Co-simulation of hybrid systems: Signal-Simulink

Stéphane Tudoret[1], Simin Nadjm-Tehrani[1*], Albert Benveniste[2], Jan-Erik Strömberg[3]

[1] Dept. of Computer & Information Science, Linköping University, S-581 83 Linköping, Sweden, e-mail: simin@ida.liu.se
[2] IRISA-INRIA, Campus de Beaulieu, Rennes, France
[3] DST Control AB, Mjärdevi Science Park, Linköping, Sweden

Abstract. This article 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. In particular, progress in formal analysis of SIGNAL programs and the common availability of the SIMULINK tool makes these an interesting choice for combination. We present various alternatives for implementing communication between the underlying submodels. Finally, we present interesting scenarios in the co-simulation of a discrete controller with its environment: a non-linear siphon pump originally designed by the Swedish engineer Christofer Polhem in 1697.

1 Introduction

The use of software and embedded electronics in many control applications leads to higher demands on analysis of system properties due to added complexity. Simple controller blocks in MATLAB are increasingly replaced by large programs with discrete mode changes realising non-linear, hierarchical control and supervision. The analysis of these design structures benefits from modelling environments using languages with formal semantics – for example, finite state machines (e.g. STATECHARTS [11], ESTEREL [5]), or clocked data flows (e.g. LUSTRE [9], SIGNAL [8]).

These (discrete-time) languages and associated tools provide support in programming the controller in many ways. To begin with, they provide an architectural view of the program in terms of hierarchical state machines or block diagrams. In recent years, certain modelling environments for continuous systems have also been augmented with versions inspired by these languages, e.g. MATLAB STATEFLOW [20] and MATRIXX [12] discrete-time superblocks.

* This work was supported by the Esprit LTR research project SYRF. The second author was also supported by the Swedish research council for engineering sciences (TFR).

In addition, formal semantics for the underlying languages allows the controller design to be formally analysed. Constructive semantics in ESTEREL and clock calculi in LUSTRE and SIGNAL, enable formal analysis directly at compilation stage [4]. Properties otherwise checked by formal verification at later stages of development [6], e.g. causal consistency or determinism, are checked much earlier. Also, results of these analyses are used at later stages of development – in particular, for automatic code generation (code optimisation) and code distribution [2, 3, 7, 13]. Note that these types of formal analysis of a discrete controller are so far not supported in the traditional modelling environments (e.g. MATLAB and MATRIXX).

However, properties at the system level still have to be addressed by the analysis of the closed loop system. Formal verification of hybrid models is generating new techniques for this purpose. Restrictions on the class of differential and algebraic equations (DAE) for the plant or approximations on the model to get decidability are active areas of research [10, 26, 14].

In this paper we explore another direction aimed at applications where the DAE plant model is directly used for controller testing within the engineering design process. That is, we study the question of co-simulation. Formal verification can be a complement to, or make use of the knowledge obtained by integrated simulation environments. In this set-up the plant is specified as a set of DAE and the controller specified in a high level design language. The controller is subjected to formal verification supported by the discrete modelling tools, and the closed loop system is analysed by co-simulation. To this end, we propose a framework in which SIGNAL programs and MATLAB-SIMULINK [22] models can be co-simulated using automatically generated C-code. We present the application of the framework to a non-trivial example suggested earlier [27, 28].

2 Introduction to SIGNAL

SIGNAL is a data-flow style synchronous language specially suited for signal processing and control applications [1, 16, 18]. A SIGNAL program 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** characterised 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, the SIGNAL language provides the operation: *synchro* X, Y. The absence of a signal is noted $\perp$.

2.1 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.

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, with initial value $V0$:

$$ZX := X \ \$1 \ init \ V0$$

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.

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

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.

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

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 written:

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

Each equation from SIGNAL is like an elementary process. 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 denoted: $(|\ P1 \mid P2\ |)$.

2.2 Tools

All the different tools which make up the SIGNAL environment use only one tree-like representation of programs, thus we can go from one tool to another without using 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 logic specifications, the verification tool SIGALI.

The most interesting tool from a formal verification point of view is the SIGALI tool supporting the formal calculus. It contains a verification and controller synthesis tool-box [17, 15], and facilitates proving correctness of the dynamical behaviour of a system with respect to a temporal logic specification.

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 behaviour. 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 characterising 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.

3 Introduction to SIMULINK

SIMULINK is the part of the MATLAB toolbox for modelling, simulating, and analysing dynamical systems. It provides several solvers for the simulation of numeric integration of sets of Ordinary Differential Equations (ODEs). 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. For code generation, however, currently it is not possible to use variable-step solvers to build the stand-alone program. Thus, we had to use the fixed step size solvers, and therefore, the step size needs to be set accurately.

SIMULINK Real-Time Workshop (RTW) [19] is the setting for automatic C code generation from SIMULINK block diagrams via a Target Language Compiler (TLC) [21]. By default[1], the RTW gives mainly four C files : `<Model>.c`, `<Model>.h`, `<Model>.prm` and `<Model>.reg`. The function of these files in stand-alone simulation is fully described in [29]. Figure 1 summarises the architecture of the stand-alone code generation 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).

By default, the run of a stand-alone program provides a MATLAB data file (`<Model>.mat`). Before building of the stand-alone program, it is possible to select which data we want to include in the MATLAB file. Then, one can use MATLAB to plot the result.

[1] It is possible to customise the C code generated from any SIMULINK model with the TLC which is a tool that is included in RTW.

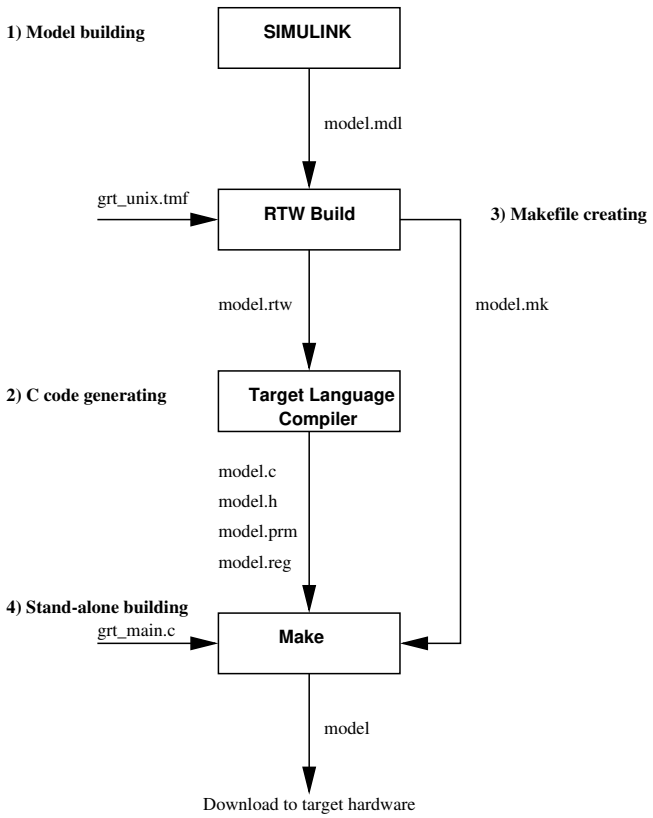

Fig. 1. Automatic code-generation within the Real-Time Workshop architecture

4 Modelling multi-mode hybrid systems

SIGNAL and SIMULINK have both a data-flow oriented style. Here we present a mathematical framework in which both SIGNAL and SIMULINK sub-models can be plugged in to form a hybrid system.
Hybrid systems can be mathematically represented as follows:

$$\dot{x}_i = f_i(q, x_i, u_i, d_i)\ ,\ x_i \in \mathbb{R}^{n_i}\ ,\ q \in Q \tag{1}$$
$$y_i = h_i(q, x_i, u_i) \tag{2}$$
$$e_i = s_i(q, x_i, u_i, y_i) \tag{3}$$
$$\tau_i = e.\mathbf{1}_{\{e_i \neq e_{i_}\}} \tag{4}$$
$$q' = T(q, \tau)\ ,\ \tau = (\tau_i, i = 1, \ldots, I) \tag{5}$$

Where:

(1): $i = 1, \ldots, I$ indexes a collection of continuous time subsystems (CTS),
$q \in Q$ is the discrete state, where Q is a finite alphabet,
$x_i \in \mathbb{R}^{n_i}$ is the vector continuous state of the ith CTS,
$u_i \in \mathbb{R}^{m_i}$ is the vector continuous control of the ith CTS,
$d_i \in \mathbb{R}^{o_i}$ is the vector continuous disturbance of the ith CTS.
(2): $y_i \in \mathbb{R}^{p_i}$ is the vector continuous output of the ith CTS,
(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 j 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 $g(q, ., ., .) : \mathbb{R}^{n_i+m_i+p_i} \longmapsto \mathbb{R}$, and so on.
(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 $e_{i_}^k(t) \neq e_i^k(t)$ means that the kth predicate changes its status at instant t; this generates an event τ_i^k. The marked events τ_i^k together form a vector event τ_i (and the latter form the vector event τ). Thus trajectories e_i are piecewise constant.
(5): q, q' are the current and next discrete automaton state.

We use an architectural decomposition earlier used for several case studies [25]. Here we use it to discuss the way the communication between the two sub-models can be implemented for co-simulation.

In the generic architecture shown in Figure 2, 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, including A/D converters. The *Selector (S)* is the purely discrete part of the controller – with discrete, input, state and output. The *Effector (E)* is the interface between the discrete selector commands and the continuous physical variables including actuators.

This architecture is a good starting point for hybrid system modelling. It remains to decide:

- How to map the mathematical representation above on the architecture?
- Which parts should be modelled in SIGNAL and which parts in SIMULINK?
- How the SIGNAL part should be activated? Which mechanism should be used including A/D convertors.

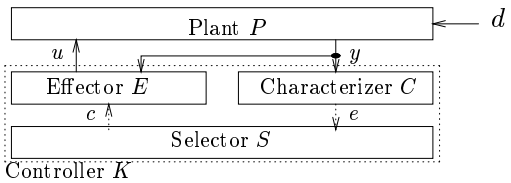

Fig. 2. General hybrid system architecture. Solid (dotted) arrows represent continuous (discrete) flows

From our introductory remarks it should be fairly obvious that selector modelling is best done in SIGNAL, and that SIMULINK is best for modelling the plant. Thus, it remains to determine how to implement the interface between the two, or rather, where and how to model the characterizer and the effector. Next, we need to determine how to generate runs of the hybrid system.

In this paper we adopt the scheme whereby the main module of the SIMULINK model is the master and the SIGNAL automaton is one of the many processes run in a pseudo-parallel fashion. This is realisable using the translation scheme in RTW. The SIMULINK model then contains input ports allowing SIMULINK subsystem blocks to be enabled and disabled, and output ports allowing subsystems to emit events to the controller. The connection can now be made by means of global variable passing.

5 Computational model with global variable passing

The mathematical model in section 4 is a natural way to conceptualise and model a multi-mode hybrid system. To implement such a system we have to transform these equations into a computational model. In this section we cast the generic mathematical model into the architectural framework presented earlier. In section 6 we provide three protocols for activation of the SIGNAL part of the model.

The plant is made of a collection of finite continuous time subsystems. As in the mathematical representation of section 4, let I be the cardinality of the collection and let i index over I. Each subsystem i contains a vector $x_i \in \mathbb{R}^{n_i}$ of n_i continuous state and also n_i differential equations.This set of equations can be rewritten as follows:

$$\begin{pmatrix} \dot{x}_i^1 \\ \vdots \\ \dot{x}_i^{n_i} \end{pmatrix} = \begin{pmatrix} f_i^1(q, x_i^1, u_i, d_i) \\ \vdots \\ f_i^{n_i}(q, x_i^{n_i}, u_i, d_i) \end{pmatrix} \tag{6}$$

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

At any time t, one or several equations among this collection forms the basis for computation. Consider the whole set of system equations as follows:

$$\begin{aligned} F_1(x_1^1, u_1, d_1) &= f_1^1(q_1, x_1^1, u_1, d_1) \\ F_2(x_1^1, u_1, d_1) &= f_1^1(q_2, x_1^1, u_1, d_1) \\ &\vdots \\ F_{|Q|}(x_1^1, u_1, d_1) &= f_1^1(q_{|Q|}, x_1^1, u_1, d_1) \\ F_{|Q|+1}(x_1^2, u_1, d_1) &= f_1^2(q_1, x_1^2, u_1, d_1) \\ &\vdots \end{aligned} \tag{7}$$

Let j be a new index for indexing the system equations. Then, we can define a new function F_j for the jth equation in the above list. Now we can rewrite each differential equation as follows:

$$\dot{x}_j = F_j(x_j, u_j, d_j) \tag{8}$$

which allows to calculate the vector continuous state x and the vector continuous output y thanks to equation $y = h(x, u)$. Then y feeds the characterizer, and the equation $e = s(y)$ defines the detection of event e.

Figure 3 shows one possible mapping of the mathematical representation into the architecture (later, we will see that this is not the only mapping). In comparison with Figure 2, a new component has been added in the controller, it is the *Edge detector* which corresponds to equation (4). The discrete state q is defined only in the selector which is the only purely discrete part. So, the selector contains the rewritten form of equation (5):

$$q' = T(q, \tau) \quad \tau = (\tau_j, j = 1, \ldots, J) \tag{9}$$

and the new equation below:

$$c = g(q') \tag{10}$$

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

$$(u_j, enabl_j) = k(c_j) \tag{11}$$

$enabl_j$ 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 equation.

Since the discrete controller (the automaton) is in one state at any one computation point[2], 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.

[2] This is a property of the data-flow program ensured by formal analysis built-in in the compilers for synchronous languages.

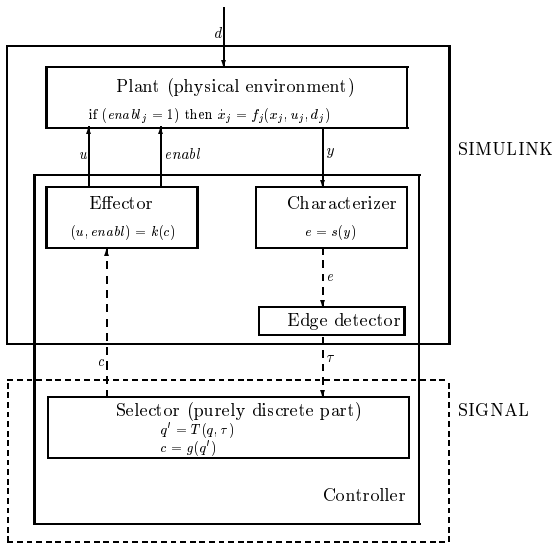

Fig. 3. Hybrid system representation

6 Selector activations

The selector, i.e. the union of equations (9) and (10) 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 for checking the event τ_j by the selector. These possibilities depend on how the selector is activated. Here we discuss three activation methods – i.e., periodic, aperiodic and asynchronous selector activations.

6.1 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 (9).*

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.

6.2 Aperiodic synchronous selector activations

Protocol 2 *Here the continuous time system (equations (8)) is the master, driven by continuous real time t. Each time some τ_j occurs a "*wake_up*" event is generated by the j^{th} continuous time system in which τ_j was generated. Then selector (equation (9)) 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 (9).*

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.

6.3 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 own 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 (9) and finally, it outputs the state changes to the effector following equation (10).*

It is important to note that with Protocol 3 the τ generation should be done in the SIGNAL part instead of the SIMULINK part (compare with Figure 3). 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 recognised with a delay of one tick in the selector.

7 Application: the siphon pump

The protocols for aperiodic and asynchronous selector activations have been implemented in our co-simulation environment [29]. In this section we give a brief exposition to application of the aperiodic protocol to a non-trivial example earlier introduced in [27, 28]. This is a model of a siphon pump machine invented by the Swedish engineer Christofer Polhem in 1697. The purpose of the pump was to drain water from the Swedish copper mines with almost no movable parts. This works by having a system of interconnected open and closed tanks, and driving the water up to the ground level by adjusting the pressure in the closed tanks via shunt valves. The idea of the pump was so revolutionary in those times that the pump was never built. However, a model of the pump going back to the 17th century is the basis of the dimensions (and therefore the coefficients in the model) that we have used in our down-scaled model. Figure 4 shows a fragment of the pump consisting of the bottom three tanks.

The plant model has several interesting characteristics. First, even without the discrete controller, there are some discrete dynamic changes in the plant. These are brought about by the two check valves (hydro-mechanically) controlling the flow of water between each open and closed container. Secondly, the plant dynamics (and also the closed loop dynamics) is non-linear. When the check valve between container i and container $i+1$ is cracked, the flow of water in that pipe, denoted by $q_{i(i+1)}$, is defined by $\dot{q}_{i(i+1)} = f(p_{i(i+1)}, q_{i(i+1)})$ where f is a non-linear function, and $p_{i(i+1)}$ is the pressure in the pipe between container i and container $i+1$.

For closed-loop simulation we thus had to make an appropriate decomposition, placing the purely discrete parts (including switching in the plant) in the SIGNAL environment, and the purely continuous parts in the SIMULINK environment.

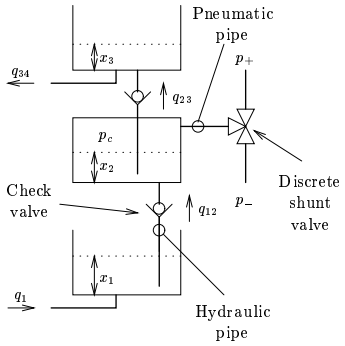

Fig. 4. A fraction of the siphon pump machine

7.1 Working principles

The purpose of the pump is to lift the water which flows 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 principle works for an arbitrary system of alternative closed and open tanks as follows.

The pull phase In the pull phase, the pressure vessels (the closed tanks) are de-pressurised by opening the p^- side of the shunt valve which drains the vessels (the p^- side is connected to a negative pressure source e.g. a vacuum tank). Now, the water will be lifted from all the open containers to the pressure vessels immediately above. Hence, as a result of this first phase, all the pressure vessels will be water-filled.

The push phase In the push phase, the pressure vessels are pressurised by opening the p^+ side of the shunt valve to fill the air-compressing vessel with air (the p^+ side represents a positive pressure source, e.g. created by an elevated lake above the mine). Now, all the pressure vessels will be emptied via the connections to the open containers immediately 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 upwards half a section. By repeating these two phases the water is sequentially lifted to the ground level.

Figure 4 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 (in between the open and closed 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). In the sequel we assume that there are only three levels to the pump and the final flow variable $q_3 = q_{34}$.

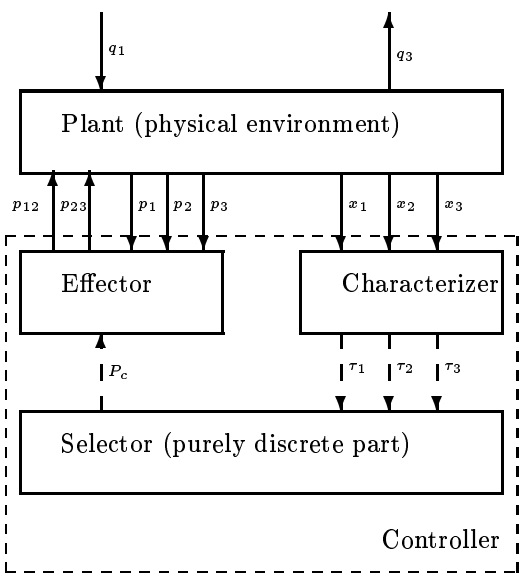

Fig. 5. General hybrid system architecture of the pump

7.2 Mathematical models

From the high level description of the pump, it is possible to represent the simulated system by means of the architecture presented earlier. Thus, the system decomposition can be depicted as in Figure 5.

At the topmost block, 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.

The closed loop system is modelled with the plant supplying control information to the effector, the characterizer and eventually to the selector. Obviously, the selector acts on the pneumatic pressure in container 2, i.e, increasing and decreasing P_c. Then the effector provides from P_c and from the gravity induced hydraulic pressure due to accumulated water in containers (p_1, p_2 and p_3) 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 levels of the containers (x_1, x_2 and x_3) and sends event τ to the selector when it is necessary.

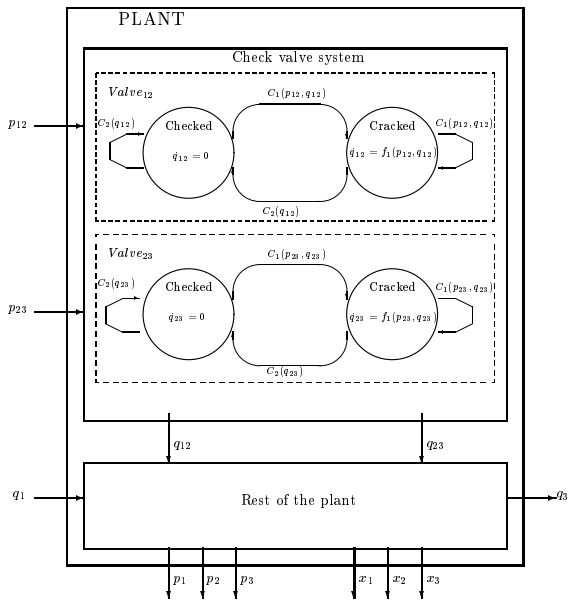

Fig. 6. The architecture of the plant

The refined model of the plant is depicted in Figure 6. 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 zero and in the cracked mode the water flow follows a non-linear differential equation (the interested reader is referred to the full report [29] for details of the plant model). Note that the

check valve can be modelled using both SIMULINK and SIGNAL: The discrete mode changes are modelled in SIGNAL and the rest in SIMULINK.

7.3 The control strategy

Finding a safe and optimal controller is far from easy. One of the more important requirements is to maximise the output flow q_3 without risking that x_i will end up outside defined safe intervals. That is, to avoid overflow in the containers (and the mine), specially under all possible disturbances (q_1).

Another important requirement is related to energy consumption and maintainability. 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.

A naive controller can be depicted by the automaton of Figure 7. This is not a robust controller and it was chosen to show the power of the co-simulation environment in illustrating its weaknesses.

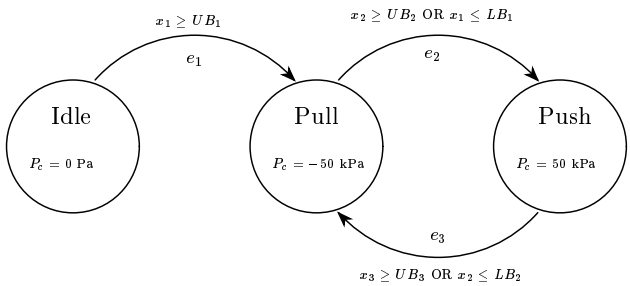

Fig. 7. Automaton implementing the control strategy in a selector, UB_i and LB_i are the level upper and lower limits in tank i respectively.

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

1. The first discrete state, i.e., the **Idle** state, is the initialisation state. At the beginning, the three 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 de-pressurised. Hence container 2 fills from 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 pressurised. 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. Container 2 is of course 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 shows which events lead to discrete state transitions of the selector and how these events are detected. Hence it is easy to model a characterizer which watches the different water levels and provides the suitable events.

8 Analysis results and future works

In this section we present some co-simulation results. We study the behaviour of the closed loop system for given disturbance signals (incoming water into the bottom container) in presence of the naive controller. It is illustrated that while certain aspects of the behaviour are as expected, we also get unsatisfactory outputs.

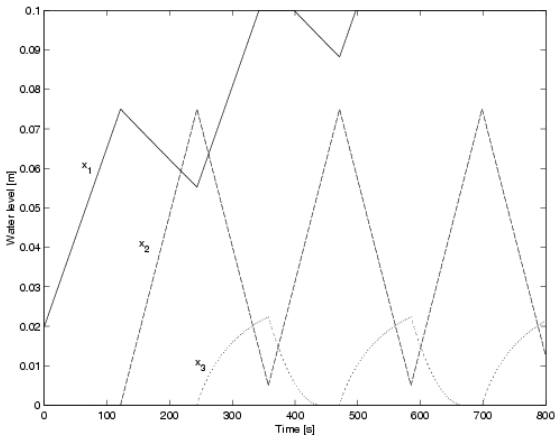

Fig. 8. Water levels of the system with $q_1 = 2.10^{-6}\, m^3/s$

First, observing the behaviour of the flow in the different pipes appears satisfactory. However, that in itself is not sufficient for correctness of the pump behaviour. Indeed, it is necessary to study the water levels in each container to check whether there is an overflow. Figure 8 shows such traces. The water level of the ith container is denoted by x_i and H denotes the height of the containers. 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. 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.

The next plot shows that even if there is no overflow, the controller has a bad behaviour. That is, an infinitely fast switching behaviour in the shunt valve controller appears. This undesired behaviour of the system is a direct result of the naive control strategy adopted, not due to the chosen communication protocol. This lack of robustness in the controller is well-illustrated by the co-simulation, see Figure 9.

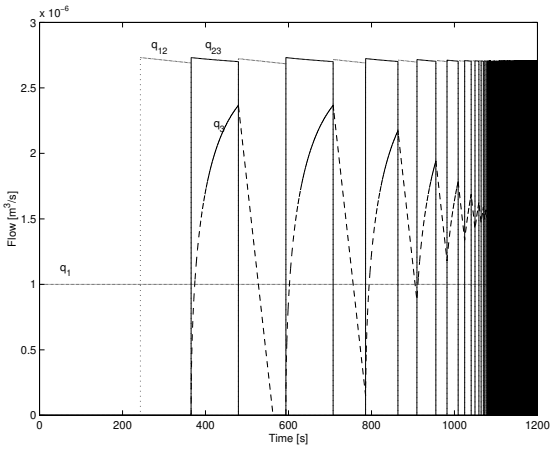

Fig. 9. The simulation result illustrating infinite switching.

Current work includes experiments using the asynchronous protocol. Another interesting problem is to study the range of values for q_1, for which the pump can work without problems; in particular, how simulation and formal verification can be combined to analyse such problems. Also, it is interesting to apply the combined environment to systems with more complex controller structure [24], where formal verification in SIGALI and co-simulation in the current environment are combined.

A survey of related works on simulation of hybrid systems can be found in [23]. A typical requirement in dealing with hybrid simulation is that systems

with uneven dynamics be simulated with variable step solvers so that rapid simulation and accuracy can be combined. Our work points out a weakness in the code generation mechanism of MATLAB which restricts the ability to use variable solvers. On the other hand, this may not be a problem in some application areas. For example, it was not considered as a critical issue when this work was presented at a forum including our industrial partners from the aerospace sector.

References

1. T. Amagbegnon, P. Le Guernic, H. Marchand, and E. Rutten. Signal- the specification of a generic, verified production cell controller. In C. Lewerentz and T. Lindner, editors, *Formal Development of Reactive Systems - Case Study Production Cell*, number 891 in Lecture Notes in Computer Science, chapter 7, pages 115–129. Springer Verlag, January 1995.
2. A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in Dataflow Synchronous Languages: Specification and Distributed Code Generation. *Information and Computation*. To appear.
3. A. Benveniste, B. Caillaud, and P. Le Guernic. From Synchrony to Asynchrony. In J.C.M. Baeten and S. Mauw, editors, *Proceedings of the 10th International Conference on Concurrency Theory, CONCUR'99, LNCS 1664*, pages 162–177. Springer Verlag, 1999.
4. G. Berry. The Constructive Semantics of Pure Esterel. Technical report, Centre de Mathematiques Appliquees, 1999. Draft book, available from `http://www-sop.inria.fr/meije/esterel/doc/main-papers.html`.
5. F. Boussinot and R. De Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.
6. W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J.D. Reese. Model Checking Large Software Specifications. *IEEE Transactions on Software Engineering*, 24:498–519, July 1998.
7. T. Gautier and P. Le Guernic. Code generation in the SACRES project. In F. Redmill and T. Andersson, editors, *Towards System Safety, Proceedings of the Safety-critical Systems Symposium, SSS'99*, pages 127–149, Huntingdon, UK, February 1999. Springer Verlag.
8. P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.
9. 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.
10. N. Halbwachs, P. Raymond, and Y.-E. Proy. Verification of Linear Hybrid Systems by means of Convex Approximations. In *In proceedings of the International Symposium on Static Analysis SAS'94, LNCS 864*. Springer Verlag, September 1993.
11. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
12. Integrated Systems Inc. *SystemBuild v 5.0 User's Guide*. Santa Clara, CA, USA, 1997.

13. A. Kountouris and C. Wolinski. Hierarchical conditional dependency graphs for mutual exclusiveness identification. In *12th International Conference on VLSI Design*, Goa, India, January 1999.
14. G. Lafferriere, G. J. Pappas, and S. Yovine. A New Class of Decidable Hybrid Systems. In *proceedings of Hybrid Systems: Computation and Control, LNCS 1569*, pages 137–151. Springer Verlag, March 1999.
15. 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.
16. E. Marchand, E. Rutten, and F. Chaumette. From data-flow task to multi-tasking: Applying the synchronous approach to active vision in robotics. *IEEE Trans. on Control Systems Technology*, 5(2):200–216, March 1997.
17. 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.
18. H. Marchand and M. Samaan. On the incremental design of a power transformer station controller using controller synthesis methodology. In *World Congress on Formal Methods (FM'99)*, volume 1709 of *LNCS*, pages 1605–1624, Toulouse, France, September 1999. Springer Verlag.
19. The MathWorks, Inc. *Real-Time Workshop User's Guide*, May 1997.
20. The MathWorks, Inc. *Stateflow User's Guide*, May 1997.
21. The MathWorks, Inc. *Target Language Compiler Reference Guide*, May 1997.
22. The MathWorks, Inc. *Using Simulink*, January 1997.
23. P. Mosterman. An Overview of Hybrid Simulation Phenomena and Their Support by Simulation Packages. In *Hybrid Systems: Computation and Control, Proceedings of the second international workshop, March 1999, LNCS 1569*, pages 168–177. Springer Verlag, March 1999.
24. S. Nadjm-Tehrani and O. Åkerlund. Combining Theorem Proving and Continuous Models in Synchronous Design. In *Proceedings of the World Congress on Formal Methods, Volume II, LNCS 1709*, pages 1384–1399. Springer Verlag, September 1999.
25. S. Nadjm-Tehrani and J-E. Strömberg. Verification of Dynamic Properties in an Aerospace application. *Formal Methods in System Design*, 14(2):135–169, March 1999.
26. A. Puri and P. Varaiya. Verificaion of Hybrid Systems Using Abstrations. In *proceedings of Hybrid Systems II, LNCS 999*, pages 359–369. Springer Verlag, 1994.
27. J.-E. Strömberg. *A mode switching modelling philosophy.* PhD thesis, Linköping University, Linköping, 1994. Dissertation no. 353.
28. 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.
29. S. Tudoret. Signal-simulink: Hybrid system co-simulation. Technical Report cis-1999-020, Dept. of Computer and Information Science, Linköpings University, December 1999. Currently available under Technical reports from `http://www.ida.liu.se/~eslab/publications.shtml`.