Pontus Boström | Lionel Morel

Formal Definition of a Mode-Automata Like Architecture in Simulink/Stateflow

Turku Centre *for* Computer Science

TUCS Technical Report
No 830 , 2007

Formal Definition of a Mode-Automata Like Architecture in Simulink/Stateflow

Pontus Boström
Åbo Akademi University, Department of Information Technologies
Turku Centre for Computer Science
Joukahaisenkatu 3-5, 20520 Turku, Finland
pontus.bostrom@abo.fi
Lionel Morel
project ESPRESSO
INRIA/IRISA - Campus universitaire de Beaulieu
35042 Rennes Cedex, France
lionel.morel@inria.fr

TUCS Technical Report
No 830 , 2007

Abstract

As embedded control systems are becoming more complex, there is a need for new software development and structuring techniques. The combination Simulink/Stateflow has become a popular tool for model-based design for this type of hybrid systems, due to the simulation and analysis tools available. To enable design and validation of large complex systems in Simulink/Stateflow, an appropriate model architecture is needed. Mode-automata is such an architecture, where control is strictly separated from signal processing. In this paper we give a formal definition of mode-automata in Simulink/Stateflow. This gives a precise definition of an architecture that restricts Simulink/Stateflow to a safe and easy to use subset that is easy to verify, but still usable in practice. We propose syntactic rules to check that a given Simulink/Stateflow model complies to our mode-automata architecture and we illustrate the approach with a controller for a digital hydraulics system.

Keywords: Simulink/Stateflow, Architecture, Mode-Automta, Control Systems

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 modeling/programming methodologies that will make the design flow clearer in itself.

The general setting of the work presented in this paper deals with improving the design flow by 1) making modeling concepts clearer and 2) introducing formal methods along the development process. In practice, we propose to study this design flow around the Stateflow/Simulink modeling tool. This latter tends to become a standard in industrial development of embedded software. However, although programming methodologies have been proposed for it, it still lacks a steady formal setting that could allow definition and application of interesting validation techniques.

1.1 Separating control and dataflow for managing systems complexity

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 in a dataflow manner.

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 [20] as implemented in the Matou tool [21]. Activity of the system is separated in different *running modes* that are described as states of a 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 [6]). 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 [14, 25]. However, Modecharts are more aimed at expressing timing properties for real-time systems, which is not considered here.

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 [9]. In this paper, we join recent work [7, 16] on the introduction of state-machine structures in SCADE (which is very similar to Simulink). The second one is the coupling of Stateflow and Simulink[1], present in the Matlab tool-set. However, the semantics behind the language is somewhat unclear, based on graphical assumptions[2]. Several works [10, 11, 27] have recently proposed formal semantics for a reasonable subset of Simulink / Stateflow. These set up a good background for our long-term goal. Our goal in this paper is not, however, to extend those semantics. Rather we try to propose sensible subset of Stateflow/Simulink that we hope can serve as a guide line for programmers and allow for the establishment of validation techniques usable in practice.

[1] Simulink and Stateflow are trademarks of *MathWorks Inc.*

[2] For example, transitions going out of a state are tested following a 12 o'clock rule: transitions are picked in a clockwise order around the originating state. This rule is purely graphical and not semantics-related.

We particularly want to define an interesting subset of Stateflow/Simulink allowing for simpler and clearer semantics from the designer point of view, and yet not reducing the expressive power too drastically. Our proposal amounts to applying the recommendation advocated by the mode-automata approach to Simulink / Stateflow designs.

Finally, as we propose an architectural guide-line for building Simulink/Stateflow applications, one important part of our work resides in the establishment of static rules that can be used to check whether the developer's design conforms to our mode-automata-like architecture. These assumptions on models are expressed as rules that can be checked automatically. Similar approaches for defining constraints on e.g. UML models have been developed (see [23, 24]). The ideas for given architectural constraints are similar, but here we focus on defining a specific architecture.

1.2 Propositions

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

The implementation of mode-automata that we propose is trying to adapt the approach proposed in [20], the most naturally possible to Simulink / Stateflow. Initial work in this direction has been published by the authors in a technical report [5]. Here we improve the formal description, consider more features in Stateflow and present a complete example. The paper shows how even complex architectural constraints can be conveniently described. Validation that the constraints indeed describe the desired model restrictions is also briefly discussed.

Benefits from this methodological design approach for validation is under study. The whole approach is being applied in a national research project on design of controller for digital hydraulics systems. The mode-automata approach has been proposed as an answer to meet explicit needs from end users to make the design methodology clearer both during the design and validation phases. The case study presented in section 5 has been re-factored from a previous version into this mode-automata structure. Ongoing work includes building a new application from scratch applying our methodology right from the beginning.

1.3 Structure of the Paper

Section 2 briefly recalls the mode-automata approach. Section 3 gives a formal definition of the subset of Simulink / Stateflow we consider. Section 4 shows how to apply the mode-automata architecture to design Stateflow / Simulink models. It gives a set of syntactic constraints to check that a model satisfies this architecture model and comments on the validation of these rules. Section 5 presents a case-study to which we apply the mode-automata paradigm. Section 6 concludes and presents briefly ongoing and future work.

2 Mode-Automata

Mode-automata have been proposed in [20] to enable the description of reactive systems in terms of *running modes*. They give a way to combine in a sound semantic framework dataflow oriented languages together with control-flow design (automata). Typical usage includes e.g. the control of an aircraft in which the same commands are directed differently whether the airplane is in take-off mode or landing mode.

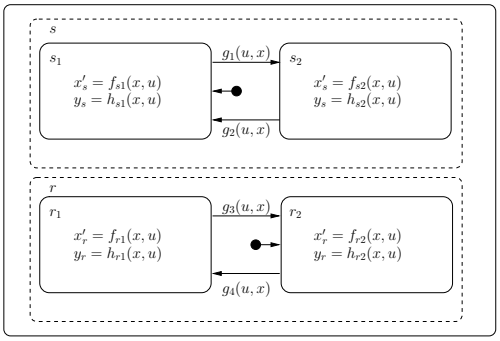


Figure 1: Example of a mode-automata

A mode-automaton consists of input variables, memory, output variables and a set of modes. Each mode has a different function for updating the value of the output variables and the memory. Transitions between modes are guarded by conditions that depends on the values of the input variables and the memory.

Figure 1 shows an example of a mode-automaton. The automaton has input variables u, memory x, output variables $y_r \cup y_s$. There are two parallel modes s and r. These two modes are further decomposed into s_1, s_2 and r_1, r_2, respectively. Transitions are guarded by the conditions g_i. Each mode has a function f that updates the output variables based on the current memory and input, as well as a function h that updates the memory based on current memory values and input values. Since modes s and r runs in parallel, the modes have to update disjoint sets of outputs y_s and y_r, as well as disjoint parts of memory x_s and x_r. Sequential modes then have to update the same variables. Note that a precise behavioural semantics of mode-automata is not given here, since the semantics is given by the underlying formalism were the mode-automata structure is implemented. In [20], behaviors associated to leaf-modes are described by dataflow equations, *à-la* Lustre [6].

3 Simulink and Stateflow

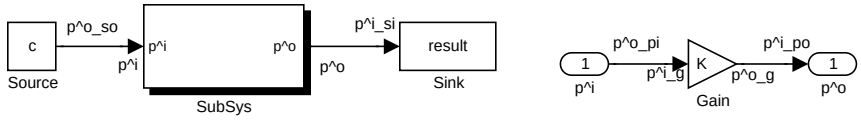
Control systems are often hybrid systems consisting of both discrete and continuous parts. *Matlab* and particularly *Simulink* developed by Mathworks Inc., have become popular tools for modelling, analysing and designing such systems. Simulink is a graphical language where different functional blocks are connected by signals in a dataflow manner.

Some discrete systems are conveniently modelled using *finite state-machines*. Simulink is shipped with *Stateflow*, which is a graphical language for creating hierarchical state-machines similar to Statecharts by Harel [12]. In order to implement the designed systems, both Simulink and Stateflow allow direct code generation from the models.

We first introduce notations used in the paper. Then we propose a formalisation of the subset of Simulink that we consider. Finally, we present the subset of Stateflow that we to use to describe the control architecture of mode-automata.

3.1 Notations

The formalisation is based on the use of higher order logic (HOL) to describe the structure of the models. The notation we use is taken from the refinement calculus



(a) (b)

Figure 2: Example of a Simulink model. The diagram (b) shows the content of the subsystem in (a)

book by Back and von Wright [2]. We use the notation $f : A \rightarrow B$ to denote a total function f from A to B. Functions are defined using λ-calculus and function application is given by the dot-notation, e.g., $f.a$. A relation between elements of two sets A and B is given as $r : A \rightarrow \mathcal{P}(B)$, i.e. each element in A maps to possibly empty set of elements in B. The domain of r is denoted by $\mathsf{dom}.r$ and the range by $\mathsf{ran}.r$. The cardinality of a set A is denoted by $\mathsf{card}.A$. Creation of a relation from the corresponding function f is denoted by $|f|$. Composition of relations r and s is denoted by $s; r$. Furthermore, we denote the transitive closure of the relation r with r^+ and the reflexive transitive closure with r^*. The composition of functions and relations makes it possible to create new higher level objects to reason about the structure of models.

3.2 Formalisation of Simulink diagram structure

The structure of a Simulink block diagram can be described as a set of blocks containing ports. The ports are then related by signals. Simulink has a large library of different blocks for mathematical and logical functions, blocks for modelling discrete and continuous systems, as well as blocks for structuring models. Simulink diagrams can be hierarchical, where subsystem blocks are used to structure the model. An example of a Simulink diagram is shown in fig. 2. The diagram contains one source block giving a value c to a signal connected to the subsystem $SubSys$. The subsystem have in- and out-blocks p_i and p_o to communicate with blocks higher in the hierarchy. The functionality of the subsystem is given by a gain block, $Gain$, that multiplies the input by a constant K. The output from the subsystem is then delivered to a sink block that consumes the given value. This diagram, hence, computes $result = Kc$.

3.2.1 Definition

A Simulink model is defined as a tuple $\mathcal{M} = (B, root^{sim}, \mathsf{subh}, P, \mathsf{blk}, \mathsf{sig}, \mathsf{subi}, \mathsf{subo})$:

- B is the set of blocks in the model. We can distinguish between; subsystem blocks B^s, in-blocks in subsystems B^i, out-blocks in subsystems B^o (representing inputs and outputs of subsystems), merge blocks B^m and blocks with memory B^{mem}.When referring to other types of "basic" blocks B^b is used in this paper. Furthermore, subsystem can be divided into, normal, *virtual subsystems* B^{vs} and *non-virtual subsystems* B^{ns}, $B^s = B^{vs} \cup B^{ns}$. The virtual subsystem do not affect the behavioural semantics of Simulink. They are used purely for structuring the diagrams, while the non-virtual subsystems can affect the semantics;
- $root^{sim} \in B^{vs}$ is the root subsystem;
- $\mathsf{subh} : B \rightarrow B^s$ is a function that describes the subsystem hierarchy. For every block b, $\mathsf{subh}.b$ gives the subsystem b is in. Note that $\mathsf{subh}.root^{sim} = root^{sim}$;
- P is the set of ports for inputs and output of data to and from blocks. The ports $P^i \subseteq P$ is the set of in-ports and $P^o \subseteq P$ is the set of out-ports;

- $\mathsf{blk} : P \rightarrow B$ is a relation that maps every port to the block it is in;
- $\mathsf{sig} : P^i \rightarrow P^o$ maps every in-port to the out-port it is connected to by a signal;
- $\mathsf{subi} : B^s \rightarrow P^o \rightarrow \mathcal{P}(P^i)$ is a partial function that describes the mapping between the in-ports of a subsystem and the out-ports of the in-blocks in that subsystem.
- $\mathsf{subo} : B^s \rightarrow P^o \rightarrow \mathcal{P}(P^i)$ is a partial function that describes the mapping between the out-ports of a subsystem and the in-ports of the out-blocks in that subsystem.

There are several constraints concerning these functions and relations in order to only consider valid Simulink models. These constraints involve e.g. valid hierarchy of subsystems and correct definition of connection over subsystem boundaries. In this paper we assume we only deal with syntactically correct Simulink / Stateflow models (ones that can be simulated),

Consider the diagram depicted in fig. 2. The blocks are defined by $B \mathrel{\hat{=}} \{source, SubSys, p_i, Gain, p_o, sink\}$. The subsystems are given as $B^{vs} \mathrel{\hat{=}} \{SubSys, root^{sim}\}$ and $B^{ns} \mathrel{\hat{=}} \emptyset$, while the hierarchy is $\mathsf{subh} \mathrel{\hat{=}} \{(Gain, SubSys), (SubSys, root^{sim}), \ldots\}$. Names of ports are usually not shown in diagrams. Here we have the following ports, $P = \{p^o_{so}, p^i_{sub}, p^o_{sub}, p^i_{si}, \ldots\}$. The function describing which block each port belongs to is then given as $\mathsf{blk} \mathrel{\hat{=}} \{(p^o_{so}, source), (p^i_g, Gain), (p^o_g, Gain), (p^i_{sub}, SubSys), \ldots\}$. The connections between the ports is defined as $\mathsf{sig} \mathrel{\hat{=}} \{(p^i_{sub}, p^o_{so}), (p^i_g, p^o_{pi}), \ldots\}$. The relations describing how ports in in/out-blocks correspond to ports of subsystems are given by subi and subo. The in-port of the subsystem is related to the out-port of the in-block, $\mathsf{subi} \mathrel{\hat{=}} \{ (SubSys, p^o_{pi}, \{p^i_{sub}\}), (SubSys, p^o_c, \emptyset), \ldots\}$. The definition of outputs of the subsystem is similar, $\mathsf{subo} \mathrel{\hat{=}} \{(SubSys, p^o_{sub}, \{p^i_{po}\}), (SubSys, p^o_c, \emptyset), \ldots\}$.

3.3 Formalisation of Stateflow

Stateflow is a Statechart implementation provided with Simulink. The main difference from Statecharts is that Stateflow is completely sequential and deterministic. A Stateflow chart is basically a hierarchical state-machine whose states are labelled with lists of actions and whose transitions are labelled with guards and actions. Both actions and guards are specified using a specific "action language". A Stateflow block is a normal block in Simulink that can have in-ports and out-ports for communicating with the rest of the Simulink model. These ports can be referred to also in the action language of Stateflow. Stateflow contains many advanced features that often lead to semantics ambiguities. One goal of this formalisation (and of the guidelines we propose in sec. 4 for implementing the mode-automata in Simulink/Stateflow) is to reduce the number of these ambiguities *to a minimum*.

We consider only a small subset of Stateflow, described below, that is defined precisely to fit the mode-automata architecture:

- *in-ports*. These ports are used for giving guard conditions to transition segments. The type of the ports have to be Boolean;
- *out-ports*. These ports are used for exporting the current activity status of states to the Simulink model;
- *Hierarchical state-machines*. We consider both sequential or-states and parallel and-states;
- *Guards on transition segments*. We only allow transition segments labelled by guards. To aid graphical analysis, each guard needs to be the name of a port. If Boolean operators were allowed in the guards, we would need a prover to e.g. decide equality of guards. Note, that if guard g is used on a transition we also often have transitions with the guard $\neg g$. Therefore, we also allow the guard

name $!g$ denoting $\neg g$. This does not complicate analysis, and it is therefore allowed for convenience;
- *Junctions*. Junctions represent decision points between different transition paths and they are, hence, used for connecting transition segments together. They can also be used for conjunction and disjunction of guard conditions.

We do not allow activities inside states nor events, actions and condition actions on transition segments. This ensures a usage which is as clear as possible. In particular, it forces for a clear separation of control from signal processing. We believe this separation is essential. Stateflow/Simulink is, to us, too permissive and allows activities (potentially with side-effects) to be expressed with the action language inside states, which is one of the biggest source of confusion and imprecision for designers. We believe the restrictions above limits Stateflow to a safe subset that is easy to formally analyse, but is sufficiently powerful to be used in practise together with Simulink.

The structure of a Stateflow chart is formalised in a similar manner as the Simulink model. Stateflow is only a block in the Simulink model.

3.3.1 Definition

The Stateflow chart $\mathcal{S}$ is here given as a tuple, $\mathcal{S} = (D, Q, Q^{and}, Q^{or}, J, root, \mathsf{sfh}, \mathsf{sfprt}, T, T^d, L, \mathsf{lbl}, \mathsf{trns})$ where:

- D is the set of objects that can be transition segment *sources* or *destinations*;
- $Q \subseteq D$ is the set of states (modes) in the Stateflow model;
- $Q^{and} \subseteq Q$ is the set of and-states. $Q^{or} \subseteq Q$ is the set of or-states. We have that $Q^{and} \cap Q^{or} = \emptyset$;
- J is the set of junctions. The sets of junctions and states are disjoint $Q \cap J = \emptyset$;
- $root \in Q^{or} \cup Q^{and}$ is the root state;
- $\mathsf{sfh} : Q \to (Q^{or} \cup Q^{and})$ is a function that maps a state to its parent. Hence, it describes the hierarchy of states;
- $\mathsf{sfprt} : (Q - (Q^{or} \cup Q^{and})) \to P^o$ is a function that maps every leaf-state to a out-port. The out-port is then used to enable the subsystem that describe the behaviour in the state. The activity of a state can exported automatically to Simulink via a port with the same name as the state;
- T is the set of transition segments;
- $T^d \subseteq T$ is the set of default transition segments. A default transition is the initialisation of an or-state. The source of a transition segment of this type is the or- state it is initialising;
- L is the set of labels on transition segments. Labels are guards that consist of either port names g_i or negation of port names $!g_i$. The value of the condition associated with the port determines when a transition segment is enabled. A transition with empty guard has the label ϵ;
- $\mathsf{lbl} : T \to L$ is a function that gives the guard of each transition segment;
- $\mathsf{trns} : T \to (D \to \mathcal{P}(D))$ gives the source and destination for each transition segment. Each transition segment has only one source and trns is therefore a partial function.

3.3.2 Example

To illustrate the formal definition consider the Stateflow chart in fig. 3. 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 $\mathsf{sfh} = \{(root, root), (q, root), (s, q), (r, q), \ldots\}$ relates the states to their parent state. There are seven transitions segments, $T \hat{=} \{t_1, \ldots, t_7\}$ (the numbers are not shown in

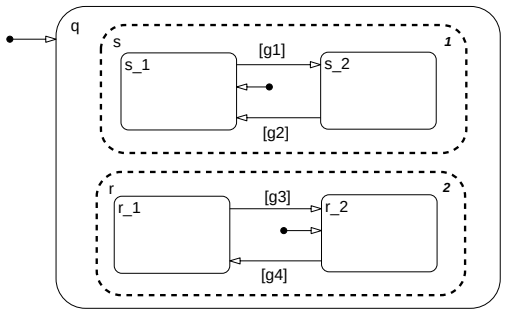


Figure 3: An example of Stateflow model.

the diagram). Three of these transitions are default transitions, $T_d \hat{=} \{t_1, t_2, t_3\}$. The function trns then gives the source and destination of the transitions $\mathsf{trns} \hat{=} \{(t_1, root, \{q\}), (t_1, q, \emptyset), \ldots, (t_4, s_1, \{s_2\}), \ldots\}$. The labels L on the transitions are given as $L \hat{=} \{\epsilon, g1, g2, g3, g4\}$. Transitions are labelled by guards as described by the function lbl. Here lbl is given as $\mathsf{lbl} \hat{=} \{(t_1, \epsilon), (t_2, \epsilon), (t_3, \epsilon), (t_4, g1) \ldots\}$. The following paragraphs establish precise rules for the design of such systems in Simulink/Stateflow.

4 Mode-Automata in Simulink / Stateflow

The Simulink / Stateflow language is a very convenient tool for system construction due to the large set of features. However, some of these features are difficult to analyse and to use correctly. Design guidelines have been developed to ensure that Simulink / Stateflow models are maintainable, readable, and use only safe constructs [15, 22]. Guidelines for using Simulink / Stateflow for production code generation have also been developed [3, 8]. However, even if these guidelines are followed the models are still difficult to formally analyse. In order to translate Simulink / Stateflow to Lustre for verification, restrictions have been adopted [26, 28] to ensure that the constructs are compatible. We also require that these restrictions apply for the controllers in this paper, but we give additional architectural constraints. 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 requirements.

4.1 Building a mode-automaton in Simulink/Stateflow

Mode-automata consist of both a state-machine part and mode-dependent computation. Stateflow is used to implement the state-machine part in Simulink. To implement the mode-specific behaviour *enabled subsystems* B^e, $B^e \subseteq B^{ns}$, are associated with each leaf-mode. The activity of each leaf-mode can be exported automatically from Stateflow to Simulink, where it is used to enable the correct subsystem. To always use the output from enabled subsystems that are currently active, we use *merge*-blocks. Merge-blocks are used to always take the value from the currently active enabled subsystem.

4.2 Syntactic Constraints for the Mode-Automata Architecture

The mode-automata architecture as presented above and in [16, 20], requires a number of constraint involving transitions, state-hierarchy, and connection between Simulink and Stateflow.

4.2.1 Preliminary definitions

To simplify the constraints for Stateflow we introduce seven new relations concerning transitions. The relation trns^t relates junctions and states that are connected by normal transition segments, trns^j gives the relation between junctions, $\mathsf{trns}^{q,j}$ from states to junctions and $\mathsf{trns}^{j,q}$ from junctions to states. The relation trns^q gives the states that are connected by transitions, i.e., by a sequence of transition segments and junctions. Default transitions are treated separately. The relation $\mathsf{trns}^{dq,j}$ gives the relation corresponding to default transitions to junctions and trns^{dq} gives the initial state for each or-state.

$$
\begin{array}{lcl}
\mathsf{trns}^t & \hat{=} & \lambda d_1 : D \cdot \{d_2 \in D | \exists t \cdot t \in T - T^d \wedge d_2 \in \mathsf{trns}.t.d_1\}\ , \\
\mathsf{trns}^j & \hat{=} & \lambda j_1 : J \cdot \{j_2 \in J | j_2 \in \mathsf{trns}^t.j_1\}\ , \\
\mathsf{trns}^{q,j} & \hat{=} & \lambda q : Q \cdot \{j \in J | j \in \mathsf{trns}^t.q\}\ , \\
\mathsf{trns}^{j,q} & \hat{=} & \lambda j : J \cdot \{q \in Q | q \in \mathsf{trns}^t.j\}\ , \\
\mathsf{trns}^q & \hat{=} & \lambda q_1 : Q \cdot \{q_2 \in Q | q_2 \in (\mathsf{trns}^{q,j}; \mathsf{trns}^{j*}; \mathsf{trns}^{j,q}).q_1 \vee q_2 \in \mathsf{trns}^t.q_1\}\ , \\
\mathsf{trns}^{dq,j} & \hat{=} & \lambda q : Q \cdot \{j \in J | \exists t \cdot t \in T_d \wedge j \in \mathsf{trns}.t.q\}\ , \\
\mathsf{trns}^{dq} & \hat{=} & \lambda q_1 : Q \cdot \{q_2 \in Q | q_2 \in (\mathsf{trns}^{dq,j}; \mathsf{trns}^{j*}; \mathsf{trns}^{j,q}).q_1 \vee \\
 & & \quad \exists t \cdot t \in T^d \wedge q_2 \in \mathsf{trns}.t.q_1\}
\end{array}
$$

For Simulink diagrams, we will need to be able to express constraints in a way that is not dependent on virtual blocks. Virtual subsystems do not affect the semantics of the model and therefore they should not affect the mode-automata constraints. Hence, to give the constraints for Simulink we need to state that a port depends on another regardless of the virtual subsystem hierarchy.

A port depends on another if there is a signal between them or they form a connection over a subsystem boundary. This is expressed by the relation dep:

$$
\begin{array}{lcl}
\mathsf{dep} \hat{=} & \lambda p_1 : P \cdot \{p_2 \in P | p_1 \neq p_2 \wedge (p_1 \in P^i \Rightarrow p_2 = \mathsf{sig}.p_1) \wedge \\
 & (p_1 \in P^o \Rightarrow (\exists b \cdot b \in B^s \wedge (\mathsf{subi}.b.p_1 = p_2 \vee \mathsf{subo}.b.p_1 = p_2)))\}
\end{array}
$$

The function ndep then gives the connections between non-virtual blocks.

$$
\begin{array}{lcl}
\mathsf{ndep} \hat{=} & \lambda p_1 : P \cdot \{p_2 \in P^o | \\
 & \mathsf{blk}.p_1 \notin (B^i \cup B^o \cup B^{vs}) \wedge \mathsf{blk}.p_2 \notin (B^i \cup B^o \cup B^{vs}) \wedge \\
 & p_2 \in \mathsf{dep}^+.p_1 \wedge \\
 & (\forall p \cdot p \in P \wedge p \in (\mathsf{dep}^+.p_1 \cap (\mathsf{dep}^{-1})^+.p_2) \Rightarrow \mathsf{blk}.p \in (B^i \cup B^o \cup B^{vs}))\}
\end{array}
$$

A port is connected to another if there is sequence of signals, virtual subsystems, in-blocks and out-blocks between them. The ndep function is the key to expressing actual connections between blocks in a manner that is not dependent on the virtual subsystem hierarchy.

4.2.2 Stateflow constraints

Stateflow allows transitions that have very complicated semantics. We here give a number of additional constraints to limit the set of legal transitions in order to only use

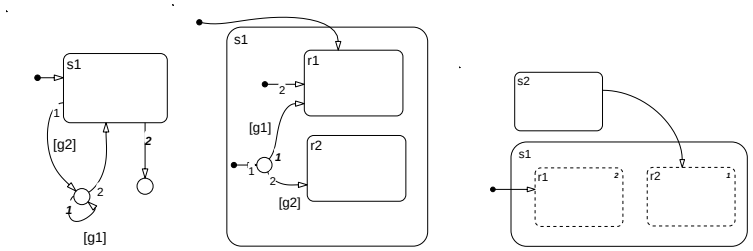


Figure 4: Example of syntactically correct Stateflow models that use constructs not allowed by our mode-automata definition. The chart (a) shows illegal transitions, (b) shows illegal initialisations, while (c) shows illegal transitions to parallel states.

transitions with intuitive and easily verifiable behaviour. A few examples of Stateflow constructs that are not allowed are shown in fig. 4.

The first constraint is that transitions should start and end in states. Otherwise the transition does nothing, since we have no actions on transitions. Transitions containing cycles with only junctions are also not allowed, since this can lead to infinite loops in the chart. Finally, if a transition has the same source and destination, it is unnecessary and therefore not allowed (see fig. 4 (a)).

$$\forall j \cdot j \in J \wedge \exists q \cdot q \in Q \wedge (\mathsf{trns}^{j*}; \mathsf{trns}^{j,q}).j \ ,$$
$$\forall j \cdot j \in J \Rightarrow j \notin \mathsf{trns}^{j+}.j \ ,$$
$$\forall q \cdot q \in Q \Rightarrow q \notin \mathsf{trns}^{q}.q$$

Each or-state should have exactly one default transition. This is stricter than Stateflow, since Stateflow also allows several default transitions or no initialisation at all (see fig. 4 (b)). Furthermore, it should always be possible to execute the initialisation, which can here be guaranteed by only syntactic rules. The condition can be ensured by having one unguarded transition for each junction reached from the default transition segment. Alternatively, for each transition segment with guard g from a junction, there exists another transition segment from the same junction with guard $\neg g$.

$$\begin{array}{l}
\forall q \cdot q \in Q^{or} \Rightarrow \exists t \cdot t \in T^{d} \wedge \mathsf{card}.(\mathsf{trns}.t.q) = 1 \ , \\
\forall q \cdot q \in Q^{or} \Rightarrow \\
\quad \forall j \cdot j \in (\mathsf{trns}^{dq,j}; \mathsf{trns}^{j*}).q \\
\qquad \Rightarrow ((\forall t_1 \cdot t_1 \in T \wedge \mathsf{trns}.t_1.j \neq \emptyset \\
\qquad\quad \Rightarrow \exists t_2 \cdot t_2 \in T \wedge \mathsf{trns}.t_2.j \neq \emptyset \\
\qquad\qquad \wedge t_1 \neq t_2 \wedge \mathsf{lbl}.t_1 = \neg \mathsf{lbl}.t_2) \vee \\
\qquad\qquad (\exists t \cdot t \in T \wedge \mathsf{trns}.t.j \wedge \mathsf{lbl}.t = \epsilon))
\end{array}$$

To enforce creation of more structured models, transitions that cross the boundary of a composite state are not allowed (see fig. 4 (b) and (c)). In order to discover if a transition crosses a composite state boundary, we check that if there is a transition between two states then these two states have the same parent. The parent also has to be an or-state, since transitions between parallel states are not allowed. Default transitions have to lead to a direct sub-state of the state they are initialising.

$$\forall q_1, q_2 \cdot q_1 \in Q \wedge q_2 \in \mathsf{trns}^{q}.q_1 \Rightarrow \mathsf{sfh}.q_1 = \mathsf{sfh}.q_2 \wedge \mathsf{sfh}.q_1 \in Q^{or} \ ,$$
$$\forall q_1, q_2 \cdot q_1 \in Q^{or} \wedge q_2 \in \mathsf{trns}^{dq}.q_1 \Rightarrow q_1 = \mathsf{sfh}.q_2$$

4.2.3 Simulink constraints

Rules for the Simulink part of the model are also needed in order for the model to conform to the mode-automata architecture. The only output from the Stateflow chart is the current activity of the leaf states. Each out-port should be connected to the *enable port*, $p_e \in P^e$, of the enabled subsystem in Simulink that defines the behaviour in that state (mode).

$$\forall p_o \cdot p_o \in P^o \wedge \mathsf{blk}.p_o = \mathcal{S} \Rightarrow \exists p_e \cdot p_e \in P^e \wedge \mathsf{blk}.p_e \in B^e \wedge p_o \in \mathsf{ndep}.p_e$$

To ensure that the mode-dependent behaviour conforms to the mode-automata architecture we need to constrain how outputs of enabled subsystems are used. Each enabled subsystem, b_e, is followed by a *merge block*, b_m. The merge block is used to obtain the latest result from different enabled subsystems connected to it. Exactly one port in the merge block has to be updated regardless of the states (modes) the system is in, otherwise the value of the output signal would be undefined in certain modes. To express this property we give a function that states which subsystems can be enabled by a state q or one of its sub-states.

$$\begin{aligned}\mathsf{stesub} \mathrel{\hat{=}} \; & \lambda q : Q \cdot \{b_e \in B^e | \\ & \exists q_1 \cdot q_1 \in ((\mathsf{sfh}^{-1})^*.q - (Q^{or} \cup Q^{and})) \wedge \\ & \exists p_e \cdot p_e \in P^e \wedge \mathsf{ndep}.p_e = \mathsf{sfprt}.q_1 \wedge \mathsf{blk}.p_e = b_e\}\end{aligned}$$

The set of merge-blocks affected by these states can then be computed.

$$\begin{aligned}\mathsf{stmrg} \mathrel{\hat{=}} \; & \lambda q : Q \cdot \{b_m \in B^m | \exists p_i \cdot p_i \in P_i \wedge \mathsf{blk}.p_i = b_m \wedge \\ & \exists p_o \cdot p_o \in P^o \wedge p_o = \mathsf{ndep}.p_i \wedge \mathsf{blk}.p_o \in \mathsf{stesub}.q\}\end{aligned}$$

This relation is then used to define the set of merge block affected by states q or its sub-states. For a hierarchical Stateflow model we have that every sub-state of an and-state is connected to a different merge block and every sub-state of an or-state are connected to the same merge block. After the signals have been merged, sub-states of and-states updates different resulting signals, while sub-states of or-states updates the same signals.

$$\begin{aligned}&\forall q \cdot q \in Q^{and} \Rightarrow (\forall q_1, q_2 \cdot q_1, q_2 \in (\mathsf{sfh}^{-1}).q \wedge q_1 \neq q_2 \\ &\quad \Rightarrow \mathsf{stmrg}.q_1 \cap \mathsf{stmrg}.q_2 = \emptyset) \,, \\ &\forall q \cdot q \in Q^{or} \Rightarrow (\forall q_1, q_2 \cdot q_1, q_2 \in (\mathsf{sfh}^{-1}).q \wedge q_1 \neq q_2 \\ &\quad \Rightarrow \mathsf{stmrg}.q_1 = \mathsf{stmrg}.q_2)\end{aligned}$$

Furthermore, we need to ensure that an enabled subsystem is connected to a merge block with only one signal.

$$\begin{aligned}&\forall p_1, p_2 \cdot p_1 \in P^i \wedge p_2 \in P^i \wedge p_1 \neq p_2 \wedge \mathsf{blk}.p_1 \in B^m \wedge \mathsf{blk}.p_1 = \mathsf{blk}.p_2 \\ &\Rightarrow \forall p_{11}, p_{22} \cdot p_{11} \in \mathsf{ndep}.p_1 \wedge p_{22} \in \mathsf{ndep}.p_2 \Rightarrow \mathsf{blk}.p_{11} \neq \mathsf{blk}.p_{22}\end{aligned}$$

The rules above, together with the definition of activity in or- and and-states, ensure that exactly one input for each merge-block is enabled at the same time. Note that to simplify the rules above, there can be no multistage merge, i.e. merge blocks connected to merge blocks. This can be ensured by checking that each merge block is connected to an enabled subsystem:

$$\forall p, b \cdot p \in P^i \wedge b \in B^m \wedge \mathsf{blk}.p = b \Rightarrow (\forall p_o \cdot p_o \in \mathsf{ndep}.p \Rightarrow \mathsf{blk}.p_o \in B^e)$$

Memory inside mode enabled subsystems needs to be handled with care. Consider a PI-controller in an enabled subsystem enabled by mode m. When the mode

is switched away from m and then after some time switched back, the integrator can contain a very old value not relevant anymore. This can lead to problems in control applications. Either blocks that contain memory should be avoided in enabled subsystems or the memory should be reset upon activation to ensure predictable behaviour. If memory is avoided, it can also potentially reduce the problem with transients when switching modes. The following rule ensures that no memory is used in enabled subsystems enabled by the Stateflow chart:

$$\forall b \cdot b \in B^{mem} \Rightarrow |\mathsf{subh}|^*.b \cap \mathsf{stesub}.root = \emptyset$$

This can be considered an optional rule, since it restricts the modeling too much to be used in general. However, conformance to this rule can greatly aid verification of the mode switching.

When all these restrictions are satisfied the Simulink / Stateflow model conforms to the mode-automata architecture. These constraints can then be used to simplify the analysis when formally analysing the behaviour of the models.

4.3 Composition of Mode-Automata

Mode-automata allow for two different types of compositions, namely AND-states (parallel compositions) and OR-states (sequential compositions). The sequential composition corresponds to the basic construction mechanism for standard automata. Parallel composition allows for introducing concurrency in a design.

The formal definition of both parallel and sequential compositions has been studied extensively in the literature. An overview can be found in [5]. For saving space, we do not get into the details here. However, we need to be able to express under which conditions our mode-automata architecture is preserved by composing mode-automata using one or the other of these compositions.

We only give an overview of how composition can be performed here. Parallel composition consists of making two state-machines in $\mathcal{M}_A$ and $\mathcal{M}_B$ sub-states of a common and-state. The behaviour of the modes in $\mathcal{M}_A$ and $\mathcal{M}_B$ is orthogonal, which means that enabled subsystems from $\mathcal{M}_A$ cannot update the same merge-blocks as the enabled subsystems from $\mathcal{M}_B$. When performing sequential composition the two models $\mathcal{M}_A$ and $\mathcal{M}_B$ becomes sub-states of a common or-state. The mode-dependent behaviours should now update the same outputs, meaning that they should be connected to the same merge-blocks. We can also give syntactic rules similar to the ones previously presented, in order to describe how the models $\mathcal{M}_A$ and $\mathcal{M}_B$ can be composed to form $\mathcal{M}_C$. These rules can be used to ensure that the mode-automata constraints given earlier are preserved by the composition.

4.4 Validation of the architectural rules

The mode-automata architecture should allow a subset of Simulink / Stateflow that is safe to use and easy to verify, which also allow interesting models to be created. Hence, the rules we have given need to be consistent, i.e., it should be possible to create models satisfying the constraints. They should also be complete, meaning that no undesirable models satisfy the constraints. To investigate if the formalisation of Simulink / Stateflow and our restrictions on the models work as intended, we have studied their properties using the Alloy Analyzer[3] [13]. Alloy is a model-checking tool based on first order logic, where systems can be modeled using relations and constraints on relations. It can then be used to generate models satisfying the constraints and to check validity

[3] Alloy Analyzer, `http://alloy.mit.edu`

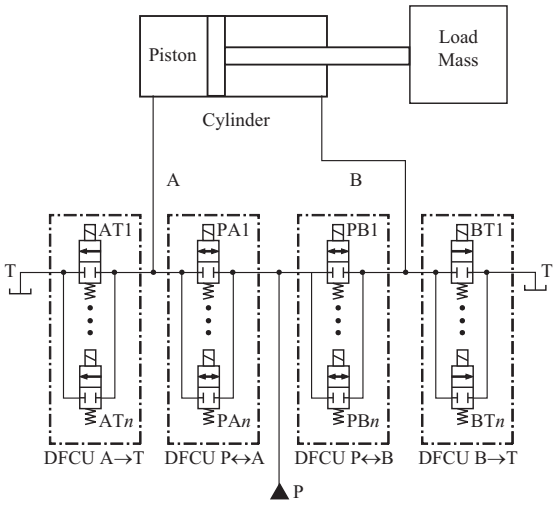


Figure 5: Example of a digital hydraulics system controlled by four DFCUs [18]

of assertions in models. The constraints we have given can be translated to Alloy, since they are actually first order properties. Using this tool we have checked that the constraints are consistent, i.e., there are models that satisfies them. Completeness of the rules depends on the informal description of how we like Simulink to be restricted. By generating models satisfying constraints and checking that undesirable models cannot be generated, confidence in that the formalisation works as intended is gained.

5 Case Study

To investigate the suitability of the model architecture we have tested it on a case study. The case study is a digital hydraulics system with energy saving [4, 17]. In this paper we focus on the architectural rules for Simulink / Stateflow, while the previous two publications describes the algorithms used, as well as design and verification techniques. The aim of digital hydraulics [17, 18] is to use cheap and simple on/off valves instead of expensive and complicated servo- or proportional valves. This has the potential to lead to cheaper, more flexible and robust hydraulics systems. The downside is that digital hydraulics require complicated controllers to achieve good performance. To ensure reliability and performance of such systems adequate software structuring and design methods need to be used.

The system developed in this case study consists of a hydraulic cylinder that moves a load mass either to a desired position or with a desired speed. An overview of the system is shown in fig. 5. The speed of the load mass is controlled by the pressure on each side of the piston in the cylinder (*A*- and *B*-side). The pressures are controlled by opening and closing suitable combinations of valves in the Digital Flow Control Units (DFCU). The system has several running modes for normal motion and energy saving motion. Each mode requires different types of computation. Hence, mode automata is a good solution for structuring the controller.

Model-based design is used to construct the controller in order to be able to investigate the behaviour of the entire system. An overview of the system with a model of the physical system is given in fig. 6. Here the controller is given in the subsystem *Controller*, a model of the valve dynamics is given in *Valves*, the pump is modeled

in *Pump*, the hydraulic cylinder in *Cylinders* and the load of the cylinder is modeled in *Mechanism*. The entire system can then be simulated and the performance of the controller can be evaluated.

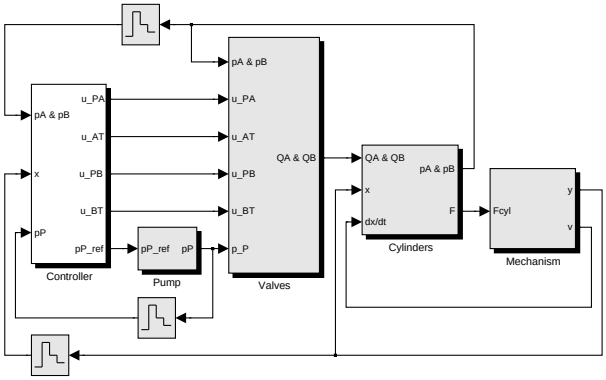


Figure 6: Model based design of a digital hydraulics system in Simulink

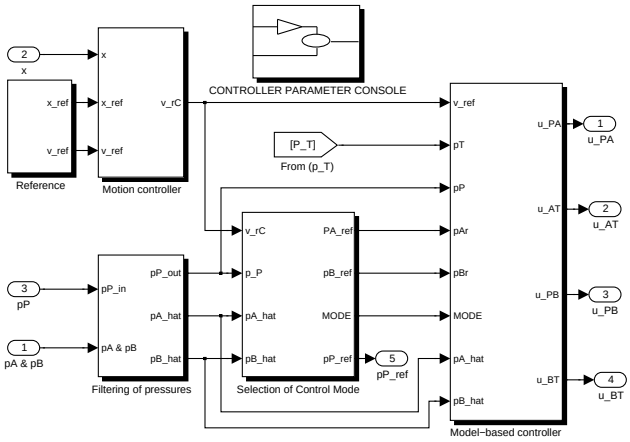


Figure 7: Overview of the controller of the digital hydraulics system

An overview of the controller is given in fig. 7. It is fairly large and complex consisting of over 4000 blocks. The controller has the chamber pressures p_A and p_B, the piston position x and pump pressure p_P as sensor inputs. From these sensor inputs the controller computes the optimal valve configurations u_{PA}, u_{AT}, u_{PB}, u_{BT}, as well as the pump reference pressure p_{Pref}. The mode specific computation is encapsulated in the subsystem *Selection of Control Mode*. The subsystem computes chamber reference pressures p_{Aref} and p_{Bref}, mode specific parameters in $MODE$, and the pump reference pressure, p_{Pref}. Based on this information, the subsystem *Model-based Controller* then computes the optimal valve configuration independently of the mode.

5.1 Mode-automata architecture

The mode selection subsystem shown in fig. 8 has been designed to conform to the mode-automata architecture. The subsystem *Conditions* computes a set of conditions that are used for the mode switching. These conditions are based on the sensor values

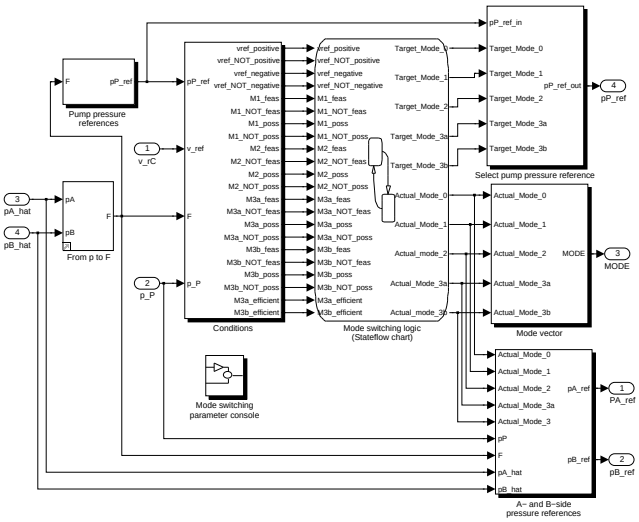


Figure 8: Overview of the mode switching and definition of mode dependent behaviour

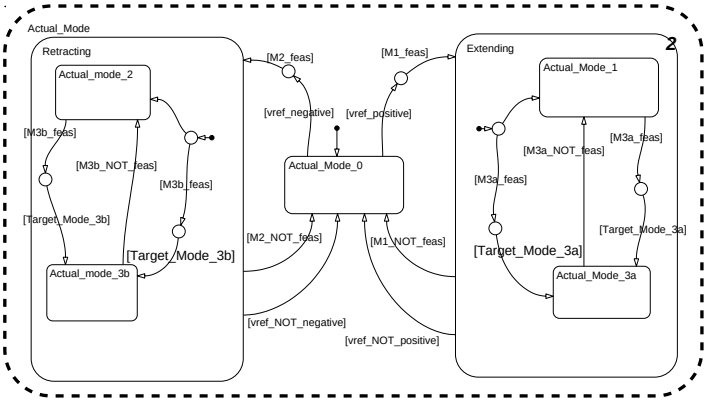


Figure 9: Switching structure of the *actual* mode

read from the environment. Based on the conditions modes are switched in the Stateflow chart. The activity of the leaf states are automatically exported to the Simulink diagram to enable the mode specific behaviour. There are two parallel modes, *target mode* and *actual mode*, in the Stateflow chart. The target mode determines the pump pressure reference, while the actual mode determines the cylinder chamber pressure references and the mode specific parameters.

The actual mode consists of five leaf modes. The switching between them is shown in fig. 9. Target mode is also switched in a similar manner.

0 *Stopped motion*. If the reference speed is close to zero or no other mode is feasible this mode is used;

1 *Normal extending motion*. This mode is selected if the reference speed is greater than a threshold value, the mode is feasible and energy saving should not be used;

2 *Retracting motion*. This mode is similar to mode 1, but it concerns movement

in the opposite direction;

3a *Extending energy saving motion*. If the reference speed is greater than a threshold value and energy saving can be used, this mode is selected;

3b *Retracting energy saving motion*. This mode is similar to mode 3a, but it concerns movement in the opposite direction.

Note that the conditions containing the prefix NOT are not the negation of their corresponding conditions, but contain other additional features to prevent excessive mode switching. Observe also that there are only in-port names as guards on transition segments, and that all transitions follow the mode-automata rules given for Stateflow in Subsection 4.2.

Consider the computation of chamber pressure references, p_{Aref} and p_{Bref}. Each leaf state of *actual mode* is connected to enable ports of enabled subsystems inside the subsystem *A- and B-side pressure references* shown in fig. 10. Every enabled subsystem then computes the value of p_{Aref} and p_{Bref} when enabled. The results from these subsystems are merged according to the mode-automata rules to give the final pressure references.

As is, the merge block accepts that several or none of its inputs to be active at the same time. However this situation needs to be prevented in order to ensure predictability. The architecture proposed here does prevent this: all the inputs of the merge blocks are obtained from enabled blocks whose enabling signals are directly linked to activity of the exclusive (*or*) states of the Stateflow diagram.

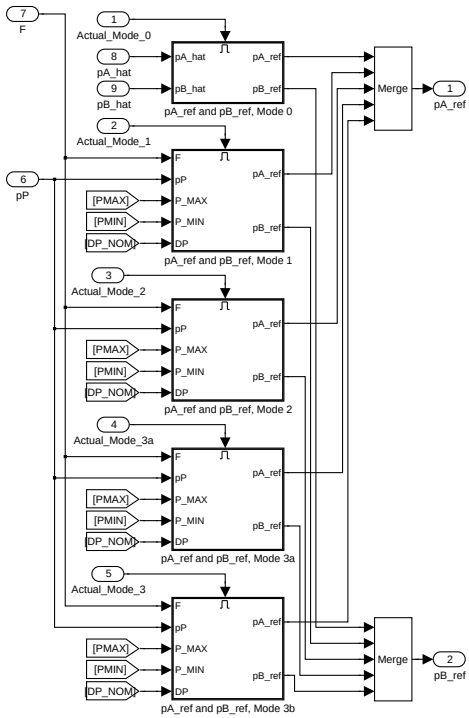


Figure 10: Mode dependent computation of pressure references p_{Aref} and p_{Bref}.

5.2 Lessons learnt

We have used pre- and post-conditions to state correctness constraints for subsystems [4]. By exploiting the structure given by the mode-automata architecture proof obligations for the mode switching system can be derived. This has been done to verify the correctness of the mode selection subsystem [4].

Composition can be used to construct the controller from smaller parts. The controller behaviour of *target mode* and *actual mode* are orthogonal and can be developed separately. The final system is then obtained by the and-composition. The behaviour in actual mode can also be constructed from smaller parts using or-composition.

It seems to us reasonable to use mode-automata for designing *parts* of an application. Imposing them as a top-most architecture is too restrictive. In the case-study, the mode-automaton controls signals p_{Aref}, p_{Bref} and $MODE$. The signals are used to control the "model-based controller" (see fig. 7), which then computes the actual actuator values.

6 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 Simulink / Stateflow to a safe kernel with clear semantics. The aim is to allow enough features for the architecture to be usable in practise, while simplifying the analysis of the models. The mode-automata model architecture provides a structured and maintainable model architecture for mode-based systems. It can also be exploited for validating desirable properties of the controller. Furthermore, we mentioned two methods for composing different mode-automata. In order to validate the formalisation of the architecture, its properties have been investigated with the Alloy Analyzer. This enabled creation of the complex constraints needed, while still ensuring that they are consistent and adequate. The case study showed that the architecture is suitable for developing complex controllers and aid their verification [4].

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. Similar approaches have been used for UML diagrams [23]. Another solution would consists, as pointed out earlier, in implementing a conformance rules checker using the Matlab scripting language.

More generally, 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 Simulink / Stateflow taking advantage of the mode-automata architecture and the formalisation of the structure. This will be done by first expressing the semantics of the considered subset of Simulink / Stateflow in the refinement calculus [2] in order to benefit from this framework. Ultimately, this will give us strong formal support for stepwise refinement of Simulink / Stateflow models.

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 Simulink / Stateflow models. These techniques will contain, among other things local specification in the form of assume-guarantee contracts [4, 19] and compositional verification rules [1, 4].

Simulink / Stateflow has become a popular tool for model-based design of control systems. The architecture of the Simulink / Stateflow designs is important in order to ensure that the constructed control system is maintainable, reliable, and the logic of it is easy to follow. Mode-automata is such an architecture for separating control

and signal processing in mode-based systems. The mode-automata architecture also simplifies verification, since models are guaranteed to have a certain simple structure.

References

[1] M. Abadi and L. Lamport. Composing specification. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.

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

[3] M. Beine, U. Eisemann, and R. Otterbach. Transforming a control design model into an efficient production application. In *Proceedings of the 2006 IEEE Conference on Computer Aided Control System Design*, pages 3019–3023, 2006.

[4] P. Boström, M. Linjama, L. Morel, L. Siivonen, and M. Waldén. Design and validation of digital controllers for hydraulics systems. In *The 10th Scandinavian International Conference on Fluid Power*, volume 1, pages 227–241, Tampere, Finland, 2007.

[5] P. Boström and L. Morel. Mode-automata in Simulink/Stateflow. Technical Report 772, Turku Centre for Computer Science, 2006.

[6] 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.

[7] 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.

[8] U. Eisemann. Modeling guidelines for function development and production code generation. Distributed on the occasion of Embedded World Conference 2006, `http://www.dspace.de/ftp/papers/dspace_embWorld_0602_e_p344.pdf`, 2006.

[9] Esterel Technologies. SCADE, `http://www.esterel-technologies.com/products/scade-suite/`, 2006.

[10] 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.

[11] 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.

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

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

[14] 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.

[15] Japan Mathworks Automotive Advisory Board (J-MAAB). Simulink styleguide. `http://www.embeddedcmmi.at/index.php?id=10`, 2003.

[16] 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.

[17] M. Linjama, M. Huova, P. Boström, A. Laamanen, L. Siivonen, L. Morel, M. Waldén, and M. Vilenius. Design and implementation of energy saving digital hydraulic control system. In *The 10th Scandinavian International Conference on Fluid Power*, volume 2, pages 341–359, Tampere, Finland, 2007.

[18] M. Linjama and M. Vilenius. Improved digital hydraulic tracking control of water hydraulic cylinder drive. *International Journal of Fluid Power*, 6(1):29–39, 2005.

[19] 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.

[20] 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.

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

[22] Mathworks Automotive Advisory Board (MAAB). Controller style guidelines for production intent using Matlab, Simulink and Stateflow. `http://www.mathworks.com/matlabcentral/fileexchange/`, 2001.

[23] I. Porres. A toolkit for model manipulation. *Software and Systems Modeling*, 2(4):262–277, 2003.

[24] I. Porres. Rule-based update transformations and their application to model refactorings. *Software and Systems Modeling*, 4(4):368–385, 2005.

[25] C. Puchol, D. Stuart, and A. Mok. An operational semantics and compiler for real-time specifications. *Integrated Computer-Aided Engineering*, 5(3):187–206, 1998.

[26] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a "safe" subset of Simulink/Stateflow into Lustre. In *Proceedings of the 4th ACM international conference on Embedded software*, pages 259–268. ACM Press, 2004.

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

[28] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating discrete-time Simulink to Lustre. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(4):779–818, 2005.

Lemminkäisenkatu 14 A, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
- Institute of Information Systems Sciences

ISBN 978-952-12-1922-1
ISSN 1239-1891