Pontus Boström | Lionel Morel

Mode-Automata in Simulink/Stateflow

Turku Centre *for* Computer Science

TUCS Technical Report
No 772, September 2006

Mode-Automata in Simulink/Stateflow

Pontus Boström
Åbo Akademi University, Department of Information Technologies
Turku Centre for Computer Science (TUCS)
Lemminkäisenkatu 14 A, 20520 Turku, Finland
`pontus.bostrom@abo.fi`
Lionel Morel
IRISA - Campus universitaire de Beaulieu
35042 Rennes Cedex, France
`lionel.morel@gmail.com`

TUCS Technical Report
No 772, September 2006

Abstract

This paper presents an application of the mode-automata based design method to Stateflow/Simulink. The observation we make is two fold. First, we realized that mode-automata, being one of the most convincing proposition made recently to separate control from signal processing, is only starting to be applied to industrial tools. Second, although the separation of control and signal processing is somehow effective in Stateflow/Simulink, the lack of formal definition does not lead to a valuable interpretation.

The goal of the work presented in this paper is to make these two approaches converge. We introduce a formalisation of Stateflow/Simulink where the language has been reduced so that to fit the mode-automata approach and thus restrict the expressive power of Stateflow in a way still suitable to real-life application. We then illustrate the approach with a small application in digital hydraulics controller development.

Keywords: Control Systems, Formalisation, System Architecture, Model-Based Design

TUCS Laboratory
Distributed Systems Design Laboratory

1 Introduction

The design of computerized embedded control systems has become an important activity in the last decades. As complexity has increased, the need for clearer methodologies and paradigms has become greater. Correctness of control systems can be improved first by means of formal techniques introduced in the design flow, but also by proposing modelling/programming methodologies that will make the design flow clearer in itself. All these have as a strong common goal to take care of the growing complexity of the applications.

One way to tackle complexity in embedded control systems design is to separate the expression of control and computations. During the last 15 years or so, we have seen an emergence of different paradigms allowing to separate these aspects. The idea underlying all these paradigms is to express control using hierarchical state-machines and computation with block diagrams, connecting different subsystem of a systems in a dataflow manner.

1.1 Related Works

On the academic ground, several works have emerged that try to find the best compromise between expressiveness and complexity.

One of the most significant is the mode-automata of Maraninchi and Rémond [13, 14] as implemented in the Matou tool [15]. Activity of the system is separated in different *running modes* that are described as states of a possibly hierarchical state machine. The behaviour of the system in each of these modes is described by a set of dataflow equations (e.g. using the syntax of the Lustre language [2]). This notion of mode is actually significant in that it corresponds exactly to what end-users have in mind when asking for a clear separation of control and signal processing. However this approach has lacked for a long time a successful transfer to industrial tools.

Another approach comparable to mode-automata is Modecharts [9, 17]. The biggest advantage of mode-automata compared to Modecharts was that it allowed the expression of both control and signal-processing in the same language. Our approach can be found to have the same inconvenience as Modecharts, since the state-structure of the mode-automata is described in Stateflow and the signal processing part in Simulink. But this decoupling is now a de facto method in industry and should not be seen as such a drawback. Moreover, the graphical syntax we adopted makes the approach very similar to mode-automata.

Among the industrial tools dedicated to the design of control systems, there are two successful examples of separation of control and signal processing. The first one is the introduction of hierarchical state-machines to the SCADE environment [1]. In this paper, we join recent work [3] on the introduction of state-machine structures in SCADE (which is very similar to Simulink). We are also very close to [10] in the methodology we propose. However, this latter does not concentrate on a formal semantics as we do. The second one is the coupling of Stateflow and Simulink[2], present in the Matlab toolset. However, the semantics behind the language is somewhat unclear, based graphical assumption (like the famous 12 o'clock rule of Stateflow). Several works [4, 5, 22] have recently proposed formal semantics for a subset of Stateflow/Simulink. The semantics we give here is very close to those, but we try to keep in mind the mode-automata architecture, which leads to some restrictions on the part of Stateflow/Simulink we need and thus simplifies the semantics.

[1]http://www.esterel-technologies.com/products/scade-suite/overview.html

[2]Trademarks of the *MathWorks* company, http://www.mathworks.com

1.2 Propositions

The work presented in this paper goes more into the "methodology" direction. We aim at 1) the definition of a sound subset of Stateflow/Simulink sufficient for allowing the construction of mode-automata-like structures; 2) an actual proposition of mode-automata in Stateflow/Simulink.

The semantics of Stateflow/Simulink that we give is based on different other works (notably [19]) and mainly brings technical restrictions that lead to an unambiguous semantics for mode-automata in Stateflow/Simulink. The implementation of mode-automata that we propose is trying to adapt the approach proposed in [14], the most naturally possible to Stateflow/Simulink. We do not claim that this is the only way to do this, but that following these propositions, developers will follow the formal semantics proposed.

1.3 Structure of the Paper

Section 2 gives basic notation references that will be used in the rest of the paper. Section 3 presents our formal definition of mode-automata in Stateflow/Simulink, while section 4 gives the behavioural semantics of systems built using our methodology. The formal definition of mode-automata can be used for static analysis. The behavioural semantics can then be used for runtime analysis and simulation. We try to concentrate on restrictions we put on Stateflow/Simulink in order to enforce an easy-to-understand yet expressive language for describing mode-automata. Section 5 gives some practical hints on how to build controllers in Stateflow/Simulink using that mode-automata approach. In section 6, we show the application of our methodology to the design of a simple controller case study in digital hydraulics. Section 7 first briefly sketches a possible verification methodology. Then section 8 also presents other research directions that we wish to address in the future and summarizes the contributions of that paper.

2 Notations

The formalisation is based on the use of functions and relations to describe the structure of the models. The notation we use is influenced by the formalisation of Statecharts in [21]. We use the notation $f : A \rightarrow B$ to denote a function f from A to B. Function application is given by $f(a)$ and $f(a) = b$ if and only if $(a, b) \in f$. The Cartesian product of two sets is denoted by $A \times B$. A relation between elements of two sets A and B is given as $r \in A \times B$. The domain of r is denoted by $dom(r)$ and the range by $ran(r)$. We can define the image $r[A_1]$ of r for $A_1 \subseteq A$. The image is defined as $r[A_1] = \{b | b \in B \wedge (\exists a.a \in A_1 \wedge (a, b) \in r)\}$. Furthermore, we denote the transitive closure of the relation r with r^+ and the reflexive transitive closure with r^*. The cardinality of a set A is denoted by $|A|$.

3 Mode-Automata in Stateflow/Simulink

Control systems are often hybrid systems consisting of both discrete and continuous parts. *Matlab* and *Simulink* developed by Mathworks Inc., have become popular tools for modelling, analysing and designing such systems. Simulink is a graphical language where different functional blocks are connected by signals. A large library of functional blocks is included for modelling both continuous, discrete and hybrid systems. Models can be numerically simulated using one of several numerical differential equation solvers included in Simulink.

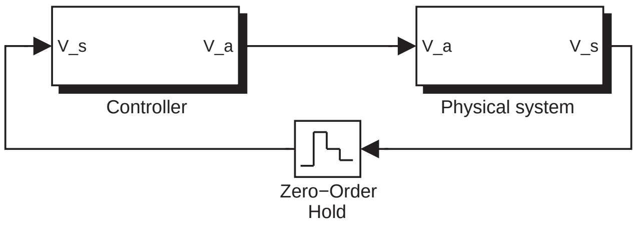

Figure 1: Overview of control system modelled in Simulink

Discrete systems are often conveniently modelled using *finite state-machines*. Simulink contains *Stateflow*, which is a graphical language for creating hierarchical state-machines similar to Statecharts by Harel [6]. The main difference between Statecharts and Stateflow is that Stateflow is completely sequential and deterministic. In order to implement the designed systems, both Simulink and Stateflow allow direct code generation from the models.

A Simulink model of a control system consists of two parts. A controller implemented on a computer and a model of the physical system, as shown in Figure 1. The controller and physical system communicate via sensors and actuators. A Simulink model $\mathcal{M}$ can then be given as a tuple $\mathcal{M} = (\mathcal{C}, \mathcal{P}, V_s, V_a)$ where:

- $\mathcal{C}$ is the part of the Simulink model that implements the controller, which is here constructed as a mode-automata.
- $\mathcal{P}$ is the part of the Simulink model that models the physical system
- V_s is the set of sensors
- V_a is the set of actuators

We first concentrate on describing the mode-automata architecture of the controller. This involves a formal description of the features in the Simulink/Stateflow model, as well as, restrictions to Simulink/Stateflow required by the mode-automata architecture. Finally, we describe how to consider the continuous model of the physical system in this framework.

3.1 The Controller

The controller is an implementation of mode-automata in Simulink. The Simulink/ Stateflow language is a very convenient tool for system construction due to the large set of features. However, some of these features have complicated or counter-intuitive semantics. We like to restrict the language to a safe kernel that is expressive enough to be conveniently used in practice, while the models are still easy to understand and (formally) analyse. Furthermore, we would like to provide an architecture that simplifies the construction of systems consisting of both discrete control logic and signal processing. Restricting Simulink/Stateflow to the mode-automata architecture seems to be a good solution for satisfying these properties. Stateflow is there used to describe the transitions between modes. Simulink block diagrams are used to define the behaviour in each mode and the definition of guards on the transitions driving the state- machine in the Stateflow model.

3.1.1 Definitions

We first define the Simulink block diagram modelling the controller with inputs and outputs and then consider the Stateflow model. The input to the Stateflow model from the Simulink model is a set of guard values. Which behaviour should be executed in the Simulink model is then decided by the active modes in the Stateflow model.

Simulink Controller The controller is defined as a tuple $\mathcal{C} = (V_i, V_o, X, i, M, f_x, \mathcal{S}, G, \phi)$.

- V_i is the input variables (source ports) and V_o is the output variables (output ports). The set of input and output variables are disjoint, $V_i \cap V_o = \emptyset$. The sensors V_s in the Simulink model are a subset of the input variables, $V_s \subseteq V_i$, and the actuators V_a are a subset of the output variables, $V_a \subseteq V_o$.
- X is the state-space of the controller.
- $i : X \rightarrow \Sigma$ is the initialization of the state variables, where Σ denotes the initial values.
- $M \subseteq V_o \rightarrow F(V_i, X)$ is a set of subsystems defining mode dependent behaviour. Each subsystem assigns an update function to a subset of output variables. The update functions are implemented as block diagrams in Simulink. Furthermore, due to implementation issues explained in Section 5 behaviour in modes cannot modify state variables.
- $f_x : X \rightarrow F(V_i, X, V_o)$ is a function that assign an update function to each state variable $x_i \in X$.
- $\mathcal{S}$ is a Stateflow model.
- G is the set of names for the guards input to the Stateflow model.
- $\phi : G \rightarrow \Phi(V_i, X)$ associates every guard name with a condition.

Most of the computation is defined using Simulink and, hence we do not need to take into account the entire Stateflow specification in our formalisation. Especially, we try to use as little as possible of the Stateflow action language. The action language makes it possible to write program statements on transitions and actions that are executed upon entry, exit, as well as, inside states. The reason for using a minimal set of features of the Stateflow action language is that we then can analyse certain properties of the computation in Simulink and Stateflow by analysing the graph formed by the diagrams, e.g., behavioural equality of two models. Simulink block diagrams are also often easier to understand and easier to use correctly than many of the features in the action language of Stateflow.

We consider hierarchical state-machines containing both or-states and and-states in Stateflow, but we do not consider activities inside states. Transitions in Stateflow can be labelled by events, guards, and actions and they can contain junctions. Here we only consider transitions labelled by guards. Each guard only consists of a guard name, since we can then use the properties of the Stateflow graphs for analysis. If boolean operators were allowed in the guards we would need a prover to e.g. decide equality of guards. Note, that if Φ_i is a guard for a transition we also often have transitions with the guard $\neg\Phi_i$. Therefore, if Φ_i is associated with the name g we we also allow the guard name $!g$ denoting $\neg\Phi_i$. This does not complicate analysis, and it is therefore allowed for convenience.

The Stateflow language is deterministic. If two transition guards are enabled at the same time the priority of the transtions decides which transition is executed. Transitions with a source higher in the state hierarchy have higher priority. The priority of two transitions with the same source state is either determined by the internal rules of Stateflow or explicitely given. For clarity, we assume that explicit ordering of transitions is used in the Stateflow models, since the internal rules relies on the graphical layout of the diagram and are not considered safe.

Stateflow Model The Stateflow model $\mathcal{S}$ is here given as a tuple $\mathcal{S} = (Q, Q_{and}, Q_{or}, root, p, i, b, T, \rho)$ where:

- Q is the set of states (modes) in the Stateflow model
- $Q_{and} \subseteq Q$ is the and-states. $Q_{or} \subseteq Q$ is the or-states. We have that $Q_{and} \cap Q_{or} = \emptyset$.
- $root \in Q_{or}$ is the root state
- $p : Q \to Q$ is a function that maps a state to its parent.
- $i : (Q_{or} \cup Q_{and}) \to Q$ is the initial states in the or-states and and-states.
- $b : (Q - (Q_{and} \cup Q_{or})) \to M$ is a function used to define the subsystem that describe the behavior in each state. Only leaf states can have behaviors.
- $T \in Q \times G \times Q$ is the set of transitions, labelled by guard names G. The value of the condition associated with the guard name determines when a transition is enabled.
- $\rho : T \to \mathbb{N}$ is a total function that orders the transitions. Transitions with lower numbers have higher priority.

3.1.2 Properties and Constraints

We model the structure of Stateflow models using sets and relations. To illustrate this consider figure 2. The figure to the left shows a Stateflow model as drawn in Simulink and the figure to the right shows the formalisation of the state-hierarchy. The root state of the diagram is an or-state with one sub-state q_1. This state is an and-state that has two sub-states, q_2 and q_3, which then have two sub-states each. The function $p = \{(q_1, root), (q_2, q_1), (q_3, q_1), \ldots\}$ relates the states to their parent state, while the relation $i = \{(root, q_1), (q_1, q_2), (q_1, q_3), \ldots\}$ gives the initialisation for each composite state. The transitions are not shown in the figure describing the formalisation.

In order to form a valid Stateflow model the state hierarchy has to satisfy certain properties. The set of valid transitions is also limited by additional constraints. To simplify these rules we define a child-relation $c \mathrel{\hat{=}} p^{-1}$. The first set of restrictions concern the nesting of states. These constraints are automatically ensured by the Stateflow notation. Every sub-state of an and-state is an or-state, every composite state has more than zero sub-states and the parent of every state is a composite state:

$$\begin{array}{l} \forall q . q \in Q_{and} \Rightarrow c(q) \subseteq Q_{or} \wedge c[\{q\}] \neq \emptyset \\ \forall q . q \in Q_{or} \Rightarrow c[\{q\}] \neq \emptyset \\ \forall q . q \in Q \Rightarrow p(q) \in Q_{or} \cup Q_{and} \end{array}$$

The graph formed by the parent relation is connected and acyclic, which is also ensured by Stateflow:

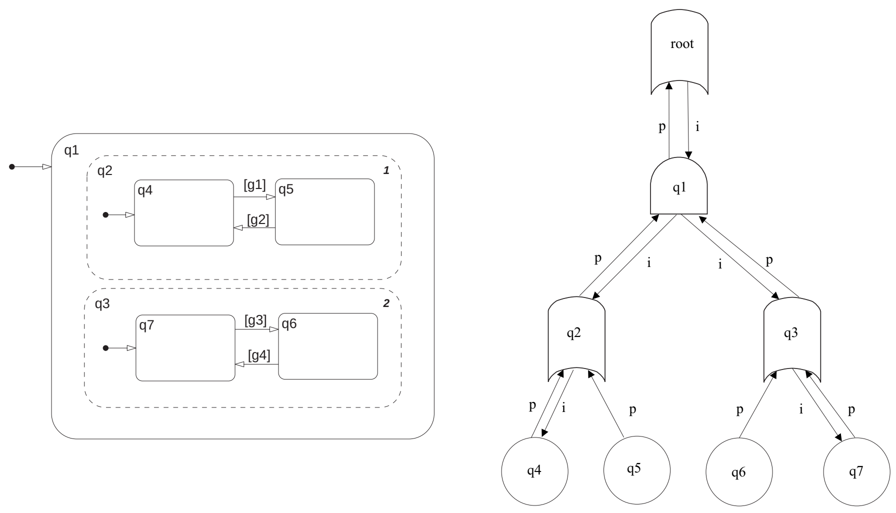

Figure 2: An example of Stateflow model and a formal description of its state-hierarchy.

$$Q = c^*[\{root\}]$$
$$\forall q.q \in Q \Rightarrow q \notin c^+[\{q\}]$$

Only one child state can be the initial state (destination of a default transition) of an or-state and all child states are initial states of an and-state. This requirement is stronger than for Stateflow, since Stateflow allows also no initial state in or-states :

$$\forall q.q \in Q_{or} \Rightarrow i[\{q\}] \subseteq c[\{q\}] \wedge |i[\{q\}]| = 1$$
$$\forall q.q \in Q_{and} \Rightarrow i[\{q\}] = c[\{q\}]$$

Each leaf state is associated with an update function $m_i \in M$ in the Simulink block diagram. All variables V_o have to be updated regardless of the states (modes) the system is in, otherwise the value of some output variables would be undefined in certain modes. However, each variable should be updated by only one function at the time. This rule is not ensured by Simulink and it needs to be verified by other means. Let $V(m)$ be the set of variables modified by the set of update functions $m \subseteq M$. We have that $V(m) \hat{=} \{v | v \subseteq V_o \wedge \exists m_1.m_1 \in m \wedge v \in dom(m_1)\}$. For a hierarchical Stateflow model we have that every sub-state of an and-state modifies different variables and every sub-state of an or-state modifies the same variables:

$$\begin{array}{l}\forall q.q \in Q_{and} \Rightarrow \\ (\forall q_1, q_2.q_1, q_2 \in c[\{q\}] \wedge q_1 \neq q_2 \Rightarrow \\ \{V | V(b(c^*[\{q_1\}]))\} \cap \{V | V(b(c^*[\{q_2\}]))\} = \emptyset)\end{array}$$

$$\begin{array}{l}\forall q.q \in Q_{or} \Rightarrow \\ (\forall q_1, q_2.q_1, q_2 \in c[\{q\}] \wedge q_1 \neq q_2 \Rightarrow \\ \{V | V(b(c^*[\{q_1\}]))\} = \{V | V(b(c^*[\{q_2\}]))\})\end{array}$$

We do not allow arbitrary transitions. Stateflow allows transitions that have very complicated semantics. However, we here give a number of additional constraints to limit the set of legal transitions in order to only use transitions with intuitive behaviour. To simplify the rules we first define a singleton set giving the state that is the closest common ancestor (cca) of a set of states q. This is the state lowest in the state hierarchy that is a parent of every state in q:

$$\begin{aligned} cca(q) \mathrel{\hat{=}} & \{r \in Q | (\forall s . s \in q \Rightarrow r \in p^*[\{s\}]) \wedge \\ & (\forall s . s \in Q \wedge q \subseteq c^*[\{s\}] \Rightarrow s \in p^*[\{r\}])\} \end{aligned}$$

We do not allow transitions to self or root, since transitions to root are forbidden in Stateflow and transition to self does nothing in our definition of mode-automata. Transitions that cross the boundary of a composite state are not allowed either. This restriction is introduced to enforce creation of more structured models and it is not enforced by Stateflow. In order to find if a transition cross a composite state boundary we check that there is no composite state on the path between the source (or destination) and the closest common ancestor of its source and destination. Consider a transition with source q_1 and destination q_2:

$$\begin{aligned} & q_1 \neq q_2 \\ & q_1 \neq root \wedge q_2 \neq root \\ & (Q_{and} \cup Q_{or}) \cap \{r \in Q | r \in (c^+[cca(q_1 \cup q_2)] \cap p^+[\{q_1\}])\} = \emptyset \\ & (Q_{and} \cup Q_{or}) \cap \{r \in Q | r \in (c^+[cca(q_1 \cup q_2)] \cap p^+[\{q_2\}])\} = \emptyset \end{aligned}$$

The final constraint concerns the ordering of transitions. Transitions that have the same source state have a fixed priority. The two transitions cannot have the same priority, which is also ensured by Stateflow:

$$\begin{aligned} & \forall (q_1, g_1, q_2), (q_1, g_2, q_3) . (q_1, g_1, q_2) \in T \wedge (q_1, g_2, q_3) \in T \wedge q_2 \neq q_3 \\ & \Rightarrow \rho((q_1, g_1, q_2)) \neq \rho((q_1, g_2, q_3)) \end{aligned}$$

3.2 The Model of the Physical System

The physical system to be controlled is also modelled using Simulink. The physical system or environment is usually continuous and it is assumed to be described using a system of ordinary differential equations.

$$\begin{array}{l} \dot{z}(t) = g_c(z(t), w_i(t)) \\ w_o(t) = h_c(z(t), w_i(t)) \end{array} \quad , \quad z(t_0) = z_0$$

Here z is the state of the system, w_i is the input of the system and w_o is the output of the system. In this paper we focus on the development of the controller. We only need to have a description of the physical system to get a complete model of the entire system that can be simulated. The controller only measures the values of the sensors and sets the values of the actuators at discrete time intervals. From the point of view of the controller the physical system can, hence, be viewed as a discrete system with the same sampling time t_s as the controller.

$$\begin{array}{l} y((k+1)t_s) = g_d(y(kt_s), w_i(kt_s)) \\ w_o(kt_s) = h_d(y(kt_s), w_i(kt_s)) \end{array} \quad , \quad y(k_0 t_s) = y_0$$

Here y is the discrete state space and $y((k+1)t_s)$ is the value of y after $k+1$ samples. The state space y and functions g_d and h_d are chosen in such a way that the output w_o is the same for both systems at the sampling time instances kt_s for the same input signals. The system also has to have the same initial time, $t_0 = k_0 t_s$. Note, that we do not consider how to construct the discrete system from the continuous one. We only focus on the controller and therefore this abstraction is sufficient for our purpose.

Physical System A discrete model of the physical system can be viewed as a tuple $\mathcal{P} = (W_i, W_o, Y, i, g_d, h_d, p)$ where:

- W_i is the input variables (source ports), W_o is the output variables (output ports) and Y is the state space of the controller. The set of variables are disjoint, $W_i \cap W_o = \emptyset$, $W_i \cap Y = \emptyset$ and $W_o \cap Y = \emptyset$. The sensors in the Simulink model V_s are a subset of the output variables, $V_s \subseteq W_o$, and the actuators V_a are a subset of the input variables, $V_a \subseteq W_i$.
- $i : Y \to \Sigma$ is the initialization of the state variables.
- $g_d : Y \to G_d(W_i, Y)$ is a function that maps every state variable to an update function.
- $h_d : W_o \to H_d(W_i, Y)$ is a function that maps every output variable to an update function.

3.3 Composition of Mode-Automata

The semantics that we have given above is useful for ensuring that a given Simulink model satisfies our definition of mode-automata. Now it is also important to be able to ensure that this semantics is preserved "*by construction*". To do that, we need to define a constructive semantics of the only two construction mechanisms that the language offer, the parallel "AND" composition and the classical automaton "OR" composition.

We can compose two Simulink models together in two different ways. The two models can become sub-states of an and-state (AND Composition) or an or-state (OR Composition). Assume we have two Simulink models $\mathcal{M}_A = (\mathcal{C}_A, \mathcal{P}_A, V_{sA}, V_{aA})$ and $\mathcal{M}_B = (\mathcal{C}_B, \mathcal{P}_B, V_{sB}, V_{aB})$. Their controllers are given as $\mathcal{C}_A = (V_{iA}, V_{oA}, X_A, i_A, M_A, f_{xA}, \mathcal{S}_A, G_A, \phi_A)$ and $\mathcal{C}_B = (V_{iB}, V_{oB}, X_B, i_B, M_B, f_{xB}, \mathcal{S}_B, G_B, \phi_B)$. We illustrate the compositions with two models, but it can be generalised to an arbitrary number of models.

3.3.1 AND Composition

The output variables of the controller need to be uniquely defined. Hence, the sensors, actuators and all output variables of the controllers need to be disjoint, $V_{sA} \cap V_{sB} = \emptyset$, $V_{aA} \cap V_{aB} = \emptyset$ and $V_{oA} \cap V_{oB} = \emptyset$. The assignment to the memory (state) have to be compatible in both models. This means that $\forall v. v \in X_A \cap X_B \Rightarrow f_{xA}(v) = f_{xB}(v)$. The variables in the physical systems $\mathcal{P}_A$ and $\mathcal{P}_B$ in both models also have to be disjoint, $W_{iA} \cap W_{iB} = \emptyset$, $W_{oA} \cap W_{oB} = \emptyset$ and $Y_A \cap Y_B = \emptyset$.

We denote the composition $\mathcal{M}_C = \mathcal{M}_A \parallel \mathcal{M}_B$. The Simulink model becomes $\mathcal{M}_C = (\mathcal{C}_C, \mathcal{P}_C, V_{sA} \cup V_{sB}, V_{aA} \cup V_{aB})$. The controller $\mathcal{C}_C$ is then the tuple $\mathcal{C}_C = (V_{iA} \cup V_{iB}, V_{oA} \cup V_{oB}, X_A \cup X_B, i_C, M_C, f_{xC}, \mathcal{S}_C)$ where:

- $i_C = i_A \cup i_B$
- $M_C = M_A \cup M_B$
- $f_{xC} = f_{xA} \cup f_{xB}$
- $\mathcal{S}_C$ is the stateflow model of $\mathcal{M}_C$
- $G_C = G_A \cup G_B$
- $\phi_C = \phi_A \cup \phi_B$

The Stateflow model $\mathcal{S}_C$ becomes the tuple $\mathcal{S}_C = (Q_C, Q_{andC}, Q_{orC}, root_C, p_C, i_C, \ldots)$, where:

- $Q_C = Q_A \cup Q_B \cup \{q_{and}, root_C\}$. The set of states of $\mathcal{S}_C$ is the union of the states of $\mathcal{S}_A$ and $\mathcal{S}_B$ with an additional state for modelling parallel execution of the two. An illustration of the formalisation of the state hierarchy is found in Figure 2 in Section 3.
- $Q_{andC} = Q_{andA} \cup Q_{andB} \cup \{q_{and}\}$ and $Q_{orC} = Q_{orA} \cup Q_{orB} \cup \{root_C\}$.
- $p_C = (p_A \cup p_B \cup \{(q_{and}, root_C), (root_A, q_{and}), (root_B, q_{and})\})$
- $i_C = (i_A \cup i_B \cup \{(root_C, q_{and}), (q_{and}, root_A), (q_{and}, root_B)\})$
- $b_C = b_A \cup b_B$
- $T_C = T_A \cup T_B$
- $\rho_C = \rho_A \cup \rho_B$

The physical system $\mathcal{P}_C$ is also composed from the physical systems $\mathcal{P}_A$ and $\mathcal{P}_B$. $\mathcal{P}_C = (W_{iA} \cup W_{iB}, W_{oA} \cup W_{oB}, Y_A \cup Y_B, g_{dA} \cup g_{dB}, h_{dA} \cup h_{dB})$.

3.3.2 OR Composition

The second option when composing two models is to make them sub-states of a common or-state. The output and state variables of the controller again need to be uniquely defined. Hence, the variables need to be the same in both models, $V_{sA} = V_{sB}$, $V_{aA} = V_{aB}$ and $V_{oA} = V_{oB}$. The behaviour updating common variables and the physical system also need to be the same, $f_{xA} = f_{xB}$ and $\mathcal{P}_A = \mathcal{P}_B$.

This composition requires that we add a set of transitions in the composition in order to change from modes in $\mathcal{M}_A$ to modes in $\mathcal{M}_B$. We denote the composition $\mathcal{M}_C = \mathcal{M}_A \circ_U \mathcal{M}_B$, where $U = \{(root_A, e_1, root_B), \ldots, (root_B, e_n, root_A)\}$ is a set of transitions between the root states of the two composed models. The Simulink model becomes $\mathcal{M}_C = (\mathcal{C}_C, \mathcal{P}_A, V_{sA}, V_{aA})$. The controller $\mathcal{C}_C$ is then the tuple $\mathcal{C}_C = (V_{iA} \cup V_{iB}, V_{oA}, X_A, i_A, M_C, f_{xA}, \mathcal{S}_C, G_C, \phi_C)$, where

- $M_C = M_A \cup M_B$
- $G_C = G_A \cup G_B$
- $\phi_C = \phi_A \cup \phi_B$

The Stateflow model $\mathcal{S}_C$ becomes the tuple $\mathcal{S}_C = (Q_C, Q_{andC}, Q_{orC}, root_C, p_C, i_C, b_C, \ldots)$, where

- $Q_C = Q_A \cup Q_B \cup \{root_C\}$. The set of states in $\mathcal{S}_C$ is the union of the sets of states in $\mathcal{S}_A$ and $\mathcal{S}_B$ with an additional or-state for composition.
- $Q_{andC} = Q_{andA} \cup Q_{andB}$ and $Q_{orC} = Q_{orA} \cup Q_{orB} \cup \{root_C\}$
- $p_C = (p_A \cup p_B \cup \{(root_A, root_C), (root_B, root_C)\})$
- $i_C = i_A \cup i_B \cup \{(root_C, root_A)\}$ is the initialization of states in $\mathcal{M}_C$. It is assumed that $root_A$ is the initial state.
- $b_C = b_A \cup b_B$
- $T_C = T_A \cup T_B \cup U$.
- $\rho_C = \rho_A \cup \rho_B \cup \rho_U$

4 Behavioural Semantics

The behavioural semantics of a Simulink/Stateflow model conforming to the mode-automata architecture is given as a set of traces over the variables. This is similar to how semantics is defined for synchronous languages such as Lustre [2]. The semantics gives a very compact description of the intended behaviour. Since we have restricted Simulink/Stateflow to a subset with clear semantics, it is relatively easy to manually verify that the behavioural semantics given here corresponds to the behaviour of Simulink. We cannot prove the equivalence between this formal description of mode-automata behaviour and the behaviour of Simulink/Stateflow, since Simulink/Stateflow does not have a formal semantics. However, the formal semantics given here is a useful guideline for implementing mode-automata using Simulink/Stateflow, as well as for its implementation in other modelling languages.

The controller of the system is assumed to be executed periodically with a fixed sampling time t_s. The behaviour of the continuous physical system is only observed at discrete time instants in the controller. Let $V(t)$ denote the values of the variables V at time t. A simulation of the system having variables V is a sequence of variable values $V(t_0), V(t_0 + t_s), \ldots, V(t_0 + nt_s)$. The mode-automata has a set of active states (modes) that changes during the execution of the system. The set of active states at time t is here given as $Q_a(t) \subseteq Q$. We have the following constraints. In Stateflow all sub-states of an active and-state are active and only one sub-state of an active or-state is active:

$$\begin{array}{l} \forall q_i . q_i \in Q_a(t) \cap Q_{and} \Rightarrow c[\{q_i\}] \subseteq Q_a(t) \\ \forall q_i . q_i \in Q_a(t) \cap Q_{or} \Rightarrow |(c[\{q_i\}] \cap Q_a(t))| = 1 \end{array}$$

The initial configuration at t_0 for a model of a control system with controller conforming to the mode-automata is given below.

Initialization of a system with a mode-automata controller

$$\begin{array}{l} Q_a(t_0) = i^*(root) \\ X(t_0) = i(X) \\ V_o(t_0) = b[Q_a(t_0) - (Q_{or} \cup Q_{and})](X(t_0), V_i(t_0)) \\ Y(t_0) = i(Y) \\ W_o(t_0) = h_d(W_o)(Y(t_0), W_i(t_0)) \end{array}$$

The active states $Q_a(t_0)$ are set to the initial states of the Stateflow model. This behaviour can be achieved by choosing to initialize the Stateflow diagram at the start of the simulation. The memory variables X and Y are set to their initial values. The output variables V_o and W_o are computed from the initialised memory and the input variables.

In order to give the semantics of the evolution of the system from time t to time $t + t_s$ we need to find the transitions that are executed at each time instant. Transitions can be executed when their corresponding guard evaluates to *true*. We first compute the set of these enabled transitions:

$$\begin{array}{l} T_e \hat{=} \\ \quad \{(q_1, g, q_2) | (q_1, g, q_2) \in T \\ \quad \wedge q_1 \in Q_a \wedge \phi(g)(V_i(t + t_s), X(t))\} \end{array}$$

The transitions that are executed are concurrent and have the highest priority. Transitions higher in the state-hierarchy have higher priority and we therefore need to compute the set of transitions that are highest in the hierarchy:

$$\begin{aligned}&T_{hierarchy} \hat{=} \\ &\quad \{(q_1, g, q_2) | (q_1, g, q_2) \in T_e \wedge q_1 \neq q_2 \wedge \\ &\quad \forall (q_{11}, g_1, q_{22}).(q_{11}, g_1, q_{22}) \in T_e \Rightarrow q_1 \notin c^+[\{q_{11}\}]\}\end{aligned}$$

There can still be enabled transitions with the same source state. Which one of these transitions is executed depends on the fixed priority of the transitions:

$$\begin{aligned}&T_{prio} \hat{=} \\ &\quad \{(q_1, g, q_2) | (q_1, g, q_2) \in T_{hierarchy} \wedge \\ &\quad \forall (q_1, g_1, q_{22}).(q_1, g_1, q_{22}) \in T_{hierarchy} \wedge q_2 \neq q_{22} \Rightarrow \\ &\quad \rho((q_1, g, q_2)) < \rho((q_1, g_1, q_{22}))\}\end{aligned}$$

We now have a set of transitions that can be executed at the same time.

A transition modifies the set of active states. To describe how the set of active states is modified we define a number of sets to enhance readability similar to [21]. First we need to define the set of states on the path between the two sets of states q_1 and q_2. Sets q_1 and q_2 are both sets with only one element.

$$path(q_1, q_2) \hat{=} \{r \in Q | r \in c^+[q_1] \cap p^*[q_2]\}$$

We need the active states exited when the transition is executed.

$$exited(q_1, q_2) \hat{=} \{r \in Q_a | r \in c^+[cca(q_1 \cup q_2)]\}$$

The states entered when the transition is executed are also needed.

$$entered(q_1, q_2) \hat{=} path(cca(q_1 \cup q_2), q_2) \cup i^*[q_2]$$

The definition of the system's evolution over time can now be given. Transitions are executed first, while all variables are updated according to the update functions defined in the model.

Evolution of the system from t to time $t + t_s$.

$$\begin{aligned}&T_{prio} \neq \emptyset \Rightarrow Q_a(t + t_s) = \\ &\quad \{Q_n \in Q | Q_n = (Q_a(t) - exited(\{q_1\}, \{q_2\})) \cup \\ &\quad entered(\{q_1\}, \{q_2\}) \wedge \exists g.(q_1, g, q_2) \in T_{prio}\} \\ &T_{prio} = \emptyset \Rightarrow Q_a(t + t_s) = Q_a(t) \\ \\ &X(t + t_s) = f_x(X)(X(t), V_i(t + t_s), V_o(t + t_s)) \\ &V_o(t + t_s) = b[Q_a(t + t_s) - (Q_{or} \cup Q_{and})](V_o)(X(t), V_i(t + t_s)) \\ &Y(t + t_s) = g_d(Y)(Y(t), W_i(t + t_s)) \\ &W_o(t + t_s) = h_d(W_o)(Y(t), W_i(t + t_s))\end{aligned}$$

This uniquely defines the evolution of the system from time t_0 to $t_0 + nt_s$. The semantics corresponds to the behaviour that can be observed in the controller during the simulation of a corresponding Simulink model. First all transitions that can be executed are executed to find the current active modes. All variables are then updated according to the update functions given earlier.

5 Practical Implementation in Simulink

There are a number of practical considerations when implementing a controller with mode-automata architecture in Simulink/Stateflow. These issues involve architecture,

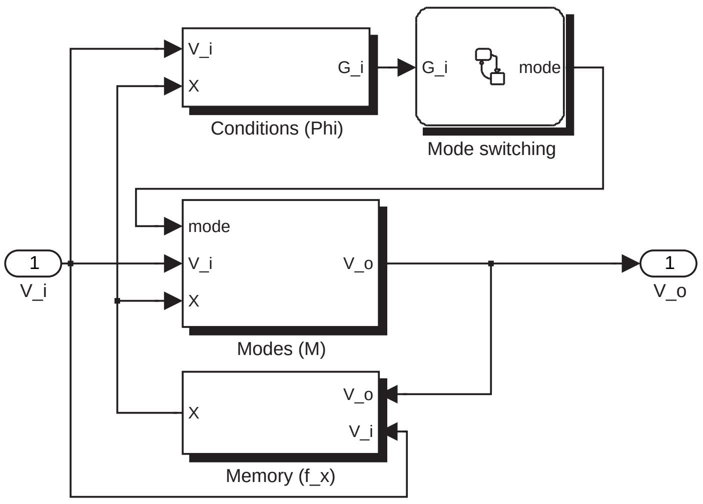

Figure 3: Overview of the controller

guard definition, mode switching and memory updates. The controller can be considered to consists of four disjoint parts as shown if Figure 3; 1) A state machine modelled in Stateflow, 2) Simulink subsystems for modelling different modes of operations (M), 3) block diagrams for defining the guard conditions (Phi) enabling transitions in the Stateflow diagram and 4) block diagrams for updating the memory of the controller (f_x). Additionally, we assume that all blocks in the controller have the same sampling time and that the blocks are not allowed to have side effects.

5.1 Guard conditions

Each guard name g_i is associated with a condition $\phi_i(V_i, X)$. The condition can be modelled by a Simulink block diagram with one output of type *boolean*. The output has the value *true* when the condition holds and *false* otherwise. The outputs of the block diagrams that computes the conditions are added as inputs to the Stateflow model.

5.2 Implementing Automata and Mode Switching Using Stateflow

We use a subset of Stateflow for implementing the automata we need. The current active states can be automatically exported from the Stateflow model to the Simulink block diagram. A port with the same name as the state is then available, which has the value *true* when the state is active and *false* otherwise.

Each active state in the Stateflow model is associated with an update function $m \subseteq M$ in the Simulink model. We use *Enabled Subsystems* to associate an active state with the computation that should be performed in that state. The correct values of the output variables is obtained by using a *Merge*-block to merge the signals from different enabled subsystems.

5.3 Updating the Memory

The memory of the controller should be updated regardless of which mode the system is in. To illustrate this, consider an example where the previous value of output variable v_o is stored in variable x. If memory is allowed in an enabled subsystem and the

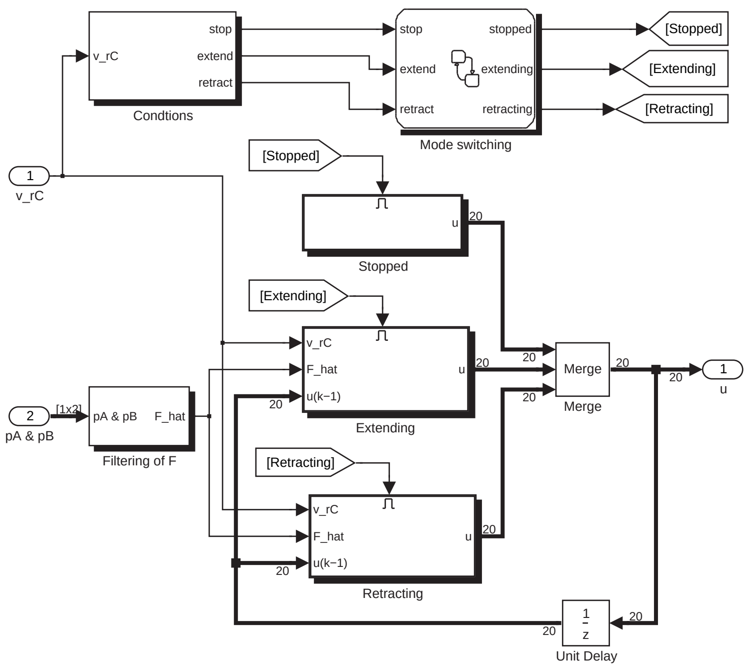

Figure 4: Simulink implementation of the mode-automata in the case study

mode is switched, x will not store the updated value of v_o. If the mode is switched back then x will contain an old version of v_o, which can lead to problems in control algorithms. This means that memory in the controller cannot be updated in mode specific behaviour, i.e., any enabled subsystem enabled by a mode.

6 Case Study

To investigate the suitability of the formalisation we have tested it on a case study. The case study is a digital hydraulics system [11] developed for research in digital hydraulics at the Tampere University of Technology. The system consists of a hydraulic cylinder that moves a load mass either to a desired position or with a desired speed. The speed of the load mass is controlled by the pressure on each side of the piston in the cylinder. A digital controller controls the pressures in the cylinder using a system of on/off valves. One of the main ideas of digital hydraulics is to use simple, cheap mechanical components and more advanced control algorithms instead. Hence, the controller is fairly large and contains rather sophisticated algorithms. The complexity is increased by several different modes of operation in the controller. In the final controller there will be modes for considering, direction of movement, fault tolerance and energy saving.

The Simulink model for the mode-automata controlling the system is shown in Figure 4. Here we only consider a simplified controller for normal operation with three modes *stopped*, *extending* and *retracting*. These modes corresponds to the modes in subsystem M in Figure 3. The blocks *Unit Delay* and *Filtering of F* corresponds to the subsystem f_x in Figure 3. The input port v_{rC} gives the difference between the desired speed and the actual current speed. The port $pA\&pB$ gives the pressures on either side of the piston in the hydraulic cylinder. From the pressures the force F acting on the load mass can be estimated. The output port u then gives the new positions of the

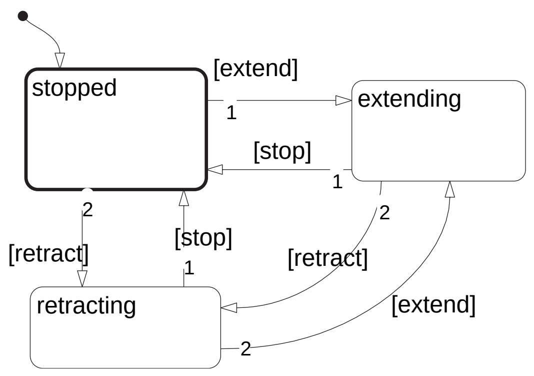

Figure 5: Stateflow model in the case study

valves computed by the active mode.

The guard conditions concerns the direction of movements. The guard *extend* is associated with the condition $v_{rC} > v_{tol}$, *retract* with $v_{rC} < -v_{tol}$ and *stop* with $\neg(v_{rC} > v_{tol}) \wedge \neg(v_{rC} < -v_{tol})$. The constant v_{tol} gives the speed for which the system is assumed to be stopped. The mode transitions are modelled using Stateflow as shown in Figure 5. Each mode of the controller is associated with an enabled subsystem that defines the mode specific behaviour in the Simulink model. The system in the case study does not use hierarchical modes at the moment, but it is possible according to our formalisation.

The people working with the digital hydraulics system have found the mode automata architecture very suitable for their purpose. It provides an architecture where it easy to update behaviour in modes, which is important for a model used for research. It also enables collaboration where different people can focus on different modes.

7 Towards Model Verification

The formalisation of mode-automata should only allow a subset of Simulink/Stateflow, but it should also allow interesting models to be created. Simulink and Stateflow do not have a formal semantics and it is, therefore, impossible to prove equivalence between the formalisation and Simulink. To investigate if the formalisation works as intended we have studied its properties using the Alloy Analyzer [8]. Alloy is a tool based on first order logic, where systems can be modelled using relations and constraints on relations. It can then be used to generate models satisfying the constraints and to check validity of assertions in models. Using this tool we have checked that we can generate models that correspond to Simulink/Stateflow models satisfying the constraints in the formalisation. We have also checked that the conditions in our formalisations are sufficient in order to guarantee the properties we like to ensure. The Alloy Analyzer is not a theorem prover and, hence, we can only investigate models of limited size. However, the tool can give confidence that the formalisation works as intended.

As future work we intend to create a tool for checking that a Simulink/Stateflow model conforms to our definition of mode-automata. This can be done by translating the Simulink/Stateflow models to the representation given in the formalisation. This representation is then used to verify that all constraints are satisfied.

Verification of properties of the complete controller can be extremely difficult due to size and complexity. When the model conforms to the mode-automata definition we can take advantage of this architecture to verify certain properties of the controller. It is possible to verify that the guarded mode transitions function correctly. To enable this type of verification, an invariant is assigned to each mode. We only need to consider the conditions associated with the guards, the Stateflow model and the invariants. A model checker can then be used to ensure that the invariant in each mode is maintained by the transitions. For example the *ss2lus* translator [19] from Simulink/Stateflow to Lustre [2] can be used in conjunction with the model checkers for Lustre [18].

8 Conclusions and Further Work

In this paper we have given a formal definition of mode-automata implemented using Simulink and Stateflow. The mode-automata architecture restricts the allowed constructs from Simulinik/Stateflow to a safe kernel with clear semantics. The aim is to have allow enough features for the architecture to be usable in practice, while simplifying the analysis of the models. The formalisation considers state-machines with both or-states and and-states. We also give definition of two methods for composing different mode-automata. In order to validate the formalisation, its properties has been investigated with the Alloy Analyzer. The mode-automata model architecture provides a structured and maintainable model architecture for mode-based systems. Furthermore, it can also be exploited for validating certain desirable properties of the controller.

Future Work We plan to extend this work in several directions. Stepwise development and refinement can be beneficial for developing complex systems. We plan to introduce the notion of refinement into our formalisation. This will be done by first expressing the semantics given in section 3 in the refinement calculus [1] in order to benefit from it. This approach has already been explored for Statecharts alone [20], but now we wish to extend these works to the couple Stateflow/Simulink. Ultimately, this will give us strong formal support for stepwise refinement of Stateflow/Simulink models.

The suitability of the formalisation will also be further investigated through case studies. The example given in section 6 is a first simple example taken from our collaboration on designing correct controllers for digital hydraulics applications. As the project is going on, we are currently refining this example and are confident that it will finally reach a reasonable size so as to demonstrate the full applicability of mode-automata to industrial applications. It will also help us validate the restriction we wish to apply on Stateflow/Simulink in order to get a stable semantics.

Verification and Testing methods based on this architecture are also interesting topics for further research. In this context, we plan to assemble a set of well-established techniques and apply them to Stateflow/Simulink models. These techniques will contain, among other things local specification in the form of assume-guarantee contracts [12] and compositional verification rules [7, 16]. The overall goal of the project is not to design brand new verification techniques, but rather to see how it is possible to integrate existing ones in a global development and validation framework for Stateflow/Simulink.

Acknowledgement

This work is carried out in the context of the ITCEE (Improving transient control and energy efficiency by digital hydraulics) project funded by TEKES (Finnish Funding Agency for Technology and Innovation).

References

[1] R.-J. Back and J.von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[2] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. Lustre, a declarative language for programming synchronous systems. In *14th ACM Conf. on Principles of Programming Languages*, Munich, Germany, 1987.

[3] J.-L. Colaço, B. Pagano, and M. Pouzet. A conservative extension of synchronous data-flow with state machines. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 173–182, New York, NY, USA, 2005. ACM Press.

[4] G. Hamon. A denotational semantics for stateflow. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 164–172, New York, NY, USA, 2005. ACM Press.

[5] G. Hamon and J. Rushby. An operational semantics for stateflow. In *Fundamental Approaches to Software Engineering, FASE 2004*, volume 2984 of *LNCS*, pages 229–243. Springer Verlag, 2004.

[6] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[7] Holenderski. Compositional verification of synchronous networks. In *FTRTFTS: Formal Techniques in Real-Time and Fault-Tolerant Systems: International Symposium Organized Jointly with the Working Group Provably Correct Systems – ProCoS*. LNCS, Springer-Verlag, 2000.

[8] D. Jackson. *Alloy 3.0 Reference Manual*, 2004. `http://alloy.mit.edu`.

[9] F. Jahanian and A. K. Mok. Modechart: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 20(12):933–947, December 1994.

[10] O. Labbani, J.-L. Dekeyser, and P. Boulet. Mode-automata based methodology for scade. In *Hybrid Systems: Computation and Control: 8th international workshop, HSCC 2005*, volume 3414 of *LNCS*, pages 386–401. Springer Verlag, 2005.

[11] M. Linjama, K. T. Koskinen, and M. Vilenius. Accurate trajectory tracking control of water hydraulic cylinder with non-ideal on/off valves. *International Journal of Fluid Power*, 4(1):7–16, 2003.

[12] F. Maraninchi and L. Morel. Logical-time contracts for reactive embedded components. In *30th EUROMICRO Conference on Component-Based Software Engineering Track, ECBSE'04*, Rennes, France, August 2004.

[13] F. Maraninchi and Y. Rémond. Mode-automata: About modes and states for reactive systems. In *European Symposium on Programming*, volume 1381 of *LNCS*. Springer Verlag, 1998.

[14] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, 46(3):219–254, 2003.

[15] F. Maraninchi, Y. Rémond, and Y. Raoul. Matou : An implementation of mode-automata into dc. In *Compiler Construction*, Berlin (Germany), March 2000. Springer verlag.

[16] K. L. McMillan. A methodology for hardware verification using compositional model checking. *Science of Computer Programming*, 37(1-3):279–309, 2000.

[17] C. Puchol, D. Stuart, and A. Mok. An operational semantics and compiler for real-time specifications, 1998.

[18] P. Raymond. *LUSTRE-V4 manual*, 2000. `http://www-verimag.imag.fr/SYNCHRONE/tools.html`.

[19] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a "safe" subset of simulink/stateflow into lustre. Technical Report TR-2004-16, Verimag, Centre Équation, 38610 Gières, July 2004. `http://www-verimag.imag.fr/index.php?page=techrep-list`.

[20] P. Scholz. A refinement calculus for statecharts. In *FASE: International Conference on Fundamental Approaches to Software Engineering (FASE)*. LNCS, Springer-Verlag, 1998.

[21] E. Sekerinski and R. Zurob. Translating statecharts to B. In *Integrated Formal Methods (IFM 2002)*, volume 2335 of *LNCS*. Springer Verlag, May 2002.

[22] A. Tiwari. Formal semantics and analysis methods for Simulink Stateflow models. Technical report, SRI International, 2002. `http://www.csl.sri.com/~tiwari/stateflow.html`.

ISBN 952-12-1737-5
ISSN 1239-1891