HLang: Compositional Representation of Hybrid Systems via Predicates

by

Martin Fränzle Hardi Hungar Christian Schmitt Boris Wirtz

AVACS Technical Report No. 20
July 2007
ISSN: 1860-9821

Publisher: Sonderforschungsbereich/Transregio 14 AVACS
(Automatic Verification and Analysis of Complex Systems)
Editors: Bernd Becker, Werner Damm, Martin Fränzle, Ernst-Rüdiger Olderog,
Andreas Podelski, Reinhard Wilhelm
ATRs (AVACS Technical Reports) are freely downloadable from `www.avacs.org`

HLang: Compositional Representation of Hybrid Systems via Predicates

Martin Fränzle* Hardi Hungar† Christian Schmitt* Boris Wirtz‡

April 2005 – July 2007

Given the structural diversity of modeling paradigms for hybrid systems that are in widespread use, which ranges from a signal-transducer-based view like in Simulink/Stateflow to an automaton-based view in various hybrid automata frameworks, there is need for an intermediate-level language that can accommodate these structures in a concise way, thereby isolating verification backends from the particular modeling frontend used (cf. Fig. 1). This technical report provides a description of the intermediate-level modeling language for hybrid systems used in project area H of AVACS as a gateway to verification backends. The lnguage is distinctly different from many other proposals in the literature, e.g. HSIF as proposed by the MoBIES consortium [HSI, MoB]. The reasons for such deviations need to be well-argued. Basically, our reasons coincide with the analysis results obtained by the COLUMBUS project [Lea04] in their strife for finding a suitable hybrid system interchange format:

> "HSIF is the obvious candidate for recommendation. However, we believe that some actions must be taken to adjust HSIF or to adopt another approach since some limitations to its semantics make the interchange of data between foreign tools difficult (for example, HSIF does not support some of the features of Simulink/Stateflow model).
>
> [. . .] As we argued before, HSIF does not support some of the models of important tools and it does not allow hierarchical representations. In our opinion, HSIF is an excellent model for supporting clean design of hybrid systems but not yet a true interchange format. Simulink/Stateflow internal format could be a de facto standard but it is not open nor does it have features that favor easy import and export.
>
> Modelica has full support of hierarchy and of general semantics that subsumes most if not all existing languages and tools. As such, it is indeed an excellent candidate but it is not open.
>
> On the other hand, the Metropolis Meta-Model (MMM) has generality and can be used to represent a very wide class of models of computation. It has a clear separation between communication and computation as well as architecture and function. However, we have limited experience (if any) with the use of the meta-model to represent hybrid system control. We have plans to extend the environment and the model to cover adequately continuous time systems. The meta-model itself is perfectly capable to express continuous time systems. However, there is no tool that can manage at this time this information in Metropolis.
>
> In conclusion, we believe that a full fledged recommendation at this time is premature given the state of maturity of the available approaches."
>
> (cited from [Lea04])

This state of affairs forced us to develop an own format satisfying the following demands:

- it has to accommodate the plethora of modeling paradigms for hybrid systems in order to be able to draw benchmarks and case studies from a variety of sources,

*Carl von Ossietzky Universität Oldenburg, Faculty II, Dpt. of Informatics, Research Group Hybrid Systems

†Kuratorium OFFIS e.V., Oldenburg

‡Carl von Ossietzky Universität Oldenburg, Faculty II, Dpt. of Informatics, Research Group Safety-Critical Embedded Systems

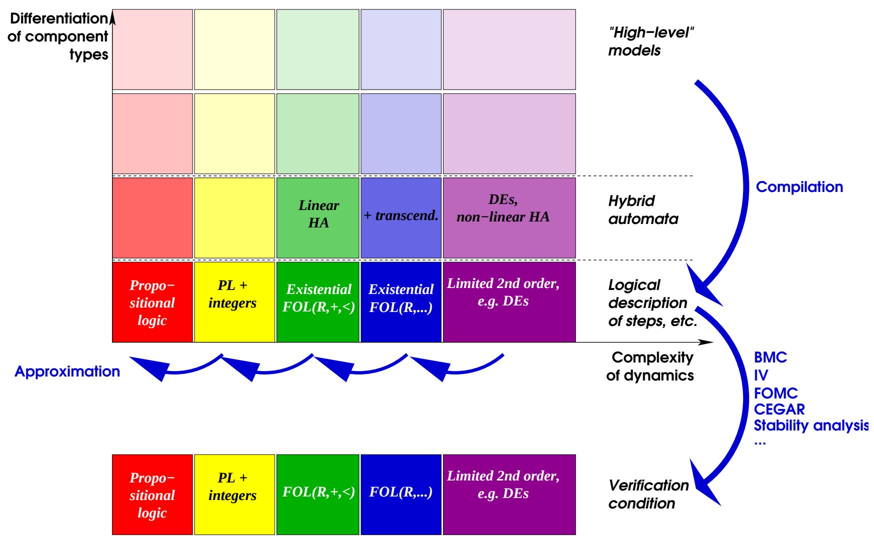


Figure 1: Layered approach to hybrid system modeling — using a concise and compositional intermediate-level model, model transformations as well as verification backends can be isolated from the pecularities of the structurally diverse high-level modeling paradigms.

- it should do so concisely, providing a linear-size encoding for the relevant paradigms in order to prevent "model explosion",
- the encodings need to be compositional, thus
  1. reducing the computational complexity of computing the encodings and
  2. providing the basis for compositional analysis and verification methods,
- its semantics need to be unambiguously defined,
- its vocabulary should be logic-based in order to simplify interfacing to the decision procedures constituting the verification backends.

This manuscript presents such a model, as developed within AVACS as result of intense cross-project discussions.

Structure of the paper. In the next section, we will define the general predicative model, while later sections detail the encodings of well-known hybrid system's models into the predicative model. In particular, Section 2 presents an embedding of flat hybrid automata and demonstrates compositionality of that encoding by providing a linear-size syntactic counterpart of the product construction. Section 3 provides an encoding of the most relevant Simulink and Stateflow blocks. Section 4 describes the concrete syntax of the language realizing the abstract model. Appendix A, finally, provides an overview over an implementation of a translator for Simulink/Stateflow models.

1 The model

Our overall model uses predicates to describe the possible behavior of a hybrid system. For the sake of convenience, these predicates are structured into different groups: a system is described in terms of an initiation predicate constraining the possible initial states, a flow predicate describing the permissible continuous actions, and a jump predicate defining the possible jumps (= instantaneous transitions).

1.1 Flows and jumps

Following the tradition of the domain, we draw a distinction between "jumps" and "flows". A *flow* is a continuous and differentiable action over time which leads to a continuous change of the *continuous variables*[1] of the system. It leaves all other variables unaffected. A *jump*, in contrast, is an instantaneous action that may exert discontinuous changes to all kinds of variables. In order to model this, we assume that our system operates on two disjoint sets of variables:

1. a set CV of *continuous variables*, with each variable $x \in CV$ being interpreted over an associated domain D_x, which can be the reals $\mathbb{R}$, the complex numbers $\mathbb{C}$, or subranges thereof —i.e., open, halfopen, or closed intervals in $\mathbb{R}$ or $\mathbb{C}$. It does also make sense to allow x to be of vector type, i.e. have $D_x = \mathbb{R}^n$ or $D_x = \mathbb{C}^n$ or D_x a vector space over some subrange thereof, yet we will not elaborate on this —except for the treatment of ill-formed vector subscriptions— straightforward extension here.

2. a set DV of *discrete variables* which is disjoint from CV. Each $v \in DV$ is interpreted over an associated domain D_v. D_v can be finite or infinite (even uncountably so). In general, we may encounter enumeration types, the Boolean values, the integers or subranges thereof, the reals or sub-intervals thereof, or the complex numbers as domains of discrete variables. The term "discrete variable" thus does refer to the mode of variable update, not to the type of value domain.

These two sets of variables constitute the *state space* of the system.

1.1.1 Flow constraints

We describe flows by first-order predicates over the continuous variables and their derivatives. We call such a predicate a *flow constraint*. In more detail, a flow constraint is a predicate in FOL($\mathbb{R}, <, +, \ldots$) (the set of function and predicate symbols in the signature may vary depending on the application) over the following variables

- undecorated continuous variables $x \in CV$ denoting the *current value of continuous variable x along* a continuous flow,
- decorated continuous variables $\overleftarrow{x}$, for $x \in CV$, denoting the *value of x when entering the current mode*,
- decorated continuous variables $\overrightarrow{x}$, for $x \in CV$, denoting the *value of x when leaving the current mode*,[2]
- decorated continuous variables $\frac{\mathrm{d}x}{\mathrm{d}t}$, for $x \in CV$, denoting the *current derivative* over time of x (which may degenerate to a left or right derivative upon jumps).

Typical examples of flow constraints are

A differential equation: $\frac{\mathrm{d}x}{\mathrm{d}t} = v \wedge \frac{\mathrm{d}v}{\mathrm{d}t} = a$

An algebraic equation: $a = \frac{F}{m} \wedge F = -kx - 9.81m$

A conservation law: $E_{\text{kin}} + E_{\text{pot}} = \overleftarrow{E}_{\text{kin}} + \overleftarrow{E}_{\text{pot}}$

An invariant: $x^2 + y^2 < 1$

[1]Note that the term "continuous variable" is a bit fuzzy in the domain. It does neither imply that these variables are confined to a continuous evolution, as they may well be subjected to discontinuous updates, nor does it refer to the whole vector of continuously-valued variables, as some of these may be owned by (in the sense of their values being controlled by) discretely evolving subsystems, thus not being affected by any flow. It is generally considered redundant to mention these variables within flows.

[2]Note that flow predicates will be interpreted by continuous functions such that it is often redundant to refer to the initial or final values. E.g., $x = y$ along the flow implies $\overleftarrow{x} = \overleftarrow{y}$ and $\overrightarrow{x} = \overrightarrow{y}$. Hence, the decorated variables will not flood the model; they will only be used under special circumstances.

Please note that such constraints, if taken from physics, often lack any notion of free (= input) variable and dependent (= output) variable, which makes them awkward —if not impossible— to model compositionally in many hybrid automata frameworks. Simple examples are the algebraic equation $a = \frac{F}{m}$ above from Newtonian mechanics, where there is no directed causality between force F and acceleration a, or in the same vein Ohm's law $R = \frac{U}{I}$, where the relation between voltage U and current I is an undirected relation. This is our primary reason for not equipping our variable space $CV \cup DV$ with a classification into input and output variables.

Flow constraints are interpreted by differentiable valuations of the continuous variables over the continuous time domain $\mathit{Time} = \mathbb{R}_{\geq 0}$ or a closed prefix interval $[0, e]$, with $e \in \mathit{Time}$, thereof. I.e., the set of interpretations is the set Flow of functions of type $\mathit{Time} \overset{\text{part.}}{\to} CV \overset{\text{tot.}}{\to} \bigcup_{x \in CV} D_x$ such that for each $f \in \mathit{Flow}$

- f is well-typed, i.e. for each $t \in \operatorname{dom} f$ and each $x \in CV$ we have $f(t)(x) \in D_x$,
- the domain of f is a closed interval in Time starting at 0, i.e. $\operatorname{dom} f = \mathit{Time}$ or $\operatorname{dom} f = [0, e]$ for some $e \in \mathit{Time}$,
- for each $x \in CV$, the function $f_x = t \mapsto f(t)(x)$ is differentiable inside $\operatorname{dom} f$.

We denote the derivative of f_x in time instant t by $\dot{f}_x(t)$. In case of a so-called *point flow*, i.e. a flow f with $\sup \operatorname{dom} f = 0$, the derivative is undefined.

We say that a flow constraint ϕ is satisfied by a *non-point* flow f, denoted $f \models \phi$, iff $\rho_{f(t)} \models_{\mathrm{FOL}(\mathbb{R},<,+,\ldots)} \phi$ for each $t \in (0, \sup \operatorname{dom} f)$, where $\rho_{f(t)}$ is the interpretation

$$\rho_{f(t)}(v) = \begin{cases} f(v)(t) & \text{iff } v \in CV \\ \dot{f}_x(t) & \text{iff } v = \frac{\mathrm{d}x}{\mathrm{d}t} \text{ for some } x \in CV \\ f(x)(0) & \text{iff } v = \overleftarrow{x} \text{ for some } x \in CV \\ f(x)(\sup \operatorname{dom} f) & \text{iff } v = \overrightarrow{x} \text{ for some } x \in CV \end{cases}$$

of continuous variables and their decorated variants. Note that the current values v of variables are interpreted in the interval $t \in (0, \sup \operatorname{dom} f)$, skipping the start and end points of the flow. This adds to expressiveness: invariants covering the start/end point also can always be formulated by use of $\overleftarrow{v}$ or $\overrightarrow{v}$; however, start/end values and inner values can be related through non-reflexive relations only if inner values are distinct from the start/end values, which is enforced by interpreting inner values only on $(0, \sup \operatorname{dom} f)$.

If f happens to be a *point flow*, i.e. if $\sup \operatorname{dom} f = 0$, then $f \models \phi$, iff $\rho \models_{\mathrm{FOL}(\mathbb{R},<,+,\ldots)} \phi$ for some $\rho : CV \cup DV \cup \frac{\mathrm{d}DV}{\mathrm{d}t} \cup \overleftarrow{CV} \overset{\text{tot.}}{\to} V$ with

$$\rho(v) = \begin{cases} f(v)(0) & \text{iff } v \in CV \\ f(x)(0) & \text{iff } v = \overleftarrow{x} \text{ for some } x \in CV \\ f(x)(0) & \text{iff } v = \overrightarrow{x} \text{ for some } x \in CV \end{cases}$$

and $\rho(\frac{\mathrm{d}x}{\mathrm{d}t}) \in \mathbb{R}$ arbitraily chosen for each $x \in CV$. I.e., a point flow satisfies a flow constraint if there exists an arbitrary valuation of the derivatives (which are undefined on point flows) such that the valuations of the genuine variables (which are determined by the point flow) together with that free-floating valuation of the derivatives satisfies the constraint.

In order to avoid anomalies in parallel composition, we will usually use flow constraints that feature a certain form of stutter invariance, namely that stopping the flow for a τ jump (i.e., a jump not changing state) and restarting it from fresh thereafter has no impact on the trajectory. Formally, a flow constraint ϕ is called *stutter-invariant* iff models of ϕ can be chopped arbitrarily without obtaining a non-model and, vice versa, concatenation of models yields models again. I.e., we call ϕ stutter-invariant iff $(f \models \phi) \iff ((g \models \phi) \wedge (h \models \phi))$ holds for all $f, g, h \in \mathit{Flow}$ with $\sup \operatorname{dom} g < \infty$ and $g(\sup \operatorname{dom} g) = h(0)$ and $g'(\sup \operatorname{dom} g) = h'(0)$ and

$$f(t) = \begin{cases} g(t) & \text{iff } t \leq \sup \operatorname{dom} g, \\ h(t - \sup \operatorname{dom} g) & \text{iff } t \geq \sup \operatorname{dom} g. \end{cases}$$

Note that the property of stutter-invariance is in general undecidable, as it requires a solution to the differential (in-)equations in ϕ as well as constraint solving over very rich signatures. It is thus a high price to be paid for a simple definition of parallel composition, which stutter invariance helps to achieve. There are, however, special cases where stutter invariance is immediately obvious: if the flow predicate ϕ does neither refer to the pre-state nor to the post-state, i.e. does not contain occurrences of decorated variables $\overleftharpoon{x}$ or $\overrightharpoon{x}$, then ϕ necessarily is stutter-invariant. This applies for, e.g., flow constraints that do only contain differential (in-)equations.

1.1.2 Anchored flow constraints

A characteristic property of hybrid systems is that the actual dynamics of the continuous variables depends on the current discrete state. In order to permit modeling of such an interdependency, we introduce "anchored flow constraints" which combine flows with a predicate over the current discrete state, thus anchoring the individual flows. I.e., an *anchored flow constraint* is a predicate in $\mathrm{FOL}(\mathbb{R}, <, +, \ldots)$ over

- undecorated continuous variables $x \in CV$ denoting the *current value of continuous variable* x,
- decorated continuous variables $\overleftharpoon{x}$, for $x \in CV$ denoting the *value of* x *when entering the current mode*,
- decorated continuous variables $\overrightharpoon{x}$, for $x \in CV$ denoting the *value of* x *when leaving the current mode*,
- decorated continuous variables $\frac{\mathrm{d}x}{\mathrm{d}t}$, for $x \in CV$ denoting the *current derivative* over time of x,
- undecorated discrete variables $c \in DV$ reflecting the *current discrete state.*

An example of an anchored flow constraint is

$$(\underbrace{\mathit{trafficlight}}_{\in DV} = \text{yellow} \wedge \underbrace{v}_{\in CV} > 0) \implies \underbrace{a}_{\in CV} = -3$$

which makes the current acceleration a dependent on both the current speed v —a continuous variable— and the current state of the traffic light, which is a discrete variable.

We say that an anchored flow constraint is *simple* iff it can be transformed to a semantically equivalent anchored flow constraint of the form $\bigwedge_{i=1}^{n} \psi_i \Rightarrow \phi_i$, where the ψ_i do only contain discrete variables and the ϕ_i are flow constraints. Note that simple constraints have a more limited dependency between discrete and continuous variables than general anchored flow constraints: the latter may well contain (in-)equations between discrete and continuous variables; a simple constraint would only allow an encoding of those if the domain of the entailed discrete variable is finite. We call an anchored flow constraint *stutter-invariant* iff it simple and the ϕ_i in its transformation $\bigwedge_{i=1}^{n} \psi_i \Rightarrow \phi_i$ are stutter-invariant flow constraints.

In order to interpret anchored flow constraints, we need to take both continuous and discrete variables into account. This leads to generalized flows which do interpret both continuous variables $x \in CV$ and discrete variables $v \in DV$. The discrete variables are, however, constant along an uninterrupted flow. I.e. the set *GFlow* of generalized flows is the set of functions $f : \mathit{Time} \overset{\text{part.}}{\to} (CV \cup DV) \overset{\text{tot.}}{\to} \bigcup_{v \in CV \cup DV} D_v$ such that

- f is well-typed, i.e. for each $t \in \operatorname{dom} f$ and each $x \in CV \cup DV \cup EV$ we have $f(t)(x) \in D_x$,
- the domain of f is a closed interval in *Time* starting at 0, i.e. $\operatorname{dom} f = \mathit{Time}$ or $\operatorname{dom} f = [0, e]$ for some $e \in \mathit{Time}$,
- for each $x \in DV$, the function $f_x = t \mapsto f(t)(x)$ is constant on $\operatorname{dom} f$,
- for each $x \in CV$, the function $f_x = t \mapsto f(t)(x)$ is differentiable inside $\operatorname{dom} f$.

An anchored flow constraint $\mathcal{F}$ is satisfied by a generalized flow g with $\sup\operatorname{dom} g > 0$ (i.e., a non-point flow), denoted $g \models \mathcal{F}$, iff $\eta_{g(t)} \models_{\mathrm{FOL}(\mathbb{R},<,+,\dots)} \mathcal{F}$ for each $t \in \operatorname{dom} f$, where $\eta_{g(t)}$ is the interpretation

$$\eta_{g(t)}(v) = \begin{cases} f(v)(t) & \text{iff } v \in CV \cup DV \\ \dot{f}_x(t) & \text{iff } v = \frac{\mathrm{d}x}{\mathrm{d}t} \text{ for some } x \in CV \\ f(x)(0) & \text{iff } v = \overleftharpoon{x} \text{ for some } x \in CV \\ f(x)(\sup\operatorname{dom} f) & \text{iff } v = \overrightharpoon{x} \text{ for some } x \in CV. \end{cases}$$

As before, point flows permit arbitrary valuation of derivatives. Therefore, a generalized flow g with $\sup\operatorname{dom} g = 0$ satisfies $\mathcal{F}$ iff $\eta \models_{\mathrm{FOL}(\mathbb{R},<,+,\dots)} \mathcal{F}$ for some $\eta : DV \cup CV \cup DV \cup \frac{\mathrm{d}DV}{\mathrm{d}t} \cup \overleftharpoon{CV} \overset{\text{tot.}}{\to} V$ with

$$\eta(v) = \begin{cases} f(v)(0) & \text{iff } v \in CV \cup DV \\ f(x)(0) & \text{iff } v = \overleftharpoon{x} \text{ for some } x \in CV \\ f(x)(0) & \text{iff } v = \overrightharpoon{x} \text{ for some } x \in CV \end{cases}$$

and $\eta(\frac{\mathrm{d}x}{\mathrm{d}t}) \in \mathbb{R}$ for each $x \in CV$. I.e., a point flow satisfies an anchored flow constraint if there exists an arbitrary valuation of the derivatives such that the flow's valuations of the genuine variables together with that free-floating valuation of the derivatives satisfies the constraint.

1.1.3 Jump constraints

Jumps (= transitions) may involve the manipulation of all kinds of state variables within an instantaneous assignment. They are thus formalized by a first-order predicate over, first, pre-states comprising both discrete and continuous variables and , second, post-states comprising both discrete and continuous variables. Such a *jump constraint* is a FOL($\mathbb{R}, <, +, \dots$) predicate with free variables from

- the undecorated discrete and continuous variables $v \in DV \cup CV$ denoting the *post-transition value of discrete variable* v,
- decorated discrete and continuous variables $\overleftharpoon{v}$, for $v \in DV \cup CV$, denoting the *value of* v *prior to the transition.*

An example of a jump constraint is

$$\underbrace{(\overleftharpoon{\mathit{trafficlight}} = \text{yellow} \wedge \overleftharpoon{v} > 0)}_{\text{guard}} \implies \underbrace{a = -3}_{\text{effect}}$$

which relates the pre-state of the traffic light and the cars velocity v to the acceleration a to be selected. Note that the state of the traffic light is not constrained in the post-state, reflecting the fact that the above jump, which is deemed to be part of the car's control system, does not control the traffic light. Vice versa, jumps in the traffic light controller would leave car dynamics (i.e., acceleration a and velocity v, a.o.) unconstrained, reflecting the lack of direct control over these entities.

In hybrid systems, jumps connect phases of continuous flow. Hence, jump constraints are interpreted by pairs of generalized flows $(g, g') \in \mathit{GFlow} \times \mathit{GFlow}$. A flow pair (g, g') satisfies a jump constraint $\mathcal{J}$, denoted $(g, g') \models \mathcal{J}$, iff $\sup\operatorname{dom} g < \infty$ and $\rho_{(g,g')} \models_{\mathrm{FOL}(\mathbb{R},<,+,\dots)} \mathcal{J}$, where

$$\rho_{(g,g')}(v) = \begin{cases} g'(0)(v) & \text{iff } v \in CV \cup DV, \\ g(\sup\operatorname{dom} g)(x) & \text{iff } v = \overleftharpoon{x} \text{ for some } x \in CV \cup DV. \end{cases}$$

This guarantees that the final state of g (i.e., the state reached in time instant $\sup\operatorname{dom} g$) and the initial state of g' (i.e., the state g' visits in time instant 0) are in the relation defined by the jump constraint.

1.2 Initial states

The set of possible initial states is described by an *initiation constraint*, which is a FOL($\mathbb{R}, <, +, \dots$) formula over $CV \cup DV$. A generalized flow g satisfies an initiation constraint $\mathcal{I}$ iff $g(0) \models_{\mathrm{FOL}(\mathbb{R},<,+,\dots)} \mathcal{I}$.

1.3 Global behavior

A *system description* is a seven-tuple entailing a finite set CV of continuous variables, a finite set DV of discrete variables with $CV \cap DV = \emptyset$, type information D for the aforementioned variables, an initiation constraint $\mathcal{I}$, an anchored flow constraint $\mathcal{F}$, and a jump constraint $\mathcal{J}$. A (finite or infinite) *run* of such a system is a finite or infinite sequence[3] $t = \langle g_0, g_1, \ldots \rangle$ of generalized flows such that

- *Initiation:* $g_0 \models \mathcal{I}$,
- *Flow consistency:* $g_i \models \mathcal{F}$ for each $i \leq \operatorname{len} t$,
- *Jump consecution:* $(g_i, g_{i+1}) \models \mathcal{J}$ for each $i < \operatorname{len} t$.

I.e., a run starts in a permissible initial state; it then engages in a sequence of generalized flows each of which is permissible wrt. the flow constraint, while the chaining of the flows is mediated by the jump constraint.

2 Encoding hybrid automata

We will now demonstrate how to encode standard hybrid automata in the model. There are various superficially similar, yet different in detail, definitions of hybrid automata. We adopt a slight generalization of the definition used by Lygeros [Lyg03], albeit make it more concrete where necessary for constructing a concrete embedding. In our context a (flat) hybrid automaton $A = (Q, X, \mathit{Init}, f, \mathit{Inv}, E, G, R)$ consists of

- a finite collection Q of discrete states,
- a finite collection X of continuous variables, giving rise to valuations $V : X \to \mathbb{R}$,
- a subset $Y \subset X$ of continuous state variables being controlled by the automaton, interpreted by valuations $W : Y \to \mathbb{R}$,
- a set $\mathit{Init} \subseteq Q \times W$ of initial states, here assumed to be defined by a family $(I_q)_{q \in Q}$ of first-order predicates over Y,
- a mapping $f : Q \times V \to W$ assigns a vector field to each discrete state $q \in Q$,
- a mapping $\mathit{Inv} : Q \to 2^W$ assigning a first-order-definable invariant J_q to each discrete state $q \in Q$,
- a collection $E \subseteq Q \times Q$ of discrete transitions,
- $G : E \to 2^V$ assigns to each $e \in E$ a first-order-definable guard G_e, and
- $R : E \times V \to W$ assigns an assignment function to each transition $e \in E$.

For each $q \in Q$ and each $y \in Y$, let $t_{q,y}$ be a term with free variables X such that $f(q,u)(y) = [\![t_{q,y}]\!](u)$ for each $u \in V$. Similarly, for each $e \in E$ and each $y \in Y$, let $r_{q,y}$ be a term with free variables X such that $R(e,u)(y) = [\![r_{e,y}]\!](u)$ for each $u \in V$.

Then the automaton can be embedded into our model as follows

1. we take a single discrete variable *loc* of finite type $D_{loc} = Q$,
2. the set of continuous variables is X, with the type D_x of each continuous variable x being $\mathbb{R}$,
3. the initiation constraint is $\bigvee_{q \in Q} loc = q \wedge I_q$,
4. the flow constraint is
$$\bigwedge_{q \in Q} loc = q \implies \left(J_q \wedge \bigwedge_{y \in Y} \frac{\mathrm{d}y}{\mathrm{d}t} = t_{q,y} \right) ,$$
and

[3] An extension to transfinite sequences is straightforward and makes sense if Zeno behaviors with well-defined limit points are present.

5. the jump constraint is

$$\bigwedge_{e=(q,q')\in E} (\overleftarrow{loc} = q \wedge loc = q') \implies (\overleftarrow{G_e} \wedge \bigwedge_{y\in Y} y = r_{e,y}[\overleftarrow{X}/X]) \ .$$

Note that the flow constraint does not mention decorated variables. This, together with its syntactic form, enforces that its is simple and stutter-invariant.

2.1 Parallel composition

The usual[4] interleaving-style parallel composition of n hybrid automata $A_i = (Q_i, X_i, Init_i, f_i, Inv_i, E_i, G_i, R_i)$ with $i \leq n$ can be defined via a product construction. We will show that this corresponds to union of variable sets, conjunction of initiation constraints, conjunction of anchored flow constraints, and disjunction of appropriately framed jump constraints in our description. Thus, our model permits a linear-size, compositional representation of the interleaving product of hybrid automata, avoiding the blow-up of the product construction.

We start from defining the classical interleaving product: given the n automata $A_i = (Q_i, X_i, Init_i, f_i, Inv_i, E_i, G_i, R_i)$, their *interleaving product* is the automaton $A = (Q, X, Init, f, Inv, E, G, R)$ with

- discrete state space $Q = \prod_{i=1}^{n} Q_i$,
- continuous variables $X = \bigcup_{i=1}^{n} X_i$,
- controlled continuous variables $Y = \bigcup_{i=1}^{n} Y_i$,
- initial state set $Init = \{((q_1, \ldots, q_n), y) \in Q \times V \mid y \models \bigwedge_{i=1}^{n} I_{q_i}\}$,
- the mapping $f(q,u)(y) = f_i(q, u|_{X_i})$ if $y \in Y_i$ assigns a vector field to each discrete state $q \in Q$,[5]
- a mapping $Inv : Q \to 2^V$ assigning a the first-order-defined invariant $J_{(q_1,\ldots,q_n)} = \bigwedge_{i=1}^{n} (J_i)_{q_i}$ to each discrete state $(q_1, \ldots, q_n) \in Q$,
- the collection $E = \bigcup_{i=1}^{n} Id_1 \times \ldots \times Id_{i-1} \times E_i \times Id_{i+1} \ldots \times Id_n$ of discrete transitions, where Id_i is the identity mapping on Q_i,
- $G : E \to 2^V$ assigns to $e \in E$ the first-order-definable guard $(G_i)_{(q_i,q_i')}$ if $e = ((q_1, \ldots, q_n), (q_1', \ldots, q_n'))$ is the identity mapping on all except the ith component (i.e., $q_j = q_j'$ for all $j \neq i$), and
- $R : E \times V \to V$ assigns the assignment function $(R_i)_{(q_i,q_i')}$ to $e \in E$ if $e = ((q_1, \ldots, q_n), (q_1', \ldots, q_n'))$ is the identity mapping on all except the ith component (i.e., $q_j = q_j'$ for all $j \neq i$). !!!!!!!!!!!!!!!!!!!!!!

Now, let $(CV_i, DV_i, \mathcal{I}_i, \xi_i, \psi_i)$ be the encodings for the automata A_i in our model obtained through the encoding technique explained in the previous section, where where CV_i denotes the continuous variable sets, $DV_i = \{loc_i\}$ is a the one-element set of discrete variables, ξ_i is a stutter-invariant anchored flow constraint, and psi_i is a system of jump constraints. W.l.o.g., we assume that the names of the location variables loc_i are distinct for the different automata A_i. Likewise, let $(CV, DV, \mathcal{I}, \xi, \psi)$ be the encoding of the product automaton A.

As an alternative to computing the product and encoding it, we can build a model $(CV', DV', \mathcal{I}', \xi', \psi')$

[4]In the domain of hybrid automata

[5]Note that this implies that the vector fields $f_i(.,.)(y)$ and $f_j(.,.)(y)$ should coincide if y is member of both Y_i and Y_j.

of the product construction compositionally from the component models $(CV_i, DV_i, \mathcal{I}_i, \xi_i, \psi_i)$ as follows:

$$\begin{aligned}
CV' &= \bigcup_{i=1}^{n} CV_i, \\
DV' &= \{loc_1, \ldots, loc_n\}, \text{ with } D_{loc_i} = Q_i, \\
\mathcal{I}' &= \bigwedge_{i=1}^{n} \mathcal{I}_i \\
\xi' &= \bigwedge_{i=1}^{n} \xi_i \\
\psi' &= \bigvee_{i=1}^{n} \left(\psi_i \wedge \bigwedge_{j \neq i} loc_j = \overleftarrow{loc_j} \wedge \bigwedge_{y \in Y \setminus Y_i} y = \overleftarrow{y} \right)
\end{aligned}$$

It is a straightforward proof that the runs of $(CV', DV', \mathcal{I}', \xi', \psi')$, when projected to all variables distinct from loc, agree with the runs of $(CV, DV, \mathcal{I}, \xi, \psi)$ projected to the variables distinct from loc_1 to loc_n. Hence, interleaving-style parallel composition can be achieved through straightforward application of Boolean connectives, as can be seen the definitions of $\mathcal{I}'$, ξ', and ψ'.

3 Embedding Simulink

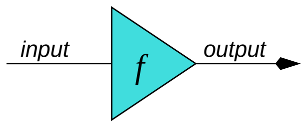
All that needs to be done for embedding Simulink models into the proposed predicative framework is to provide models for the various atomic Simulink blocks. The overall dynamics is then adequately by taking the conjunction of the individual models.

3.1 Basic blocks

We provide encodings for some sample basic blocks; other blocks can be represented analogously.

3.1.1 'Algebraic' blocks

For every time instant t, the output of an algebraic block is determined via a time-invariant function applied to the input at the very same time instant. Thus, an algebraic block

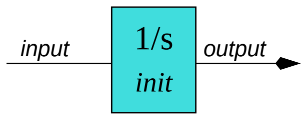


can be characterized by the transfer function $output(t) = f(input(t))$. This is captured in our modeling framework by a predicative model with

1. two continuous variables *input* and *output* of type $\mathbb{R}$,
2. no discrete variables,
3. initiation constraint `true`,
4. flow constraint $f(input) = output$,
5. jump constraint `true`, thus permitting arbitrary jumps. Note, however, that the flow constraint together with continuity of the flows enforces the functional dependency of outputs on inputs within the pre- as well as the post-states of a jump, yet not across the jump.

3.1.2 Integrators

An integrator

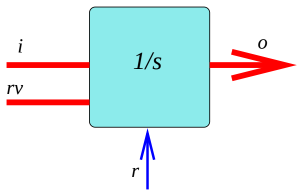


integrates its input over time. Its dynamics is thus characterized by the time-dependent function $output(t) = init + \int_0^t input(u)\,\mathrm{d}u$. This gives rise to the predicative encoding entailing

1. two continuous variables $input$ and $output$ of type $\mathbb{R}$,
2. no discrete variables,
3. initiation constraint $output = init$, where $init$ is a constant obtained from the annotation of the integrator block,
4. flow constraint $input = \frac{\mathrm{d}output}{\mathrm{d}t}$,
5. jump constraint $output = \overleftarrow{output}$, thus enforcing output continuity over jumps.

3.1.3 Resettable integrators

A resettable integrator



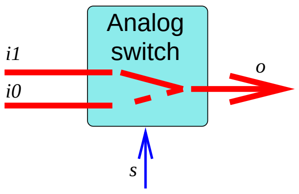
with analog inputs i and rv for the integration input and the reset value, analog output o, and a digital input r triggering the reset can be formalized by a predicative model with

1. continuous variables i, rv, o of type $\mathbb{R}$,
2. a single discrete variable r of type $\mathbb{B}$,
3. an initiation constraint $o = 0$,[6]
4. flow constraint $\frac{\mathrm{d}o}{\mathrm{d}t} = i$
5. jump constraint $\left(\left((\neg \overleftarrow{r}) \wedge r\right) \implies o = rv\right) \wedge \left(\left(\overleftarrow{r} \vee (\neg r)\right) \implies o = \overleftarrow{o}\right)$, enforcing reset of o's current value to rv's current value upon positive edges of r and continuity of o otherwise. Note that it would be more elegant to reset o to rv's value immediately prior to the jump through the equation $o = \overleftarrow{rv}$ instead of $o = rv$, yet Simulink does not do so, leading to many modeling problems in practice which are reported as "algebraic loops".

[6]Any other initial value could be used, if desired.

3.1.4 Analog switch

An analog switch with analog inputs $i_{0,1}$, selector input s (digital), and analog output o, as in

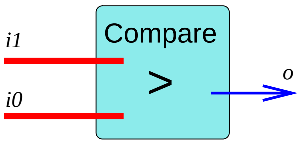


is characterized by the dynamics $o(t) = \begin{cases} i_1(t) & \text{, if } s(t) \\ i_0(t) & \text{, if } \neg s(t) \end{cases}$ in each time instant t. It can thus be modeled as a predicative model with

1. continuous variables i_0, i_1, o of type $\mathbb{R}$,
2. a discrete variable s of type $\mathbb{B}$,
3. initiation constraint `true`,
4. flow constraint $(s \implies i_1 = o) \wedge (\neg s \implies i_2 = o)$,
5. jump constraint `true`.

3.1.5 Comparator blocks

A comparator with analog inputs $i_{0,1}$ and discrete Boolean output o, as in



is characterized by the dynamics $o(t) \iff (i_1(t) > i_0(t))$ in each time instant t. It can thus be modeled as a predicative model with

1. continuous variables i_0, i_1 of type $\mathbb{R}$,
2. a discrete variable o of type $\mathbb{B}$,
3. initiation constraint `true`,
4. flow constraint $o \iff (i_1 > i_0)$
5. jump constraint `true`.

Note that the fact that o is a discrete variable enforces a discrete step whenever $i_1(t) > i_0(t)$ changes truth value, thus interrupting the continuous flow. Observe, furthermore, how the fact that undecorated variables are only interpreted in the inner part of the domain of a flow (except for point flows) simplifies the definition. Otherwise, $o \iff (i_1 > i_0)$ could not be used as an invariant as $i_1(t) > i_0(t)$ is not an admissible predicate wrt. limits of continuous functions. Finally, the alternation of flows and jumps together with the constancy of discrete variables along flows renders an explicit update of o within jumps superfluous. While it would cause no harm, it would be completely redundant to use jump constraint $o \iff (i_1 > i_0)$ instead of `true`.

3.1.6 Stateflow blocks

A Stateflow block in Simulink is a container for a statechart. It interfaces the statechart to its simulink environment via shared-state concurrency. Furthermore, it provides a triggering mechanism which determines when continuous flows are interrupted for discrete computations to be performed by the entailed statechart. Simulink offers a variety of such triggering mechanisms. The most important are

- periodic activation of the statechart and
- edge-triggered activation.

In the latter case, the Stateflow block comes equipped with an input x and is activated upon zero-crossings of x's value. I.e., the chart is triggered whenever continuous evolution lets x pass from negative to non-negative value. This activation mode is called "positive edge triggering" and naturally there are also the variants "negative edge triggering", where triggering occurs iff x transits from non-positive to positive, and "triggered on both edges" referring to triggering whenever x crosses 0. We will only formalize triggering on positive edges here, as the other edge-triggering modes as well as the periodic activation have similar formalizations.

The essence of the formalization is that time is allowed to pass, i.e. an uninterrupted flow is possible, if x does not exhibit a zero crossing. Once a zero-crossing is detected, a special start event (encoded by a discrete variable *trigger*) is generated which "fuels" the entailed statechart for one step. Such a step does basically entail one transition per parallel component at each currently active chart level.[7] A stateflow chart C included in a simulink block with positively edge-triggered trigger condition x, input variables I, and output variables O is thus represented by

1. the set $CV = \{x\} \cup I$ of continuous variables, representing the fact that all chart-internal variables and chart outputs are subject to discrete updates,
2. the set $DV = \mathit{Internal} \cup O \cup \{\mathit{trigger}, \mathit{negative}\}$, where *Internal* is the set of chart-internal variables, of discrete variables collects all discrete variables from the entailed stateflow chart plus the chart ouputs, yet extends this set with two additional variables *trigger*, *negative*,
3. the type information is inherited from the entailed chart for the discrete variables, from the simulink environment for the continuous variables, and assigns type $\mathbb{B}$ to *trigger*, *negative*,
4. the initiation constraint $\mathcal{I}$ is inherited from the entailed chart,
5. the flow constraints is

$$(\mathit{negative} \iff x < 0) \wedge (\mathit{trigger} \iff x < 0 \wedge \overrightharpoon{x} = 0) \ ,$$

   i.e.

   (a) a flow cannot extend beyond a zero crossing of x as otherwise *negative* would have to change truth value within an uninterrrupted flow, which it cannot, and

   (b) *trigger* and thus $\overrightharpoon{\mathit{trigger}}$ is true iff the flow ends with such a positive edge of x,[8]

6. the jump constraint coincides to the jump constraint of the outermost level of the chart iff *trigger* is true and to the identity transformation iff *trigger* is false or no transition in the chart is enabled:

$$\mathcal{J} = \begin{cases} \mathcal{J}_n & \text{iff } \mathit{trigger} \wedge \exists \vec{v}. \mathcal{J}_n \\ l_n = \overleftharpoon{l}_n \wedge \bigwedge_{v \in DV \setminus \{\mathit{trigger}, \mathit{negative}\}} v = \overleftharpoon{v} & \text{otherwise,} \end{cases}$$

   where n is the top-level location of the statechart and $\mathcal{J}_n$ is the jump constraint of the statechart.

[7]It is a peculiarity of the Stateflow semantics that chart activations do neither implement a run-to-completion step (i.e., a transition sequence running until the statechart stabilizes due to no further transitions being enabled) nor a single transition per chart activation, but the intermediate concept of taking one transition within each level and each parallel component. Both other concepts would be much easier to formalize. In particular, they could do without dedicated start events.

[8]Note that *trigger* actually anticipates the zero crossing as it is true throughout the whole flow.

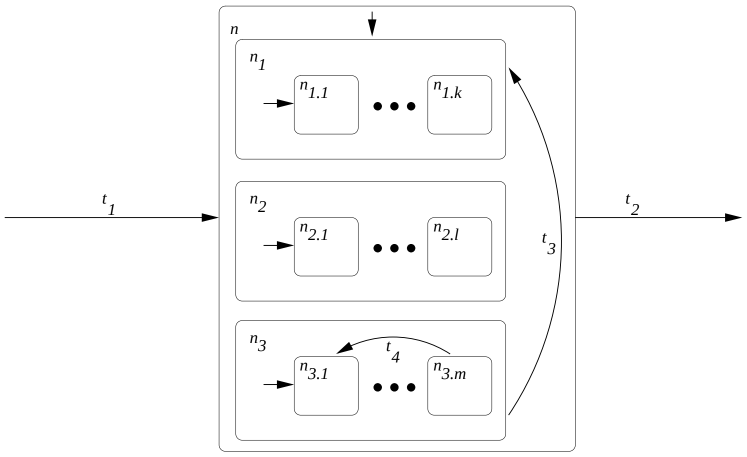


Figure 2: "Or" decomposition in Stateflow

3.1.7 Stateflow charts

Statechart-like notations are known to have a complex semantics. We provide a compositional encoding of a large subset of the Stateflow formalism by recursively translating them according to the sub-chart hierarchy. This requires an appropriate encoding of the different mode decomposition types ("and" vs. "or" decomposition) and the transition prioritization mechanisms attached to these. We start with atomic modes, i.e. those which do not entail sub-modes, and will then show how to translate compound modes by recursively translating their sub-modes.

Atomic modes. An atomic mode, i.e. a mode not decomposed into further sub-modes, is translated to the "empty" model, i.e. a predicative model with

1. no variables at all (neither discrete nor event nor continuous),
2. initiation constraint `true`,
3. jump constraint `false`, as there are no inner jumps possible,
4. flow constraint `true`, as it does not impose constraints on the flows.

"Or" decomposition, i.e. sequential composition of sub-locations. A location n which is "or"-composed from sub-locations $n_1, \dots, n_m$ (see Fig. 2), can be translated as follows:

1. as variables we take all variables obtained from translating the sub-modes, plus
   - a fresh (in the sense of not being used otherwise) discrete variable l_n of enumeration type $\{n_1, \dots, n_m\}$,
   - for each local variable x declared at this level a fresh discrete variable $\tilde{x}$ of matching type,
   - for each local event e a fresh discrete variable $\tilde{e}$ of type $\mathbb{B}$,[9]
2. the initiation constraint is $\mathcal{I}_n \equiv (l_n = n_1 \wedge \mathcal{I}_{n_1})$, where $\mathcal{I}_{n_1}$ is the initiation constraint obtained from translating n_1 (which is or `true` if n_1 has no sub-structure),

[9]Note that Stateflow events are represented by discrete state variables rather than event variables. This reflects the fact that synchronisation on events is neither symmetric nor fully synchronous: instead of an agreement upon events between parallel components, an event is generated by one partner in one computational step and consumed by the other partners in the *next* computational step.

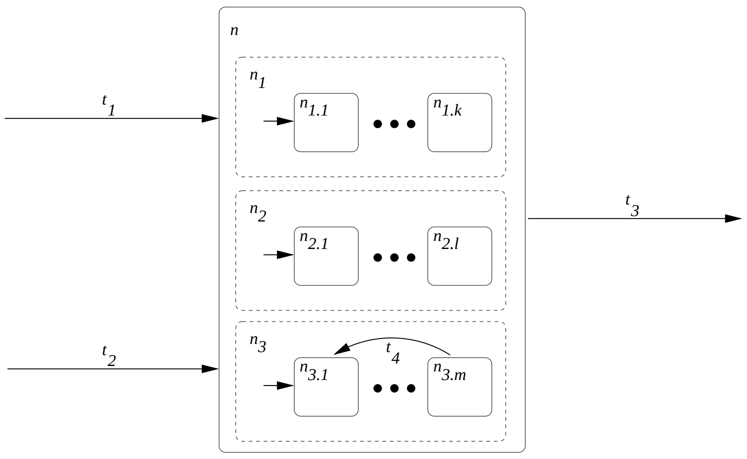


Figure 3: "And" decomposition in Stateflow

3. the jump predicate $\mathcal{J}_n$ is composed from those of the sub-modes as well as the transitions on level n —for simplicity we do not deal with inter-level transitions here— as follows:

$$\begin{aligned}
\mathcal{J}_n &\equiv \left(\bigvee_{i=1}^{m} \mathcal{J}_{n_i} \wedge \overleftarrow{l_n} = l_n = i\right) \vee \neg \mathit{submode\_transition\_enabled} \wedge \mathit{level\_transition} \\
\mathit{submode\_transition\_enabled} &\equiv \exists \vec{v}. \bigvee_{i=1}^{n} \mathcal{J}_{n_i} \wedge l_n = i
\end{aligned}$$

where $\vec{v}$ denotes the vector of variables of the predicative model such that *submode_transition_enabled* tests whether there exists a post-state that is consistent with at least one transition from the subordinate levels. As inner transitions have priority in Stateflow, a transition at level n may only be taken if *submode_transition_enabled* fails. The effect of the level transitions is described by predicate *level_transition*, which is the disjunction of predicates $trans_{t_i}$ formalizing the individual transitions t_i at level n. For a transition t with source node n_s and target node n_t, both at the same hierarchy level immediately below n, furthermore with enabling events $e_1, \dots, e_u$, guard g, effect $(x_1, \dots, x_v) := (t_1, \dots, t_v)$, and generates events $e'_1, \dots, e'_w$, the predicate $trans_t$ is defined as

$$trans_t \equiv \overleftarrow{l_n} = n_s \wedge l_n = n_t \wedge \bigwedge_{i=1}^{u} \overleftarrow{\tilde{e_i}} \wedge \overleftarrow{g} \wedge \bigwedge_{i=1}^{v} \tilde{x_i} = \overleftarrow{t_i} \wedge \bigwedge_{x \notin \{x_1, \dots, x_v\}} \tilde{x} = \overleftarrow{\tilde{x}} \wedge \bigwedge_{i=1}^{w} \widetilde{e'_i} \wedge \mathcal{I}_{n_t}$$

where $\overleftarrow{t_i}$ denotes the term t_i with all occurrences of variables x replaced by $\overleftarrow{\tilde{x}}$.

This specification can easily be extended to cover inter-level transitions, as well as special "connectors" overriding the standard activation mechanisms of sub-modes. A so-called "history connector" may, e.g., be modeled by preserving location information concerning sub-modes instead of re-initializing them upon entry to n_t. For the sake of simplicity, we have also omitted the more idiosyncratic parts of Stateflow's plethora of priority rules between transitions, namely clockwise priority between transitions within a level. These could, however, be added by strengthening the guards appropriately.

4. the flow constraint is `true`.

"And" decomposition, i.e. concurrent sub-locations. A location n "and"-composed from sub-locations $n_1, \dots, n_m$ (see Fig. 3), can be translated as follows:

1. first, we translate the sub-modes, whereby we enforce that fresh variables are taken for all declarations s.t. the predicative models obtained use unique names for local variables,

2. as variables space we then take all variables obtained from translating the sub-modes, plus
   - for each local variable x declared at the new level n a fresh discrete variable $\tilde{x}$ of matching type,
   - for each local event e declared at this level a fresh discrete variable $\tilde{e}$ of type $\mathbb{B}$,
3. the initiation constraint is $\mathcal{I}_n \equiv \bigwedge_{i_1}^{m} \mathcal{I}_{n_i}$, where $\mathcal{I}_{n_1}$ is the initiation constraint obtained from recursively translating n_i,
4. in Stateflow, and decomposition is equivalent to an *accelerated* sequential composition where the sub-locations perform there steps sequentially in a fixed order, yet each sub-location does one step per activation. Thus, the the jump constraint $\mathcal{J}_n$ is the sequential composition

$$\mathcal{J}_n \equiv \exists\, \vec{v}_0, \ldots, \vec{v}_m \,. \left(\begin{array}{l} \bigwedge_{i=1}^{m} (\mathcal{J}_{n_i}[\vec{v}_{i-1}/\overleftharpoon{\vec{v}}][\vec{v}_i/\vec{v}] \wedge \bigwedge_{j \neq i} \overleftharpoon{l}_{n_j} = l_{n_j}) \\ \wedge\ \overleftharpoon{\vec{v}} = \vec{v}_0 \\ \wedge\, \vec{v} = \vec{v}_m \end{array} \right)$$

   of the jump constraints $\mathcal{J}_{n_i}$ obtained from translating the sub-modes n_1 to n_m, where $\vec{v}$ is the vector of all variables occurring in the sub-modes,[10]
5. the flow constraint is `true`.

3.2 Composition of simulink blocks

A complete Simulink model is a parallel composition of basic blocks.[11] Given predicative models for the entailed basic blocks, the predicative model of the overall system can be obtained by

1. joining the individual sets of continuous variables in order to obtain the overall set of continuous variables,[12]
2. joining the individual sets of discrete variables in order to obtain the overall set of discrete variables,
3. collecting together the type information,[13]
4. using conjunction of the initiation constraints to obtain the overall initiation constraint,
5. using conjunction of the flow constraints to obtain the overall flow constraint,
6. using conjunction of the jump constraints to obtain the overall jump constraint.

Hence, the predicative model is perfectly compositional wrt. Simulink-style building-block composition.

4 The `HLang` implementation

The following section contains a detailed description of the `HLang` syntax.

[10]Note that some of the more natural notions of parallel composition, like e.g. interleaving, have considerably simpler axiomatizations. But in a strive to protect engineers from experiencing the intricacies of concurrency, Stateflow opts for a deterministic interpretation of parallel composition. This leads to this idiosyncratic deterministic sequencing of parallel components.

[11]Though there is a notion of hierarchically nested subsystems, it is semantically irrelevant. Hierarchy at the Simulink level (yet, not within Stateflow blocks) merely is a means of structural organization of a model for documentation purposes. Hence, we ignore hierarchy information and consider the overall system a flat parallel composition of basic blocks.

[12]Here and in the sequel we assume that the local variables of different blocks have different names, which can be achieved by renaming.

[13]Note that composition in Simulink enforces type consistency wrt. shared variables s.t. the types assigned in the individual component models coincide. Hence, type information can simply be collected.

4.1 HSystem

A system consists of optional parts :

- ***declaration part* :**
  The declaration part contains the declarations of the symbols used in a system.
- ***initial part* :**
  The initial part contains the initial state description of the system.
- ***invariant part* :**
  The invariant part contains properties of the system, which invariantly hold during flows and jumps.
- ***jump part* :**
  The jump part contains the description of the jumps of a system.
- ***flow part* :**
  The flow part contains the description of the jumps of a system.
- ***target part* :**
  The target part contains targets for tool applications on the system.

```
<hsystem> := <declaration_part>?
             <init_part>?
             <invariant_part>?
             <jump_part>?
             <flow_part>?
             <target_part>?
```

4.2 Type

The `HLang` type system currently consists of the boolean type, the integer type and the real type. More complex types (like enumerations, tuples or records) are planned for the future.

4.2.1 Type

A type is either a bool type, an integer type or a real type.

```
<type>        :=   <bool_type>
                 | <int_type>
                 | <real_type>
```

4.2.2 Boolean Type

Syntactically, the boolean type is described by the keyword '`bool`'.

```
<bool_type> := 'bool'
```

4.2.3 Integer Type

The integer type is optionally bounded. If there are bounds, they constrain the minimum and maximum value variables of that type can take.

```
<int_type> := <int_type_bounds>?  'int'
```

4.2.4 Integer Type Bounds

Integer type bounds consist of a lower and an upper bound, syntactically separated by a comma.
The bounds consist of a value and brace, which signals whether the bound value is included in the type range ('`[`' for lower bounds, '`]`' for upper bounds) or excluded ('`(`' for lower bounds, '`)`' for upper bounds). For integer types, bound values are always excluded, with the exception of the special lower bound value '`-inf`', which is used when the type has no lower bound, and the special upper bound value '`inf`', which is used when the type has no upper bound.

```
<int_type_bounds> :=   <int_lower_bound> ',' <int_upper_bound>
<int_lower_bound> :=   '(' '-' 'inf'
                     | '[' <int_expression>
<int_upper_bound>      'inf' ')'
                     | <int_expression> ']'
```

4.2.5 Real Type

The real type is optionally bounded. If there are bounds, they constrain the minimum and maximum value variables of that type can take.

```
<real_type> := <real_type_bounds>?  'real'
```

4.2.6 Real Type Bounds

Real type bounds consist of a lower bound and an upper bound, syntactically separated by a comma. The bounds consist of a value and brace, which signals whether the bound value is included in the type range ('`[`' for lower bounds, '`]`' for upper bounds) or excluded ('`(`' for lower bounds, '`)`' for upper bounds). The special lower bound value '`-inf`' (which is never included) is used when the type has no lower bound. The special upper bound value '`inf`' (which is never included) is used when the type has no upper bound.
Semantically a lower bound value must be lower than the upper bound value.

```
<real_type_bounds> :=   <real_lower_bound> ',' <real_upper_bound>
<real_lower_bound> :=   '(' <real_expression>
                        '(' '-' 'inf'
                      | '[' <real_expression>
<real_upper_bound> :=   <real_expression> ')'
                        'inf' ')'
                      | <real_expression> ']'
```

4.3 Declaration Part

In the declaration part, all symbols used in the system are declared.
Syntactically, the declaration part consists of the keyword `DECL`, followed by a (possibly empty) sequence of declarations.

```
<declaration_part> := 'DECL'
                      ( <declaration> ';' )*
```

4.3.1 Declaration

With each declaration one symbol of the system is declared. Each declaration connects an identifier name with a type, optional modifiers and an optional initialization expression.
For each declaration each modifier must occur at most once. A symbol must not be declared twice.

```
<declaration> := <modifier>* <type> <id> <initializer>?  ';'
```

4.3.2 Modifier

A modifier adds some information to a symbol.
Currently, the following modifiers are present in `HLang` :

- ***const*** **:**
  The symbol is a constant.
  A constant symbol is a symbolic name for a constant value, which has to be set with an initializer. The value cannot change during system evolution.
- ***input*** **:**
  The symbol is an input.
- ***param*** **:**
  The symbol is a parameter.
  A parameter is a variable, whose value can be initially chosen, but which cannot change during further system evolution.

```
<modifier>    :=   'const'
                 | 'input'
                 | 'param'
```

4.3.3 Initializer

An initializer describes the initial values of the corresponding variable.
The type of the initial expression must match the corresponding variable type.

```
<initializer> :=   '=' <expression>
```

4.4 Initial Part

The initial part contains a predicative description of the initial condition of the system. It consists of a sequence of predicates, each of which holds for the initial condition.
The initial part contains only undecorated variables.
Syntactically, the initial part consists of the keyword `INIT`, followed by a (possibly empty) sequence of predicates separated by ';' :

```
<init_part> := 'INIT'
               ( <predicate> ';' )*
```

4.5 Invariant Part

The invariant part contains a predicative description of invariant properties of the system. It consists of a sequence of predicates, each of which is an invariant property.
The predicates contain only undecorated variables.
Syntactically, the invariant part consists of the keyword `INVAR`, followed by a (possibly empty) sequence of predicates separated by ';' :

```
<invariant_part> := 'INVAR'
                    ( <predicate> ';' )*
```

4.6 Flow Part

The flow part contains a predicative description of the flows of the system. It consists of a sequence of predicates, which contain only undecorated, enter decorated and leave decorated variables.
Syntactically, the flow part consists of the keyword `FLOW`, followed by a (possibly empty) sequence of predicates separated by ';' :

```
<flow_part> := 'FLOW'
               ( <predicate> ';' )*
```

4.7 Jump Part

The jump part contains a predicative description of the jumps of the system. It consists of a sequence of predicates, which contain only undecorated and primed variables.
Syntactically, the flow part consists of the keyword `JUMP`, followed by a (possibly empty) sequence of predicates separated by ';' :

```
<jump_part> := 'JUMP'
               ( <predicate> ';' )*
```

4.8 Target Part

The target part contains targets for tool applications. It consists of a sequence of expressions.
Syntactically, the transition part consists of the keyword `TARGET`, followed by a (possibly empty) sequence of expressions separated by ';' :

```
<flow_part> := 'TARGET'
               ( <expression> ';' )*
```

4.9 Predicates

A predicate is an expression of type bool.

```
<predicate> :=   <expression>
```

4.10 Expressions

In `HLang` expressions either are literals, variables or operator applications. Most of the typical logical, relational and arithmetical operators are supported.

```
<expression> :=   <literal_expression>
                | <variable_expression>
                | <logic_expression>
                | <relational_expression>
                | <arithmetic_expression>
                | <misc_expression>
```

4.10.1 Integer expression

An integer expression is an expression of type integer.

```
<int_expresssion> :=   <expression>
```

4.10.2 Real expression

A real expression is an expression of type real.

```
<real_expresssion> :=   <expression>
```

4.11 Literal Expressions

A literal expression is a representation of a concrete value for a particular type.

```
<literal_expression> :=   <bool_literal>
                        | <int_literal>
                        | <real_literal>
```

4.11.1 Bool Literal

A bool literal is a representation of a concrete boolean value.
A boolean literal is either true or false.
The type of a boolean literal is the boolean type.

```
<bool_literal> :=   'true'
                  | 'false'
```

4.11.2 Integer Literal

An integer literal is a representation of a concrete integer value.
An integer literal is a sequence of digits, and can optionally be negative.
The type of an integer literal is the integer type.

```
<int_literal> :=   '-'?  <digit>+
```

4.11.3 Real Literal

A real literal is a representation of a concrete real value.
A real literal is a sequence of digits, and can optionally be negative and/or have an exponent.
The type of a real literal is the real type.

```
<real_literal>  :=   '-'?  <digit>+ '.'  <digit>+ <real_exponent>?
                   | '-'?  <digit>+ <real_exponent>
<real_exponent> :=   ('e' | 'E') ('+' | '-')?  <digit>+
```

4.12 Variable Expressions

A variable expression is a representation of a variable. Variables can either be decorated or undecorated.

```
<variable_expression> :=   <undecorated_variable>
                         | <decorated_variable>
```

4.12.1 Undecorated Variables

An undecorated variable is an identifier, for which a corresponding symbol must be declared in the associated symbol table.

```
<undecorated_variable> :=   <identifier>
```

4.12.2 Decorated Variables

A decorated variable is either a prime decorated variable, an enter decorated variable or a leave decorated variable.

```
<decorated_variable>  :=   <prime_decorated_variable>
                         | <enter_decorated_variable>
                         | <leave_decorated_variable>
```

4.12.3 Prime Decorated Variables

A prime decorated variable describes the value of a variable after a jump. It is an identifier followed by a prime. The variable must be declared in the associated symbol table.

```
<prime_decorated_variable>  :=   <identifier> '''
```

4.12.4 Enter Decorated Variables

An enter decorated variable describes the value of a variable at the beginning of a flow. The variable must be declared in the associated symbol table.

```
<prime_decorated_variable>  :=   'enter' '(' <identifier> ')'
```

4.12.5 Leave Decorated Variables

An leave decorated variable describes the value of a variable at the end of a flow. The variable must be declared in the associated symbol table.

```
<prime_decorated_variable>  :=   'leave' '(' <identifier> ')'
```

4.13 Logic Expressions

HLang supports the typical logical operators.

```
<logic_expression> :=   <not_expression>
                      | <and_expression>
                      | <or_expression>
                      | <xor_expression>
                      | <nxor_expression>
                      | <implication_expression>
```

4.13.1 Negation

The negation expression describes the negation of an expression.
The operator maps a boolean expression to a boolean expression.

```
<not_expression> :=   'not' <expression>
                    | '!'  <expression>
```

4.13.2 Conjunction

The conjunction expression describes the conjunction of two expressions.
The operator maps two boolean expressions to a boolean expression.

```
<not_expression> :=   <expression> 'and' <expression>
```

4.13.3 Disjunction

The disjunction expression describes the disjunction of two expressions.
The operator maps two boolean expressions to a boolean expression.

```
<or_expression> :=   <expression> 'or' <expression>
```

4.13.4 Exclusive Disjunction

The xor expression describes the exclusive disjunction of two expressions.
The operator maps two boolean expressions to a boolean expression.

```
<xor_expression> :=   <expression> 'xor' <expression>
```

4.13.5 Negated Exclusive Disjunction

The nxor expression describes the negated exclusive disjunction (aka boolean equivalence) of two expressions.
The operator maps two boolean expressions to a boolean expression.

```
<nxor_expression> :=   <expression> 'nxor' <expression>
                     | <expression> '<->' <expression>
```

4.13.6 Implication

The implication expression describes the implication between two expressions.
The operator maps two boolean expressions to a boolean expression.

```
<implication_expression> :=   <expression> 'implies' <expression>
                            | <expression> '->' <expression>
```

4.14 Relational Expressions

`HLang` support the typical relational operators.

```
<relational_expression> :=   <lesser_expression>
                           | <lesser_equal_expression>
                           | <greater_expression>
                           | <greater_equal_expression>
                           | <equal_expression>
                           | <unequal_expression>
```

4.14.1 Lesser Expression

The lesser expression is the application of the lesser operator to two expressions.
The lesser operator maps

- two integer expressions to a boolean expression, or
- two real expressions to a boolean expression.

```
<lesser_expression> := <expression> '<' <expression>
```

4.14.2 Lesser Equal Expression

The lesser equal expression is the application of the lesser equal operator to two expressions.
The lesser equal operator maps

- two integer expressions to a boolean expression, or
- two real expressions to a boolean expression.

```
<lesser_equal_expression> := <expression> '<=' <expression>
```

4.14.3 Greater Expression

The greater expression is the application of the greater operator to two expressions.
The greater operator maps

- two integer expressions to a boolean expression, or
- two real expressions to a boolean expression.

```
<greater_expression> := <expression> '>' <expression>
```

4.14.4 Greater Equal Expression

The greater equal expression is the application of the greater equal operator to two expressions.
The greater equal operator maps

- two integer expressions to a boolean expression, or
- two real expressions to a boolean expression.

```
<greater_equal_expression> := <expression> '>=' <expression>
```

4.14.5 Equal Expression

The equal expression is the application of the equal operator to two expressions.
The equal operator maps

- two boolean expressions to a boolean expression, or
- two integer expressions to a boolean expression, or
- two real expressions to a boolean expression.

```
<equal_expression> := <expression> '=' <expression>
```

4.14.6 Unequal Expression

The unequal expression is the application of the unequal operator to two expressions.
The unequal operator maps

- two boolean expressions to a boolean expression, or
- two integer expressions to a boolean expression, or
- two real expressions to a boolean expression.

```
<unequal_expression> := <expression> '!=' <expression>
```

4.15 Arithmetic Expressions

`HLang` support the usual arithmetic operators.

```
<arithmetic_expression> :=   <plus_expression>
                            | <minus_expression>
                            | <mult_expression>
                            | <div_expression>
                            | <mod_expression>
                            | <power_expression>
                            | <unary_plus_expression>
                            | <unary_minus_expression>
```

4.15.1 Plus Expression

The plus expression is the application of the plus operator to two expressions and describes the sum of the operands.
The plus operator maps

- two integer expressions to an integer expression, or
- two real expressions to a real expression.

```
<plus_expression> := <expression> '+' <expression>
```

4.15.2 Minus Expression

The minus expression is the application of the minus operator to two expressions and describes the difference of the operands.
The minus operator maps

- two integer expressions to an integer expression, or
- two real expressions to a real expression.

```
<minus_expression> := <expression> '-' <expression>
```

4.15.3 Multiplication Expression

The mult expression is the application of the mult operator to two expressions and describes the product of the operands.
The mult operator maps

- two integer expressions to an integer expression, or
- two real expressions to a real expression.

```
<mult_expression> := <expression> '*' <expression>
```

4.15.4 Division Expression

The div expression is the application of the div operator to two expressions and describes the division of the operands.
The div operator maps

- two integer expressions to an integer expression, or
- two real expressions to a real expression.

```
<div_expression> := <expression> '/' <expression>
```

4.15.5 Modulo Expression

The modulo expression is the application of the modulo operator to two expressions and describes the remainder of a division of the operands.
The modulo operator maps

- two integer expressions to an integer expression.

```
<mod_expression> := <expression> '%' <expression>
```

4.15.6 Power Expression

The power expression is the application of the power operator to two expressions, whereby the first expression describes the base and the second expression describes the exponent.
The power operator maps

- two integer expressions to a real expression, or
- an integer and a real expression to a real expression, or
- a real and an integer expression to a real expression, or
- two real expressions to a real expression.

```
<power_expression> :=   <expression> '**' <expression>
                      | <expression> '^' <expression>
```

4.15.7 Unary Plus Expression

The unary plus expression is the application of the unary plus operator to an expression and describes the same expression as the operand.
The unary plus operator maps

- an integer expression to an integer expression, or
- a real expression to a real expression.

```
<unary_plus_expression> := '+' <expression>
```

4.15.8 Unary Minus Expression

The unary minus expression is the application of the minus plus operator to an expression and describes the operand expression with changed sign.
The unary minus operator maps

- an integer expression to an integer expression, or
- a real expression to a real expression.

```
<unary_minus_expression> := '-' <expression>
```

4.16 Misc Expressions

Also some other expressions are available in `HLang`.

```
<misc_expression>        <parentheses_expression>
                       | <if_then_else_expression>
                       | <cast_expression>
                       | <time_derivative_expression>
                       | <xml_tagged_expression>
```

4.16.1 Parentheses Expression

A parentheses expression describes the contained expression, and the type is the type of the contained expression.

```
<parentheses_expression> := '(' <expression> ')'
```

4.16.2 If-Then-Else Expression

The if-then-else expression is the application of the if-then-else operator to a condition expression and two argument expressions and describes a functional if-then-else, which selects the first argument expression, if the condition expression is true, and the second argument expression, if the condition expression is false.
The if-then-else operator maps

- three boolean expression to a boolean expression, or
- a boolean and two integer expressions to an integer expression, or
- a boolean and two real expressions to a real expression.

```
<if_then_else_expression> := 'if' <predicate>
                             'then' <expression>
                             'else' <expression>
                             'fi'
```

4.16.3 Cast Expression

The cast expression converts an expression from one type to another.
Currently `HLang` supports casts from

- a boolean expression to an integer expression, or
- an integer expression expressions to a real expression.

```
<cast_expression> := cast '<' <type> '>' '(' <expression> ')'
```

4.16.4 Time Derivative Expression

The time derivative expression is the application of the time derivative operator to a condition expression and two argument expressions and describes the time derivative of the operand.
The time derivative operator maps

- two real expressions to a real expression.

```
<time_derivative_expression> := 'd/dt' '(' <expression> ')'
```

4.16.5 XML Tagged Expression

It is possible to annotate `HLang` expressions with XML tags. This has no semantical meaning, but can be used to transport additional information from one tool to another.
As usual for xml tags, the identifiers of opening and closing tags must match.

```
<xml_tagged_expression> := <xml_tag_open> <expression> <xml_tag_close>
<xml_tag_open>          := '<' <id> <xml_attribute>* '>'
<xml_tag_close>         := '<' <id> '>'
<xml_attribute>         := <id> '=' '"' <string> '"'
```

A The prototypic Simulink to HLang compiler

In this section, we provide an informal overview over the prototypic Simulink/Stateflow to HLang compiler implemented in AVACS. It compiles directly from Simulink's `.mdl` files, thereby supporting simulink model versions between 6.2 and 6.5. Currently, the following simulink blocks are supported: Abs, Chart, Clock, Compare to Constant, Constant, Demux, Digital Clock, Display, From, Gain, Goto, In, Integrator, Interval Test, Logic Operator, Math, Memory, Multiport Switch, Mux, Out, Product, Relational Operator, Saturation, Scope, Sign, Subsystem, Sum, Switch, Terminator, Unary Minus, and the Stateflow block, featuring a well-defined subset of Stateflow (cf. Sect. A.3).

A.1 Simulink input file

Fig. 4 shows the structure of a simulink model file. It consists of at most two parts, where the first, obligatory one is the model part defining the block structure. It contains defaults for most of the blocks used in the model. The default attributes thus assigned to the various block types are then being inherited by the blocks in the system part. The blocks in the system part can, however, override the defaults. The system part is primarily concerned with detailing the system structure, listing all blocks and connecting lines used. For a mere Simulink model without embedded Stateflow charts, the compilation can proceed structurally by traversing the `mdl` file according to this grammar.

The (optional) second part is the Stateflow part, which defines the statecharts and related entities used in Stateflow blocks in the model part. The Stateflow part uses a different structuring strategy than the model part: Instead of a grammar reflecting the nesting structure, the hierarchy here is being expressed by linking parts together using a unique identifier for each part.

A.2 Simulink translation

Translation starts with parsing a Simulink model into an object hierarchy, yielding an object per block, line, etc. that reflects the entity type and the explictly set attributes in the blocks. To have all attributes relevant for the translation locally accessible, a merge function is being called which sets all unset attributes by retrieving them from the default blocks.

In the next step, we abstract from the concrete the lines and branches. We convert all lines and branches to uniform connections objects which store the source block, the source port, the destination block, and the destination port. These connections are then attached to the block objects such that for each block, the precursors and successors (i.e., the blocks driving an input of th current block and the blocks being driven) can be determined locally.

A.2.1 Example

We will now explain the translation by means of example. Fig. 5 provides an example of a simulink model containing a resettable integrator. Its input stemming from a constant block is being integrated over time. If the output reaches a value of 3, the integrator is being reset to its initial value (here 0).

The model comprises four block, which are translated as follows.

Constant. A constant block is translated by adding a constant (of type real or bool depending on the settings made in the model) to the HLang declaration part. Here, we obtain the declaration

```
const real Constant_out1 = 1.0;
```

Scope. The scope block is a sink in the model which has no effect on the overall dynamics. Being functionally irrelevant, compilation boils down to declaration of the input port of this block as a real-valued flow variable.

```
real Scope_in1;
```

Compare to constant. This block compares its actual input value to a defined constant value (here 3) according to the selected relation (here $\geq$).

```
DECL
real CompareToConstant_in1;
bool CompareToConstant_out1;

FLOW
CompareToConstant_in1 >= 3.0 <-> CompareToConstant_out1;
```

Note that this definition, though lacking any explicit definition of the jump predicate (thus defaulting to true), nevertheless enforces a jump whenever the input passes the threshold.

Integrator. The integrator in this model has two inputs, one for the input to be integrated and one for triggering a reset. The reset is triggered on a "rising edge", i.e., a zero crossing from below of the real-valued trigger input. The reset value may be an external value originating form an input or an internal value originating from a block attribute. Here, it is internal and 0.

```
DECL
real Integrator_in1;
real Integrator_in2;
real Integrator_out;

INIT
Integrator_out = 0.0;
```

```
JUMP
enter(Integrator_in2) <= 0.0 and Integrator_in2 > 0.0 -> Integrator_out = 0.0;
enter(Integrator_in2) > 0.0 or Integrator_in2 <= 0.0 -> Integrator_out = enter(Integrator_out);

FLOW
Integrator_in1 = d/dt(Integrator_out);
```

Lines. Each line connecting two blocks mediates an invariant between its source port and its driven port. Most of these invariants can, as they are actually equivalences, be used for simplifying the model by substitution (cf. next paragraph). Without such optimizations, these invariants, as stemming from our example, induce the translation

```
INVAR
Integrator_in2 = Constant_out1;
Scope_in1 = Integrator_out;
CompareToConstant_in1 = Integrator_out;
CompareToConstant_out1  -> Integrator_in2 = 1.0;
!CompareToConstant_out1 -> Integrator_in2 = 0.0;
```

Note that the invariant entails type casting due to the incompatible types of the output of the compare to constant block (type Boolean) and the reset input to the integrator (type real), which prevents the direct translation

```
Integrator_in2 = CompareToConstant_out1;
```

A.2.2 Actual translation.

The actual automatic translation follows the aforementioned scheme, yet saves variables and code by substituting a unique representative for equivalent entities induced by connecting lines and branches. In particular, variables for output ports do only exist for sources like constant blocks and clock blocks or the output of an integrator, while all other outputs are substituted by a term representing the invariant of the dataflow network driving it. For the example, the resulting translation is:

```
DECL
real global_time;
real reset_int_Compare_nTo_Constant_in1;
const real reset_int_Constant_out1 = 1.0000000000;
real reset_int_Integrator_in1;
real reset_int_Integrator_in2;
real reset_int_Integrator_out;
real reset_int_Scope_in1;

INIT
global_time = 0.0;
reset_int_Integrator_out = 0.0000000000;

INVAR
reset_int_Integrator_in1 = reset_int_Constant_out1;
reset_int_Scope_in1 = reset_int_Integrator_out;
reset_int_Compare_nTo_Constant_in1 = reset_int_Integrator_out;

JUMP
enter(reset_int_Integrator_in2) <= 0.0000000000 and
reset_int_Integrator_in2 > 0.0000000000 ->
reset_int_Integrator_out = 0.0000000000;
enter(reset_int_Integrator_in2) > 0.0000000000 or
```

```
reset_int_Integrator_in2 <= 0.0000000000 ->
reset_int_Integrator_out = enter(reset_int_Integrator_out);

FLOW
d/dt (global_time) = 1.0;
reset_int_Compare_nTo_Constant_in1 >= 3.0000000000 ->
reset_int_Integrator_in2 = 1.0;
reset_int_Compare_nTo_Constant_in1 < 3.0000000000 ->
reset_int_Integrator_in2 = 0.0;
reset_int_Integrator_in1 = d/dt(reset_int_Integrator_out);
```

A.3 Stateflow translation

For statecharts we will again explain the translation by means of example. Fig. 6 depicts a flat stateflow diagram with two graphica locations (a.k.a. "states"), three variables and three transitions. Each location has a unique name and may enclose four different section label defining different types of in-location actions. The actions attached to the entry label are being executed as soon as the location is entered. If a location is being left, the actions assigned to the exit label are being executed. If a chart is triggered and the active location cannot be left, the during actions arise. If this triggering was by the event denoted directly after the on label, the on label actions are also being executed.

All local and output variables have an initial value of 0 for integers and reals and of false for boolean variables. If a variable is not effected by an action, its value will remain unchanged.

A.3.1 Encoding execution states

For a sound compilation of hierarchical statecharts and the induced priority and sequencing rules amongst transitions, we need to need to know about the activation and execution state of each graphical locations. This amounts to an encoding of what action label (entry, during, etc.) currently is being executed and to a scheduling of that sequence by means of alternating execution states. The following execution states can be recognized:

- idle : all possible actions have been finished
- post_idle : decide whether to leave a location or not
- pre_entry : execute default transition actions
- entry : execute entry actions
- post_entry : wait for child entry actions
- during : execute during and on actions
- post_during : wait for child actions
- pre_exit : wait for child exit
- exit : execute exit action
- post_exit : execute actions for outgoing transition

A.3.2 Encoding activity and execution of locations

In order to mediate the flow of control, we need to know which location currently is active. Furthermore, we need to ensure that always only one location (per hierarchy level) is active and only one location can perform an action. Activity of only one location can be enforced by demanding that the sum of active locations be always at most one.

```
DECL
bool s0_active;
bool s1_active;

INIT
!s0_active;
!s1_active;

INVAR
<int>(s0_active) + cast<int>(s1_active) <= 1;
```

If we want to guarantee that at most one location is non-idle, it's enough to guarantee that all other locations are idle. To ensure this we need to decide how to encode the execution states for each location. A simple solution is to add an integer variable for each location telling its current execution state. To increase the readability of the model, one can add constant values for each execution state:

```
DECL
const int post_idle = 1;
const int pre_entry = 2;
const int entry = 3;
const int post_entry = 4;
const int during = 5;
const int post_during = 6;
const int pre_exit = 7;
const int exit = 8;
const int post_exit = 9;
const int idle = -9;
```

Beyond we need to add an execution state variable for each location:

```
DECL
[-9,9]int s0_state;
[-9,9]int s1_state;

INIT
s0_state = idle;
s1_state = idle;

INVAR
s0_state = idle or s0_state >= post_idle;
s1_state = idle or s1_state >= post_idle;
```

Using this encoding at most one location is not idle if the sum of all execution state variables is less or equal the number of locations minus two times idle value

```
INVAR
//COMMENT: s0_state + s1_state <= (2-2)*idle;
s0_state + s1_state <= 0;
```

In addition we know that an active location can only be deactivated if its execution state if post exit and that the idle state can only be left if a trigger occurs.

```
INVAR
s0_active and s0_state != post_exit -> s0_active';
s1_active and s1_state != post_exit -> s1_active';
s0_state = idle and !trigger -> s0_state' = idle and s0_active' = s0_active;
s1_state = idle and !trigger -> s1_state' = idle and s1_active' = s1_active;
```

Encoding variables and events. All stateflow variables need to be declared in the HLang representation. For the declarations we use the same datatype as in stateflow. For the example this yields:

```
DECL
real x;
real y;
real z;
```

Events (local, input and output) are encoded by Boolean variables.
Input events. Input events are generated during zero crossing of the value connected to the trigger port of the stateflow chart. In this example, we have one input event named `event`. Translation of the trigger port in the Simulink model has thus generated two variables and one invariant predicate as follows.

```
DECL
//defined by triggerport
real TriggerPort_in1;
real Stateflow_event1;
INVAR
Stateflow_event1 = TriggerPort_in1;
```

We thus have a real valued variable for each input event. We can now generate a boolean variable with a name similar to the stateflow event name and force its value to true whenever a zero crossing respecting the trigger type occurs. In our case, the `event` is defined as a rising edge event, giving

```
DECL
bool event;
FLOW
event <-> enter(Stateflow_event1) <= 0.0 and leave(Stateflow_event1) > 0.0;
```

Because a Stateflow chart may be triggered by a couple of events, we define an overall trigger variable providing the disjunction of all input events, reflecting the Stateflow semantics of disjunctive triggering:

```
DECL
bool trigger;
INVAR
trigger <-> event;
```

Local and output events. Those two event types are either either edge events (local and output) or function call events (output). The latter are not currently supported by our translation. For output edge events, if the event is being generated in a transition or location, which we encode by simply switching the boolean output value at the port corresponding to the event.

```
DECL
bool the_outport;

JUMP
transition_executed_where_event_generated -> the_outport' = !the_outport;
```

For local events its not that easy, as local events can be generated as well as consumed during a trigger. For the comsumption we need to check whether the value of the event has changed or not, for generation we use the same procedure as for the output events:

```
DECL
bool local_event;

JUMP
//generation
transition_executed_where_event_generated -> local_event' = !local_event;
//consumption
local_event' != local_event and some_condition -> action;
```

For the stateflow input events in this example we get an overall HLang representantion like

```
DECL
bool event;            //the event defined in stateflow
real TriggerPort_in1;  //input to the chart's triggerport
real Stateflow_event1; //connection variable simulink to stateflow
bool trigger;          //the overall stateflow trigger

INVAR
Stateflow_event1 = TriggerPort_in1;
trigger <-> event;

FLOW
event <-> enter(Stateflow_event1) <= 0.0 and leave(Stateflow_event1) > 0.0;
```

Encoding stateflow behaviour. Normally, a stateflow chart is being initialized only after the first trigger. This means that initially no location is active and time passes by. The chart is stable. As soon as the chart is triggered, time stops flowing and all possible statechart actions are being executed. We can represent this behaviour as follows:

```
DECL
bool stable;

INVAR
stable <-> s0_idle and s1_idle;
trigger <-> event;

JUMP

stable and !trigger -> stable' and
x' = x and y' = y and z' = z;

trigger and (!s0_active and s0_state = idle) and
(!s1_active and s1_state = idle) -> s0_state' = pre_entry
and x' = x and y' = y and z' = z;

//execute transition action
s0_state = pre_entry -> y' = 7.0 and s0_active' and s0_state' = entry
and x' = x and z' = z;
//execute entry action
s0_state = entry -> x' = 0.0 and y' = 1.0 and s0_state' = idle and z' = z;

FLOW
event <-> enter(Stateflow_event1) <= 0.0 and leave(Stateflow_event1) > 0.0;
stable -> d/dt(time) = 1.0;
!stable -> leave(time) = enter(time);
```

At this point the system is stable again, because no action can be executed anymore. If the chart is triggered again, we either need to execute the during action or (if possible) leave the current location.

```
JUMP
//go to a state where to decide, if the location must be left
trigger and s0_active and s0_state = idle -> s0_state' = post_idle
and x' = x and y' = y and z' = z;

//condition not satisfied, do during action
```

```
s0_state = post_idle and !x > 3.0 -> s0_state' = during
and x' = x and y' = y and z' = z;

//condition satisfied, leave the location
s0_state = post_idle and x > 3.0 -> s0_state' = exit
and x' = x and y' = y and z' = z;

//execute during and go to idle
s0_state = during -> x' = x + 1.0 and s0_state' = idle
and y' = y and z' = z;

//execute exit action
s0_state = exit -> x' = 0.0 and s0_state' = post_exit
and y' = y and z' = z;

//execute transition action
//   execution state     event    condition     action
s0_state = post_exit and true and (x > 3.0) -> x' = 2.0 and z' = 1.0 and
s0_state' = idle and s1_active' and s1_state' = entry and y' = y;
```

For the second location we generate all predicates exactly the same way.

```
JUMP

s1_state = entry -> s1_state' = idle and x' = 10.0 and y' = 14.0 and z' = z;

//go to a state where to decide, if the state must be left
trigger and s1_active and s1_state = idle -> s1_state' = post_idle
and x' = x and y' = y and z' = z;

//condition not satisfied, do during action
s1_state = post_idle and !y < 2.0 -> s1_state' = during
and x' = x and y' = y and z' = z;

//condition satisfied, leave the state
s1_state = post_idle and y < 2.0 -> s1_state' = exit
and x' = x and y' = y and z' = z;

//execute during and go to idle
s1_state = during -> s1_state' = idle and y' = y - 1 and   x' = x + 1 and
(!event or z' = z * 2.0) and (event or z' = z);

//execute exit action
s1_state = exit -> s1_state' = post_exit and x' = 0.0 and y' = y and z' = z;

//execute transition action
s1_state = post_exit and true and y < 2.0000000000 -> x' = 12.0 and s1_state' = idle and
s0_active' and s0_state' = entry and y' = y and z' = z;
```

Full implementation of this translation scheme, which yields a concise representation of all hierarchy-related priority rules, is currently underway. Though definitely possible, we have tacitly avoided an encoding of the more idiosyncratic intra-level priority rules of Stateflow (12-o-clock rule, sequentialization and event-based rescheduling between parallel components), as their motivation is simulation-oriented (achieving a deterministic semantics) rather than induced by the intuitive semantics of the underlying concepts of transition selection and concurrency.

References

[HSI] The HSIF hybrid system interchange format. Available from `micc.isis.vanderbilt.edu:8080`.

[Lea04] John Lygeros and et al. COLUMBUS: Design of embedded controllers for safety critical systems — project final report. Available from `http://www.columbus.gr/finalreport/index.htm#`, August 2004.

[Lyg03] John Lygeros. *Lecture Notes on Hybrid Systems.* Cambridge, 2003.

[MoB] Homepage of the model-based integration of embedded software (mobies) project. Available from `http://www.rl.af.mil/tech/programs/MoBIES/`.

[Pas04] Roberto Passerone. *Semantic Foundations for Heterogeneous Systems.* PhD thesis, EECS Dpt., Univ. of California, Berkeley, 2004.

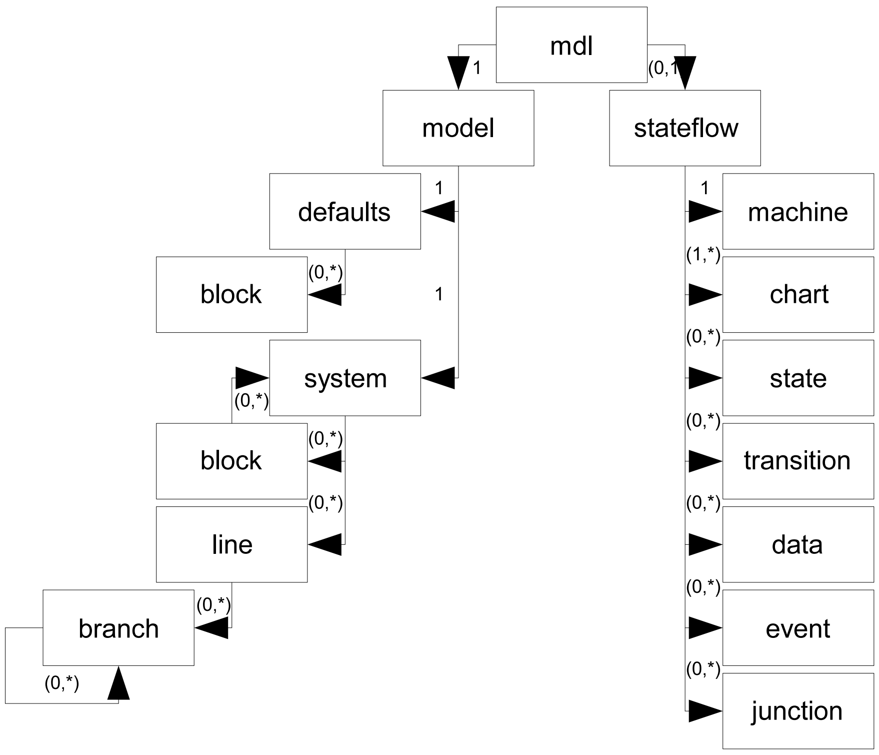


Figure 4: Layout of simulink mdl file

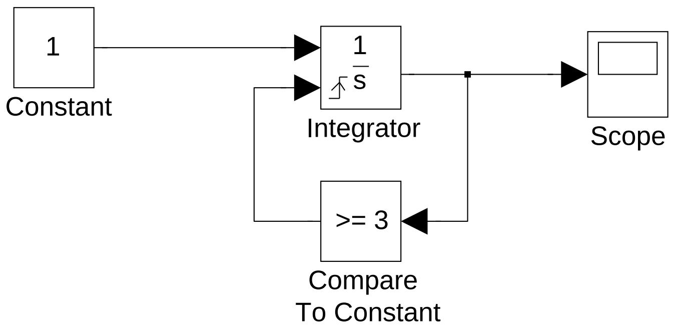


Figure 5: A sample Simulink model

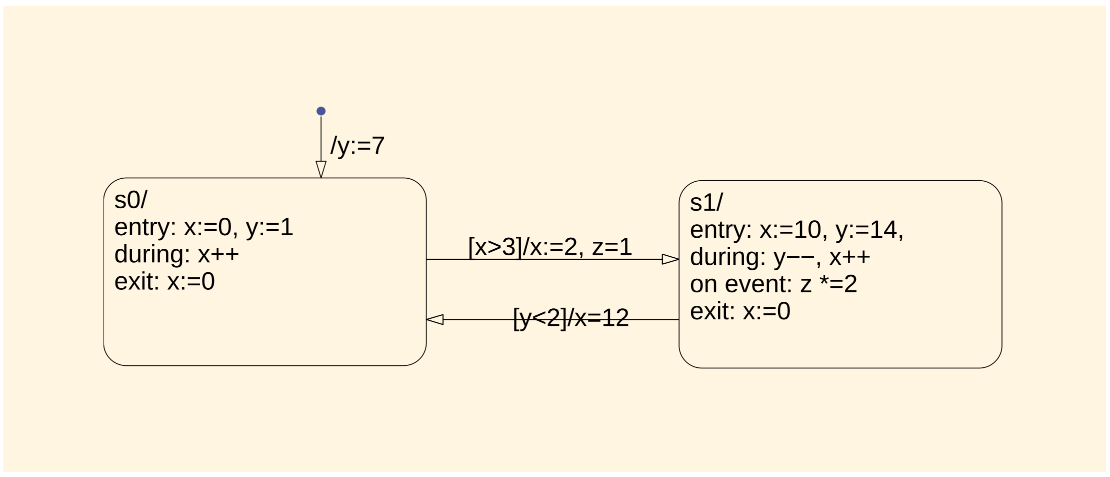


Figure 6: A sample (and simple) Stateflow chart