Annex 7.3.B
Co-simulation of hybrid systems: Signal-Simulink (Extended Abstract)

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
tel: +46-13-282411, fax: +46-13-282666
[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. 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-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 and hierarchical control and supervision.

The need for analysis of these complex design structures has lead to development of several modelling environments based on languages with formal semantics – for example, finite state machines (e.g. STATECHARTS [12], ESTEREL [6]), or clocked data flows (e.g. LUSTRE [10], SIGNAL [9]).

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

* Corresponding author

diagrams. In recent years, certain modelling environments for continuous systems have also been augmented with versions inspired by these languages, e.g. MATLAB STATEFLOW [21] and MATRIXX [13] discrete-time superblocks.

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 [7], 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, 8, 14]. Note that these types of formal analysis are so far not supported in the continuous modelling environments.

However, properties at the system level still have to be addressed by the analysis of the closed loop system, including some mathematical representation of the plant. Formal verification of hybrid models is generating new techniques for this purpose. Restrictions on the class of DAE equations for the plant or approximations on the model to get decidability are active areas of research [11, 25, 15].

In this paper we explore another direction aimed at applications where the DAE plant model is directly used for controller design and test within the engineering design process. In cases where the plant model is typically too complex to rule out formal verification approaches, we study the question of co-simulation. Here the plant is specified as a set of DAE and a controller specified in a high level design language. More specifically, we propose a framework in which SIGNAL programs and MATLAB-SIMULINK [23] models can be co-simulated using automatically generated C-code. We report on a number of issues considered when developing such a co-simulation environment and present the application of the framework to a non-trivial example earlier suggested and modelled using Bond Graphs [26, 27].

2 Introduction to SIGNAL

SIGNAL is a data-flow style synchronous language specially suited for signal processing and control applications [1, 17, 19]. 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** 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: $synchroX, Y$. The absence of a signal is noted $\perp$.

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. In the full paper we describe the kernel language functions in more detail.

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: $(| \; P1 \mid P2 \; |)$.

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 SIGNAL-V4 graphical interface:
  A SIGNAL program lends itself well to a graphical block-diagram oriented user interface. Thus, a graphical editor [5] is available in the SIGNAL environment. This allows to build SIGNAL programs mixing texts and graphics. The editor supports a hierarchical conception and modular implementation of algorithms.
- SIGALI: a formal calculus software:
  The SIGNAL environment contains a verification and controller synthesis tool-box, named SIGALI[18, 16]. This tool facilitates proving correctness of the dynamical behavior of the system with respect to a temporal logic specification.
- Code generation:
  To generate a stand-alone program with SIGNAL four steps are needed: specify a model, generate C code, generate makefile, and generate stand-alone program. Once a <Model>.sig file is specified it can be compiled into three C components. To get an executable file, a makefile is automatically created using the command BuildMakefile <Model>. Finally the stand alone program is produced using the make command and the makefile: make -f <MakefileName>

3 Introduction to SIMULINK

SIMULINK is the part of the MATLAB toolbox for modeling, simulating, and analysing dynamical systems. It supports linear and nonlinear systems, modeled in continuous time, sampled time, or a hybrid of the two. In the full paper we include an introduction to SIMULINK elementary building blocks.

Once the model is built, it is typically analysed by simulation. But before doing that it is necessary to fix some parameters, especially for the solver options. SIMULINK provides several solvers for the simulation of numeric integration of sets of Ordinary Differential Equations (ODEs). 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. Hence, the step size needs to be set accurately.

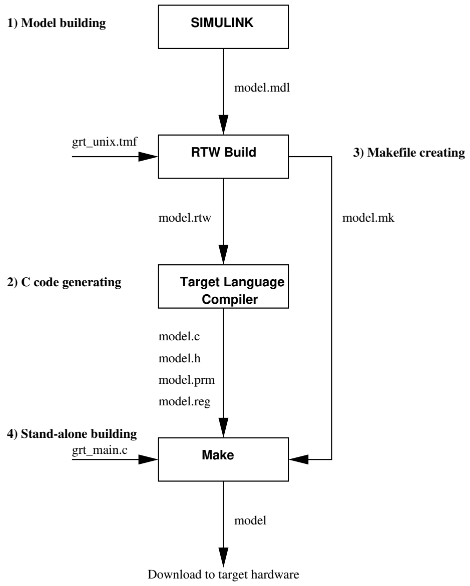


Fig. 1. Automatic code-generation within the Real-Time Workshop architecture

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. Figure 1 summarizes the architecture of the stand-alone code generation with SIMULINK.

SIMULINK Real-Time Workshop (RTW) [20] is the setting for automatic C code generation from SIMULINK block diagrams via a Target Language Compiler (TLC) [22]. 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 [28].

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). As for stand alone code compilation, just as in the case of SIGNAL , 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.

By default, the run of 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 customize the C code generated from any SIMULINK model with the TLC which is a tool that is included in RTW.

4 Our framework for co-simulation

Here we present the mathematical framework in which both SIGNAL and SIMULINK sub-models can be plugged in to form a hybrid system. We use an architectural decomposition earlier used for several case studies [24]. Here we use the architecture to discuss the way the communication between the two sub-models can be implemented for co-simulation.

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 of 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.

An implementation framework for hybrid system can be described by the generic architecture shown in Figure 2. The *Plant (P)* is the physical environment under control. The *Characterizer (C)* is the interface between the continuous plant and the discrete selector, including A/D convertors. 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 modeling. 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?

– Which mechanism should be used for activation of different sub-models?

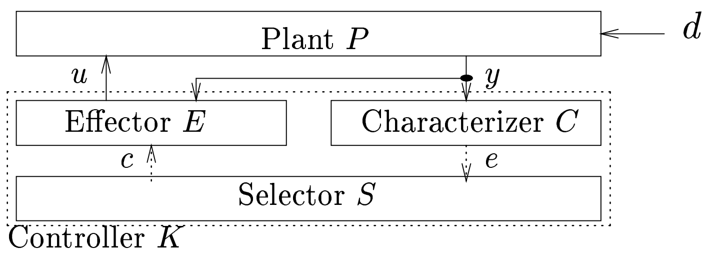


Fig. 2. General hybrid system architecture. Solid (dotted) arrows represent continuous (discrete) flows

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. 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. In the full paper the implementation details will be explained in more detail.

5 Generic model with global variable passing

In this section we provide three protocols for activation of the SIGNAL part of the model. But first, we cast the generic mathematical representation in section 4 into the architectural framework presented earlier.

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:

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

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, 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.

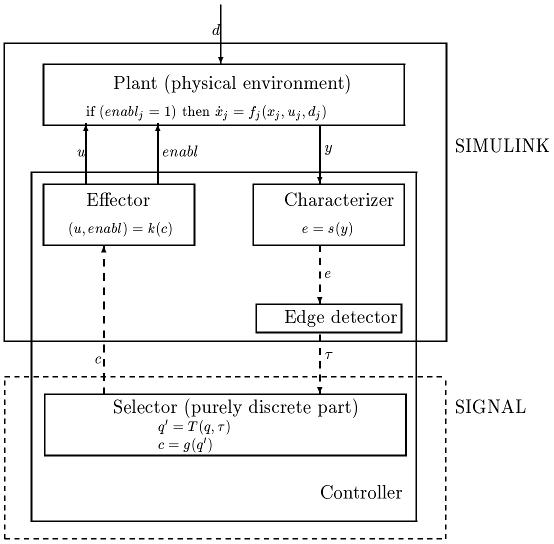


Fig. 3. Hybrid system representation

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 of checking the event τ_j for 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

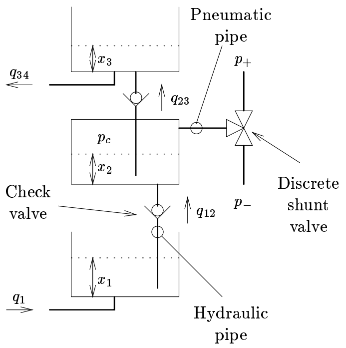
Here, continuous subsystems and the selector have independent SIMULINK threads, that means above all the selector has its own thread and its own activation clock.

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).*

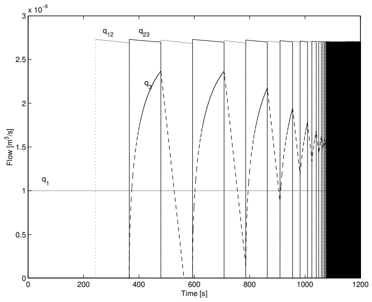
It is important to note that with Protocol 3 the τ generation should be done in the SIGNAL part instead of the SIMULINK part (compare with Figure 3). Indeed, if the τ is provided by SIMULINK, there is a risk that the selector will miss some τ because no assumption can be made about when the selector will check its input channels. In the best case some τ are recognized 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 [28]. In this section we give a brief exposition to application of the aperiodic protocol to a non-trivial example earlier introduced in [26, 27]. 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. Figure 4(a) shows a fragment of the pump consisting of the bottom three tanks.



(a) A fraction of the siphon pump machine copied from [26]



(b) The co-simulation depicting infinitely fast switching

Fig. 4. A schematic diagram of the pump and one simulation result

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-mecanically) 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 the pipe $q_{i(i+1)}$ 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 (even in the plant) in the SIGNAL environment, and the purely continuous parts in the SIMULINK environment. In the full paper we provide the pump and controller models in detail, the exact decomposition in SIGNAL and SIMULINK, and 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 a naive controller. It is illustrated that while certain aspects of the behaviour are as expected, we also get unsatisfactory outputs. First, we get an overflow in the lowest container after a certain point in time, and secondly 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 co-simulation (Figure 4(b)). Current work includes experiments using the asynchronous protocol.

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. 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.
6. F. Boussinot and R. De Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.
7. 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.
8. 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.
9. 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.
10. 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.
11. 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.
12. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
13. Integrated Systems Inc. *SystemBuild v 5.0 User's Guide*. Santa Clara, CA, USA, 1997.
14. A. Kountouris and C. Wolinski. Hierarchical conditional dependency graphs for mutual exclusiveness identification. In *12th International Conference on VLSI Design*, Goa, India, January 1999.
15. 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.

16. 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.
17. 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.
18. 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.
19. 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.
20. The MathWorks, Inc. *Real-Time Workshop User's Guide*, May 1997.
21. The MathWorks, Inc. *Stateflow User's Guide*, May 1997.
22. The MathWorks, Inc. *Target Language Compiler Reference Guide*, May 1997.
23. The MathWorks, Inc. *Using Simulink*, January 1997.
24. 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.
25. 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.
26. J.-E. Strömberg. *A mode switching modelling philosophy*. PhD thesis, Linköping University, Linköping, 1994. Dissertation no. 353.
27. 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.
28. S. Tudoret. Signal-simulink: Hybrid system co-simulation. Technical Report http://www.ep.liu.se/ea/cis/1999/020, Linköpings University E-press, 1999. To appear as electronic publication.