconnected (smaller) continuous systems, and a discrete controller.

Organization of this report and contributions This report contains three parts with nine chapters. The first part is the background includes brief introductions to SIGNAL and to SIMULINK (chapter 1 and 2). The second part begins by a chapter showing generic models for hybrid systems and a brief state of the art of hybrid systems. The contribution of the author for this part, is in three following chapters (4,5 and 6). The chapter 4 compare two different approaches depend-ing on which is the master between SIGNAL and SIMULINK/MATLAB stand-alone program. The second approach, i.e., SIMULINK/MATLAB stand-alone program as master, is kept in the following chapters. The chapter 5 specifies the generic models for the co-simulation with SIGNAL and SIMULINK. Finally chapter 6 gives implementing methods which are based on the new generic model of chapter 5. The third and last part deals with a non-trivial case study. Chapter 7 presents the Polhem pump example. Then the contribution of the author for this part, is in the two last chapters. Chapter 8 deals with the modeling of the pump and chapter 9 shows how co-simulation points out some deficiencies of the controller.

Part I

Background

Chapter 1

Introduction to SIGNAL-V4

The aim of SIGNAL[AGMR95, BBM97, Hou98] is to support the design of safety critical applications, especially those involving signal processing and process control. The synchronous approach [Hal93] guarantees the determinism of the specified systems, and supports techniques for the detection of causality cycles and logical incoherences. The design environment of SIGNAL features a block-diagram graphical interface, a formal verification tool SIGALI, and a compiler that computes a hierarchy of inclusion of logical clocks (representing the temporal characteristics of discrete events), checks for the consistency of the inter-dependencies, and automatically generates optimized executable code ready to be embedded in environments for simulation, test, prototyping or the system itself[1].

1.1 The SIGNAL language

SIGNAL is a programming language to Specify, verify and implement real-time reactive systems. Just like LUSTRE [HCRP91], it belongs to the synchronous data flow languages. It manipulates signals, which are unbounded series of typed values (**logical**, **integer**...), with an associated clock denoting the set of instants when values are present. Signals of a special kind called **event** are characterized only by their clock i.e., their presence (when they occur, they give the boolean value **true**). Given a signal X, its clock is obtained by the language expression *event* X, resulting in the event that is present simultaneously with X. To constrain signals X and Y to be synchronous SIGNAL language provide the operation: $synchro X, Y$. The absence of a signal is noted $\perp$.

The compiler performs the analysis of the consistency of the system of equations, and determines whether the synchronization constraints among the signals are verified or not. If the program is constrained so as to compute a deterministic solution, then executable code can be produced automatically in C. This code basically con-

[1] The SIGNAL language is developed at IRISA in France, and it is sold by TNI (Brest in France) within the Sildex environment.

sists of a cyclic call to a transfer function, which computes an output, and updates state variables, according to the presence and value of input signals. The stand-alone code generation is summarized in 1.3.

1.1.1 The kernel of SIGNAL

SIGNAL is built around a small kernel comprising five basic operators (functions, delay, selection, deterministic merge, and parallel composition). These operators allow to specify in an equational style the relations between signals, i.e., between their values and between their clocks. Each equation from SIGNAL is like an elementary process.

Functions (e.g., addition, multiplication, conjunction, ...) are defined on the type of the language. For example, the boolean negation of a signal E is *not* E.

$$X := f(X1, X2, \cdots, Xn)$$

The signals X,$X1$,$X2$,$\cdots$,Xn must all be present at the same time, so they are constrained to have the same clock.

Delay gives the previous value ZX of a signal X:

$$ZX := X\ \$1$$

with initialization at the declaration of ZX: $ZX\ init\ V0$. Signals X and ZX have the same clock. The array below shows a trace of use of delay.

$$\begin{array}{rcllll} X & : & 5 & 2 & 3 & 20 \\ ZX := X\ \$1 & : & V0 & 5 & 2 & 3 \end{array}$$

Selection of a signal Y is possible according to a boolean condition C:

$$X := Y\ when\ C$$

The clock of signal X is the intersection of the clock of Y and the clock of occurrences of C at the value *true*. When X is present, its value is that of Y.

$$\begin{array}{rcccccccc} Y & : & \perp & 1 & 2 & 3 & 4 & \perp & 5 \\ C & : & \mathrm{t} & \perp & \mathrm{t} & \mathrm{f} & \perp & \mathrm{t} & \mathrm{t} \\ X := Y\ when\ C & : & \perp & \perp & 2 & \perp & \perp & \perp & \mathrm{t} \end{array}$$

Deterministic merge defines the union of two signals of the same type, with a priority on the first one if both are present simultaneously:

$$X := Y\ default\ Z$$

The clock of signal X is the union of that of Y and of that Z. The value of X is the value of Y when Y is present, or else the value of Z if Z is present and Y is not.

$$\begin{array}{rccccccccc} Y & : & 1 & \perp & 2 & 3 & \perp & 4 & 5 \\ Z & : & \perp & 10 & 20 & \perp & 30 & \perp & 50 \\ X := Y\ default\ Z & : & 1 & 10 & 2 & 3 & 30 & 4 & 5 \end{array}$$

Parallel composition of processes is made by the associative and commutative operator "|", denoting the union of the equation systems. In SIGNAL, the parallel composition of $P1$ and $P2$ is writing:

$$(|\ P1 \mid P2\ |)$$

Extensions: The rest of the language is build upon the kernel. Processes have been defined from the primitive operators, providing programming comfort.

Repetition upon another clock: To memorize Y values and to output them also when C is *true*, the additional operator *cell* has been introduced in SIGNAL:

$$X := Y\ cell\ C$$

C is a boolean signal. The X value is the Y value when Y is present, or the last Y value when Y is absent and C is *true*, or an initialization value of X when C is *true* before the first value of Y. The clock of signal X is the union of that of Y and of instant where C is *true*.

$$\begin{array}{rcccccccccc} Y & : & \perp & 1 & \perp & \perp & 2 & \perp & 3 & \perp & \perp \\ C & : & \mathrm{t} & \perp & \mathrm{t} & \mathrm{f} & \perp & \mathrm{t} & \mathrm{f} & \mathrm{f} & \mathrm{t} \\ X := Y\ cell\ C\ (init0) & : & 0 & 1 & 1 & \perp & 2 & 2 & 3 & \perp & 3 \end{array}$$

The equivalent of $X := Y\ cell\ C$ is :

```
(| synchro { X, event Y default when C }
 | X:=Y default X $1
 |)
```

Finally, note that the value to memorize can be a parameter given once time at beginning of calculation.

1.1.2 Introductory Example

The following example is a process which mixes two input signals $X1$ and $X2$ in one output signal Y equal to $X1$ or $X2$, and delay $X2$ when $X1$ and $X2$ are simultaneous, so that, no input is lost.

```
process MIXER =
  { ? integer X1, X2
```

```
    ! integer Y % Y= X1 or X2, split X1 and X2 %
                % if they are simultaneous %
  }
  (| B := not (event X1) when event X2
          default event X1
          default event X2
          default true
     % if X1 and X2 are simultaneous %
     %    then B = false, else B = true %
   | ZB := B $1
   | synchro {X1 default X2, when ZB }
   | Y := X1 default (X2 cell (not ZB))
   |)
  where
    logical B, ZB init true
  end
end
```

In this example, we can obtain a clock of output signal (Y) upper[2] than the union of inputs signals ($X1$ and $X2$) like in the trace bellow. This oversampling results from the synchronization of $X1$ $default$ $X2$ and of $when$ ZB. This synchronization leads to: when the value of ZB is false then we have a new tick of clock which does neither coincide with the clock of signal $X1$ nor the clock of signal $X2$.

$X1$	:	1	$\perp$	3	$\perp$	$\perp$	5	7	$\perp$	9
$X2$	:	$\perp$	2	4	$\perp$	6	$\perp$	8	$\perp$	$\perp$
B	:	t	t	f	t	t	t	f	t	t
ZB	:	t	t	t	f	t	t	t	f	t
not ZB	:	f	f	f	t	f	f	f	t	f
Y	:	1	2	3	4	6	5	7	8	9

1.2 Tools

The different tools which make up the SIGNAL environment use all only one tree-like representation of programs, thus we can go from one tool to another without use an intermediate data structure. The principal tools are the compiler which allows to translate SIGNAL programs into C, the graphical interface and, for the classic temporal logics specifications, the tool SIGALI.

1.2.1 The SIGNAL-V4 graphical interface

A SIGNAL program lends itself well to a graphical block-diagram oriented user interface. It is the why, in order to ease programming,

[2]There is an ordering relation between clocks: a clock C_1 is upper, i.e, higher up in the order relation, than a clock C_2 if all C_2 instants are also C_1 instants and C_1 is more frequent than C_2.

an editor [BG97] is available in the SIGNAL environment. This one allows to build SIGNAL programs with mixing texts and graphics. The editor also allows a hierarchical conception (with a block within another block) and modular implementation of algorithms (we can build a model, save it and use it in several programs). A SIGNAL program is built in two ways, either top down: we draw the higher level blocks and then we define them, or bottom up: we define the basic blocks and then we group them.

We can emphasize the supervisor of the graphical interface has been written in SIGNAL; this supervisor manages the interaction between the user and the window manager (X-Window), in the same way it manages the chain of user actions.

1.2.2 SIGALI: a formal calculus software

The SIGNAL environment contains a verification and controller synthesis tool-box, named SIGALI[MBLL98, LMRS96, BBM97]. This tool allows us to prove the correctness of the dynamical behavior of the system. The equational nature of the SIGNAL language leads to the use of polynomial dynamical equation systems (PDS) over $\mathbb{Z}/_{3\mathbb{Z}}$ as a formal model of program behavior. Polynomial functions over $\mathbb{Z}/_{3\mathbb{Z}}$ provides us with efficient algorithms to represent these functions and polynomial equations. Hence, instead of enumerating the elements of sets and manipulating them explicitly, this approach manipulates the polynomial functions characterizing their set. This way, various properties can be efficiently proved on polynomial dynamical systems. The same formalism can also be efficiently used for solving the supervisory control problem[ML98].

1.3 Stand-alone code generation

To generate a stand-alone program with SIGNAL is a four-step action: specify a model, generate C code, generate makefile and generate stand-alone program. The Figure 1.1 summarizes the architecture of the stand-alone code generation with SIGNAL.

1.3.1 Model specification

With SIGNAL, it is possible to specify a model either by writing a SIGNAL program (in `<Model>.SIG` file form) which can be directly compiled or by using the SIGNAL Graphical User Interface (GUI) (see [BG97] for more details). In the last case it is necessary to convert the graphical file of the model (`<Model>.sig`) to an ascii SIGNAL program file (`<Model>.SIG`).

1.3.2 C code generation

Once the SIGNAL model is specified, C code programs can be obtained by compiling from `<Model>.SIG`. By default, the SIGNAL compilater

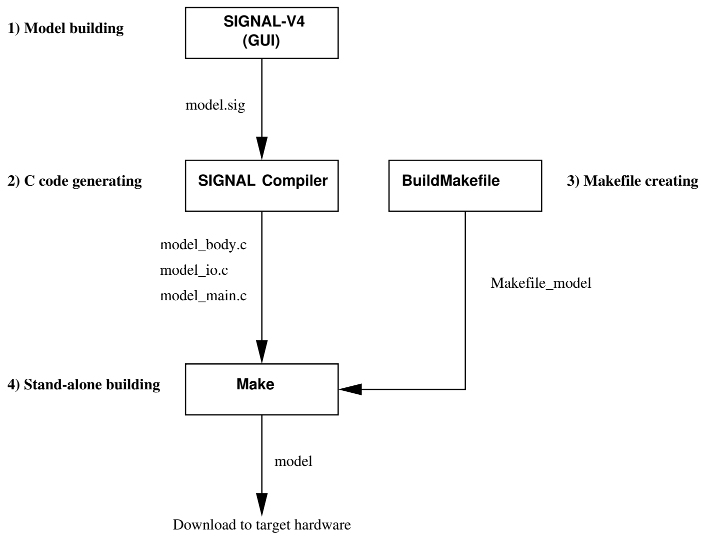


Figure 1.1: The stand-alone code generation with SIGNAL

gives mainly three C files : `<Model>_body.c`, `<Model>_io.c` and `<Model>_main.c`.

- `<Model>_body.c` contains three functions:
  - `logical <Model>_initialize(void)` to initialize the SIGNAL model.
  - `logical <Model>_iterate(void)` to update the SIGNAL model, this function gets (resp. sets) those input (resp. output) signals in files through the `<Model>_io.c` functions.
  - `logical <Model>_iterate_Black_Box(inputs,outputs)` to update the SIGNAL model, this function gets (resp. sets) those input (resp. output) signals by means of parameters.
- `<Model>_io.c` consists of input/output functions of the model. Those input/output functions are made by reading and writing in files.
- `<Model>_main.c` consists of the C main function of the stand-alone program. The main function calls the initialization function and loops calling the iterate function (the one that deals with input/output files) at each logical step.

1.3.3 Makefile creation

In order to build an executable file from the generated code from SIGNAL the make command is used. The makefile shows the compiler how to link generated files and how to compile them.

With SIGNAL, this makefile is automatically made by the command:

```
BuildMakefile <Model>
```

that provides the following make file `Makefile_<Model>`. If needed (e.g. in order to change `<Model>_main.c` to another main file), the make file can be improved by adding C user's source includes.

1.3.4 Stand-alone code compilation

In order to build the stand-alone program, the last thing to do, is to run the make command with the makefile:

```
make -f <MakefileName>
```

After that one can start the stand-alone program as any stand-alone program.

Chapter 2

Introduction to SIMULINK

This is a short introduction beginning by a quotation taken from [Mat97d] where a complete description of SIMULINK, can be found. The latter is too large to be completely included here.

> "SIMULINK is a software package for modeling, simulating, and analyzing dynamical systems. It supports linear and nonlinear systems, modeled in continuous time, sampled time, or a hybrid of the two. Systems can be also multi-rate, i.e, have different parts that are sampled or update at different rates."

Since SIMULINK is a MATLAB [Mat98] component, all of the tools in MATLAB are usable with it. So it is possible to take the SIMULINK simulation results and analyze and visualize them with MATLAB tools.

2.1 SIMULINK libraries

SIMULINK use dataflow oriented block diagrams and provides a library of six blocks by default:

- Sources are blocks with vector outputs and no vector inputs, e.g, **Constant**, **Signal generator**, and so on.
- Sinks are blocks with vector inputs and no vector outputs, e.g, **Scope**, **Display**, **Stop**, and so on.
- Discrete are blocks with discrete vector outputs and discrete or continuous vector inputs.
- Linear components are blocks with both vector outputs and inputs, where outputs are a linear function of inputs. Examples are **Sum**, **Integrator**, **Transfer Function** , and so on.
- Nonlinear components are blocks with both vector outputs and inputs, where outputs are a nonlinear function of inputs. Examples of these are **Logical Operator**, **Product**, **Abs**, and so on.

- Connections are some blocks which do not belong to the above library, e.g, **Mux**, **Demux**, **Enable**, and so on.

As Signal Simulink allows stand-alone generation, in four steps, i.e, specify a model, generate C code, generate makefile and generate stand-alone program. The Figure 2.1 summarizes the architecture of the stand-alone code generation with Simulink.

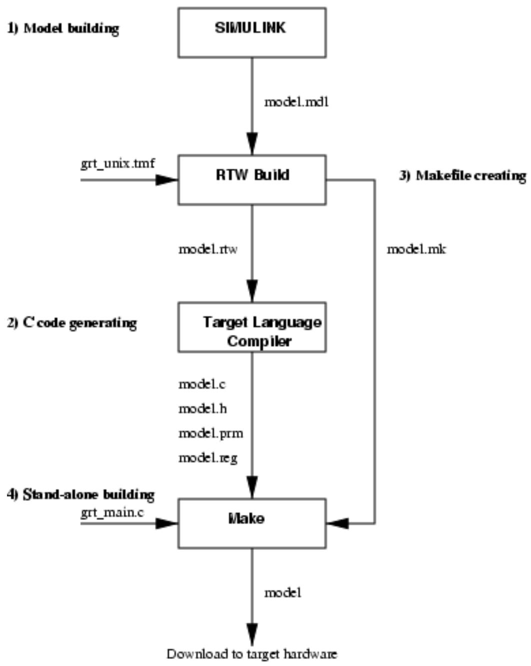


Figure 2.1: The Real-Time Workshop's architecture

2.2 Using Simulink

2.2.1 Building a model

It is possible to customize and create new blocks, and to collect together several blocks in order to build a new block, i.e, a **Subsystem**. To create a model one simply performs the following steps:

1. Chose blocks in the suitable library.
2. Click and drag these blocks in the current Simulink model window.
3. Link blocks together.
4. Save the model in a file `<Model>.mdl`.

The Figure 2.2 shows a simple model with four blocks. This model is more exactly a submodel since it posseses an input block

(**In1**) and an outputs block (**Out1**) and thus it can be represented as a SIMULINK **Subsystem** block itself.

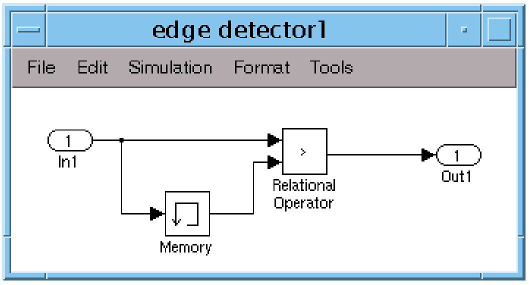


Figure 2.2: Block diagram of edge detector

This submodel is an edge detector and it will be used further in the report. It works as follows: A binary piecewise constant signal is in input, and at each tick of the clock this signal value is kept in the SIMULINK **Memory** block and compared with the preceding signal value thanks to the **Relational Operator** block. Thus, if the new value is higher than the old value then the output signal is 1, otherwise it is 0.

Before starting the simulation, it is necessary to take care to fix correctly some parameters, especially for the solver options. SIMULINK provides several solvers for the simulation of numeric integration of sets of Ordinary Differential Equations(ODEs). Because of the diversity of dynamic system behaviors, some solver may be more efficient than others at solving a particular problem. It is also possible to choose between two kind of solvers, the variable-step and the fixed-step solvers. Since, at the moment, it is not possible to build stand-alone program with variable-step solvers, only fixed-step solvers are used in this work. Hence, the step size needs to be set accurately.

The Figure 2.3 shows a complete simple model which test the above edge detector block. The **Start** command of the **Simulation** menu starts the simulation and then a click on the **Scope** block displays signals generated during the simulation in a new window, see Figure 2.3.

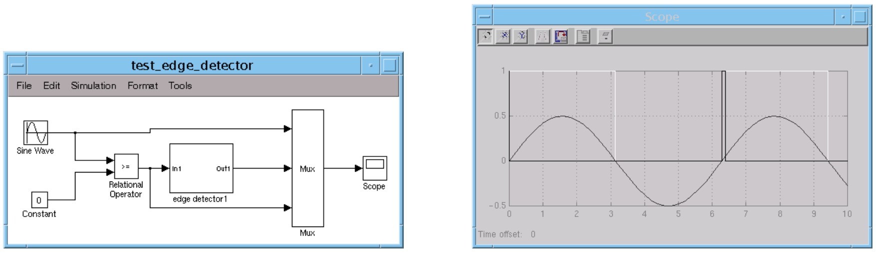


Figure 2.3: The test environment of the edge detector with its traces

This simulation is not a stand alone-simulation since it runs on top of SIMULINK. Since we need C code to link SIMULINK with SIGNAL and later stand-alone simulation, these points are dealt with in the next sections.

2.2.2 C code generation

It is the SIMULINK Real-Time Workshop (RTW) [Mat97a] which is used to automatically generate C code from SIMULINK block diagrams.
By default[1], the RTW gives mainly four C files :
<Model>.c, <Model>.h,
<Model>.prm and <Model>.reg.
Those files are briefly described below and described in detail in chapter 5 of the Real-Time Workshop User's Guide[Mat97a].

- <Model>.c contains the procedures that implement the algorithm defined by the SIMULINK block diagram:
  - void MdlStart(int_T tid) to initialize the model,
  - void MdlOutput(int_T tid) to update the output of blocks at appropriate times,
  - void MdlUpdate(int_T tid) to update the discrete states,
  - void MdlDerivatives(void) contains derivatives provided by each block that has continuous states. The derivatives are used by the solver to integrate the continuous state to produce the next value.
  - void MdlTerminate(void)
- <Model>.h contains the structure definitions of the block diagrams (i.e, the model),
- <Model>.p contains the structure declarations of the block diagrams,
- <Model>.reg contains the model registration function,
  - void MdlInitialiseSizes(void) to set size of various data structures in a SIMULINK structure named *SimStruct*,
  - void MdlInitialiseSampleTimes(void) to set sample times for the model in the SimStruct,
  - void <Model>(void) to set up working areas in the hboxSimStruct

[1] It is possible to customize the C code generated from any SIMULINK model with the Target Language Compiler (TLC) [Mat97c]. The TLC is a tool that is included in RTW.

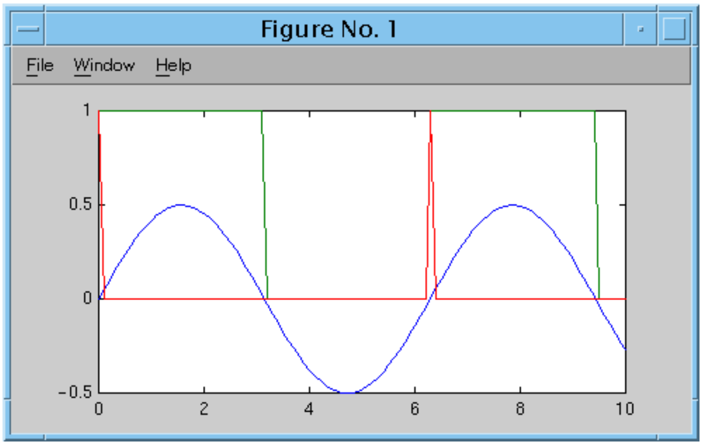


Figure 2.4: Matlab traces of the edge detector test

By default, the main C file is generic (cf `grt_main.c`) and does not depend on the block diagrams defined in the first step. The main C file contains the main function that performs the initialization, the model execution and the program termination.

2.2.3 Makefile creation

With SIMULINK, the makefile is automatically made from a template makefile (for example `grt_unix.tmf` is the generic real-time template makefile for UNIX), see chapter 3 of the *Real-Time Workshop User's Guide*[Mat97a].

2.2.4 Stand-alone code compilation

Just like in section 1.3.4, the last thing to do is to run the make command with the makefile:

```
make -f <MakefileName>
```

in order to build the stand-alone program. After that you can start the stand-alone program as any stand-alone program.

2.3 Using the Stand-alone program

By default, the run of stand-alone program provides a MATLAB data file (`<Model>.mat`). Before the building of the stand-alone program, it is possible to select which data we want in the MATLAB file, see chapter 2 of the *Real-Time Workshop User's Guide*[Mat97a]. Then, you can use MATLAB to plot the result. The Figure 2.4 shows the traces of the edge detector test obtained by MATLAB from the `test_edge_detector.mat` file.

Part II

Modeling and co-simulation

Chapter 3

Generic model for hybrid systems

3.1 A mathematical representation of hybrid systems

Hybrid systems can be mathematically represented as follows:

$$\begin{aligned}
\dot{x}_i &= f_i(q, x_i, u_i, d_i)\ ,\ x_i \in \mathbb{R}^{n_i}\ ,\ q \in Q && (3.1)\\
y_i &= h_i(q, x_i, u_i) && (3.2)\\
e_i &= s_i(q, x_i, u_i, y_i) && (3.3)\\
\tau_i &= e.\mathbf{1}_{\{e_i \neq e_{i_}\}} && (3.4)\\
q' &= T(q, \tau)\ ,\ \tau = (\tau_i, i = 1, \ldots, I) && (3.5)
\end{aligned}$$

Where:

(3.1): $i = 1, \ldots, I$ indexes a collection of continuous time sub-systems,
$q \in Q$ is the discrete state, where Q is a finite alphabet,
$x_i \in R^{n_i}$ is the vector continuous state of the ith continuous time sub-systems,
$u_i \in R^{m_i}$ is the vector continuous control of the ith continuous time sub-systems,
$d_i \in R^{o_i}$ is the vector continuous of disturbance of the ith continuous time sub-systems.

(3.2): $y_i \in R^{p_i}$ is the vector continuous output of the ith continuous time sub-systems,

(3.3): $e_i \in B^{r_i}$ where B is the boolean domain. Thus at each instant an r-tuple of predicates depending on the current values of (q, x_i, u_i, y_i) is evaluated.
Examples are $x_i^k > 0$ where superscript k refers to the kth component of x_i, if $x_i = (x^{1_i}, \ldots, x^{n_i})$, or $g(q, x_i, u_i, y_i) > 0$ for $g(q, ., ., .) : R^{n_i+m_i+p_i} \longmapsto R$, and so on.

(3.4): $e_{i\_}(t)$ denotes the left limit of e_i at t, i.e., the limit of $e_i(s)$ for $s < t, s \nearrow t$. Assume that $e^k_{i\_}(t) \neq e^k_i(t)$ means that the kth predicate changes its status at instant t; this generates an event τ^k_i. Those marked events τ^k_i are collected into a vector event τ_i (and those latter are collected into vector event τ). Thus trajectories e_i are piecewise constant.

(3.5): q, q' are the current and next discrete automaton state.

The thermostat example The thermostat example, which is taken from [NSY91, ACH+95, BS97], is simple but it will used all along this report to illustrate several different formalisms. The temperature is governed by differential equations. The thermostat is provided with two parameters: m and M which determine the interval in which the temperature of the room is supposed to remain. In order to satisfy this requirement, the thermostat switches a heater OFF and ON. Thus the system has two locations:

- At the location OFF, the temperature decreases according to the exponential function $x(t) = \theta e^{Kt}$, where t is the time, θ is the initial temperature, and K is a constant determined by the room.

- At the location ON, the temperature increases according to the function
$x(t) = \theta e^{Kt} + h(1 - e^{Kt})$, where h is a constant that depends on the power of the heater.

So, the thermostat can be mathematically represented with the following equations.

$$Q = \{ON, OFF\} \tag{3.6}$$

For the continuous subsystem ON:

$$\begin{aligned}
\dot{x}_{ON} &= f_{ON}(q, x_{ON})\ ,\ x_{ON} \in \mathbb{R}\ ,\ q \in Q && (3.7)\\
f_{ON}(ON, x_{ON}) &= K(h - x_{ON}) && (3.8)\\
f_{ON}(OFF, x_{ON}) &= 0 && (3.9)\\
y_{ON} &= x_{ON} && (3.10)\\
e_{ON} &= s_{ON}(q, x_{ON}) && (3.11)\\
s_{ON}(ON, x_{ON}) &= x_{ON} \geq M && (3.12)\\
s_{ON}(OFF, x_{ON}) &= 0 && (3.13)\\
\tau_{ON} &= e.\mathbf{1}_{\{e_{ON} \neq e_{ON_}\}} && (3.14)
\end{aligned}$$

For the continuous subsystem OFF:

$$\begin{aligned}
\dot{x}_{OFF} &= f_{OFF}(q, x_{OFF}) \text{ , } x_{OFF} \in \mathbb{R} && (3.15)\\
f_{OFF}(ON, x_{OFF}) &= 0 && (3.16)\\
f_{OFF}(OFF, x_{OFF}) &= -Kx_{OFF} && (3.17)\\
y_{OFF} &= x_{OFF} && (3.18)\\
e_{OFF} &= s_{OFF}(q, x_{OFF}) && (3.19)\\
s_{OFF}(ON, x_{OFF}) &= 0 && (3.20)\\
s_{OFF}(OFF, x_{OFF}) &= x_{OFF} \geq M && (3.21)\\
\tau_{OFF} &= e.\mathbf{1}_{\{e_{OFF} \neq e_{OFF_}\}} && (3.22)
\end{aligned}$$

For the discrete automaton:

$$\begin{aligned}
q' = OFF &\Leftrightarrow q = ON \wedge \tau_{ON} = 1 && (3.23)\\
q' = ON &\Leftrightarrow q = OFF \wedge \tau_{OFF} = 1 && (3.24)
\end{aligned}$$

Note the continuous temperature x can be obtained by adding x_ON and x_OFF.

3.2 A hybrid system architecture

Simin Nadjm-Tehrani et.al. proposed to organized a hybrid system as in Figure 3.1 and as described below[SNT94]:

- The *Plant (P)* is the physical environment under control. The inputs u, the outputs y and the disturbances d all have continuous domains.

- The *Characterizer (C)* is the interface between the continuous plant and the discrete selector in terms of, e.g., discrete sensors.

- The *Selector (S)* is the purely discrete part of the controller. The inputs as the outputs have discrete domains.

- The *Effector (E)* is the interface between the discrete selector commands and the continuous physical variables in terms of, e.g., continuous controllers.

The *Controller (K)* contains the characterizer, the selector and the effector. The composition of the controller and the plant forms the *closed loop system.*

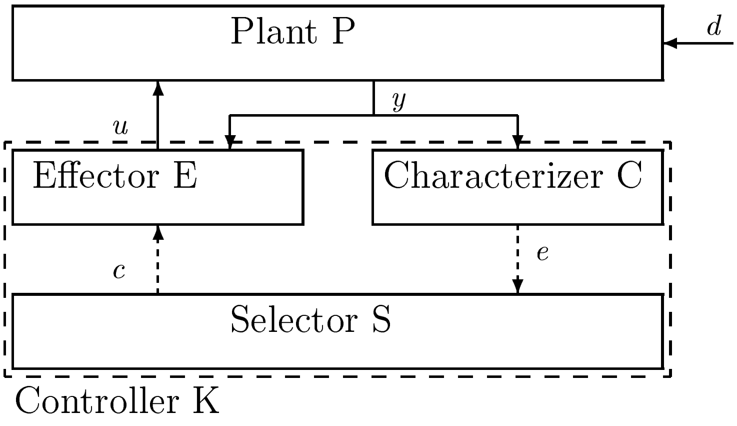


Figure 3.1: General hybrid system architecture. Solid arrows represent continuous and dashed discrete variable

The hybrid system architecture presented above, is a good starting point for hybrid system modeling. It remains just to answer to the following questions:

- How the mathematical representation of section 3.1 can be mapped on the architecture of the section 3.2?
- Which parts should be done in SIGNAL and which ones in SIMULINK?
- How the SIGNAL part should be activated?

The next section presents a brief collection of hybrid system works. Then, the chapter 4 shows two different approach for modeling hybrid systems, and finally, answers the above questions using the best approach.

3.3 Related works

Hybrid system can be seen as dynamical systems consisting of interacting and continuous components. They are traditionally used to specify the combined behavior of several embedded real-time systems as well as their physical environment.

The first extensions of the methodology in the formal specification and analysis of reactive systems to deal with real time and continuously varying elements date from the beginning of nineties, *Automata* (resp. STATECHARTS) have been extended to *timed automata* (resp. *timed* STATECHARTS) and *hybrid automata* (resp. *hybrid* STATECHARTS). *Bond graph* gave *switched bond graph* in order to take account of abrupt changes in the dynamic behavior of physical systems which are ubiquitous in engineering applications.
There are also some new languages which do not extend but use older concept, e.g, SHIFT is a high level language which models networks of hybrid automata.

3.3.1 Hybrid automata

Hybrid automata have been introduced by Xavier Nicollin, Joseph Sifakis and Sergio Yovine in [NSY91] and an analysis of linear hybrid automata are described in [ACH$^+$95]. They modeled a hybrid system as a finite automaton that is expended with a set of real valued variables. These variables can be tested and modified at transitions. At a automaton location Q the values of the variables change continuously with time according to evolution laws which are associated with Q. The transition relations are specified by guarded commands; the activities, by differential equations; and the location invariants, by logical formulas.

Figure 3.2 illustrates a hybrid automaton of the thermostat presented in section 3.1.

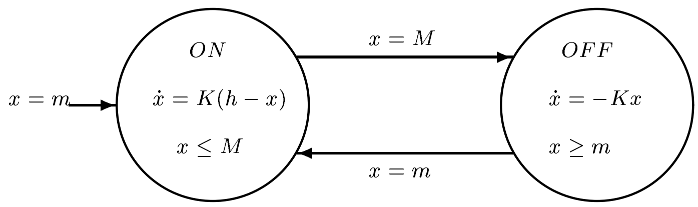


Figure 3.2: Hybrid automaton of the thermostat

Initially the heater is ON and the temperature of the room is m.

3.3.2 Hybrid STATECHARTS

If we consider particular STATECHARTS semantics like that implemented by STATEMATE [HN96], STATECHARTS belong to the class of the imperative synchronous languages. STATECHARTS have been used for the specification of complex reactive systems. The visual formalism of STATECHARTS has been proposed by David Harel in [Har87].

> "Our diagrams, which we call *statecharts* extend conventional state-transition diagrams with essentially three elements, dealing, respectively, with the notions of hierarchy, concurrency and communication."

Kesten and Pnueli have extended STATECHARTS, first adding the element of metric time (Timed STATECHARTS) and secondly adding a notion that allows to annotate a basic state by a differential equation (Hybrid STATECHARTS)[KP91].
Thus, the activity associated with the differential equation is active precisely when the state it labels is active.

3.3.3 Switched bond graphs

A methodology for physical modeling of systems in three steps, i.e, reticulation, equations generation and composition has been proposed by Lennart Ljung and Torkel Glad in [Pay61, LG94].

- Reticulation phase consists in repeatedly dividing the system into sub-systems, until sub-systems can be described by equations without too much effort.
- Equations generation consists in finding the equations that describe the sub-systems and that describe the sub-systems connection.
- Composition phase consists in sorting the equations in the right order and combining them.

L. Ljung shows also that the composition in *bond graphs* can be automatized.

Bond graphs

Bond graphs were created in 1959 by Henry M. Paynter [Pay61] to support physical modeling of systems. A bond graph is a graphical model of the system. It is based on energy conservation and it describes the energy flow through the modeled system. The energy flow is described by two quantities, i.e., effort e and flow f.

One of the main aspects is that different domains, like mechanical, hydraulic and electrical ones, are modeled in the same way. Because energy behaves in the same way regardless of the domain. Actually, energy can be fed into, stored or transformed in, or dissipated from the system. Thus all models are built by a small number of ideal elements, each describing a certain aspect of the behavior of energy.

Other advantages are:

- bond graphs allow some analysis of the models, e.g., causality analysis, mathematical model generation,
- bond graphs can be the user's interface in a modeling and simulation environment,
- equations associated with bond graph elements can be automatically converted into simulation code.

Bond graph switch element

The bond graph language is originally designed to support the derivation of continuous models. However, many practical engineering systems incorporate fast switching devices such as relays, thyristors, mechanical clutches, hydraulic valves and so on. These systems are referred to as "*mode switching systems*" [Str94]. One extension, among several bond graph extensions proposed to allow modeling of discrete phenomena, is to add to bond graph a new element called *ideal switch* or by "misnomer" *switch*. The switch has two states, zero effort and zero flow. In the zero effort state the switch behaves like an effort source and in the zero flow state the switch behaves like an flow source. Thus switched bond graphs capture these hybrid models, where the discrete part is simple and the continuous part is arbitrary complex.

Strömberg and Edström extended the analysis of bond graphs to switched bond graphs [Eds99, Str94]. Causality analysis now depends on the state of the switch. The algorithm for extracting mathematical representation of the model has also been modified.
The specific use of switched bond graphs in formal verification of embedded systems is dealt with in [SNTT96]. The modeling, the analysis and the design of hybrid systems in automatic control with bond graphs are also resumed in [LGG+96].

3.3.4 SHIFT

A. Deshpande, A. Gôllû and L. Semenzato present SHIFT in [DGS98] as following:

> " SHIFT is a programming language for describing and simulating dynamic networks of hybrid automata. Such systems consist of components which can be created, interconnected and destroyed as the system evolves. Components exhibit hybrid behavior, consisting of continuous-time phases separated by discrete-event transitions. Components may evolve independently, or they may interact through selected state variables and events. The interaction network itself may evolve."

The SHIFT language is used as a system description, integration and simulation environment in the Automated Highway Systems project of the National AHS Consortium.
A compiler to translate SHIFT programs into C and a run-time system for SHIFT have been implemented by the PATH project (Partners for Advanced Transit and Highways) at Berkeley in California.
SHIFT is resolutely simple and small. It has only one number type, no functions (however it can use external C functions) and it has a garbage collector in its implementation. SHIFT has high-level system abstractions which include differential equations, state transitions and synchronous compositions. Thus it can be used to describe models with differential equations which can be stopped or replaced with other differential equations.

General principle

In the SHIFT model, the *world* W consists of a set of hybrid *components*:
$W = \{h_1, \dots, h_w\}$. Each components h is in a *configuration* C_h (discrete state). The world evolves in a sequence of phases. During each phase, time flows continuously while the configuration of the world $C_W = \{C_{h_1}, \dots, C_{h_w}\}$ remains fixed. In the transition between phases, time stops and the set of components in the world and their configurations are allowed to change. Thus, components obey continuous-time dynamics within each phase and discrete-event dynamics in phase transitions.

Example of SHIFT program

A component prototype is defined by the SHIFT **type** declaration. The structure of a component is partly given by its **input**s, **output**s, and **state**s. State variables are not visible outside the component. The value of an input variable of a component may only be specified by another component. The **discrete** clause defines the possible values for the type's discrete state variable and associates a set of differential equations and algebraic definitions to each discrete state. Groups of common flow equations are given a name through the **flow** clause. Transitions between states are defined in the **transition** clause. Transitions are labeled by a (possibly empty) set of event labels. These labels allow transitions to synchronize with each other.
SHIFT contains other facilities, however the above sub-set of SHIFT is enough to specify and understand the following program of the thermostat example of the section 3.1.

Program 1 The SHIFT specification of the thermostat

```
type Thermostat {
  output continuous number x;
  discrete
    ON {heating},
    OFF {cooling};
  flow heating{
    x' = K * (h - x);
  }
  flow cooling{
    x' = - K * x;
  }
  transition
    ON -> OFF {} when x = M,
    OFF -> ON {} when x = m;
}
```

Complex SHIFT program examples and more information about SHIFT can be found at
http://www.path.berkeley.edu/shift/.

To conclude this section, it seems SHIFT has good properties since it is data-flow and object oriented. With regard to the verification, note that projects were underway to interface SHIFT to KRONOS[1]. However, there seems to be fewer works related to SHIFT in recent years after the end of the original PATH project.

3.3.5 MODELICA

In September 1997, the first version of MODELICA [EOS97, OEM99] was finished, its goal effort is to unify the concepts and design a new uniform language for hybrid model representation. The central property of MODELICA is the usage of *synchronous* differential, algebraic and discrete equations, see [OEM99].
MODELICA is an object-oriented modeling language, hence, it allows *hierarchical modeling*, *encapsulation*, and *inheritance*. In order to increase re-usability of model components, the MODELICA modeling is based on equations instead of statements as in traditional input/output block abstractions, i.e, instead of functions (methods in traditional object-oriented languages) MODELICA uses *equations* to specify behavior.
Note that MODELICA has possibilities to describe hierarchical bond graph models thanks to a MODELICA bond graph library defined in [Bro97].

3.4 Chapter summary

This chapter gives a brief survey of some related works and of course it is not exhaustive for lack of space and of time. There are many other tools for modeling and making simulation of dynamical systems with both continuous and discrete sub-systems, as SCICOS (Scilab Connected Object Simulator) [NS98]. Other tools for hybrid systems verification are emerging as, e.g., the deductive checker The Stanford Temporal Prover (STeP) [MS98] or the symbolic model checker for linear hybrid automata HyTech [HHWT97].

Though the hybrid community is young, i.e., about ten years old, it is widely spread through out the world. Moreover, this community seems to have been divided in two, on the one hand the computer scientists and on the other hand automatic control engineers. Krister Edström wrote in [Eds99] the following quotation.

[1] KRONOS [DOTY96] is a model checker based on the model of timed-automata for verifying temporal properties on the behavior of the system, such as reachability, invariance and bounded response.

> Computer scientists have focused on systems with complex discrete dynamics and simpler continuous dynamics and looked at properties like reachability, safety, etc. ... In the area of Automatic control, the focus has been on systems with complex continuous dynamics, with simpler discrete dynamics. Issues are for example stability and control.

Indeed, hybrid automata, STATECHARTS and SHIFT seem to be more suitable for dealing with complex discrete dynamics and simpler continuous dynamics whereas bond graphs seem to be geared to deal with complex continuous dynamics and simpler discrete dynamics. However, new languages like MODELICA aims to merge the two directions in order to enable analysis of systems with both complex continuous and complex discrete dynamics. Development in this area is still in very preminary stages.

Using SIGNAL with SIMULINK to model complex hybrid systems seems a good idea since SIGNAL is very efficient to model complex discrete behaviors and SIMULINK is well-suited to deal with complex continuous dynamics, this is the direction explored in this report. The underlying mathematical model presented at the begining of this chapter serves this purpose.

Chapter 4

Co-simulation issues: Simulink procedure calls versus global variable passing

This chapter shows two approaches allowing the simulation of hybrid systems. In the first approach, the discrete part modeled with Signal is the master and controls the changes in the continuous part by using Simulink procedure calls. In the second approach, the stand alone program provided by Simulink/Matlab is the master, the links between the discrete part and the continuous part is done by means of global variables. The rest of the report gives up the first approach and explores in detail the second approach.

4.1 Using Simulink procedure calls: a naive approach

Since the Signal program should be the controller of the system, the first idea is to embed Simulink update functions into the Signal stand-alone program. It is possible since Signal allows to directly use external functions with Signal code. In order to simplify the complete system, the Simulink model consists of several Simulink sub-models (i.e. several `<SubmodelSimulink>.mdl` files should be specified).
The triggering of a sub-model is made by calling of the main function of the sub-model. To do that the main function should be renamed (for example, to `<Sub-model>_main`). Theses sub-models behave like following:

```
function(<Sub-model>_main){
  -- initialize
  while true{
    -- resolve sub-model equations
```

```
    if event{ /* for example a threshold reached */
      exit of the loop
    }
  }
  -- terminate
}
```

where the exit is done by means of SIMULINK Stop block.

The working scheme of the system is very simple, at each iteration step of the discrete SIGNAL automaton,

- the automaton checks its states and in particular if a (SIMULINK) sub-system has just finished its run,
- with above information the automaton solves the transition system and (if needed) triggers a different (SIMULINK) sub-system,
- to trigger a sub-system, the automaton calls a function which sets a boolean variable,
  e.g., `<Sub-model>_stop`, to `false` before it calls the sub-system main function,
- to check if a sub-system has just finished its run, the automaton calls a function which returns the value of the `<Sub-model>_stop` variable and sets this variable to `true`.

Since all `<SubmodelSimulink>.c` generated by the RTW contains common function and variable identifiers (like for example `MdlStart`, `MdlOutput` and so on) it is necessary to rename them. The Target Language Compiler (TLC) [Mat97c] allows to make a new target for RTW where the common identifiers are replaced by new identifiers. Thus new identifiers can consist of sub-model names plus old identifiers. For compiling from SIMULINK block diagram to C code, the Real-Time Workshop (RTW) [Mat97a] used "target files", which specify particular code for each block. It is the TLC which is used to write these "target files" (`*.tlc`).

4.1.1 The thermostat example

The thermostat of the section 3.1 has two discrete state ON and OFF within the temperature moves continuously. It is easy to model these two modes with two SIMULINK block diagrams ON and OFF as in the Figure 4.1. Here, in order to simplify the example, m and M are constants, but it is also possible to pass them in parameter.

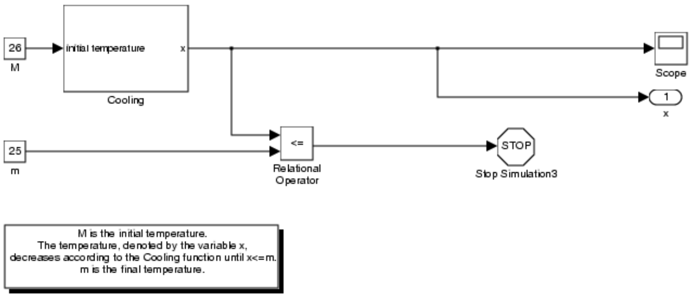


Figure 4.1: Block diagram of OFF

Figure 4.2 illustrates the SIGNAL controller. This SIGNAL program contains four external function calls. These functions are written in C, see appendix A.1 which describes their functioning and their code.
When the event `x_eq_inf` is emitted, the sub-system OFF is stopped because its variable x has reached its low bound m. So, if the event `x_eq_inf` is emitted the automata (the biggest processus of the Figure 4.2) is able to change its state and then provides the starting of the sub-system ON.

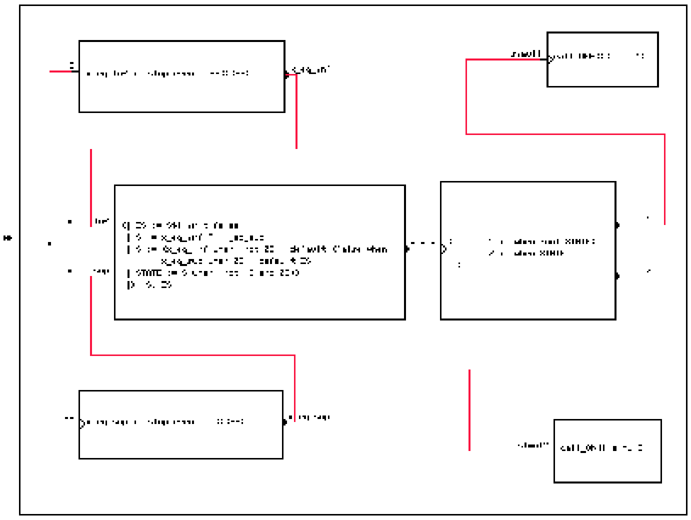


Figure 4.2: SIGNAL thermostat using SIMULINK calls

4.1.2 Discussion

In all cases, the above system is very restricted because its execution is *purely sequential*. This means that :

- two sub-systems can not run at the same time
- when a sub-system runs the controller is frozen,
- when the controller runs no sub-system can be executed.

Thus this system is not suitable to simulate hybrid systems.

One way to resolve those problems is to use threads. If it is possible to encapsulate main C subs-system functions and the automaton into threads then it is possible to run in a concurrent way several subsystems and the automaton. The multi-threading will be seen later. Another way to avoid the purely sequential running is shown in the next section (4.2).

4.2 Using global variable passing

Another way to link SIGNAL model with SIMULINK model(s) is based on the facilities provided by RTW. Actually, it is possible with SIMULINK and RTW to build a system with several sub-systems which seem to be executed at the same time[1]. Thus, the SIMULINK main file (e.g., `grt_main.c`[2]) is a judicious starting point to build the standalone hybrid system simulator since the pseudo-parallelism of the sub-systems is already taken in to account. In this way, instead of several SIMULINK sub-model `<SubmodelSimulink>.mdl`s, only one model `<ModelSimulink>.mdl` is specified. The continuous sub-model are described in the main model by means of SIMULINK Subsystem blocks. The Subsystem blocks are enable by means of SIMULINK Trigger blocks and SIMULINK Enable blocks. Thus the system simulator consists on the one hand of the SIGNAL controller model, and on the other hand of a big SIMULINK model (i.e. the main SIMULINK model). The big SIMULINK model contains input ports allowing SIMULINK Subsystem blocks to be enabled and disabled, and output ports allowing sub-systems to emit events to the controller. The SIGNAL controller model is simpler than in section (4.1) since input signals are replaced by external C functions checking termination and output trigger signals are replaced by external trigger C functions.

In this way, SIGNAL inputs are connected to some SIMULINK outputs and vice versa SIMULINK inputs are connected to SIMULINK outputs. This connection can be made by means of global variable passing.

The SIMULINK input variables can be accessed through `ExternalInputs rtU` and the SIMULINK output variables can be accessed through `ExternalOutputs rtY`.
The `ExternalInputs` and `ExternalOutputs` structure types are automatically defined in the `<ModelSimulink>.h` file. The `rtU` and `rtY` variables are automatically defined in the `<ModelSimulink>.prm`.

[1]RTW provides two basic environments: a singletasking operating system and a multitasking real-time operating system(e.g., Tornado) refer to chapter 7 of the *Real-Time Workshop User's Guide*[Mat97a].

[2]`grt_main.c` is a generic singletasking target main file.

The SIGNAL input variables can be passed by value to the SIGNAL iterate function (i.e., `<ModelSignal>_iterate_Black_Box(inputs, outputs)`). Thus it is possible to pass directly the SIMULINK output variables to SIGNAL iterate function. The SIGNAL output variables should be passed by reference to SIGNAL iterate function.

Now, the question is how the SIGNAL automaton can be embedded in the main C file of the big SIMULINK system. The two next chapters deal with some answers to the above question in details by showing several ways to implement hybrid system simulators.

Chapter 5

Generic model for the global variable passing approach

This chapter gives the generic model for the global variable passing approach and provides three protocols for activation of the SIGNAL part. The generic model is based on the mathematical representation and the architecture of hybrid systems presented in chapter 3.

5.1 Hybrid system with SIGNAL and SIMULINK representation

5.1.1 The mapping of the mathematical representation

The plant is made of a collection of finite Continuous Time Sub-Systems ($CTSS$). As in the mathematical representation of the chapter 3, let I be the cardinality of the collection and let i index over I (the ith $CTSS$ noted $CTSS_i$). Such $CTSS_i$ contains a vector $x_i \in R^{n_i}$ of n_i continuous state and also n_i differential equations (3.1). This last equation can be rewritten as follows:

$$\begin{pmatrix} \dot{x}_i^1 \\ \dot{x}_i^2 \\ \vdots \\ \dot{x}_i^{n_i} \end{pmatrix} = \begin{pmatrix} f_i^1(q, x_i^1, u_i, d_i) \\ f_i^2(q, x_i^2, u_i, d_i) \\ \vdots \\ f_i^{n_i}(q, x_i^{n_i}, u_i, d_i) \end{pmatrix} \tag{5.1}$$

Hence, the system contains $\sum_{k=1}^{I} n_k$ differential equations for each q. However, the implementation needs to extract the discrete parameter $q \in Q$ of these differential equations. Thus we have $J = |Q| \sum_{k=1}^{I} n_k$

differential equations in the continuous system DE[1]. At any time t, one or several equations among this collection forms the basis for computation. Let j be a new index for indexing the DE equations as follows:

$$\begin{array}{rcl} F_1(x_1^1, u_1, d_1) & = & f_1^1(q_1, x_1^1, u_1, d_1) \\ F_2(x_1^1, u_1, d_1) & = & f_1^1(q_2, x_1^1, u_1, d_1) \\ \vdots & & \\ F_{|Q|}(x_1^1, u_1, d_1) & = & f_1^1(q_{|Q|}, x_1^1, u_1, d_1) \\ F_{|Q|+1}(x_1^2, u_1, d_1) & = & f_1^2(q_1, x_1^2, u_1, d_1) \\ F_{|Q|+2}(x_1^2, u_1, d_1) & = & f_1^2(q_2, x_1^2, u_1, d_1) \\ \vdots & & \end{array} \tag{5.2}$$

Thus, we get a new function F_j for the jth equation in the old list of equation DE. That is, by substituting, duplicating and renaming it is possible to find a new indexing system so that only the j subscript remains. Now we can write each differential equation as follows:

$$\dot{x}_j = F_j(x_j, u_j, d_j) \tag{5.3}$$

which allows to calculate the vector continuous state x and the vector continuous output y thanks to equation

$$y = h(x, u) \tag{5.4}$$

which is a rewriting of equation (3.2). Then y feeds the characterizer and the equation

$$e = s(y) \tag{5.5}$$

allows the detection of event e.
In comparison with Figure 3.1, a new component has been added in the controller, it is the *Edge detector* which corresponds to equation (3.4). The discrete state q is defined only in the selector which is the only purely discrete part. So, the selector contains the equation (3.5) and the new equation below:

$$c = g(q') \tag{5.6}$$

where $c \in R^J$ is the vector discrete control of the effector.
The effector deduces from its input c two continuous vectors $u \in R^J$ and $enabl \in B^J$ thanks to:

$$(u_j, enabl_j) = k(c_j) \tag{5.7}$$

$enabl_j$ is used by the plant to enable or disable the jth differential equation and u_j is the vector continuous control of the jth differential

[1] DE is used as the name for set of Differential Equations and J is used as cardinality of DE

equation.

Now that the loop is closed, we can put the above equations (eqn. (5.3) to eqn. (5.7) plus eqn. (3.4) and eqn. (3.5)) together as follow.

$$
\begin{aligned}
\text{if } (enabl_j = 1) \text{ then } \quad \dot{x}_j &= f_j(x_j, u_j, d_j) && (5.8)\\
y &= h(x, u) && (5.9)\\
e &= s(y) && (5.10)\\
\tau_j &= e.\mathbf{1}_{\{e_j \neq e_{j\_}\}} && (5.11)\\
q' &= T(q, \tau) && (5.12)\\
c &= g(q') && (5.13)\\
(u_j, enabl_j) &= k(c_j) && (5.14)
\end{aligned}
$$

Since the discrete controller (the automaton) is in one state at any one computation point, it follows that the change in continuous state is well-defined, i.e. although several equations are enabled in parallel, only one equation at a time is chosen for *each* continuous state variable.

5.1.2 Distribution of computations

It is obvious that the plant should be specified by SIMULINK and the selector by SIGNAL. But for the other components of the system it is not so clear.
There are several possibilities and the Figure 5.1 shows one of them in which the effector, the characterizer and the edge detector are specified by SIMULINK.
Later, we will see that it can be indispensable to specify the edge detector by SIGNAL instead of SIMULINK according to how the selector is activated.

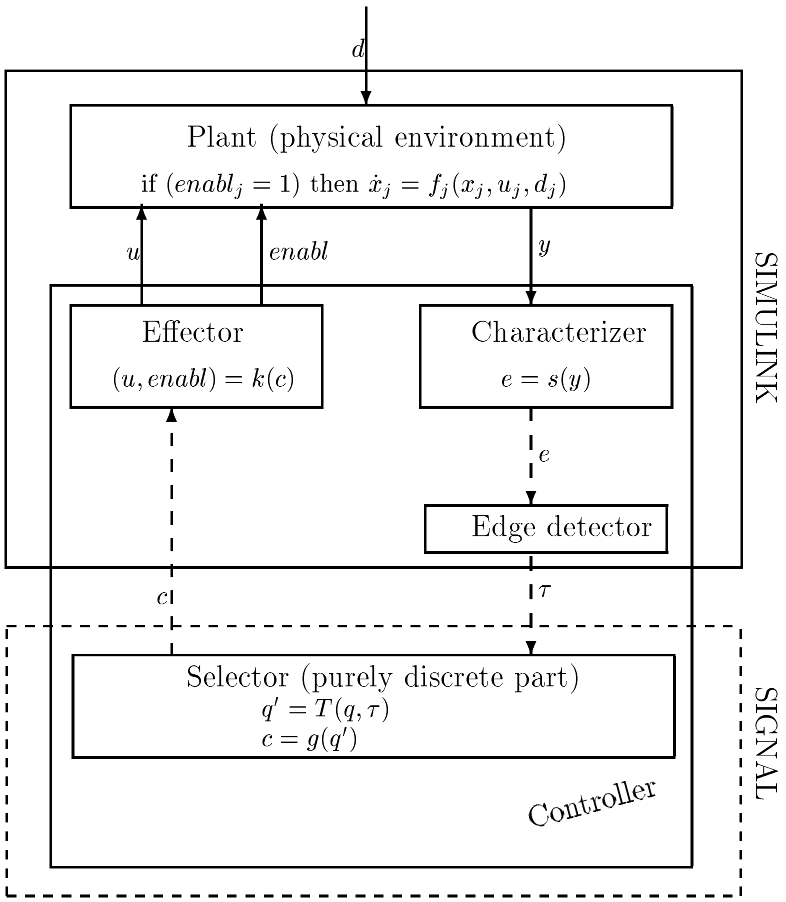


Figure 5.1: Hybrid system representation

5.2 Selector activations

The selector, i.e., the union of equations (5.12) and (5.13) is assumed to work in discrete time, meaning that continuous time t is sampled with period Δt. During each sampling period, the $(e_j(t), e_j(t+\Delta t))$ trajectory is recorded, and it is hoped that each component of e_j changes at most once during the sampling period. If e_j changes during the sampling period then the event τ_j is emitted. Then, there are several possibilities of checking the event τ_j for the selector. These possibilities depend of how the selector is activated. Three activation methods are discussed here, i.e., periodic, aperiodic and asynchronous selector activations.

5.2.1 Periodic synchronous selector activations

Synchronous means here that the selector activation coincides with a tick of the clock of the sampled continuous system.

Protocol 1 *At each sampling period Δt, the selector senses the final value of vector τ_j, and applies its transition according to (5.12).*

This protocol is simple, but assumes that sampling period Δt is small enough to avoid missing events. This may typically lead to taking a Δt much smaller than really needed, i.e., to activate the automaton for nothing most of the time.

5.2.2 Aperiodic synchronous selector activations

Protocol 2 *Here the continuous time system (equations (5.8–5.11)) is the master, driven by continuous real time t. Each time some τ_j occurs in equation (5.11), a "*wake_up*" event is generated by the* j*th continuous time system in which τ_j was generated. Then selector (equation (5.12)) awaits for* wake_up*, so* wake_up *is the activation clock of the selector. When activated, the automaton checks which event τ_j is received, and moves accordingly, following equation (5.12).*

Within this protocol, the master is the continuous time system, and the selector reacts to the events output by the continuous time system. More precisely, the continuous time system outputs *wake_up* (in addition to τ_j), which in turn activates the selector.

5.2.3 Asynchronous selector activations

Here, continuous subsystems and the selector have independent SIMULINK threads, that means above all the selector has its own thread and its activation clock.

Protocol 3 *At each round, the selector senses whether there is some event τ, if it is the case then the selector moves accordingly, following equation (5.12) and finally, it outputs the state changes to the effector following equation (5.13).*

It is important to note that with the Protocol 3 the τ generation should be done in the SIGNAL part instead of in the SIMULINK part as in the Figure 5.1.
Indeed, if the τ is provided by SIMULINK, there is a risk that the selector will miss some τ because no assumption can be made about when the selector will check its input channels. In the best case some τ are recognized with a delay of one tick in the selector
In order to illustrate this problem, Figure 5.2 shows both methods, i.e, τ provided by SIMULINK and τ provided by SIGNAL.
In the two cases,

- the horizontal arrows coincide with the ticks of the selector activations;
- e_j, τ_j, $e_{j'}$, and $\tau_{j'}$ are boolean signals;
- e_j, and τ_j have same sample period Δt_j;
- $e_{j'}$, and $\tau_{j'}$ have same sample period $\Delta t_{j'}$.

In Figure 5.2(a), where τ is provided by SIMULINK, there are the following features:

- if e_j equals 0 at time t_j and 1 at time $t_j + 1$ (i.e the following sample) then τ_j equals 0 at time t_j and 1 between $t_j + 1$ and $t_j + 2$
- if τ_j equals 1 at time t_j then at time $t_j + 1$, τ_j equals 0.
- $e_{j'}$ and $\tau_{j'}$ behave as e_j and τ_j

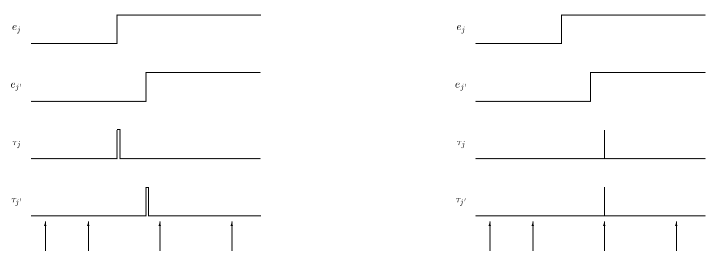
So, τ_j is not synchronized with the selector but only with the sample rate of e_j. Hence in the Figure τ_j and $\tau_{j'}$ are lost.

On the other hand, in Figure 5.2(b), where τ is provided by SIGNAL, there are the following features:

- if e_j equals 0 at round k of the selector and 1 at following round $k + 1$ then τ_j equals 0 at round k and 1 at round $k + 1$
- if τ_j equals 1 at round k then at round $k + 1$, τ_j equals 0.
- $e_{j'}$ and $\tau_{j'}$ behave as e_j and τ_j
- τ_j and $\tau_{j'}$ are not defined between two rounds.

So, τ_j and $\tau_{j'}$ are synchronized with the selector clock. Hence it is impossible for the selector to know which of the two events e_j and $e_{j'}$ has occurred first. However, if it needs, it is possible to time-stamp τ_j and $\tau_{j'}$ with an absolute date and then the selector can manage with the time stamp τ. This holds only if e_j and $e_{j'}$ are generated in the same subsystem. In practice, one needs complex clock synchronization routines to ensure that time-stamped events generated in different sub-systems are meaningful.

Though it is still possible for the selector to miss some event (e.g., if e_j moves from 0 to 1 and then to 0 between two rounds of the selector), this protocol has the advantage of being fully distributed.



(a) in the SIMULINK part (b) in the SIGNAL part

Figure 5.2: τ detection

Chapter 6

Implementing the co-simulation

In order to show how the hybrid system representation of the Figure 5.1 can be implemented with SIGNAL and SIMULINK this section uses the simple example of the thermostat already presented before. The implementation in the synchronous case is dealt with in section 6.1, while the asynchronous case is dealt with in the following section 6.2.

6.1 synchronous selector activations

6.1.1 SIMULINK modeling

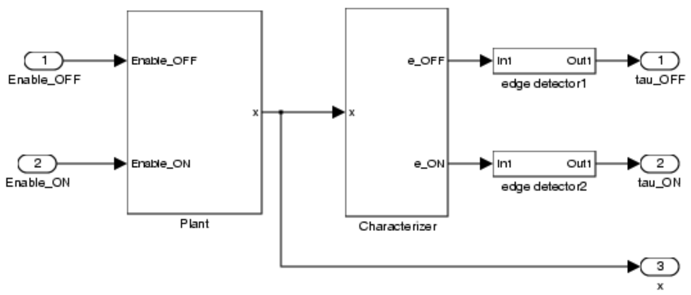


Figure 6.1: SIMULINK part of the thermostat

Figure 6.1 shows the main SIMULINK block diagrams of the thermostat. This block diagram contains a plant block, a characterizer block and two edge detector blocks. It does not contains the effector block because this example is very simple and the effector is not needed here.

The Plant

The plant of the thermostat (Figure 6.2) contains the following features.

- Two input variables, i.e., $Enable\_OFF$ and $Enable\_ON$.
- Two continuous time sub-systems, i.e., **OFF** and **ON**. These subsystems are activated according to the $Enable\_OFF$ and $Enable\_ON$ values as in the equation 5.8 and thanks to the enabling ports at the top of the sub-systems **OFF** and **ON**. A block diagram having such an enabling port, executes while the input received at the enabling port is greater than zero. For more information about SIMULINK enabled subsystems, see Chapter 7 on *Using* SIMULINK [Mat97d]. Thus, if the input signal *enable_OFF* is 1 then **OFF** is running.
- An output variable x, i.e., the temperature. At time t, the output variable x equals the output variable of the activated sub-system as following:

$$x = (x\_OFF * Enable\_OFF) + (x\_ON * Enable\_ON) \quad (6.1)$$

So, the selector prevents **OFF** and **ON** from being activated at the same time or else x will get a wrong value.

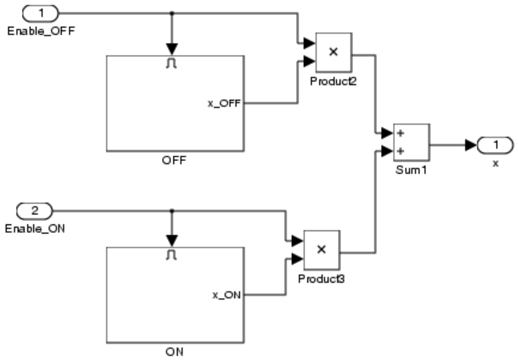
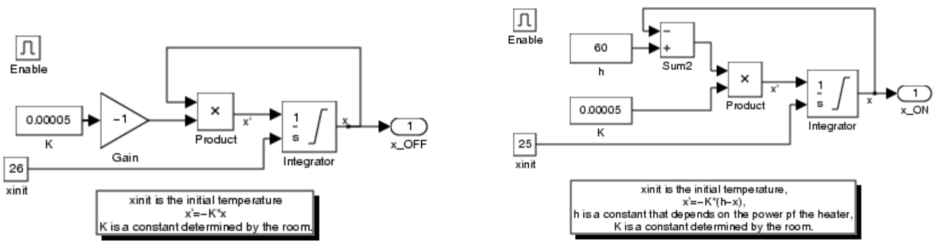


Figure 6.2: The Plant of the thermostat

The continuous sub-systems



(a) The sub-system **OFF** (b) The sub-system **ON**

Figure 6.3: The sub-systems

Figure 6.3 shows the block diagram of sub-systems **OFF** and **ON**. Each sub-system calculates a differential equation, see the end of chapter 3 of *Using* SIMULINK [Mat97d] where the modeling of differential equations is explained. Note that each sub-system **ON** and **OFF** has an enabling port at the left top.

The Characterizer

Figure 6.4 shows the block diagrams of characterizer of the thermostat. Its input is the temperature x and it has two output ports to be able to broadcast the event e_OFF if the temperature reaches the lower threshold m and the event e_ON if the temperature reaches the upper threshold M.

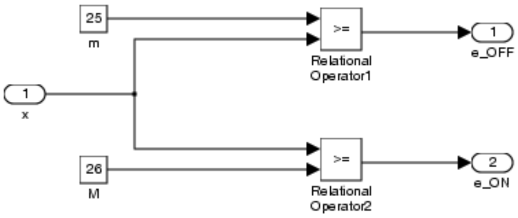


Figure 6.4: The Characterizer of the thermostat

The edge detectors

The modeling of edge detectors is described in the section 2.2.1.

6.1.2 SIGNAL modeling

The SIGNAL part contains just the selector since the edge detector part is implemented in the SIMULINK part. Figure 6.5 shows the SIGNAL model is build of a synchronizer process (the left-hand block) and of an automaton process, i.e., the selector, (the right-hand block) which is described in detail in Figure 6.6.

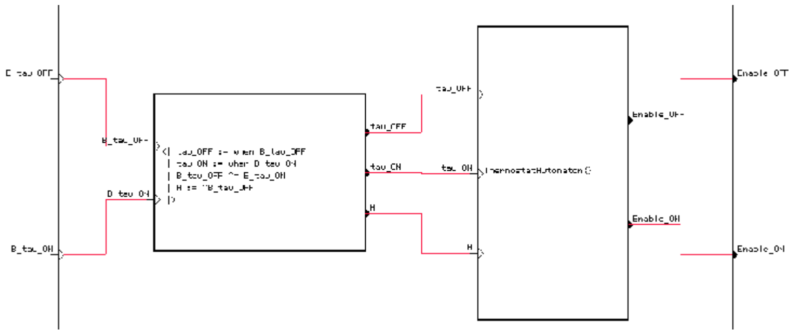


Figure 6.5: SIGNAL part of the thermostat

The synchronizer block is necessary because the inputs of the selector are just boolean C variables without well defined clock. Thus, **B_tau_OFF** means the tau_OFF with boolean type and **tau_OFF** means the event tau_OFF.

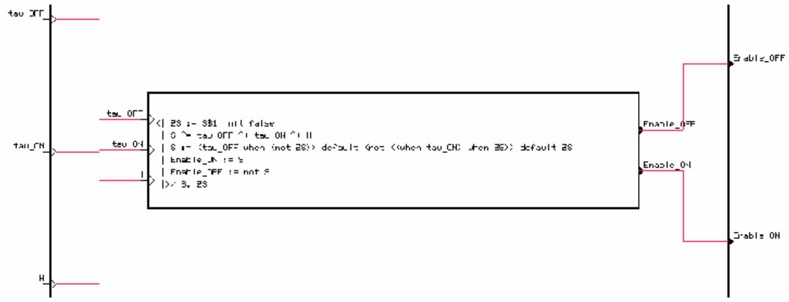


Figure 6.6: ThermostatAutomaton

Note that the inputs of the selector are identical (except for the type) to the outputs of the characterizer and the outputs of the selector suit well to the inputs of the plant. Nevertheless, after modeling there is still a need to link the selector and the SIMULINK part together. Such a work depends on how the selector is activated.

6.1.3 Periodic synchronous selector activations

In order to execute a model, the main function of the SIMULINK main file, i.e., `grt_main.c`, runs an infinite loop such as :

```
while true{
  OneStep_Simulink_model;
}
```

So, to implement protocol 1, it is just needed to add the iteration function of the SIGNAL part of the controller in the loop as below:

```
while true{
  OneStep_Simulink_model;
  OneStep_Signal_controller;
}
```

The main function of the `grt_main.c` file of the thermostat application modified according to Protocol 1 is showen in appendix B.2.1.

6.1.4 Aperiodic synchronous selector activations

For the aperiodic synchronous selector activations, the main loop becomes:

```
while true{
  OneStep_Simulink_model;
  if "wake_up" { OneStep_Signal_controller;}
}
```

The "wake_up" should be produced only if any τ_j occures. So, such "wake_up" can be expressed by means of tests upon the τ_j as follow:

if $(\exists \tau_j \mid \tau_j = 1)$ OneStep_Signal_controller;

The main function of the `grt_main.c` file of the thermostat application modified according to Protocol 2 is showed in appendix B.2.2.

After compiling and running the model defined above, it is possible to see the traces. In this way, Figure 6.7 shows the temperature x moving between the values 25 and 26. The vertical lines indicate when the signals tau_OFF and tau_ON equal 1, but in order to differentiate at the time of displaying, tau_OFF has been multiplyed by 20 and tau_ON by 25.

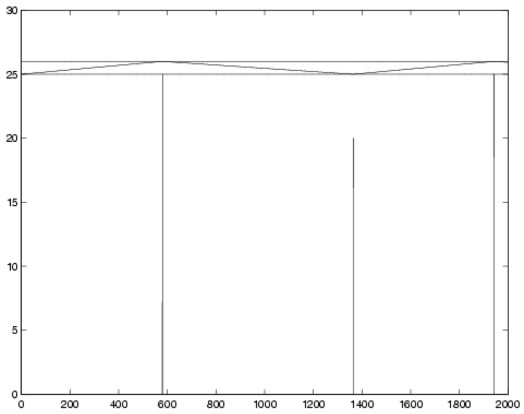


Figure 6.7: Temperature traces

The result is the same with the two protocols 1 and 2. Since the selector is assumed to have a negligible time of computation, it makes no difference which protocol among the protocols 1 and 2 is used in the application. However, considering the real time of computation, protocol 2 seems more accurate.

6.2 Asynchronous selector activations

6.2.1 SIMULINK modeling

The main changes resulting from the asynchronous model are in the SIMULINK modeling. Indeed, one needs to create a multi-rate model in order to use multi-threading or pseudo multi-threading with SIMULINK, because threads are associated with sub-systems with the same sampling rate, see Chapter 7 of *Real-Time Workshop User's Guide*[Mat97a].
The idea for simulating the wanted multi-threading, i.e., a thread for the SIGNAL part and others for the continuous sub-systems parts, is the folowing:

- set an activation clock for the SIGNAL part thanks to a tick generator whose its sampled rate is arbitrary fixed, as illustrated in Figure 6.8.
- fix sampled rate for the characterizer as illustrated in Figure 6.9

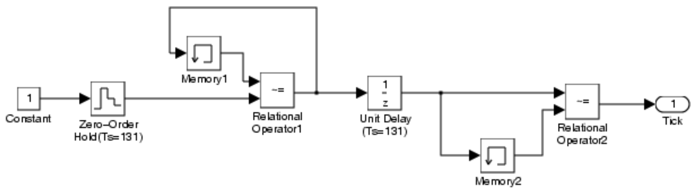


Figure 6.8: SIMULINK generator of the clock of the selector

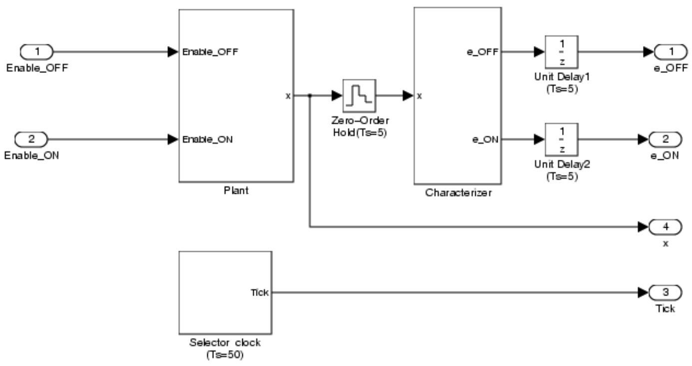


Figure 6.9: SIMULINK part of the thermostat for Protocol 3

6.2.2 SIGNAL modeling

Section 5.2.3 explains why the edge detectors have to be in the SIGNAL part when the selector activations are asynchronous with the events from the characterizer. Figure 6.10 shows the SIGNAL model of the thermostat with the two edge detector process (Figure 6.11) and the automaton (selector) process. This automaton is exactly the same as in section 6.1. Moreover, note that an external input H, i.e., a clock activation, takes place of the synchronizer process of Figure 6.6.

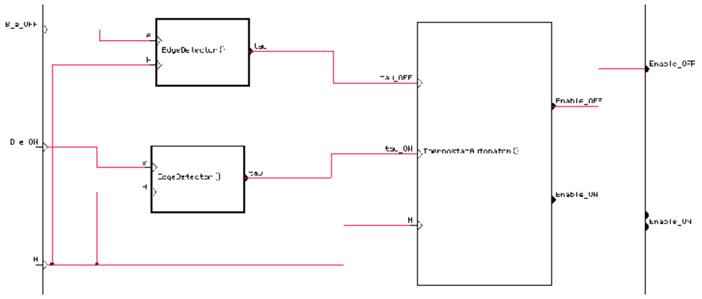


Figure 6.10: SIGNAL selector for Protocol 3

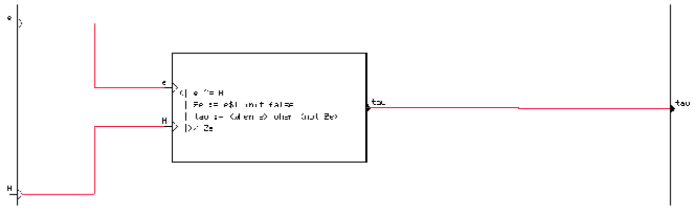


Figure 6.11: SIGNAL edge detector

6.2.3 Linking

To link the two models above, it is enough to implement the following loop :

```
while true{
  OneStep_Simulink_model;
  if Tick { OneStep_Signal_controller;}
}
```

Of course, here the selector is not really embedded in a thread (or pseudo-thread) since only the tick generator is embedded in a thread (or pseudo-thread). However, it is not a problem if the selector computes in negligible time.

After compiling and running the model defined above, it is possible to see the traces. Figure 6.12 shows the temperature x, which should move between the values 25 and 26, moves over these bounds. The vertical lines going up the value 5 indicate the $Ticks$ of the selector. The vertical rectangles indicate when the signals e_OFF and

e_ON equal 1, but in order to differentiate at the time of displaying, e_OFF has been multiplied by 20 and e_ON by 25. Furthermore, the left edge of these rectangles indicates the moment where the signals e_OFF and e_ON are emitted for the first time. The very moment of the discrete transition is on the $Tick$ taking place between the left and the right edges of these rectangles.The delay between the edges and the $Ticks$ are due to the difference of the frequencies of the selector and of the characterizer. So the larger the rectangle is the longer the delay between the event and the discrete transition is and the more the temperature moves over the bounds.

This protocol seems worse than protocol 1 or 2, however it remains interesting to study because it is like distributed simulation.

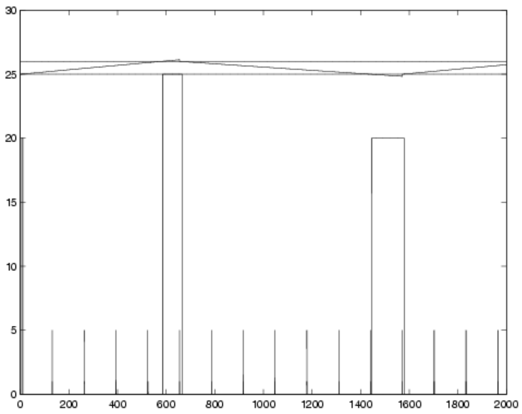


Figure 6.12: Temperature traces with the Protocol 3

It should be possible to find other means to implement each of the three protocols. Especially for the third, it should be interesting to implement it on top of a real-time kernel which deals with true multi-threading and really embedded the selector in a thread. A good thing will be to build a real-time multi-threaded SIGNAL system which allows to simulate hybrid systems with SIMULINK or other tools dealing with differential equations.

Part III

Summarizing application

Chapter 7

The siphon pump machine

There are two kinds of discrete dynamic change in a plant. On the one hand there are those which are the consequence of discrete controllers interacting explicitly via binary actuators such as hydraulic shunt valves, or mechanical clutches, electrical relays; see e.g. the thermostat example. On the other hand there are discrete dynamic changes which are due to internally controlled discrete devices such as diodes, free wheeling devices or hydraulic check valves. So it is interesting to see how it is possible to simulate with the methods proposed in chapter 6 a system having both explicit control and internal mode change. The case study presented here provides to such a system. Indeed the siphon pump machine contains hydraulic shunt valves and hydraulic check valves. This example is modeled carefully in order to obtain a simulation as realistic as possible.

7.1 Historical background

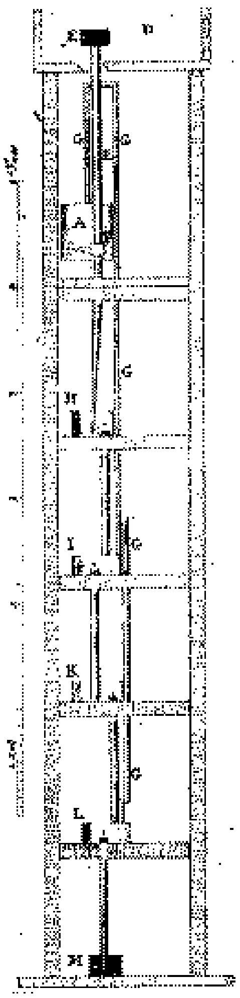


Figure 7.1: Gabriel Polhem's model of the siphon pump machine, copied from [Lin51, Str94]

The *siphon pump machine* was proposed in 1697 by the Swedish engineer Christofer Polhem (1661-1751). The pre– and post–history of the pump machine is covered in [Lin51] in Swedish. More recently, Jan-Erik Strömberg has used the Polhem's siphon pump machine to illustrate his thesis [Str94] and several parts of papers [SNT94, SST94, SNTT96] which I use in this section verbatim.
The siphon machine was designed to replace the contemporary, quite complex mechanical devices for draining the copper mines in Falun (Sweden). The main feature was that it contained almost no movable parts and especially it did not contain the normally indispensable water-wheel.
Hence, the maintenance cost was so considerably reduced that Christofer Polhem offered Bergskollegium[1] to personally take care of all the maintenance at a 'modest' yearly cost.
However, Polhem's proposal was rejected by the Bergskollegium because his pump seems too good to be true and he refused to reveal any of the details before he was contracted for the project. There is maybe another reason, it seems also that Christofer Polhem was "quite greedy" and his "modest" payment, he had in mind for the maintenance, just happened to slightly exceed the cost of maintaining the old pumps.
Near 1727, the idea of Polhem's pump eventually reached the prospectors of the Harz mines in
Klausthal (Germany). After some negotiation with Christofer Polhem, it was finally decided that a small-scale model (Figure 7.1 taken from [Lin51]) of the siphon machine was to be built. This model was in fact finished by Polhem's son Gabriel and delivered to prospectors of the Harz in 1747. In Jan-Erik Strömberg's opinion, it is not likely that a full scale version of Polhem's invention was ever built, though a machine based on similar principles was later used in Klausthal.

[1]The Bergskollegium is the Board of Mining and Metallurgical Industries

7.2 Principles of operation

7.2.1 The pump description

The informal abstraction of the pump of the Figure 7.2 shows the pump machine consists of a hydraulic system and a separate pneumatic supply system.

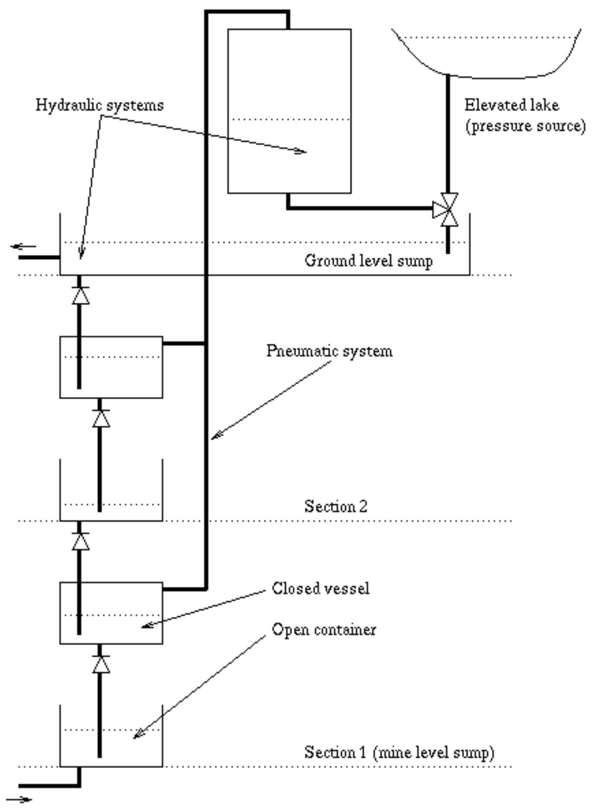


Figure 7.2: An informal model of the siphon pump machine copied from [Str94]

The hydraulic system

The hydraulic system consists of a mounted cascade of elementary sections where a section consists of one open container connected on its top with a closed container by a check valve. Actually closed container is pressure vessel, such vessel is also connected on its top by a check valve with the open containers of the next section. The check valves are directed in such a way that the water will flow upward only. Figure 7.2 shows only two such sections. The bottom open container is an abstraction of the sump at the bottom of the mine and the top open container is an abstraction of the drained ground level sump.

The pneumatic supply system

The pneumatic supply system is depicted to the right and to the top of the hydraulic part of the diagram and consists of

- an air-compressing vessel,
- a three-way valve (i.e., discrete shunt valve),
- an elevated lake supplying near constant over-pressure.

7.2.2 Working principles

The purpose of the pump is to lift the water flowing into the sump at the bottom of the mine to the drained ground level sump. This pump works in a two-phase (pull and push) manner as follows.

The pull phase

In the pull phase, the pressure vessels are de-pressurized by opening the shunt valve to drain the air-compressing vessel at the ground level. Now the water will be lifted from all the open containers to the pressure vessels next above. Hence, as a result of this first phase, all the pressure vessels will be water filled.

The push phase

In the pull phase, the pressure vessels are pressurized by opening the shunt valve to fill the air-compressing vessel from the elevated lake. Now all the pressure vessels will be emptied via the connections to the open containers next above. Hence, as a result of this second phase, all the open containers will again be filled with water. However, the water has now been shifted up-wards half a section.

By repeating these two phases the water is sequentially lifted to the ground level.

Figure 7.3 depicts a fraction of the siphon pump machine. The water entering the bottom container (flow q_1) is lifted to the top container by lowering and raising the pneumatic pressure p_c in the closed vessel. Due to the check valves, the water is forced to move upwards only. The reason why more than three containers and vessels are needed in practice, is that the vertical distance between any pair of vessel and container is strictly less than 10 meters since water can be lifted no higher than $\approx$ 10 meters by means of the atmospheric pressure ($\approx$ 1 bar).

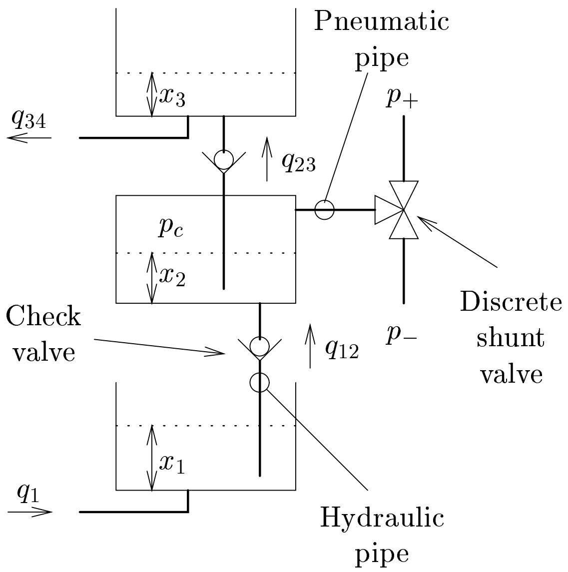


Figure 7.3: A fraction of the siphon pump machine copied from [Str94]

Chapter 8

The pump modeling

In order to provide as genuine parameter values as possible, and for the sake of validation, the model of the pump described in this chapter is a "scaled pump", i.e. a model of the real pump down-sized a factor ten or so. The size of this suggested pump almost coincides with the size of the model built in 1727 by Gabriel Polhem (son of Christofer). Naturally the model built by Gabriel was not a mathematical model but a three dimensional physical model built in wood, steel and copper, see Figure 7.1.

The first step, in order to build a simulator with SIGNAL and SIMULINK of a system, is to build a mathematical model of this system and to gather the parameter values. Then it should be already possible to model the plant with SIMULINK. The second step is to choose a strategy for controlling the system and then build the controller according to the strategy chosen and also according to the protocol of selector activation wanted. Concerning the pump, the mathematical model, the setting of the parameter values as well as a naive controller have been given by Jan-Erik Strömberg and this work is reported in Appendix C.

This chapter 8 shows the architecture of a simulator built from Jan-Erik Strömberg's mathematical specification of the pump. Note that only a fraction of the pump (which is illustrated in Figure 7.3) is modeled here and not the whole pump with the pneumatic system. Moreover, the model presented here has been made by means of protocol 2 provided in chapter 5.

8.1 The hybrid system architecture of the pump

After having the mathematical specification of the pump, it is possible to represent the simulator by means of the architecture presented in section 5.1.1. Then the architecture obtained in this way is such as the diagram of the Figure 8.1.

At the highest level, the pump has the external flow q_1 $[m^3/s]$ entering container 1 as input and the external flow q_3 leaving the container 3 as output.
The flow q_1 entering container 1 is determined by the environment (ground water entering the mine cannot be controlled but is defined by mother nature). Hence q_1 is a *disturbance* signal.

At a lower level, the pump is modeled with a plant, an effector, a characterizer and a selector. Obviously, the selector acts on the pneumatic pressure in container 2, i.e, p_c $[Pa]$. Then the effector provides from p_c and from the gravity induced hydraulic pressure due to accumulated water in containers ($p1$, $p2$ and $p3$) the net driving pressure of the vertical pipes (p_{12} and p_{23}). Hence, in addition to q_1, the plant uses p_{12} and p_{23} to calculate the output flow q_3.
In order to stimulate the selector, the characterizer "watches" continuously the water level depth of the container (x_1, x_2 and x_3) and sends event τ to the characterizer when it is necessary.

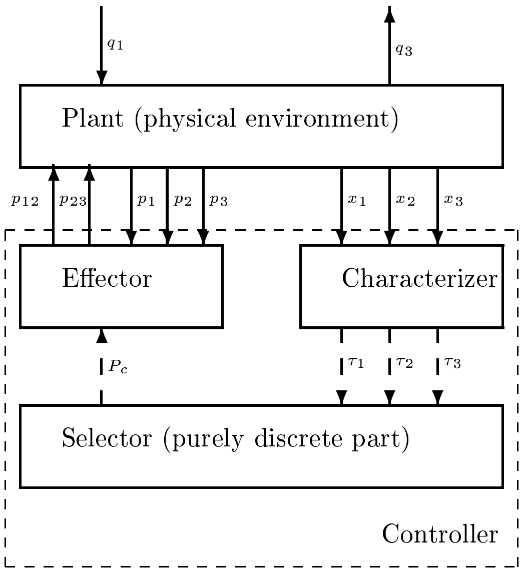


Figure 8.1: General hybrid system architecture of the pump

8.2 The plant

The plant, i.e. the physical environment is depicted in Figure 8.2. It contains mainly two check valve systems. Each check valve system is a hybrid system. Indeed, the water flow through a check valve behaves differently according to the mode of the latter. In the checked mode the water flow is null and in the cracked mode the water flow follows a differential equation.
Hence, the check valve can be also modeled using both SIMULINK and

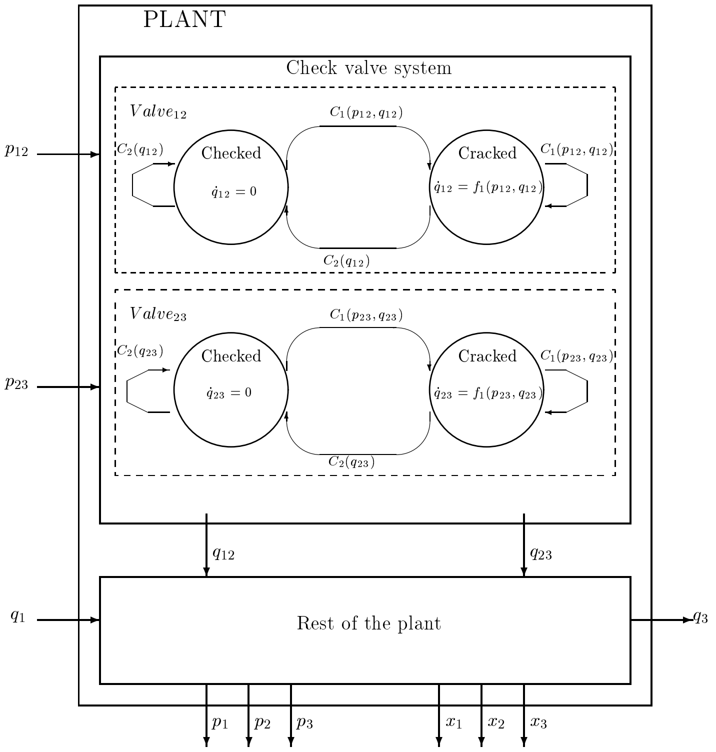


Figure 8.2: The architecture of the plant

SIGNAL. The discrete mode selector is modeled in SIGNAL and the rest in SIMULINK.

The check valve

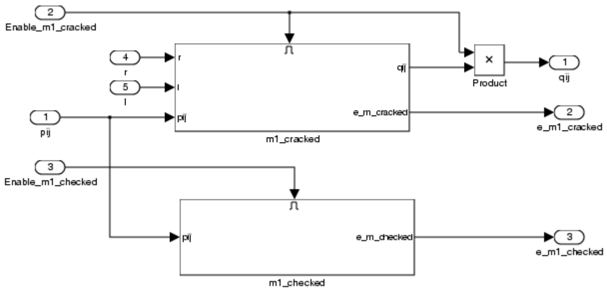


Figure 8.3: SIMULINK block diagram of the check valve m1

The principle of the check valve m_1 is similar to the thermostat example since they have both two discrete states. However, though it is possible to separate the characterizer part from the "plant" part of the check valve, in Figure 8.3 the characterizer is embedded in the sub-system (**m1_cracked** and **m2_cracked**).

Concerning the SIGNAL part, the Figure 8.4 shows that the selector of the check valve is the same as the selector of the thermostat example, only the identifiers change.

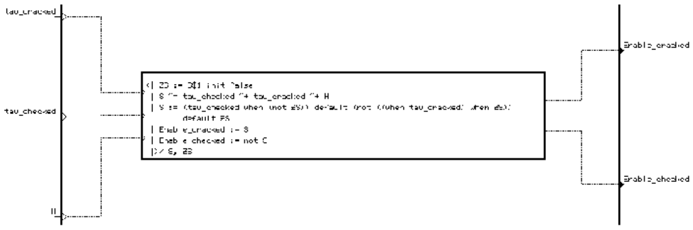


Figure 8.4: SIGNAL selector of a check valve

The SIMULINK block diagram of the plant

Once the check valve part is done, the plant can be completely modeled such as in the SIMULINK block diagram of the plant (Figure 8.5). Note that all inputs and outputs which were present in Figure 8.2 are also present in the SIMULINK block diagram. Moreover, this last has also some inputs and outputs in order to allow the mode switching of the check valves by the external SIGNAL check valve selector. The SIMULINK block diagram of the plant has also *q12* and *q23* as outputs in order to display their trajectories after running the simulation.

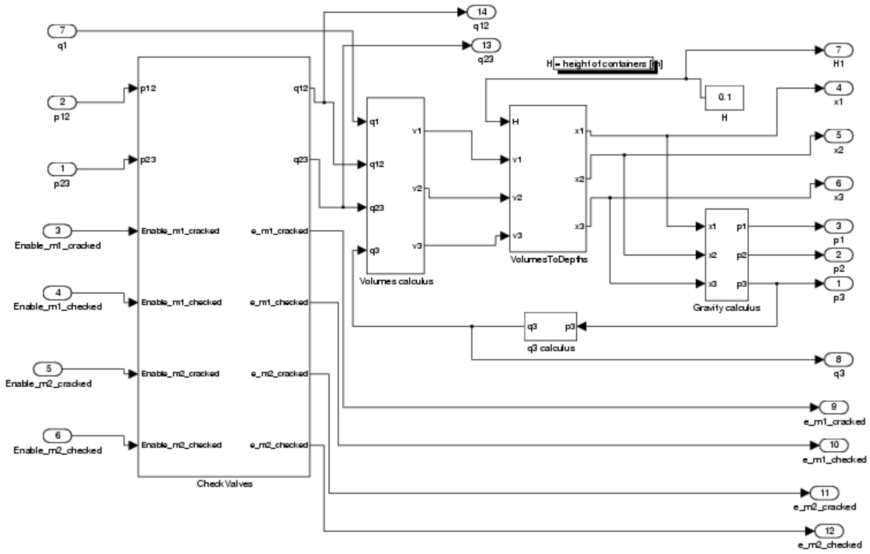


Figure 8.5: SIMULINK block diagram of the plant of the pump

8.3 The control strategy

Finding a safe and optimal controller is far from easy. One of the more important requirements is the following. It is important to minimize the number of switches of the value of p_c. Changing p_c from $+50kPa$ to $-50kPa$ and vice versa results in a significant amount of energy loss. One solution is to maintain p_c constant over as long periods as possible.
Another important requirement is to maximize the output flow q_3 without risking that x_i will end up outside the safe interval defined. Especially under all possible disturbances (q_1).

The naive controller proposed by Jan-Erik Strömberg in appendix C.3 can be depicted by the automaton of Figure 8.6. Note Jan-Erik Strömberg was aware of this is *not* a robust controller.

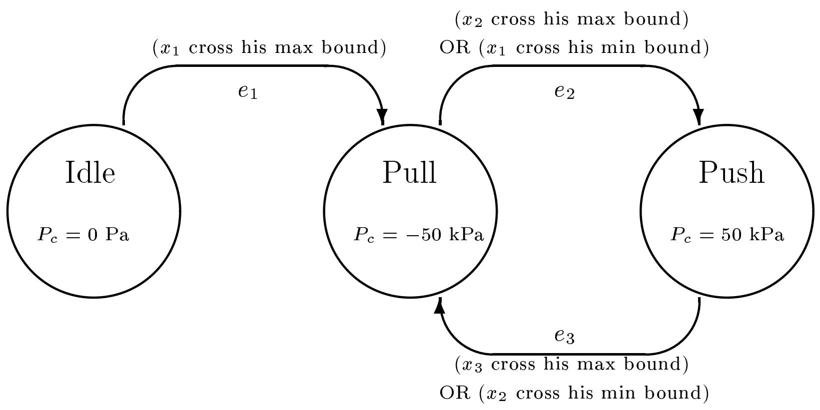


Figure 8.6: Automaton of the control strategy of the selector

The principle of this controller can be informally described as follows.

1. First of all, the first discrete state, i.e., the **Idle** state, is the initialization state. At the beginning the tree containers are empty. So it is necessary first to let the bottom containers fill. This is what is done in the **Idle** state.

2. When the first container is full enough, an event is broadcast by a level sensor (which is simulated by the characterizer) and the pump moves from the **Idle** state to the **Pull** state.

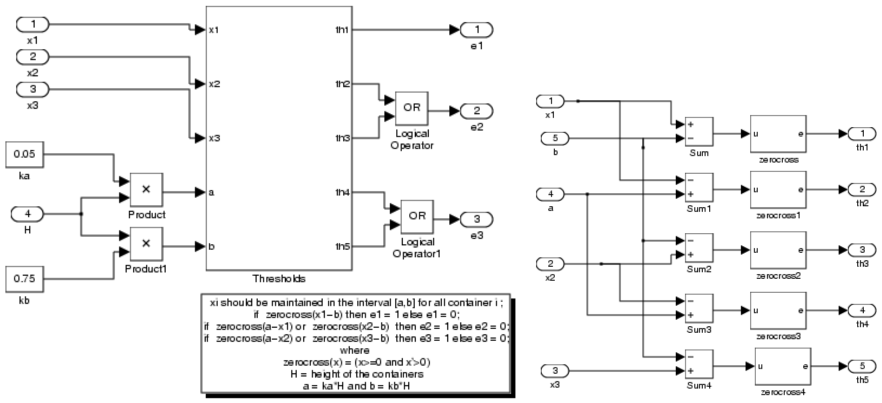
3. In the **Pull** state, container 2, i.e., the pressure vessel, is depressurized. Hence container 2 fills from the container 1. Note that container 1 is continuously filled by the input flow q_1 which is uncontrollable. So the water level of container 1 moves according to the input flow q_1 and the flow q_{12} in the pipe between the two containers 1 and 2. When both are possible the level of container 1 either rises or falls.

4. If the water level of container 1 moves down until a given minimum threshold (detected by a sensor) or if the water level of container 2 is high enough then the pump moves from the **Pull** state to the **Push** state.

5. In the **Push** state, container 2 is pressurized. Hence container 2 stops filling from container 1 and fills container 3. So, container 1 continues to fill according to the flow q_1 and container 3 fills according to the flow q_{23} (in the pipe between the two containers 2 and 3) and the output flow q_3. And of course container 2 is emptied.

6. Finally, if the water level of container 2 reaches its minimum threshold or if the water level of container 3 is high enough then the pump comes back from the **Push** state to the **Pull** state. Thus, the loop is closed.

The above automaton gives which events lead to discrete state transitions of the selector and how these events are deduced. Hence it is easy to model a characterizer which watches the different water levels and provides the suitable events. Such a characterizer is depicted in Figure 8.7.



(a) The characterizer

(b) The SIMULINK block diagram which simulates the sensors

Figure 8.7: SIMULINK block diagram of the characterizer of the pump

In the same way the effector is depicted in Figure 8.8.

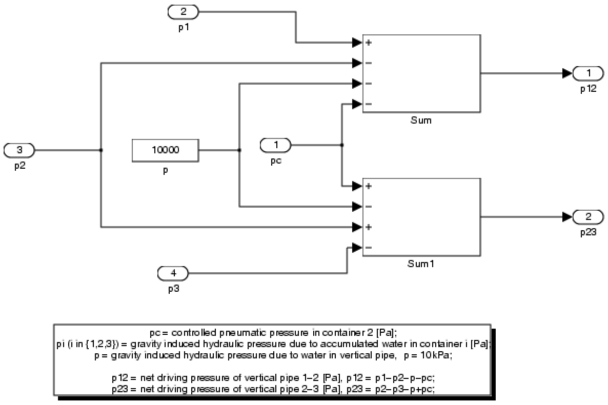


Figure 8.8: SIMULINK block diagram of the effector of the pump

Once the plant, the characterizer and the selector are built, all these components should be linked together such as in the Figure 8.9.

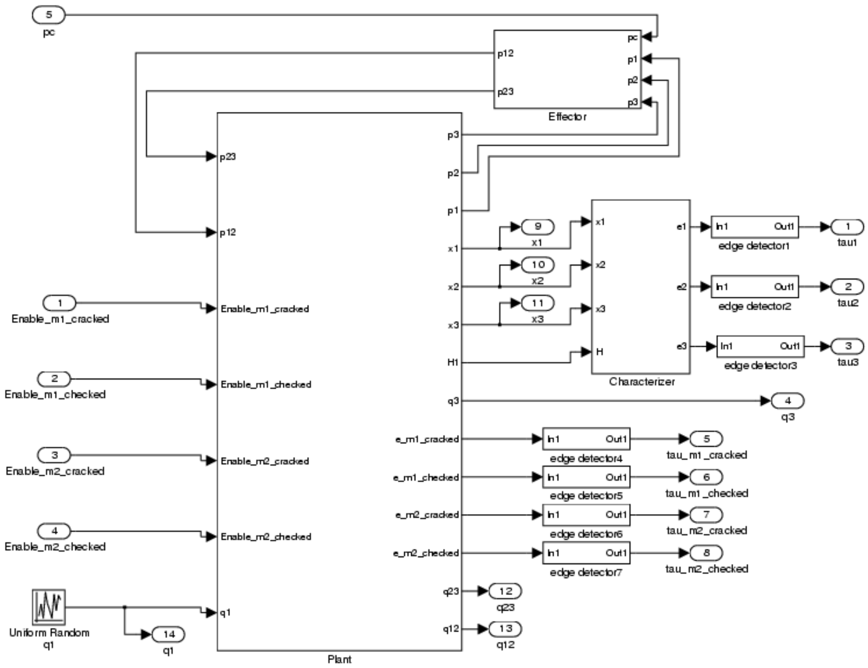


Figure 8.9: SIMULINK block diagram of the whole pump

The last thing to do to conclude modeling of the pump, is to implement the controller automaton with SIGNAL, see Figure 8.10. Instead of having an automaton with two states as the selector of the check valves, here there is three states. Moreover the output of the selector is now the pressure p_c which is an integer. There is also a second output (**event_pc**) which is a SIGNAL event emitted when a discrete transition occurs. This variable is used by the main function of the `grt_main.c` to update the input *pc* of the Effector only when discrete state of the selector changes.

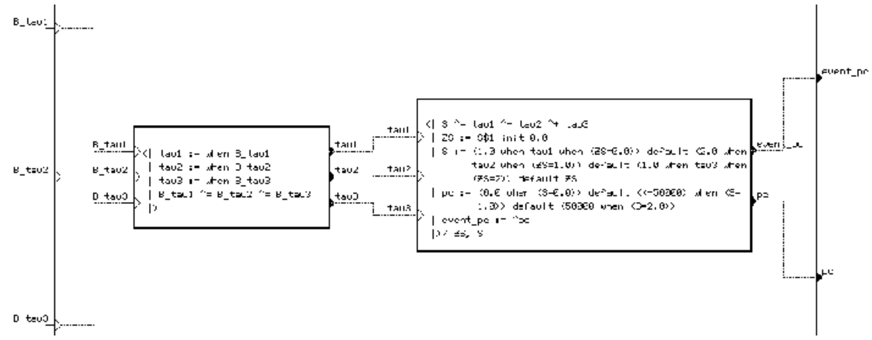


Figure 8.10: SIGNAL selector of the pump

Once the model of the pump is built, it remains just to experiment with it. This part of the work is dealt with in the next chapter.

Chapter 9

The pump simulation

It was a priori known that the naive controller is not robust. This chapter illustrates this fact by simulations of the pump with different constant values for the input flow q_1. Indeed, simulations of the pump show two kinds of behavior. If the flow q_1 is too large then container 1 overflows. If the flow q_1 is too weak then the switches of the value of p_c happens faster and faster.

9.1 Overflow

Figure 9.1 shows traces of the input flow q_1, the flows in the pipes q_{12} and q_{23}, and the output flow q_3 while a simulation where q_1 is constant and equals $2.10^{-6}\,m^3/s$. These flow traces do not show a bad behavior. Indeed they show at the first step (the **Idle** state) the flow q_{12} in the pipe between containers 1 and 2 and the flow in the pipe q_{23} between containers 2 and 3 are null. Then after a while, the system moves from the **Idle** state to the **Pull** state and q_{12} is positive and q_{23} is still null until the system moves from the **Pull** state to the **Push** state. In the **Push** state q_{23} is positive and q_{12} becomes null and the output flow q_3 becomes also positive since container 3 is filling. In the step following, the system moves from the **Push** state to the **Pull** state. Then $q_1 2$ becomes positive and q_{23} becomes null so q_3 decreases because container 3 empties. Then the switching between **Push** state and **Pull** state goes on.

Such behavior of the pump seems correct, however the observation of only the flows in the different pipes is not sufficient to say this pump behaves well. Indeed it needs also to watch the water levels in each container to check whether there is overflow. Figure 9.2 shows such traces. The water level of the ith container is denoted by x_i. The height H of the containers is equal to $0.1\,m$. The upper sensors are situated at $0.075\,m$ from the bottom of the container. The lower sensors are situated at $0.005\,m$ from the bottom of the container. At the beginning of the simulation, i.e., at time $t = 0$, the water level in container 1 is $0.02\,m$ and all the other containers are empty.

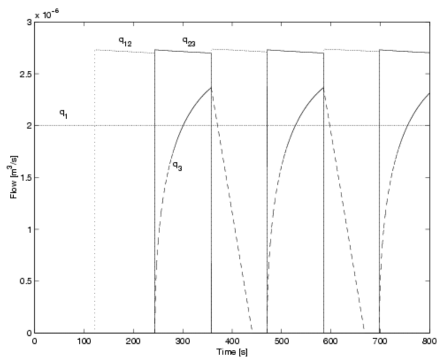


Figure 9.1: Water flows of the system with $q_1 = 2.10^{-6}\, m^3/s$

Note the fact that container 1 being not empty changes nothing at the simulation since in the **Idle** state) container 1 fills. What is important in these traces is that around $t = 350\, s$ container 1 overflows since x_1 reaches the value of H. Because water was not lifted fast enough against the input water flow q_1. The controller is not to blame, since overflow is due to q_1 which is uncontrollable. Moreover, overflow is easily detectable. So it should be interesting to find in witch domain of q_1, the pump can work without problems. However, the next section shows that even if there is no overflow, the controller has a bad behavior.

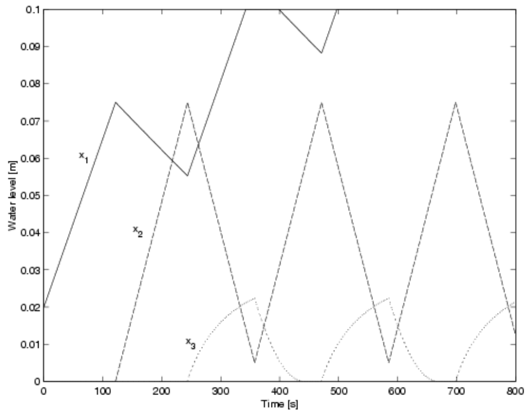


Figure 9.2: Water levels of the system with $q_1 = 2.10^{-6}\, m^3/s$

9.2 Explosion of switches

A new simulation has been done with q_1 constant and equal to $1.10^{-6}\, m^3/s$. Figure 9.3 shows that the output flow q_3 after a while tends towards the q_1 value. That sounds good. However if we examine the other flows in Figure 9.4 or the water levels in Figure 9.5 it appears obviously that the switches between **Pull** state and **Push** state are

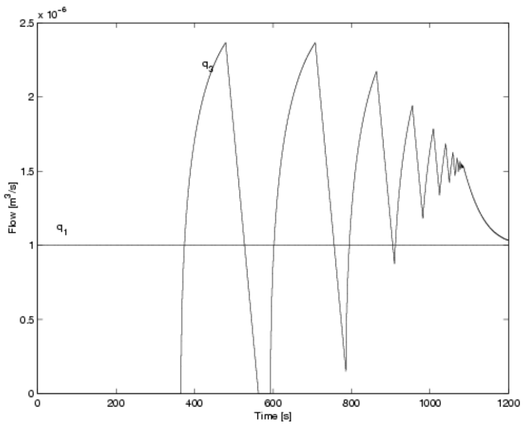


Figure 9.3: Water flows q_3 of the system with $q_1 = 1.10^{-6}\, m^3/s$

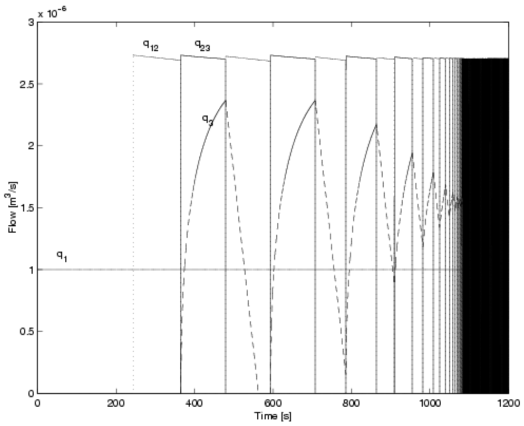


Figure 9.4: Water flows of the system with $q_1 = 1.10^{-6}\, m^3/s$

faster and faster. Recall that one of the important requirements was to minimize the number of switches of the value of p_c, (see the section 8.3).

The intention of this study was not to find an optimal controller. The first goal was to show that it is possible to build a realistic simulator of a fairly complicated physical model with the method proposed in the preceding chapters. The second goal was to show that hybrid system behavior is not easy to understand and to foresee. So the simulation of such systems is very good to show and analyze their behaviors.

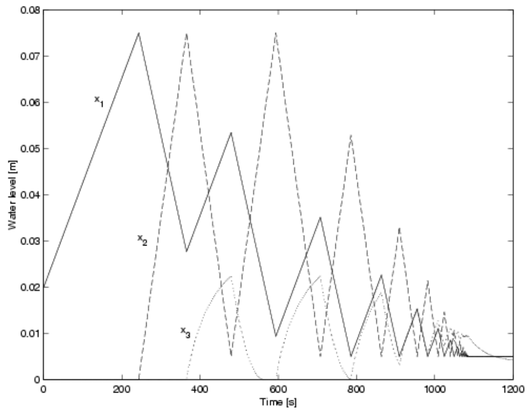


Figure 9.5: Water levels of the system with $q_1 = 1.10^{-6}\, m^3/s$

Conclusion

Report summary: This report has presented hybrid systems which are systems consisting of a mixture of discrete and continuous components, such as a discrete controller that controls a physical environment. It showed how such systems can be mathematically represented, modeled and simulated. Mainly, it dealt with the co-modeling and the co-simulation with SIGNAL and SIMULINK. Several methods have been proposed and examined, namely:

- embed SIMULINK update functions into the SIGNAL generated stand-alone program.
- embed SIGNAL update functions into the SIMULINK generated stand-alone program.

The second method has been kept because the MATLAB stand-alone program easily deals with concurrent components.

In order to ease the building of a hybrid system simulator, a simulator architecture was proposed where the SIMULINK part and the SIGNAL part are well defined. In this architecture the continuous environment is called the *Plant* and the purely discrete component is called the *Selector*. Then three protocols for the selector activation have been proposed, namely:

1. Periodic synchronous selector activations. The selector clock corresponds to the sample rate of the SIMULINK part. Hence the selector is activated for nothing most of the time.
2. Aperiodic synchronous selector activations. It is an improvement of the first one since the selector is activated only if it is necessary.
3. Asynchronous selector activations. The activation clock of the selector is asynchronous with any of the continuous components of the plant.

For each protocol an implementation method was proposed and illustrated with a simple example.

Finally, the non-trivial modeling example of the pump provided by "the father of Swedish engineers", Christofer Polhem, illustrated our modeling method with protocol 2 (i.e., aperiodic synchronous selector activations). Here, we have added a naive controller and shown how co-simulation points out some deficiencies of the controller.

Although this report provides already a continuous environment for SIGNAL controller simulation, this is only a beginning for the hybrid system co-simulation with SIGNAL and SIMULINK. There remains a lot of ways to explore and works to do.

Future work: The example of the pump showed that the modeling of a hybrid system is a hard work. The sources of mistakes are numerous. They can occur:

- in the physical specification,
- in the SIMULINK modeling,
- in the SIGNAL modeling,
- in the interaction between the SIMULINK and SIGNAL parts.

So it should be useful to have some tools to minimize the risk of mistakes. For example such tool could support the building of the SIMULINK and SIGNAL models, as well as the link.

Though an implementation method for protocol 3 exists, this way has not been explored fully. For example, protocol 3 should lead to a real multi-tasking implementation instead of a pseudo multi-tasking. Moreover, it should be interesting to see how the Polhem's pump modeled with this protocol behaves. It is likely that the switching explosion would be avoided if since the switches can not be faster than the selector activation clock. Hence, one can wonder how the selector activation clock affects the whole hybrid system.

Using SIGNAL with SIMULINK to model complex hybrid systems seems a good idea since SIGNAL is very efficient to model complex discrete behaviors and SIMULINK is well-suited to deal with complex continuous dynamics, this is the direction explored in this report. However, the methods presented here do not yet fully exploit the power of SIGNAL; since the selectors were very simple, even in the non-trivial example of the pump. Indeed, when an event is recognized by the selector, the selector runs only one step. Whereas a complex selector would run several discrete steps until a steady state is reached.

In [LMNT99], a *reactive rule-based system* is defined as system that reacts to the changes of its environment continuously. Such system is composed of three entities called *state*, *rules* and *inference engine*. When a stimulus comes from the environment (such a moment is called an *asynchronous computational point* (ACP)) one or more rules are triggered, producing new changes in the states, which in turn trigger other rules, and so on. This is continued until no changes are possible, i.e, a steady state is reached. Then the system starts "resting" in its new *equilibrium period*, awaiting new stimuli. The inference engine is in charge of the computations at the ACPs. So, a possible future work is to allow the simulation of complex hybrid systems by extending the proposed methods to take account of the inference engine defined above.

Appendix A

Using SIMULINK procedure calls

A.1 The C main program of the SIGNAL controller of the thermostat

The `thermostat.c` file is the main file of the thermostat application, it has been written to replace
`thermostat_main.c` which is automatically generated by the program SIGNAL (figure 4.2). The `thermostat.c` file contains four new functions:

- `call_OFF` is used to call the SIMULINK model OFF, actually it calls the function `sim_OFF` provided by compiling the SIMULINK model OFF. Just after `sim_OFF` ends its running, the global variable `stop_OFF` take the value 1, then `call_OFF` finishes.
- `call_ON` is used like `call_OFF` but for the SIMULINK model ON.
- `stop_event_OFF` is used to check if the global variable `stop_OFF` has moved to 1 since the last call of `stop_event_OFF`. If yes, an event `x_eq_inf` is broadcast from the box containing the `stop_event_OFF`.
- `stop_event_ON` is used like `stop_event_OFF` but for the SIMULINK model ON.

This is the `thermostat.c` file:

```
#include "thermostat_types.h"
#include "thermostat_externals.h"
#include "thermostat_body.h"
extern void thermostat_OpenIO();
extern void thermostat_CloseIO();

logical stop_OFF = 1; /* initialisation */
logical stop_ON = 0;
```

```
extern void call_OFF(logical start){
 int i;
 i=sim_OFF(); /* call the simulink model OFF*/
 stop_OFF = 1;
}

logical stop_event_OFF(logical H){
 logical stop=stop_OFF;
 stop_OFF = 0;
 return stop;
}

extern void call_ON(logical start){
 int i;
 i=sim_ON(); /* call the simulink model ON*/
 stop_ON = 1;
}

logical stop_event_ON(logical H){
 logical stop=stop_ON;
 stop_ON = 0;
 return stop;
}

extern int main()
{   int i;
    logical code;
    code = thermostat_initialize();
    /* to run 10 steps of SIGNAL*/
    for (i=1; i<10; i+=1) thermostat_iterate_Black_Box(1);
}
```

Appendix B

Using global variable passing

B.1 The SIGNAL part

To link C code obtained from the SIGNAL model with the C code from the RTW, it is necessary after compiling the model SIGNAL with the compiler to do the following actions.

- inhibit the `<ModelSignal>_iterate()` function from the `<ModelSignal>_body.c` file since it is the `<ModelSignal>_iterate_Black_Box()` which will be use instead.
- inhibit the call to `<ModelSignal>_iterate()` function from the `<ModelSignal>_main.c` file.
- use the command `make -f <Makefile>` in order to build the `<ModelSignal>_body.o` which will be linked latter to the SIMULINK `grt_main.c` file.

B.1.1 The `Thermostat_body.c` file

```
#include "Thermostat_types.h"
#include "Thermostat_externals.h"
static logical H_15_H;
/*  ==> parameters and indexes */
/*  ==> input signals     */
static logical B_tau_OFF, B_tau_ON;
/*  ==> output signals    */
static logical Enable_OFF, Enable_ON;
/*  ==> local signals     */
static logical S, ZS;
static logical XZX_40;
static logical XZX_60, XZX_66, BCLOCK_XZX, XZX_72, XZX_88;

EXTERN logical Thermostat_initialize()
```

```
{
    ZS = FALSE;
    XZX_40 = TRUE;
    H_15_H = TRUE;
    return TRUE;
}

EXTERN void Thermostat_iterate_Black_Box(
   logical _B_tau_OFF_, logical _B_tau_ON_,
   logical *_Enable_OFF_, logical *_Enable_ON_)
{
    B_tau_OFF = _B_tau_OFF_;
    B_tau_ON = _B_tau_ON_;
    XZX_60 = !ZS;
    XZX_66 = B_tau_OFF && XZX_60;
    BCLOCK_XZX = B_tau_ON && ZS;
    XZX_72 = BCLOCK_XZX || XZX_66;
    XZX_88 = !XZX_72;
    if (XZX_66)
        S = TRUE;
    else if (BCLOCK_XZX)
        S = FALSE;
    else
        S = ZS;
    Enable_OFF = !S;
    Enable_ON = S;
    *_Enable_OFF_ = Enable_OFF;
    *_Enable_ON_ = Enable_ON;
    ZS = S;
}
```

B.1.2 The Thermostat_main.c file

```
#include "Thermostat_types.h"
#include "Thermostat_externals.h"
#include "Thermostat_body.h"
extern void Thermostat_OpenIO();
extern void Thermostat_CloseIO();

extern int main()
{
    logical code;
    Thermostat_OpenIO();
    code = Thermostat_initialize();
  /*  while(code)code = Thermostat_iterate();*/
    Thermostat_CloseIO();
}
```

B.2 The grt_main.c file

To implement the hybrid system simulator with SIGNAL and SIMULINK see section 4.2, the SIMULINK main file, i.e., grt_main.c file, can be used as a starting point.

Next, the following elements have to be added at the appropriate places:

- include SIGNAL header files
- include <ModelSimulink>.h to allow the use of the rtU and rtY types
- extern logical <ModelSignal>_initialize();
- extern void <ModelSignal>_iterate_Black_Box(...);
- extern ExternalOutputs rtY; to allow the use of the rtY variable
- extern ExternalInputs rtU; to allow the use of the rtU variable

One needs to initialize the SIGNAL automaton in the main function before the MdlStart() call. The SIGNAL automaton initialization is made in two steps:
<ModelSignal>_initialize() is called in the first step and
the <ModelSignal>_iterate_Black_Box(...) is called with required inputs in the second step.
After that, the use of the <ModelSignal>_iterate_Black_Box(...) function depends on whether automaton activations are periodic, aperiodic or asynchronous.

B.2.1 Periodic synchronous selector activations

This is the main function of the grt_main.c file modified according to the Protocol 1: This file has been renamed therm_prot1_grt_main.c.

```
int_T main(int_T argc, char_T *argv[])
{
    SimStruct  *S;
    const char *status;
    char_T     str2[2];
    real_T     finaltime = 0.0;
    int_T      port      = 17725;
    double     tmpDouble;

    logical code;                                 /*!!*/
    logical enable_OFF;                           /*!!*/
    logical enable_ON;                            /*!!*/
```

```
/*******************
 * Parse arguments *
 *******************/
...

/************************
 * Initialize the model *
 ************************/

rt_InitInfAndNaN(sizeof(real_T));
...
/* Initialize the Thermostat automaton */
code = Thermostat_initialize();             /*!!*/
Thermostat_iterate_Black_Box(1,0,           /*!!*/
                  &enable_OFF,&enable_ON);  /*!!*/
if (enable_OFF == 1 || enable_ON == 1){     /*!!*/
  rtU.root_Enable_OFF = (real_T)enable_OFF; /*!!*/
  rtU.root_Enable_ON = (real_T)enable_ON;   /*!!*/
}                                           /*!!*/

MdlStart();
START(S);
if (ssGetErrorStatus(S) != NULL) {
  GBLbuf.stopExecutionFlag = 1;
}

/*************************************
 * Execute the model.                *
 *************************************/

while (!GBLbuf.stopExecutionFlag &&
       (ssGetTFinal(S) == 0.0 ||
        ssGetTFinal(S)-ssGetT(S) >
              ssGetT(S)*DBL_EPSILON)) {
    rt_OneStep(S);
    if (ssGetStopRequested(S)) break;

    /* thermostat automaton iteration */    /*!!*/
      Thermostat_iterate_Black_Box(         /*!!*/
        (logical)rtY.root_tau_OFF,          /*!!*/
        (logical)rtY.root_tau_ON,           /*!!*/
        &enable_OFF,&enable_ON);            /*!!*/
      if (enable_OFF == 1 ||                /*!!*/
          enable_ON == 1){                  /*!!*/
        rtU.root_Enable_OFF =               /*!!*/
                    (int_T)enable_OFF;      /*!!*/
        rtU.root_Enable_ON =                /*!!*/
                    (int_T)enable_ON;       /*!!*/
```

```
        };                                      /*!!*/
    }

    if (!GBLbuf.stopExecutionFlag &&
        !ssGetStopRequested(S)) {
        /* Execute model last time step */
        rt_OneStep(S);
    }

    /********************
     * Cleanup and exit *
     ********************/
    ...

} /* end main */
```

B.2.2 Aperiodic synchronous selector activations

This is the main function of the grt_main.c file modified according to the Protocol 2. This file has been renamed therm_prot2_grt_main.c and differs from therm_prot2_grt_main.c only in the main loop.

```
int_T main(int_T argc, char_T *argv[])
{
    ...

    while (!GBLbuf.stopExecutionFlag &&
           (ssGetTFinal(S) == 0.0 ||
            ssGetTFinal(S)-ssGetT(S) >
                ssGetT(S)*DBL_EPSILON)) {
        rt_OneStep(S);
        if (ssGetStopRequested(S)) break;

        /* if ``wake_up'' */                        /*!!*/
        if ((logical)rtY.root_tau_OFF == 1          /*!!*/
            || (logical)rtY.root_tau_ON == 1){      /*!!*/
        /* then thermostat automaton iteration */   /*!!*/
          Thermostat_iterate_Black_Box(             /*!!*/
            (logical)rtY.root_tau_OFF,              /*!!*/
            (logical)rtY.root_tau_ON,               /*!!*/
            &enable_OFF,&enable_ON);                /*!!*/
          if (enable_OFF == 1 || enable_ON == 1){   /*!!*/
            rtU.root_Enable_OFF =                   /*!!*/
                          (int_T)enable_OFF;        /*!!*/
            rtU.root_Enable_ON =                    /*!!*/
                          (int_T)enable_ON;         /*!!*/
          };                                        /*!!*/
        }
```

```
    }

    ...

} /* end main */
```

B.2.3 Asynchronous selector activations

This is the main function of the grt_main.c file modified according to the Protocol 3: This file has been renamed therm_prot3_grt_main.c.

```
int_T main(int_T argc, char_T *argv[])
{
    SimStruct  *S;
    const char *status;
    char_T     str2[2];
    real_T     finaltime = 0.0;
    int_T      port      = 17725;
    double     tmpDouble;

    logical code;                                      /*!!*/
    logical enable_OFF;                                /*!!*/
    logical enable_ON;                                 /*!!*/

    /******************
     * Parse arguments *
     ******************/
    ...

    /***********************
     * Initialize the model *
     ***********************/

    rt_InitInfAndNaN(sizeof(real_T));
    ...
   /* Initialize the Thermostat automaton */
    code = Thermostat_initialize();                    /*!!*/
    Thermostat_iterate_Black_Box(1,0,                  /*!!*/
                                 1,/* The H */         /*!!*/
                                 &enable_OFF,          /*!!*/
                                 &enable_ON);          /*!!*/
    if (enable_OFF == 1 || enable_ON == 1){            /*!!*/
      rtU.root_Enable_OFF = (real_T)enable_OFF;        /*!!*/
      rtU.root_Enable_ON = (real_T)enable_ON;          /*!!*/
   }                                                   /*!!*/

    MdlStart();
    START(S);
    if (ssGetErrorStatus(S) != NULL) {
```

```
      GBLbuf.stopExecutionFlag = 1;
    }

    /********************************
     * Execute the model.           *
     ********************************/

    while (!GBLbuf.stopExecutionFlag &&
           (ssGetTFinal(S) == 0.0 ||
            ssGetTFinal(S)-ssGetT(S) >
               ssGetT(S)*DBL_EPSILON)) {
        rt_OneStep(S);
        if (ssGetStopRequested(S)) break;

    /* test if there is a tick for the selector */ /*!!*/
        if ((logical)rtY.root_Tick == 1){          /*!!*/
          /* thermostat automaton iteration */     /*!!*/
          Thermostat_iterate_Black_Box(            /*!!*/
            (logical)rtY.root_e_OFF,               /*!!*/
            (logical)rtY.root_e_ON,                /*!!*/
            1, /* => H = 1 */                      /*!!*/
            &enable_OFF,&enable_ON);               /*!!*/
          if (enable_OFF == 1 || enable_ON == 1){  /*!!*/
            rtU.root_Enable_OFF =                  /*!!*/
                          (int_T)enable_OFF;       /*!!*/
            rtU.root_Enable_ON =                   /*!!*/
                          (int_T)enable_ON;        /*!!*/
          };                                       /*!!*/
          fprintf(stderr,"Selector step ");        /*!!*/
        }
    }

    if (!GBLbuf.stopExecutionFlag &&
        !ssGetStopRequested(S)) {
        /* Execute model last time step */
        rt_OneStep(S);
    }

    /*******************
     * Cleanup and exit *
     *******************/
    ...
} /* end main */
```

Appendix C

Specification of the Polhem's pump

C.1 Mathematical pump model

There are five (5) continuous dynamic state variables:

- v_1 = water volume accumulated in container 1 $[m^3]$
- v_2 = water volume accumulated in container 2 $[m^3]$
- v_3 = water volume accumulated in container 3 $[m^3]$
- q_{12} = volumetric flow from container 1 to 2 $[m^3/s]$
- q_{23} = volumetric flow from container 2 to 3 $[m^3/s]$

In addition there are two (2) discrete dynamic state variables:

- m_1 = mode of check valve in vertical pipe 1-2
- m_2 = mode of check valve in vertical pipe 2-3

where
$m_i \in \{checked, cracked\}, i = 1, 2;$

These are the only "true" states in the sense that the "new" values of the states can be computed based on external inputs and current values of the states.

In order to define properly the external interface we need one or two assumptions. These are our first assumptions (hopefully to be relaxed in near future):

1. The external flow $q_1[m^3/s]$ entering container 1 is determined by the environment (ground water entering the mine cannot be controlled but is defined by mother nature).

2. We have a fast pneumatic pressure controller capable of controlling the pneumatic pressure $p_c[Pa]$ at our will (within reasonable bounds). Among others, the time constant of the pneumatic controller is assumed to be a magnitude shorter than the shortest time constant of the hydraulic system (this assumption is the weakest and most questionable of all assumptions made and I do want to relax it in near future).

3. The check valves are fitted in the lower end of the vertical connecting tubes. The distance between the input port of the check valve and the bottom of the container is assumed to be approximately 0 m (neglected).

4. The water $q_{34} = q_3[m^3/s]$ flowing out of container 3 is assumed to come from a drilled draining hole in the bottom of the container. This is assumingly a good model of what Polhem had in mind.

5. The cracking pressure (opening pressure) and the leakage flow of the check valves are assumed to be negligible.

6. The vertical pipes are assumed to be truly vertical and not bent (parallel to straight lines).

7. All containers have the same physical dimensions (same cross section area and same height).

8. All vertical pipes and check valves are of the same quality, type and size (identical behavior).

9. The fluid levels x of the containers must never exceed the height of the container ($x \geq H$), nor become completely empty ($x \leq 0$); all tubes are assumed to be filled at all times.

These assumptions leads to the following observations:

1. There are only two external *inputs*: q_1 and p_c.

2. Flow q_1 is best characterized as a *disturbance* since we have no influence on it.

3. Pressure p_c is best characterized as our *control signal* since we have full control of it.

4. Since maximized water throughput is the fundamental purpose of the system, $q_3(= q_{34})$ is an appropriate external *output*.

Now we come to the actual computations. These can be derived by sorting the following constitutive relations properly:

$$\dot{v}_1 = q_1 - q_{12}; \quad \text{(C.1)}$$

$$\dot{v}_2 = q_{12} - q_{23}; \quad \text{(C.2)}$$

$$\dot{v}_3 = q_{23} - q_3; \quad \text{(C.3)}$$

```
case m_1
   cracked: q̇_12 = 1/l (p_12 − Δ_p12);
            when ⋈(−q_12) then m_1 := checked;
   checked: q_12 = 0;
            when ⋈(p_12 − Δ_p12) then m_1 := cracked;
end case;
case m_2
   cracked: q̇_23 = 1/l (p_23 − Δ_p23);
            when ⋈(−q_23) then m_2 := checked;
   checked: q_23 = 0;
            when ⋈(p_23 − Δ_p23) then m_2 := cracked;
end case;
```

where

- q_1 = external input flow to container 1 $[m^3/s]$ (external input)
- q_3 = flow to the environment from container 3 $[m^3/s]$ (external output)
- l = hydraulic inductance of vertical pipe $[kg/m^4]$ (parameter)
- p_{12} = net driving pressure of vertical pipe 1-2 $[Pa]$ (derived quantity)
- p_{23} = net driving pressure of vertical pipe 2-3 $[Pa]$ (derived quantity)
- $\Delta_{p_{12}}$ = net pressure drop due to dissipation in vertical pipe 1-2 (derived quantity)
- $\Delta_{p_{23}}$ = net pressure drop due to dissipation in vertical pipe 2-3 (derived quantity)
- $\bowtie(a) = (a \geq 0 \text{ and } \dot{a} > 0)$;

For all pressures introduced here and in the sequel we assume pressures referred to atmospheric pressure $p_0[Pa]$. By doing this "trick" we avoid adding p_0 everywhere.

For the derived quantities we have the following:

$$p_{12} = p_1 - p_2 - p - p_c; \tag{C.4}$$
$$p_{23} = p_2 - p_3 - p + p_c; \tag{C.5}$$

where

- p_c = controlled pneumatic pressure in container 2 $[Pa]$ (external input)
- p_1 = gravity induced hydraulic pressure due to accumulated water in container 1 $[Pa]$ (derived quantity)

- p_2 = gravity induced hydraulic pressure due to accumulated water in container 2 $[Pa]$ (derived quantity)
- p_3 = gravity induced hydraulic pressure due to accumulated water in container 3 $[Pa]$ (derived quantity)
- p = gravity induced hydraulic pressure due to water in vertical pipe (1-2, 2-3) $[Pa]$ (derived quantity)

For the newly introduced derived quantities we have:

$$\begin{aligned} p_1 &= \rho g\, x_1; & \text{(C.6)} \\ p_2 &= \rho g\, x_2; & \text{(C.7)} \\ p_3 &= \rho g\, x_3; & \text{(C.8)} \\ p &= \rho g\, L; & \text{(C.9)} \end{aligned}$$

where

- ρ = density of water $[kg/m^3]$ (parameter)
- g = acceleration of gravity $[m/s^2]$ (parameter)
- x_1 = water level depth of container 1 $[m]$ (derived quantity)
- x_2 = water level depth of container 2 $[m]$ (derived quantity)
- x_3 = water level depth of container 3 $[m]$ (derived quantity)
- L = length of vertical pipe (1-2 or 2-3) $[m]$ (parameter)

Again we can derive the new quantities:

$$\begin{aligned} x_1 &= \max(\min(v_1/A, H), 0); & \text{(C.10)} \\ x_3 &= \max(\min(v_2/A, H), 0); & \text{(C.11)} \\ x_2 &= \max(\min(v_3/A, H), 0); & \text{(C.12)} \end{aligned}$$

where

- A = cross section area of the containers $[m^2]$ (parameter)
- H = height of containers $[m]$ (parameter)
- $\min(a, b) = a$ if $a < b$ else b;
- $\max(a, b) = a$ if $a > b$ else b;

Now we only have two remaining quantities to deal with:

$$\begin{aligned} \Delta_{p_{12}} &= r^2\, q_{12}\, |q_{12}|; & \text{(C.13)} \\ \Delta_{p_{23}} &= r^2\, q_{23}\, |q_{23}|; & \text{(C.14)} \end{aligned}$$

where

- r = sum of hydraulic resistance in vertical pipe (1-2 or 2-3) $[\sqrt{N}s/m^4]$ (parameter)
- $|a| = a$ if $a > 0$ else $-a$;

As for the external output we finally have:

$$q_3 = \frac{1}{r_3} \sqrt[sgn]{p_3}; \tag{C.15}$$

where

- r_3 = hydraulic resistance in draining hole in container 3 $[\sqrt{N}s/m^4]$ (parameter)
- $\sqrt[sgn]{a} = \sqrt{a}$ if $a > 0$ else $-\sqrt{-a}$;

C.2 Parameter Values

And now finally for some real numbers. All physical dimensions to follow are based on a check valve Jan-Erik Strömberg has found some data for, namely the following:

Check valve data:

Manufacturer:	The LEE Company
Type code:	TKLA9501130D (125/156 MINSTAC)
Cracking pressure:	$1.0\,kPa$ (i.e. approximately negligible)
Leakage flow:	$1.0 \cdot 10^{-8}\,m^3/min$ @ $7.0\,kPa$ (definitely negligible)
Hydraulic resistance:	$1.97 \cdot 10^7\,\sqrt{N}s/m^4$

Pipe data ($1m$ long):

Manufacturer:	The LEE Company
Type code:	
Inner diameter:	$2.4\,mm$ ($0.095in$)
Outer diameter:	$3.96\,mm$ ($0.156in$)
Hydraulic resistance:	$7.12 \cdot 10^7\,\sqrt{N}s/m^4$

From these parameters and an assumed pneumatic control pressure

$$p_c \in \{p_-, p_+\}; \tag{C.16}$$

where

$$p_+ = 50\,kPa; \tag{C.17}$$
$$p_- = -50\,kPa; \tag{C.18}$$

a number of reasonable geometric container dimensions can be determined.

The approximate peak flow between the containers (q_{12} etc.) now become something in the order of $180\,ml/min$. Let us assume cylindrical containers. Then we have

$$A = \pi D^2/4; \tag{C.19}$$

where

- D = container inner diameter $[m]$ (parameter)

Reasonable dimensions of the containers may then be

$$D = 75\,mm; \tag{C.20}$$
$$H = 100\,mm; \tag{C.21}$$

The maximal volume stored is

$$V = 4.42 \cdot 10^{-4}\, m^3 = 442\, ml; \tag{C.22}$$

Let us assume a vertical distance between containers of 1 m. We then have

$$L = 1\, m; \tag{C.23}$$

We are now able to compute a set of derived parameters:

$$l = \frac{4L\rho}{\pi d_i^2} \tag{C.24}$$

where

- d_i = pipe inner diameter $[m]$ (parameter)

Hence

$$\begin{aligned} d_i &= 2.4\, mm; && \text{(C.25)} \\ l &= 2.21 \cdot 10^8\, kg/m^4; && \text{(C.26)} \\ p &= 10\, kPa; && \text{(C.27)} \end{aligned}$$

For the vertical pipes we have

$$r = \sqrt{r_v^2 + r_p^2}; \tag{C.28}$$

where

- r_v = check valve hydraulic resistance $[\sqrt{N}s/m^4]$ (parameter)
- r_p = pipe hydraulic resistance $[\sqrt{N}s/m^4]$ (parameter)

Hence

$$\begin{aligned} r_v &= 1.97 \cdot 10^7\, \sqrt{N}s/m^4; && \text{(C.29)} \\ r_p &= 7.12 \cdot 10^7\, \sqrt{N}s/m^4; && \text{(C.30)} \\ r &= 7.39 \cdot 10^7\, \sqrt{N}s/m^4; && \text{(C.31)} \end{aligned}$$

Further we may assume a draining hole in container 3 of 2.5 mm diameter. We then have

$$r_3 = 6.32 \cdot 10^6\, \sqrt{N}s/m^4; \tag{C.32}$$

Since the approximate flow between containers is $180\, ml/min$ an input flow can be assume such as

$$q_1 \in [50\, ml/min, 300\, ml/min] = [8.3 \cdot 10^{-7}\, m^3/s, 5.0 \cdot 10^{-6}\, m^3/s]; \tag{C.33}$$

For the purpose of experimentation one may for instance assume that q_1 is constant. But try other inputs for fun to see how the overall behavior changes. It is likely that it will have significant influence depending on the control strategy.

As for the global parameters the following approximations can be used:

$$\rho = 1 \cdot 10^3 \, kg/m^3; \tag{C.34}$$

$$g = 10 \, m/s^2; \tag{C.35}$$

C.3 Control strategy

And now for the pump controller.

Again, there is no obvious control law. Feel free to play around with different versions. The goal of the system is quite obvious as well as the qualitative behavior. One extremely simple, though inadequately robust (in fact, no robustness at all) control law is to oscillate the pressure p_c between $+50\,kPa$ and $-50\,kPa$ on fixed time schedule. Try for instance a simple square wave of $0.01\,Hz$ and 50 duty cycle. I.e. let $p_c = +50\,kPa$ for 50 seconds and then switch to $p_c = -50\,kPa$ for 50 seconds. Repeat the cycle and see what happens. Especially if you increase the input flow q_1.

An important assumption made in the model is that no container must ever be completely empty ($x <= 0$), nor be overfilled ($x >= H$). This is not really a restriction, since there are more practical reasons for maintaining x_i in the interval $[a, b]$ for all containers i and all time points t. For secure performance, the parameters a and b should be something like

$$a = 0.05\,H; \tag{C.36}$$

$$b = 0.75\,H; \text{(at least container 2)} \tag{C.37}$$

These requirements should provide an interesting verification task as well as an interesting control synthesis task. Especially if we assume that we have no sensors for x_i for all containers i. What do we do then?

For the time being, it is assumed that the pump has expensive level sensors for x_i. In this half-realistic case we may introduce a control law like the following:

```
case m_c
   idle: p_c := 0;
         when ⋈ (x_1 - b) then m_c := pull;
   pull: p_c := -50 kPa;
         when ⋈ (a - x_1) or ⋈ (x_2 - b) then m_c := push;
   push: p_c := +50 kPa;
         when ⋈ (a - x_2) or ⋈ (x_3 - b) then m_c := pull;
end case;
```

Bibliography

[ACH$^+$95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 6 February 1995.

[AGMR95] T. P. Amagbegnon, P. Le Guernic, Herve Marchand, and Eric Rutten. The SIGNAL data flow methodology applied to a production cell. Technical Report RR-2522, Inria, Institut National de Recherche en Informatique et en Automatique, March 1995.

[BBM97] P. Bournai, M. Le Borgne, and H. Marchand. Environnement de conception d'automatismes discrets basé sur le langage Signal. Technical Report 1124, Inria, Institut National de Recherche en Informatique et en Automatique, September 1997. In French.

[BC85] D. Berry and L. Cosserat. The Esterel synchronous programming language and its mathematical semantics. *Lecture Notes in Computer Science*, 197:389–448, February 1985.

[Ber89] G. Berry. Real time programming: Special purpose or genral purpose languages. In *IFIP World Computer Congress*, San Francisco, 1989.

[BG97] P. Bournai and P. Le Guernic. L'éditeur graphique de SIGNAL-V4. Technical report, Inria, Institut National de Recherche en Informatique et en Automatique, June 1997. In French.

[Bro97] J. F. Broenink. Bond-graph modeling in MODELICA. In *Proceedings of the 9th European Simulation Symposium*, pages 137–141, Passau, Germany, October 1997.

[BS91] F. Boussinot and R. De Simone. The ESTEREL language. *Proc. IEEE*, 79(9):1293–1304, September 1991.

[BS97] S. Bornot and J. Sifakis. Relating time progress and deadlines in hybrid systems. *Lecture Notes in Computer Science*, 1201:286–303, 1997.

[DGS98] Akash Deshpande, Aleks Göllü, and Luigi Semenzato. The SHIFT programming language for dynamic networks of hybrid automata. *IEEE transactions on automatic control*, 43(4), April 1998.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. *Lecture Notes in Computer Science*, 1066:208–220, 1996.

[Eds99] Krister Edström. *Switched Bond Graphs Simulation and Analysis*. PhD thesis, Linköping Studies in Science and Technology. Thesis No 586, June 1999.

[EOS97] H. Elmqvist, M. Otter, and C. Schlegel. Physical modeling with MODELICA and DYMOLA and real-time simulation with simulink and real time workshop. In *Proceedings of MATLAB conferences in San Jose, October 6-8 and Stockholm, October 27-28*, 1997.

[GBBG85] P. Le Guernic, A. Beneviste, P. Bournal, and T. Ganthier. SIGNAL: a data flow oriented language for signal processing. Report 246, IRISA, Rennes, France, 1985.

[Hal93] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.

[Har87] D Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HHWT97] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: A model checker for hybrid systems. *Lecture Notes in Computer Science*, 1254:460–472, 1997.

[HN96] D. Harel and A. Naamad. The STATEMATE semantics of STATECHARTS. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[Hou98] Bernard Houssais. Cours de programmation en langage temps-réel SIGNAL. Irisa, Institut de Recherche en Informatique et en Systèmes Aléatoires, in French, November 1998.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO, ASI Series*, pages 447–498. Springer-Verlag, New York, 1985.

[KP91] Y. Kesten and A. Pnueli. Timed and hybrid statecharts and their textual representation. In Jan Vyptopil, editor, *Proceedings of Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *LNCS*, pages 591–620, Berlin, Germany, January 1991. Springer-Verlag.

[LG94] Lennart Ljung and Torkel Glad. *Modeling of Dynamic Systems.* Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1994.

[LGG+96] Lennart Ljung, Roger Germundsson, Johan Gunnarsson, Inger Klein, Jonas Plantin, and Jan-Erik Strömberg. Hybrid and discrete systems in automatic control - some new (linköping) approaches. Technical Report LiTH-ISY-R-1843, Dept of EE. Linköping University, S-581 83 Linköping, Sweden, May 1996.

[LGS94] J. Lygeros, D. N. Godbole, and S. Sastry. Simulation as a tool for hybrid control. In *Proceedings of the Fith IEEE conference on AI, Simulation and Planning in High-Autonomy Systems*, Gainesville, Florida, USA, December 1994.

[Lin51] Sten Lindroth. *Christoffer Polhem och Stora Kopparberget.* Stora Kopparberget Bergslags AB, Uppsala, Sweden, 1951. In Swedish with abstract in German.

[LMNT99] M. Lin, J. Malec, and S. Nadjm-Tehrani. On semantics and correctness of reactive rule-based systems. In *Proceedings of Andrei Ershov Third International Conference "Perspectives of System Informatics"*, Novosibrisk (Russia), July 1999. Springer-Verlag, LNCS.

[LMRS96] M. Le Borgne, H. Marchand, E. Rutten, and M. Samaan. Formal verification of SIGNAL programs: Application to a power transformer station controller. In *Proceedings of AMAST'96, LNCS 1101*, pages 271–285, Munich, Germany, July 1996. Springer-Verlag.

[Mar90] F. Maraninchi. ARGOS*, a Graphical Language for the Design, Description and Validation of Reactive Systems.* Université J. Fourier, Grenoble, France, 1990. in French.

[Mat97a] The MathWorks, Inc. *Real-Time Workshop User's Guide*, May 1997.

[Mat97b] The MathWorks, Inc. *Stateflow User's Guide*, May 1997.

[Mat97c] The MathWorks, Inc. *Target Language Compiler Reference Guide*, May 1997.

[Mat97d] The MathWorks, Inc. *Using Simulink*, January 1997.

[Mat98] The MathWorks, Inc. *Using Matlab*, January 1998.

[MBLL98] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. A design environment for discrete-event controllers based on the SIGNAL language. In *1998 IEEE International Conf. On Systems, Man, And Cybernetics*, pages 770–775, San Diego, California, USA, October 1998.

[MH96] F. Maraninchi and N. Halbwachs. Compiling ARGOS into boolean equations. In *Formal Techniques for Real-Time and Fault-Tolerance (FTRTFT)*, Uppsala (Sweden), September 1996. Springer-Verlag, LNCS.

[ML98] H. Marchand and M. Le Borgne. Partial order control of discrete event systems modeled as polynomial dynamical systems. In *1998 IEEE International Conference On Control Applications*, Trieste, Italia, September 1998.

[MS98] Z. Manna and H. B. Sipma. Deductive verification of hybrid systems using STeP. *Lecture Notes in Computer Science*, 1386:305–319, 1998.

[NS98] R. Nikoukhah and S. Steer. SCICOS - *A dynamic system builder and simulator user's guide*, 1998. http://www-rocq.inria.fr/scilab/doc/scicos/scicos.html.

[NSY91] X. Nicollin, J. Sifakis, and S. Yovine. From ATP to timed graphs and hybrid systems. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Proceedings of Real-Time: Theory in Practice*, volume 600 of *LNCS*, pages 549–572, Berlin, Germany, June 1991. Springer-Verlag.

[OEM99] M. Otter, H. Elmqvist, and S.E. Mattson. Hybrid modeling in MODELICA based on the synchronous data flow principle. In *Proceedings of the IEEE International Symposium on Computer-Aided Control System Design, CACSD'99*, Hawaii, USA, August 1999.

[Pay61] Henry M. Paynter. *Analysis and Design of Engineering Systems.* MIT Press, Cambridge, Mass, 1961.

[SNT94] J.-E. Strömberg and S. Nadjm-Tehrani. On discrete and hybrid representation of hybrid systems. In *Proceedings of the SCS International Conference on Modeling and Simulation (ESM'94)*, pages 1085–1089, Barcelona, Spain, 1994.

[SNTT96] J.-E. Strömberg, S. Nadjm-Tehrani, and J. Top. Switched Bond Graphs as Front-end to Formal Verification of Hybrid Systems. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Proceedings of the third international conference on hybrid systems, Hybrid Systems III, LNCS 1106*, pages 282–293, NJ, USA, 1996. Springer Verlag.

[SST94] Jan-Erik Strömberg, Ulf Söderman, and Jan Top. Conceptual modelling of hybrid systems. Technical Report LiTH-ISY-R-1625, Dept of EE. Linköping University, S-581 83 Linköping. Sweden, 1994.

[Str94] J.-E. Strömberg. *A mode switching modelling philosophy.* PhD thesis, Linköping University, Linköping, 1994. Dissertation no. 353.

List of Figures