From Hybrid Data-Flow Languages to Hybrid Automata: A Complete Translation *

Peter Schrammel
INRIA Grenoble – Rhône-Alpes
655 Avenue de l'Europe
38330 Montbonnot St-Martin, France
peter.schrammel@inria.fr

Bertrand Jeannet
INRIA Grenoble – Rhône-Alpes
655 Avenue de l'Europe
38330 Montbonnot St-Martin, France
bertrand.jeannet@inria.fr

ABSTRACT

Hybrid systems are used to model embedded computing systems interacting with their physical environment. There is a conceptual mismatch between high-level hybrid system languages like SIMULINK, which are used for simulation, and hybrid automata, the most suitable representation for safety verification. Indeed, in simulation languages the interaction between discrete and continuous execution steps is specified using the concept of zero-crossings, whereas hybrid automata exploit the notion of staying conditions. We describe a translation from a hybrid data-flow language to logico-numerical hybrid automata that points out this issue carefully. We expose various zero-crossing semantics, propose a sound translation, and discuss to which extent the original semantics is preserved.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specification—*Languages*; D.2.4 [**Software Engineering**]: Software / Program Verification—*Formal methods*; I.6.2 [**Computing Methodologies**]: Simulation and Modelling—*Simulation Languages*

General Terms

Languages, Verification

Keywords

Data-Flow Languages, Hybrid Systems, Hybrid Automata, Verification

1. INTRODUCTION

The motivation of this paper is the *verification of safety properties* of hybrid systems, like, for example, safety-critical controllers interacting with their physical environment as found in modern transport systems. The verification of such properties amounts to checking whether the reachable state space stays within the invariant specified by the property.

Specifying hybrid systems. Languages like SIMULINK[1], MODELICA[2], and ZELUS [6] have been developed to support the modelling, implementation and simulation of hybrid systems. They offer features like modularity, hierarchy and a data-flow or equational syntax.

SIMULINK for example uses a data-flow-based description of the behavior of continuous- and discrete-time variables; STATEFLOW extends it with the ability of automata-based specifications of the discrete-time behavior. ZELUS extends the synchronous, data-flow programming language LUCID-SYNCHRONE [26] with differential equations. In these languages, discrete execution steps that interrupt the continuous-time evolution are triggered by the activation of *zero-crossings*.[3] Roughly speaking, a zero-crossing is an event occurring during the integration of an ordinary differential equation (ODE) $\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t))$, when some expression $z(\boldsymbol{x}(t))$ changes sign from negative to positive. A zero-crossing may also be triggered by a discrete execution step as in SIMULINK. All these data-flow languages are primarily designed for simulation, hence their semantics is mostly deterministic.

On the other hand, the concept of *hybrid automata* [3, 19, 20] was developed for the verification of hybrid systems. They are lower-level representations of hybrid systems with a non-deterministic semantics by default, and in which continuous-time evolution is governed by staying conditions, usually referred to as *location invariants*.

Hence, there is a conceptual mismatch between high-level hybrid system languages and hybrid automata. The main differences between *simulation* and *verification* formalisms can be summarized as follows:

– equation-based versus automata-based
– continuous modes implicitly encoded in Boolean variables versus their explicit encoding with locations
– discrete transitions triggered by zero-crossings versus combinations of staying conditions and guards
– deterministic, open systems with inputs versus non-deterministic, closed systems.

Our primary goal is to formalize the translation from a hybrid data-flow formalism to hybrid automata, and in particular to focus on the translation of zero-crossings. How-

*This work was supported by the ANR project VEDECY and the INRIA large-scale initiative SYNCHRONICS.

[1] `http://www.mathworks.com`
[2] `http://www.modelica.org`
[3] STATEFLOW also allows to trigger discrete jumps by ordinary guards that are not interpreted as zero-crossings.

HSCC'12, April 17–19, 2012, Beijing, China.

ever, a secondary aspect we have in mind is that we want to address hybrid systems specified as the composition of a discrete controller and its physical environment. This means that the discrete part of the system's state space might be complex, and defined by Boolean variables and numerical variables (counters, thresholds, etc. manipulated by the controller). The consequence is that we want to translate the data-flow input language to *logico-numerical* hybrid automata, that can manipulate symbolically discrete variables, in addition to continuous variables governed by differential equations. Such automata allow a compact representation by not requiring the enumeration of the discrete state space.

Verifying hybrid systems. There is a vast literature on hybrid system verification based on hybrid automata. Here, we cite only some selected methods, that can be classified as follows:

Bounded-time analysis methods analyze systems up to some time horizon. Systems with linear or non-linear dynamics require a time discretization: either a so-called flow-pipe (a set of convex sets over-approximating the possible trajectories) is constructed by set integration [10, 16, 17], or the discretized system is saturated by constraint propagation techniques [15, 27].

Unbounded-time methods are more challenging, because unbounded time raises a termination issue. [18] analyzes systems with *piecewise constant dynamics* with convex polyhedra and solves the termination issue by the use of *widening* [12]. [8] extends this approach, by considering the verification of hybrid systems with a large discrete state space and by combining symbolically properties on Boolean and numerical variables within the abstract interpretation framework. Recently a method exploiting max-strategy iteration on template polyhedra was proposed in [13].

Contributions. Our contributions can be summarized as follows:

1. We present the general principles behind the translation of a simple, yet complete *hybrid data-flow language* that serves as a low-level formalism for languages such as SIMULINK or ZELUS, to *logico-numerical hybrid automata*, *i.e.* an extension of classical hybrid automata by Boolean variables. This extension prepares us w.r.t. the verification of programs with a large Boolean state space.
2. We discuss the various *zero-crossing semantics* that appear in the source simulation language. We propose sound translations to hybrid automata, and we investigate the extent to which these translations preserve the original semantics.

Related work. Recent articles describe translations of hybrid system languages, like SIMULINK/STATEFLOW, to hybrid automata, but they only treat a subset of the ways in which discrete transitions may be activated. A translation of a subset of the SIMULINK/STATEFLOW language to hybrid automata is proposed in [2] for the purpose of verification. They handle STATEFLOW diagrams of which the translation is rather straightforward, but they do not handle proper zero-crossings. Another translation of a subset of the SIMULINK/STATEFLOW language to hybrid automata with the goal of improving simulation coverage is presented in [4], but they only consider deterministic models, and similarly they do not handle zero-crossings. The tool HYLINK [24], which performs a translation of SIMULINK/STATEFLOW models to hybrid automata, targets the applications of verification and controller synthesis. It is restricted to more or less the same subset as [2]. They introduce blocks for specifying non-deterministic inputs as required by verification methods. A formal definition of the translation is ongoing work. The translation of discrete-time SIMULINK models with periodic triggers to LUSTRE is presented in [31]. The inverse of what we are doing, namely the embedding of hybrid automata in a hybrid system language (here SCICOS), is the goal of [25].

Organisation of the article. §§2 and 3 introduce the hybrid data-flow formalism and logico-numerical hybrid automata respectively. §§4 to 7 describe our contributions. After discussing the results in §8 we conclude in §9.

2. HYBRID DATA-FLOW MODEL

SIMULINK and ZELUS are full programming languages with constructs for modularity. In order to abstract from such constructs, we present here a lower-level data-flow formalism that will serve as the generic input language for the translation.

As this formalism is dedicated not only to simulation, but also serves as a specification language, we use the notion of *inputs* constrained by an *assertion* as in LUSTRE [9]. This allows us to give a semantics to the components of a more general system. Simulation can still be performed by connecting a component with inputs to an input generator, see for instance [28] for the simulation of discrete synchronous systems.

Notations. We will use the following notations:

$\boldsymbol{s} = (\boldsymbol{b}, \boldsymbol{x})$	:	state variable vector, with $\boldsymbol{b}$ discrete (Boolean and numerical) and $\boldsymbol{x}$ continuous numerical subvectors, *e.g.* $((b_1, b_2, n_1), x_1, x_2, x_3) \in (\mathbb{B}^2 \times \mathbb{Z}) \times \mathbb{R}^3$
$\boldsymbol{i}$	:	input variable vector, *e.g.* $(\beta_1, \xi_1, \xi_2) \in \mathbb{B} \times \mathbb{R}^2$
$e(\boldsymbol{s}, \boldsymbol{i})$	:	an arithmetic expression without test, *e.g.* $n + 2x + \xi$
$up(e(\boldsymbol{s}, \boldsymbol{i}))$	:	a zero-crossing, *e.g.* $up(x + \xi - n)$
$\varphi^Z(\boldsymbol{s}, \boldsymbol{i})$	:	a logical combination of zero-crossings, *e.g.* $up(z_1) \wedge \neg up(z_2) \vee up(z_3)$
$\phi(\boldsymbol{b})$	:	a Boolean expression over discrete state variables
$\Phi(\boldsymbol{s}, \boldsymbol{i})$	:	an arbitrary expression without zero-crossings

Program model. A hybrid data-flow program is defined by:

$$\left\{ \begin{array}{l} \mathcal{I}(\boldsymbol{s}) \\ \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \wedge \left\{ \begin{array}{l} \dot{\boldsymbol{x}} = \boldsymbol{f}^c(\boldsymbol{s}, \boldsymbol{i}) \\ \boldsymbol{s}' = \boldsymbol{f}^d(\boldsymbol{s}, \boldsymbol{i}) \end{array} \right. \end{array} \right.$$

where the predicate $\mathcal{I}(\boldsymbol{s})$ defines the initial states, the predicate $\mathcal{A}(\boldsymbol{s}, \boldsymbol{i})$ is the global *assertion* constraining the inputs, the continous flow equations $\dot{\boldsymbol{x}} = \boldsymbol{f}^c(\boldsymbol{s}, \boldsymbol{i})$ and the discrete transition functions $\boldsymbol{s}' = \boldsymbol{f}^d(\boldsymbol{s}, \boldsymbol{i})$ are of the form:

$$\dot{\boldsymbol{x}} = \left\{ \begin{array}{c} \cdots \\ \boldsymbol{e}_\ell(\boldsymbol{s}, \boldsymbol{i}) \text{ if } \phi_\ell(\boldsymbol{b}) \\ \cdots \end{array} \right. \qquad \boldsymbol{s}' = \left\{ \begin{array}{c} \cdots \\ \boldsymbol{\Phi}_j(\boldsymbol{s}, \boldsymbol{i}) \text{ if } \varphi_j^Z(\boldsymbol{s}, \boldsymbol{i}) \\ \cdots \end{array} \right.$$

We assume that the conditions ϕ_ℓ define a partition of the discrete state space, and that $\forall \boldsymbol{s} \exists \boldsymbol{i} : \mathcal{A}(\boldsymbol{s}, \boldsymbol{i})$ (*i.e.* the assertion does not constrain the state-space).

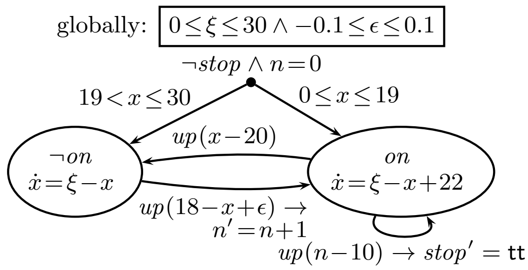
Furthermore the zero-crossing formulas φ_j^Z must be exclusive in order to guarantee determinism, *i.e.* only a single φ_j^Z may be activated at the same instant. Real languages

```
let node main xi eps = (n,x) where
  assert 0<=xi && xi<=30 &&
         -0.1<=eps && eps<=0.1 and
  der x = if on then xi-x+22 else xi-x
           init xi and
  on = (xi<=19) ->
         true every up(18-x+eps)
       | false every up(x-20) and
  n = 0 -> (last n)+1 every up(18-x+eps) and
  stop = false -> true every up(n-10)
```

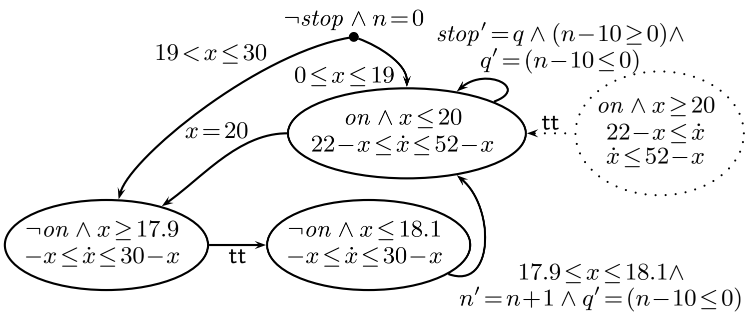
(a) Zelus thermostat

$$
\begin{aligned}
\mathcal{I}(on, stop, n, x) &= \neg stop \wedge n{=}0 \wedge 0 \le x \le 30 \wedge \\
&\quad (x \le 19 \wedge on \vee x > 19 \wedge \neg on) \\
\mathcal{A}((on, stop, n, x), (\xi, \epsilon)) &= 0 \le \xi \le 30 \wedge -0.1 \le \epsilon \le 0.1 \\
\dot{x} &= \begin{cases} \xi - x + 22 & \text{if } on \\ \xi - x & \text{if } \neg on \end{cases} \\
(on', stop', n', x') &= \begin{cases} (\mathsf{ff}, stop, n, x) & \text{if } up(x-20) \\ (\mathsf{tt}, stop, n+1, x) & \text{if } up(18-x+\epsilon) \\ (on, \mathsf{tt}, n, x) & \text{if } up(n-10) \end{cases}
\end{aligned}
$$

(b) Intermediate data-flow model



(c) Partitioned data-flow model



(d) Resulting Hybrid Automaton, with q a state Boolean variable introduced by the translation

Figure 1: Translation of the Thermostat example described in Example 1

achieve this goal using *if-then-else* constructs or with the help of priorities, *e.g.* based on the order zero-crossings occur in the source code.

A second requirement is that $\forall j : \varphi^Z_j \Rightarrow \bigvee_m up(z_m)$, where $up(z_1), \ldots, up(z_M)$ are the zero-crossings occurring in the program; this condition prevents from taking a discrete transition when no zero-crossing is activated.

Although hybrid system languages often include explicit automata representations, for uniformity of presentation we assume that they have first been transformed into data-flow equations, see [11] for instance.

Example 1. *Fig. 1a shows a variant of the classical thermostat example. The input* `xi` *represents the external temperature, the input* `eps` *models the inaccuracy of the temperature sensor*[4]*, the continuous state variable* `x` *is the room temperature, the discrete Boolean state variable* `on` *indicates the state of the heating system, and the discrete integer state variable* `n` *counts the number of times the temperature goes from below to above 18 degrees (modulo the uncertainty). At last, the state variable* `stop` *becomes true when "*`n` *reaches* 10 *from below".*

Fig. 1b shows its translation to our intermediate formalism. Observe that this translation factorizes the evolution of discrete variables according to the zero-crossing conditions.

Semantics of zero-crossings. A *zero-crossing* is an expression of the form $up(z)$ that becomes true when the sign of $z(\boldsymbol{s}, \boldsymbol{i})$, an arithmetic expression without tests, switches from negative to positive during an execution. Instead of just a valuation of variables of the form $(\boldsymbol{s}_k, \boldsymbol{i}_k)$ zero-crossings are interpreted on an execution fragment of the form $(\boldsymbol{s}_{k-1}, \boldsymbol{i}_{k-1}) \to (\boldsymbol{s}_k, \boldsymbol{i}_k)$, *i.e.* two consecutive configurations of an execution trace. We will use the notation $(\boldsymbol{s}_{k-1}, \boldsymbol{i}_{k-1}, \boldsymbol{s}_k, \boldsymbol{i}_k)$ for short. Several interpretations are possible which are discussed in §4. For now, we arbitrarily select the so-called "contact" semantics, formally defined as:

$$(\boldsymbol{s}_{k-1}, \boldsymbol{i}_{k-1}, \boldsymbol{s}_k, \boldsymbol{i}_k) \models up(z(\boldsymbol{s}, \boldsymbol{i})) \text{ iff } \begin{cases} z(\boldsymbol{s}_{k-1}, \boldsymbol{i}_{k-1}) < 0 \\ z(\boldsymbol{s}_k, \boldsymbol{i}_k) \ge 0 \end{cases} \quad (1)$$

In other words, a zero-crossing $up(z)$ is activated (and taken into account for computing the next step $k{+}1$) if the expression z was strictly negative in the previous step $k-1$ and evaluates to some positive value or zero in the current step k.

A zero-crossing formula, *i.e.* a logical combination of zero-crossings, $\varphi^Z(\boldsymbol{s}, \boldsymbol{i})$ is activated iff it evaluates to true when interpreting the zero-crossings $up(z_m)$ occurring in φ^Z using their corresponding constraints over a given $(\boldsymbol{s}_{k-1}, \boldsymbol{i}_{k-1}, \boldsymbol{s}_k, \boldsymbol{i}_k)$ as in (1).

Semantics. We define a trace semantics based on an ideal discretization of the continuous equations, following [6]. This semantics uses the theory of non-standard analysis [23, 29] to model the way typical simulators proceed, relying on a variable-step numerical integration solver (such as Sundials CVODE [21]). Such solvers are given an initial state $\boldsymbol{x}_0$, an ODE $\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t))$, and a finite set of zero-crossing expressions z_j. Then they integrate the ODE until at least one of the zero-crossings is activated. When this happens, the control is given back to the main simulation loop, which executes one or several discrete execution steps before continuing integration.

According to this approach, an execution of a hybrid data-flow program is a trace $(\boldsymbol{s}_0, \boldsymbol{i}_0) \to (\boldsymbol{s}_1, \boldsymbol{i}_1) \to (\boldsymbol{s}_2, \boldsymbol{i}_2) \to \ldots$ such that $\mathcal{I}(\boldsymbol{s}_0)$, $\to = \to_c \cup \to_d$ and

[4]We do not use `eps` in the expression $up(x{-}20)$, in order to show an example of a deterministic zero-crossing.

$$(\boldsymbol{s}_k, \boldsymbol{i}_k) \rightarrow_c (\boldsymbol{s}_{k+1}, \boldsymbol{i}_{k+1}) \iff \mathcal{A}(\boldsymbol{s}_k, \boldsymbol{i}_k) \wedge$$
$$\exists l, \exists \partial > 0 : \begin{cases} \phi_\ell(\boldsymbol{b}_k) \wedge \forall j : \neg\big((\boldsymbol{s}_{k-1}, \boldsymbol{i}_{k-1}, \boldsymbol{s}_k, \boldsymbol{i}_k) \models \varphi_j^Z(\boldsymbol{s}, \boldsymbol{i})\big) \\ (\boldsymbol{b}_{k+1}, \boldsymbol{x}_{k+1}) = (\boldsymbol{b}_k, \boldsymbol{x}_k + \boldsymbol{e}_\ell(\boldsymbol{s}_k, \boldsymbol{i}_k) \cdot \partial) \end{cases}$$
$$(\boldsymbol{s}_k, \boldsymbol{i}_k) \rightarrow_d (\boldsymbol{s}_{k+1}, \boldsymbol{i}_{k+1}) \iff \mathcal{A}(\boldsymbol{s}_k, \boldsymbol{i}_k) \wedge$$
$$\exists j : \begin{cases} (\boldsymbol{s}_{k-1}, \boldsymbol{i}_{k-1}, \boldsymbol{s}_k, \boldsymbol{i}_k) \models \varphi_j^Z(\boldsymbol{s}, \boldsymbol{i}) \wedge \\ \qquad \forall j' < j : \neg\big((\boldsymbol{s}_{k-1}, \boldsymbol{i}_{k-1}, \boldsymbol{s}_k, \boldsymbol{i}_k) \models \varphi_{j'}^Z(\boldsymbol{s}, \boldsymbol{i})\big) \\ \boldsymbol{s}_{k+1} = \Phi_j(\boldsymbol{s}_k, \boldsymbol{i}_k) \end{cases}$$

where ∂ is an infinitesimal (*c.f.* non-standard analysis).

A transition $\rightarrow_c$ corresponds to an infinitesimal continuous-time evolution, which is possible only if no zero-crossing condition φ_j^Z has been activated in the previous execution step. A transition $\rightarrow_d$ corresponds to a discrete transition triggered by the first enabled φ_j^Z.

We pinpoint some properties of this formalism:

(1) Discrete transitions are always guarded by zero-crossings, and continuous modes are always defined by a Boolean expression over discrete variables, which are piecewise constant in continuous time. This is to go sure that a mode change (change of dynamics) can only happen on discrete transitions.

(2) Furthermore, discrete transitions are *urgent*, *i.e.* they must be taken at the first point in time possible.

(3) Zero-crossings may not only be triggered by continuous evolution, but also by discrete transitions. This is the case in Example 1: if the zero-crossing $up(18 - x + \epsilon)$ occurs when $n = 9$, n is first incremented to 10, activating the zero-crossing $up(n - 10)$ which makes *stop* become true. This feature can cause infinite sequences of discrete zero-crossings. Such a behaviour can be avoided by forbidding circular dependencies between states variables through zero-crossings in the source program.

Partitioned representation. The hybrid data-flow model we have defined does not have any concept of control structure. However, for pedagogical purpose, one can partition the state space to generate an explicit automaton which may be easier to understand, see Fig. 1c. When doing this, partial evaluation may be used to simplify expressions and removing infeasible transitions. This has been done on Fig. 1c.

Standardization. As already mentioned the semantics of the hybrid data flow model is based on non-standard analysis, which allows to give an unambiguous meaning to hybrid systems even if they contain Zeno behavior for instance.

However, the semantics of the output formalism of our translation, *i.e.* hybrid automata, relies on standard analysis. Hence non-standard behaviors need to be mapped to standard behaviors. This is called *standardization*: since each standard system has a non-standard representation, a non-standard system is standardizable if it is a non-standard representation of a standard system.

In the following we give some intuitions about the relationships between non-standard and standard analysis; we refer to [23, 7] for further details.

The set ${}^*\mathbb{N}$ of non-standard positive integers is $\mathbb{N}$ augmented by infinitely large integers. The set ${}^*\mathbb{R}$ of non-standard reals contains $\mathbb{R}$, but also (positive and negative) infinitely large and infinitesimally small numbers, and non-standard infinitesimals ∂ are the inverse of infinite numbers. For each real number x, ${}^*\mathbb{R}$ contains non-standard real numbers that are only infinitesimally away from x. The standardization operator for converting non-standard reals to standard reals $st : {}^*\mathbb{R} \rightarrow \mathbb{R}$ identifies these equivalence classes of non-standard reals: $\forall x \in \mathbb{R} : \forall \partial : st(x + \partial) = x$ where ∂ is an infinitesimal $\in {}^*\mathbb{R}$.

W.r.t. continuous evolution, we have the following property: A non-standard sequence consisting of infinitesimal continuous steps

$$(\boldsymbol{s}_0, \boldsymbol{i}_0) \rightarrow \ldots \rightarrow (\boldsymbol{s}_n, \boldsymbol{i}_n)$$

with $n \in {}^*\mathbb{N}$, $\boldsymbol{x}_0 \in \mathbb{R}^p$, $\boldsymbol{x}_n \in \mathbb{R}^p$, has the following standard meaning: assuming that the input sequence $\boldsymbol{i}_0 \rightarrow \ldots \rightarrow \boldsymbol{i}_n$ forms a continuous function $\boldsymbol{i} : [0, \delta] \rightarrow I$, the sequence $\boldsymbol{s}_0 \rightarrow \ldots \rightarrow \boldsymbol{s}_n$ corresponds to a continuous function $\boldsymbol{s} : [0, \delta] \rightarrow S$ with

$$\boldsymbol{x}(\delta') = \boldsymbol{x}_0 + \int_0^{\delta'} \boldsymbol{e}_\ell((\boldsymbol{b}_0, \boldsymbol{x}(t)), \boldsymbol{i}(t))dt$$

for $\delta' \in [0, \delta]$, $\delta = st(n\partial) \in \mathbb{R}^{\geq 0}$, and I and S denote the input and state space respectively.

However, we can write programs that are not standardizable, *i.e.* non-standard and standard meaning differ: For example the program fragment $b' = (x > 0)$ if $up(x)$ with "crossing" semantics (see §4) gives us $b' = \mathtt{tt}$ in the non-standard interpretation, but $b' = \mathtt{ff}$ in the standard interpretation.

Naturally, we can only correctly translate standardizable programs.

3. LOGICO-NUMERICAL HYBRID AUTOMATA

Hybrid automata [3, 19, 20] are a well-established formalism for modelling hybrid systems. Our definition is more general in the sense that we allow also Boolean variables and tests in the expressions appearing in the automata. Fig. 1d depicts an example of a hybrid automaton.

DEFINITION 1. *A logico-numerical hybrid automaton (HA) is a directed graph defined by* $\langle L, F, J, \Sigma^0 \rangle$ *where*

- L *is the finite set of locations,*
- $F : L \rightarrow \mathcal{V}$ *is a function that returns for each location the flow relation* $V(\boldsymbol{s}, \dot{\boldsymbol{x}}) \in \mathcal{V}$ *relating the state variables* $\boldsymbol{s}$ *and the time-derivatives* $\dot{\boldsymbol{x}}$ *of the numerical state variables, and*
- $J \subseteq L \times \mathcal{R} \times L$ *defines a finite set of arcs between locations with the discrete transition relation* $R(\boldsymbol{s}, \boldsymbol{s}') \in \mathcal{R}$ *over the state variables* $\boldsymbol{s}$.
- $\Sigma^0 : L \rightarrow \mathcal{S}$ *is a function that returns for each location the set of initial states* $S^0 \in \mathcal{S}$, *which have to satisfy* $\forall \ell : \forall \boldsymbol{s} : \Sigma^0(\ell)(\boldsymbol{s}) \Rightarrow \exists \dot{\boldsymbol{x}} : F(\ell)(\boldsymbol{s}, \dot{\boldsymbol{x}})$.

Further notations:

- $C_\ell(\boldsymbol{s}) = \exists \dot{\boldsymbol{x}} : V(\boldsymbol{s}, \dot{\boldsymbol{x}})$ is the *staying condition* of the flow $V = F(\ell)$.
- $G_{\ell,\ell'}(\boldsymbol{s}) = \exists \boldsymbol{s}' : R(\boldsymbol{s}, \boldsymbol{s}')$ is the *guard* of the arc $(\ell, R, \ell') \in J$.

Semantics. We use the following definitions: Let $T_{[0,\delta]}$ be the set of differentiable trajectories $[0, \delta] \rightarrow \mathbb{R}^n$. The function $flow_V$ returns the set of end states of trajectories $\tau \in T$ starting in the given state and that obey the flow relation V:

$$flow_V(\boldsymbol{b}, \boldsymbol{x}) = \left\{ (\boldsymbol{b}, \boldsymbol{x}') \;\middle|\; \begin{array}{l} \exists \delta > 0, \exists \tau \in T_{[0,\delta]} : \\ \quad \tau(0) = \boldsymbol{x} \ \wedge \ \tau(\delta) = \boldsymbol{x}' \ \wedge \\ \quad \forall \delta' \in [0, \delta] : C(\boldsymbol{b}, \tau(\delta')) \wedge \\ \quad \forall \delta' \in (0, \delta) : V((\boldsymbol{b}, \tau(\delta')), \dot{\tau}(\delta')) \end{array} \right\}$$

We define the concrete semantics in terms of an *execution* of a hybrid automaton, which is a (possibly) infinite trace $(\ell_0, \boldsymbol{s}_0) \to (\ell_1, \boldsymbol{s}_1) \to (\ell_2, \boldsymbol{s}_2) \to \ldots$ with $\to = \to_c \cup \to_d$ and

$$\begin{array}{lcl} (\ell, \boldsymbol{s}) \to_c (\ell', \boldsymbol{s}') & \Leftrightarrow & \ell = \ell' \wedge V = F(\ell) \wedge \boldsymbol{s}' \in \mathit{flow}_V(\{\boldsymbol{s}\}) \\ (\ell, \boldsymbol{s}) \to_d (\ell', \boldsymbol{s}') & \Leftrightarrow & \exists (\ell, R, \ell') \in J : R(\boldsymbol{s}, \boldsymbol{s}') \wedge C_{\ell'}(\boldsymbol{s}') \end{array}$$

If we eliminate all Boolean variables by enumerating their valuations and encoding them with locations, the semantics above will be equivalent to the semantics of standard hybrid automata that deal only with numerical variables.

The concrete semantics of hybrid automata exhibit three kinds of non-determinism:
- Non-determinism w.r.t. *flow* transitions, *i.e.* the choice between different continuous evolutions due to the differential inclusions defined by the vector field V.
- Non-determinism w.r.t. *flow and jump* transitions: The choice between flow and jump transitions due to an overlapping of staying condition and guards.
- Non-determinism w.r.t. *jump* transitions, which is the choice between several jump transitions.

4. SEMANTICS OF ZERO-CROSSINGS

The fundamental difference between the zero-crossing concept used in our input language and the combination of staying and jump conditions in our output language is that the activation of a zero-crossings depends on the history (*i.e.* a part of the past trajectory) whereas the truth value of staying and jump conditions depends only on the current state.

Continuous vs. discrete zero-crossings. As mentioned in §2, a zero-crossing can be activated in two ways:
- It can be triggered by a continuous time evolution, as $up(x-20)$ in Fig. 1c; in this case it is active during the second step of an execution fragment $\boldsymbol{s} \xrightarrow{\boldsymbol{i}}_c \boldsymbol{s}' \xrightarrow{\boldsymbol{i}'}_d \boldsymbol{s}''$;
- It can be triggered by a discrete transition, as $up(n-10)$ in Fig. 1c; in this case it is active during the second step of an execution fragment $\boldsymbol{s} \xrightarrow{\boldsymbol{i}}_d \boldsymbol{s}' \xrightarrow{\boldsymbol{i}'}_d \boldsymbol{s}''$;

Because a zero-crossing may depend on both discrete and continuous variables, the same zero-crossing $up(e)$ can be triggered in both ways in an execution. We will use the terms *continuous* (resp. *discrete*) *zero-crossing* for indicating its source of activation.

Three semantics for zero-crossings. We consider an execution fragment $\xrightarrow{\boldsymbol{i}_{k-1}} \boldsymbol{s}_{k-1} \xrightarrow{\boldsymbol{i}_k} \boldsymbol{s}_k$ and we define $z_k = z(\boldsymbol{s}_k, \boldsymbol{i}_k)$. There are three natural choices for the semantics of zero-crossings:

- "At-zero" semantics : $z_{k-1} \le 0 \wedge z_k \ge 0$
- "Contact" semantics : $z_{k-1} < 0 \wedge z_k \ge 0$
- "Crossing" semantics : $z_{k-1} \le 0 \wedge z_k > 0$

Figs. 3b, 3e and 3h illustrate the activation of continuous zero-crossings for some typical trajectories according to each semantics.

The last two semantics are used in simulators. The zero-crossing semantics of SIMULINK is the disjunction of these semantics. In MODELICA it is up to the programmer to choose between these two semantics.

We state the first option, because it fits better to the semantics of hybrid automata (as it does not involve strict inequalities).

Chattering behaviour. An issue specific to the "crossing" semantics is that it is possible to write programs that produce executions that contain periodic sequences of infinitesimal continuous evolutions with distinct dynamics. This happens for example when a trajectory *chatters* along a surface with opposed zero-crossings, like in the following example.

EXAMPLE 2. *(see Fig. 2)*

$$\mathcal{I} = (b \wedge x = -1 \wedge y = 0) \qquad \mathtt{tt} \to \left\{ \begin{array}{l} b' = \left\{ \begin{array}{ll} \mathtt{ff} & \text{if } up(x) \\ \mathtt{tt} & \text{if } up(-x) \end{array} \right. \\ \dot{x} = 1 \\ \dot{y} = \left\{ \begin{array}{ll} 1 & \text{if } b \\ -1 & \text{if } \neg b \end{array} \right. \end{array} \right.$$

In some cases it is possible to standardize such systems by identifing the chattering behavior and replace it by a so-called *sliding mode* [14, 32, 1], *i.e.* a dynamics that defines the corresponding trajectory "in" this surface. However, this is not feasible in the general case (*i.e.* a specification which does not correspond necessarily to a physical model). Thus, we will translate such programs into hybrid automata that allow chattering in their concrete semantics.

(a) Chattering trajectory (b) Equivalent sliding mode

Figure 2: Sliding modes

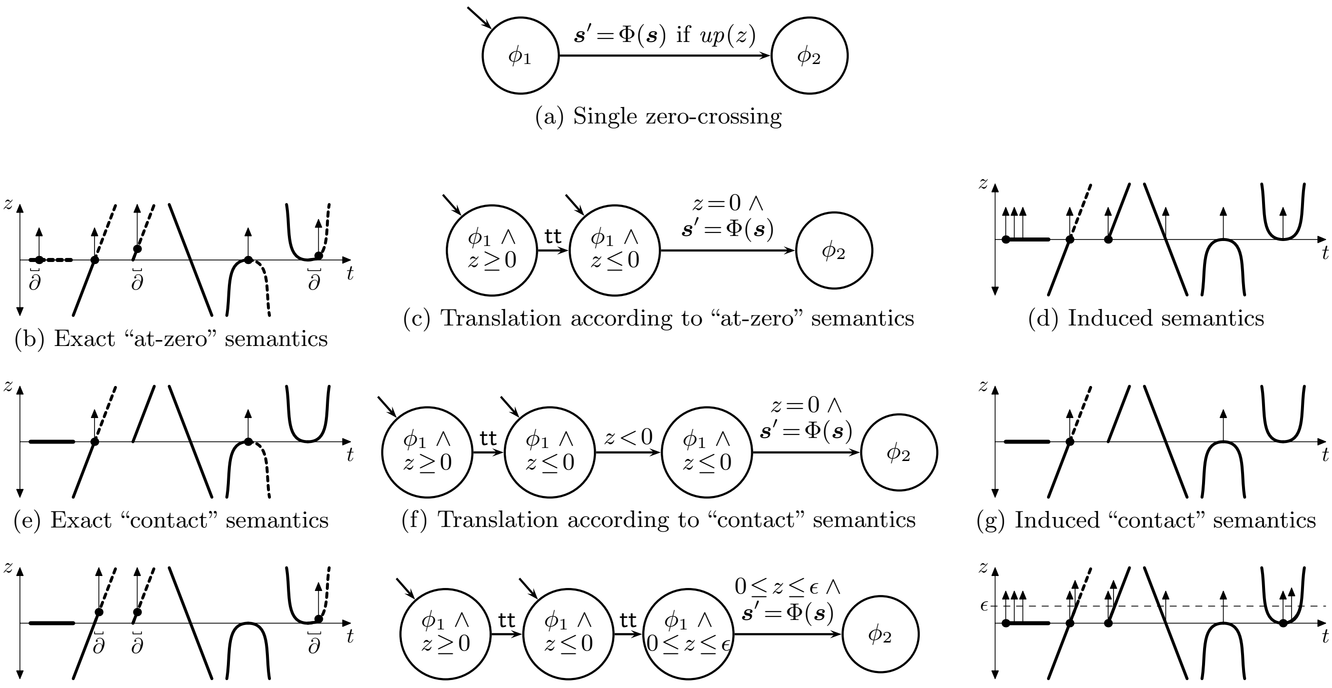
§5 focuses on the translation of continuous zero-crossings, whereas §6 will discuss the case of discrete zero-crossings, the translation of which is much less dependent on the choice of one of the three zero-crossing semantics. However, because of the limitations of the hybrid automata model, in all cases the translation will add behavior that is not present in the original program.

5. TRANSLATION OF CONTINUOUS ZERO-CROSSINGS

5.1 Simplest case: one zero-crossing, no inputs

We investigate here the translation of continuous zero-crossings of the form $up(z(\boldsymbol{x}))$: for the sake of simplicity, we assume that there are neither inputs $\boldsymbol{i}$ nor discrete variables $\boldsymbol{b}$ in z. We consider the simple case of an origin location l_1 with a single discrete transition $\boldsymbol{s}' = \Phi(\boldsymbol{s})$ if $up(z(\boldsymbol{x}))$ going from l_1 to a location l_2, such that $\phi_1(\boldsymbol{b}) \wedge (\boldsymbol{s}' = \Phi(\boldsymbol{s})) \Rightarrow \phi_2(\boldsymbol{b}')$, see Fig. 3a. As the satisfaction of a zero-crossing depends on the history, the principle of the translation is to add locations to record the history of the continuous evolution.

"At-zero" semantics. The translation of "at-zero" semantics ($z_{k-1} \le 0 \wedge z_k \ge 0$) is depicted in Fig. 3c. The origin location is partitioned in two locations: there is a discrete transition from left to right, but not from right to left to force the urgency of the discrete transition when $z = 0$ is reached from below 0. The zero-crossing condition translates to $z = 0$.



(a) Single zero-crossing

(b) Exact “at-zero” semantics

(c) Translation according to “at-zero” semantics

(d) Induced semantics

(e) Exact “contact” semantics

(f) Translation according to “contact” semantics

(g) Induced “contact” semantics

(h) Exact “crossing” semantics

(i) Translation according to “crossing” semantics

(j) Induced “crossing” semantics

Notes: (1) The arrows pointing upwards indicate on the left-hand side the points where zero-crossings are activated, and on the right-hand side the points where the jump transition may be taken non-deterministically. The dotted trajectories indicate that the preceding transition is urgent.
(2) When $\boldsymbol{s}$ does not appear in the jump condition of a HA, the equality $\boldsymbol{s}' = \boldsymbol{s}$ is implicit.
(3) The flow function $\dot{\boldsymbol{x}} = \boldsymbol{f}^c(\boldsymbol{s})$ in the first location in Fig. 3a, translates to the flow relation $V_1(\dot{\boldsymbol{x}}, \boldsymbol{s}) = (\phi_1(\boldsymbol{b}) \wedge \dot{\boldsymbol{x}} = \boldsymbol{f}^c(\boldsymbol{s}))$, not shown in the figures.

Figure 3: Zero-crossing semantics of the hybrid data-flow language and their translations. From left to right: typical trajectories in the original semantics, proposed translation to hybrid automata, and their typical trajectories.

The rationale for the condition $z = 0$ is based on the assumption of continuity of the function $z(t) = z(\boldsymbol{x}(t))$ and the urgency of the zero-crossing: $z(t_{k-1}) < 0 \wedge z(t_k) \geq 0$ with $t_{k-1} < t_k$ implies that there exists $t \in (t_{k-1}, t_k]$ such that $z(t) = 0$.

This translation induces two kinds of approximations in terms of executions:

– We lose urgency for all trajectories but the second one in Fig. 3d. In case of the first trajectory the zero-crossing may be triggered in a dense interval of time.
– We add a jump transition in the fourth trajectory because one is not able to distinguish whether the state $z = 0$ is reached from below or from above 0.

We will not consider any more the “at-zero” semantics in the sequel, as – to our knowledge – it is not used by any simulation tool.

“Contact” semantics. In order to translate the “contact” semantics defined as $z_{k-1} < 0 \wedge z_k \geq 0$, we split the original location into three locations as depicted on Fig. 3f. The two locations with the staying condition $z \leq 0$ are connected by a transition guarded by $z < 0$: this is in order to check that the trajectory was actually strictly below zero before touching zero. This prohibits the triggering of the jump transition in the first, third and last trajectory in Fig. 3g. This induces the following approximation:

– The loss of urgency for the fifth trajectory that touches (possibly several times) the line $z = 0$ from below.

Observe, that the “at-zero” translation in Fig. 3c is actually a sound translation of the “contact” semantics, though with coarser approximations.

“Crossing” semantics. The “crossing” semantics ($z_{k-1} \leq 0 \wedge z_k > 0$) is more subtle to translate. By continuity of the function $z(t) = z(\boldsymbol{x}(t))$ we can deduce that $z(t) = 0$ is valid at the zero-crossing point in standard semantics: by standardizing $z(t) \leq 0 \wedge z(t+\partial) > 0$ we get $st(z(t)) = st(z(t+\partial)) = 0$.

However, we cannot simply reuse the “at-zero” translation in Fig. 3c, because it is not sound w.r.t. chattering behaviors: in Ex. 2 time cannot advance, because only discrete transitions can be taken. Since we do not rely on standardizing chattering behaviors we have to allow chattering in the standard semantics. For that reason we allow the trajectories to actually go beyond zero, but only up to a constant $\epsilon > 0$ (see Fig. 3i). As a consequence, we have the following approximation:

– Urgency is completely lost. In case of the second, third and last trajectories the zero-crossing may be triggered in a bounded time interval with a dense interval of values for z (see Fig. 3j).

Observe, that this translation simulates the translations of the other two semantics.

REMARK 1 (CHOICE OF ϵ). *Any translation involving an ϵ close to zero is not really well-suited for verification: computations with arbitrary-precision rationals become indeed very expensive (*e.g.* least common denominators become huge).*

5.2 One zero-crossing with inputs

Now we investigate the translation of zero-crossings of the form $up(z(\boldsymbol{x}, \boldsymbol{i}))$, where the inputs $\boldsymbol{i}$ have to satisfy an assertion $\mathcal{A}(\boldsymbol{s}, \boldsymbol{i})$, see §2. We assume that in the discrete infinitesimal semantics of §2 inputs tend to continuous trajectories (between two discrete transitions). Inputs allow us to introduce non-determinism in a model, as illustrated by Fig. 1. The principle of the translation remains the same as in the previous section and depicted on Fig. 3, except that the computation of jump and flow transition relations involves an existential quantification of the inputs $\boldsymbol{i}$. The process is illustrated by Fig. 4.

We use the notation $\boxslash\psi = \overline{\neg\psi}$, where $\bar{\cdot}$ denotes the topological closure operator. We have for instance $\boxslash(z \leq 0) = z \geq 0$.

Considering the "contact" semantics and using the continuity of the function $z(\boldsymbol{x}(t), \boldsymbol{i}(t))$ during continuous evolution (see §2) the condition

$$\begin{aligned}\exists \boldsymbol{i}_{k-1}, \boldsymbol{i}_k : z(\boldsymbol{x}_{k-1}, \boldsymbol{i}_{k-1}) < 0 \wedge \mathcal{A}(\boldsymbol{s}_{k-1}, \boldsymbol{i}_{k-1}) \wedge \\ z(\boldsymbol{x}_k, \boldsymbol{i}_k) \geq 0 \quad \wedge \mathcal{A}(\boldsymbol{s}_k, \boldsymbol{i}_k) \wedge \boldsymbol{s}' = \boldsymbol{\Phi}(\boldsymbol{s}_k, \boldsymbol{i}_k)\end{aligned}$$

is equivalent to

$$\begin{aligned}\exists \boldsymbol{i}_{k-1}, \boldsymbol{i}_k : z(\boldsymbol{x}_{k-1}, \boldsymbol{i}_{k-1}) < 0 \wedge \mathcal{A}(\boldsymbol{s}_{k-1}, \boldsymbol{i}_{k-1}) \wedge \\ z(\boldsymbol{x}_k, \boldsymbol{i}_k) = 0 \quad \wedge \mathcal{A}(\boldsymbol{s}_k, \boldsymbol{i}_k) \wedge \boldsymbol{s}' = \boldsymbol{\Phi}(\boldsymbol{s}_k, \boldsymbol{i}_k)\end{aligned}$$

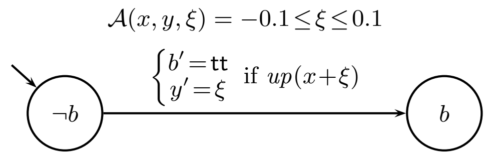
which in turn is equivalent to

$$\begin{aligned}\exists \boldsymbol{i} : z(\boldsymbol{x}_{k-1}, \boldsymbol{i}) < 0 \wedge \mathcal{A}(\boldsymbol{s}_{k-1}, \boldsymbol{i}) \wedge \\ \exists \boldsymbol{i} : z(\boldsymbol{x}_k, \boldsymbol{i}) = 0 \quad \wedge \mathcal{A}(\boldsymbol{s}_k, \boldsymbol{i}) \wedge \boldsymbol{s}' = \boldsymbol{\Phi}(\boldsymbol{s}_k, \boldsymbol{i})\end{aligned} \tag{2}$$

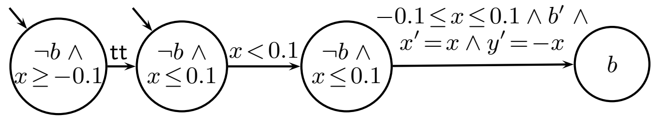
- The first line of Eqn. (2) defines the new guard of the transition between the second and third locations of Fig 3f:
$$\exists \boldsymbol{i} : \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \wedge z(\boldsymbol{x}, \boldsymbol{i}) < 0$$
- The second line gives us the new transition relation between the third and fourth locations of Fig. 3f:
$$R(\boldsymbol{s}, \boldsymbol{s}') = \exists \boldsymbol{i} : \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \wedge z(\boldsymbol{x}, \boldsymbol{i}) = 0 \wedge \boldsymbol{s}' = \Phi(\boldsymbol{s}, \boldsymbol{i})$$
- The new flow relation of the second and third locations is
$$\exists \boldsymbol{i} : \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \wedge z(\boldsymbol{x}, \boldsymbol{i}) \leq 0 \wedge \phi_1(\boldsymbol{b}) \wedge \dot{\boldsymbol{x}} = \boldsymbol{e}_1(\boldsymbol{s}, \boldsymbol{i})$$
which induces the staying condition
$$\psi_{23}(\boldsymbol{s}) = \exists \boldsymbol{i} : \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \wedge z(\boldsymbol{x}, \boldsymbol{i}) \leq 0)$$
- The new flow relation of the first location of Fig. 3f is
$$V_1(s, \dot{\boldsymbol{x}}) = \exists \boldsymbol{i} : \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \wedge z(\boldsymbol{x}, \boldsymbol{i}) \geq 0 \wedge \phi_1(\boldsymbol{b}) \wedge \dot{\boldsymbol{x}} = \boldsymbol{e}_1(\boldsymbol{s}, \boldsymbol{i})$$

The result is illustrated by Fig. 4b. One can strengthen the flow relation V_1 by conjoining it with $\boxslash\psi_{23}$, so as to minimize the non-determinism between staying in the first location or jumping to the second location, as done in Fig. 4c.[5]

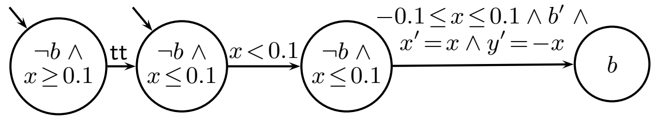
[5] We use the operator $\boxslash$ instead of $\neg$ in order to obtain a topologically closed flow relation.



(a) Zero-crossing with input ξ



(b) Translation according to "contact" semantics



(c) Strengthening the staying condition of the first location

Figure 4: Translation of a continuous zero-crossing with inputs, described in Example 3

EXAMPLE 3. *Fig. 4b illustrates this translation on the original system of Fig. 4a, where b, x, y are state variables and ξ is a numerical input variable constrained by the assertion. The jump condition of the rightmost transition is obtained from* $\exists\xi : (-0.1 \leq \xi \leq 0.1 \wedge x+\xi = 0 \wedge b' = \mathsf{tt} \wedge x' = x \wedge y' = \xi)$
$= \exists\xi : (-0.1 \leq x \leq 0.1 \wedge x = -\xi \wedge b' \wedge x' = x \wedge y' = -x)$
$= -0.1 \leq x \leq 0.1 \wedge b' \wedge x' = x \wedge y' = -x$
Observe that we obtain the non-trivial relation $y = -x$ after the jump transition.

5.3 Logical combinations of zero-crossings

We consider here a discrete transition function of the form $\boldsymbol{s}' = \boldsymbol{\Phi}(\boldsymbol{s}, \boldsymbol{i})$ if $\varphi^Z(\boldsymbol{s}, \boldsymbol{i})$ where φ^Z is a logical combination of zero-crossings $up(z_1), \ldots, up(z_M)$ satisfying the assumption of §2.

Why do we need such logical combinations? Conjunctions and negations typically occur when combining two parallel equations $s_i' = \Phi_i$ if $up(z_i)$ for $i = 1, 2$, which results in an equation

$$(s_1', s_2') = \begin{cases} (\Phi_1, \Phi_2) & \text{if } up(z_1) \wedge up(z_2) \\ (\Phi_1, s_2) & \text{if } up(z_1) \wedge \neg up(z_2) \\ (s_1, \Phi_2) & \text{if } up(z_2) \wedge \neg up(z_1) \end{cases}$$

A specification of the form $s' = \begin{cases} \Phi_1 & \text{if } up(z_1) \\ \Phi_2 & \text{else if } up(z_2) \end{cases}$

is similarly translated to $s' = \begin{cases} \Phi_1 & \text{if } up(z_1) \\ \Phi_2 & \text{if } \neg up(z_1) \wedge up(z_2) \end{cases}$

Disjunctions allow to express that the same transition may be triggered by different zero-crossings:

$$s' = \Phi \text{ if } up(z_1) \vee up(z_2)$$

Because successive graph refinements are cumbersome to describe, we reformulate the translation scheme of the previous sections by using additional discrete state variables to the system, rather than by introducing locations. This will make it easier to explain this generalization. We sketch this principle using the "contact" semantics (the translation for the general case will be presented in §7).

To encode locations, we add M discrete state variables $q_1 \dots q_M$ of the enumerated type $\{\mathsf{above}, \mathsf{below}, \mathsf{ready}\}$ for each distinct zero-crossing $up(z_m)$ occuring in the zero-crossing formulas φ^Z.

– Their transition relations are defined as

$$\begin{array}{rl} R_m = & \textbf{match } q_m \textbf{ with} \\ & \mathsf{above} \rightarrow q'_m \in \{\mathsf{above}, \mathsf{below}\} \\ & \mathsf{below} \rightarrow z_m < 0 \wedge q'_m = \mathsf{ready} \qquad\qquad \vee\, q'_m = q_m \\ & \mathsf{ready} \rightarrow z_m = 0 \wedge q'_m \in \{\mathsf{above}, \mathsf{below}\} \vee q'_m = q_m \end{array}$$

– The staying condition defined by the zero-crossing $up(z_m)$ is:

$$\begin{array}{rlll} C_m = & \multicolumn{3}{l}{\textbf{match } q_m \textbf{ with}} \\ & \mathsf{above} & \rightarrow & z_m \geq 0 \\ & \mathsf{below} & \rightarrow & z_m \leq 0 \\ & \mathsf{ready} & \rightarrow & z_m \leq 0 \end{array}$$

– The activation condition G_m associated to the zero-crossing $up(z_m)$ is

$$G_m = (q_m = \mathsf{ready}) \wedge (z_m = 0)$$

We can now build the global flow and discrete transition relations:

$$\begin{aligned} R((\boldsymbol{q}, \boldsymbol{s}), (\boldsymbol{q}', \boldsymbol{s}')) &= \left(\textstyle\bigwedge_m R_m\right) \wedge (\neg H \wedge \boldsymbol{s}' = \boldsymbol{s} \vee H \wedge \boldsymbol{s}' = \boldsymbol{\Phi}) \\ V((\boldsymbol{q}, \boldsymbol{s}), \dot{\boldsymbol{x}}) &= \left(\textstyle\bigwedge_m C_m\right) \wedge \left(\textstyle\bigvee_\ell (\phi_\ell(\boldsymbol{b}) \wedge \dot{\boldsymbol{x}} = \dot{\boldsymbol{e}}_\ell(\boldsymbol{s}))\right) \end{aligned} \tag{3}$$

with $H = \varphi^Z[\forall m : up(z_m) \leftarrow G_m]$ where $e[x \leftarrow y]$ means that x is substituted by y in expression e.

In order to obtain an explicit automaton one has to enumerate the valuations of the discrete state variables $\boldsymbol{q}$ and to encode them into explicit locations (see §8).

It is interesting to mention that this translation keeps enough information in order to preserve urgency in case of conjunctions like $s' = \Phi$ if $up(z_1) \wedge up(z_2)$ where the trajectory can move all around the intersection $z_1 = 0 \wedge z_2 = 0$ while not satisfying both zero-crossings at the same time. Fig. 5 gives an illustration of such a trajectory.

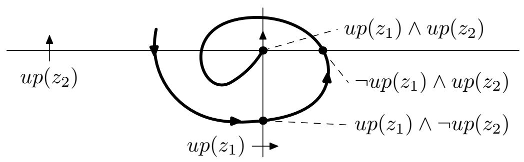


Figure 5: Conjunction of two zero-crossings

6. TRANSLATION OF DISCRETE ZERO-CROSSINGS

Discrete zero-crossings are activated by discrete transitions. Discrete zero-crossings occur in so-called *zero-crossing cascades*, which are sequences of zero-crossings of which the first one is triggered by continuous evolution, whereas the others are discrete zero-crossings. Example 1 contains such a zero-crossing cascade, which is commented in §2 point (3).

The translation explained in §5.1 is not sound for discrete zero-crossings, because we have supposed that the zero-crossings are activated by continuous evolution.

Principle of translation. The translation that we propose applies the same principle as above to encode the history of the execution into locations (using discrete state variables).

We explain it using the "contact" semantics (again without inputs and logical combinations of zero-crossings). We consider $\boldsymbol{s}' = \boldsymbol{\Phi}$ if $up(z)$ and we introduce a Boolean variable q^d, which holds at each step k the value of $z < 0$ at step $k-1$.

– The evolution of q^d is defined by the initial state $q^d = \mathsf{ff}$ and the relation $R^d = ((q^d)' = (z < 0))$;

– The condition $up(z_m)$ is translated to the activation condition $G^d = (q^d \wedge z \geq 0)$;

– The global transition relation R is generated as in Eqn. (3).

Interrupting continuous evolution. The transitions as translated above are not urgent, *i.e.* the continuous states can evolve on intermediate states of a cascade. We need to prohibit this evolution explicitly if one of the discrete zero-crossings is activated. This is done by strengthening the global flow relation $V(\boldsymbol{q}, \boldsymbol{s}, \dot{\boldsymbol{x}})$ with $V' = V \wedge \boxbslash G^d$.

In case of inputs, we have $G^d = (q^d \wedge z(\boldsymbol{s}, \boldsymbol{i}) \geq 0)$ and we take $V' = V \wedge \boxbslash(\forall \boldsymbol{i} : G^d)$: the idea is that in a state $(\boldsymbol{q}, \boldsymbol{s})$, if the discrete zero-crossing is activated for any input (*i.e.* $\forall \boldsymbol{i} : G^d$), then the continuous evolution is blocked. Otherwise, for some input, the discrete zero-crossing is not activated and the continuous evolution should be possible.

Remark 2. *A zero-crossings can be both discrete and continuous,* e.g. $up(x+n)$. *In this case it is translated twice: once as a continuous zero-crossing and a second time as a discrete one.*

Remark 3 (Compressing Cascades). *Zero-crossing cascades can be "compressed" into a single discrete transition triggered by a continuous zero-crossing by* composing *the discrete transitions forming the cascade. This is possible if there are no instantaneous cyclic dependencies between the variables. The advantage of this kind of preprocessing is that the translation does not have to deal with discrete zero-crossings. However, care must be taken w.r.t. safety verification, because this transformation does not preserve the set of reachable states (it removes intermediate states).*

7. THE COMPLETE TRANSLATION

We give here the formulas for the complete translation of a hybrid data-flow program

$$\mathcal{I}(\boldsymbol{s}) \quad , \quad \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \wedge \left\{ \begin{array}{l} \dot{\boldsymbol{x}} = \left\{ \begin{array}{c} \dots \\ \boldsymbol{e}_\ell(\boldsymbol{s}, \boldsymbol{i}) \text{ if } \phi_\ell(\boldsymbol{b}) \\ \dots \end{array} \right. \\ \boldsymbol{s}' = \left\{ \begin{array}{c} \dots \\ \boldsymbol{\Phi}_j(\boldsymbol{s}, \boldsymbol{i}) \text{ if } \varphi_j^Z(\boldsymbol{s}, \boldsymbol{i}) \\ \dots \end{array} \right. \end{array} \right.$$

as defined in §2 to a hybrid automaton by combining all the concepts presented in §5 and §6.

We use the notation ζ_m^σ to denote the constraints induced by a zero-crossing $up(z_m)$, *e.g.* $\zeta_m^{\cdot = 0} = \left(z_m = 0\right)$ or $\zeta_m^{0 < \cdot \leq \epsilon} = \left(0 < z_m \leq \epsilon\right)$.

Discrete zero-crossings. For each discrete zero-crossing $up(z_m)$ we introduce a Boolean state variable q_m^d.

$$R_m^d = ({q'}_m^d = \zeta_m^\sigma)$$
$$G_m^d = (q_j^d \wedge \zeta_m^{\overline{\sigma}})$$

	σ	$\overline{\sigma}$
"contact"	$\cdot < 0$	$\cdot \geq 0$
"crossing"	$\cdot \leq 0$	$\cdot > 0$

Continuous zero-crossings. For each continuous zero-crossing $up(z_m)$ we introduce a state variable q_m^c.

– Their transition relations are defined as follows:

	θ	σ
"contact"	$\cdot < 0$	$\cdot = 0$
"crossing"	$\cdot = 0$	$0 \leq \cdot \leq \epsilon$

$$
\begin{array}{rl}
R^c_m = & \textbf{match } q^c_m \textbf{ with} \\
& \mathsf{above} \rightarrow {q'}^c_m \in \{\mathsf{above}, \mathsf{below}\} \\
& \mathsf{below} \rightarrow ((q^c_m)' = q^c_m) \vee ((q^c_m)' = \mathsf{ready}) \wedge \zeta^\theta_m \\
& \mathsf{ready} \rightarrow (q^c_m)' \in \{\mathsf{above}, \mathsf{below}\} \wedge \zeta^\sigma_m \vee (q^c_m)' = q^c_m
\end{array}
$$

– The activation conditions G^c_m are defined as

	σ
"contact"	$\cdot = 0$
"crossing"	$0 \leq \cdot \leq \epsilon$

$$G^c_m = (q^c_m = \mathsf{ready}) \wedge \zeta^\sigma_m$$

– Using $\psi = \bigvee_\ell \big(\phi_\ell(\boldsymbol{b}) \wedge \dot{\boldsymbol{x}} = \boldsymbol{e}_\ell(\boldsymbol{s}, \boldsymbol{i})\big)$ we define the partial flow relations (containing the staying conditions):

	σ
"contact"	$\cdot \leq 0$
"crossing"	$0 \leq \cdot \leq \epsilon$

$$
\begin{array}{rl}
V_m = & \textbf{match } q^c_m \textbf{ with} \\
& \mathsf{above} \rightarrow \left\{ \begin{array}{l} \big(\exists \boldsymbol{i} : \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \wedge \zeta_m^{\cdot\geq 0} \wedge \psi\big) \wedge \\ \big(\boxslash\, \exists \boldsymbol{i} : \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \wedge \zeta_m^{\cdot\leq 0}\big) \end{array} \right. \\
& \mathsf{below} \rightarrow \exists \boldsymbol{i} : \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \wedge \zeta_m^{\cdot\leq 0} \wedge \psi \\
& \mathsf{ready} \rightarrow \exists \boldsymbol{i} : \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \wedge \zeta^\sigma_m \wedge \psi
\end{array}
$$

Transition relations. We define $G_m = G^c_m \vee G^d_m$ and $R_m = R^c_m \wedge R^d_m$. Now, we can finally put things together and define the jump and flow transition relations:

$$
R((\boldsymbol{q}, \boldsymbol{s}), (\boldsymbol{q}', \boldsymbol{s}')) = \exists \boldsymbol{i} : \left\{ \begin{array}{l} \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \wedge \big(\bigwedge_m R_m\big) \wedge \\ \Big(\neg\big(\bigvee_j H_j\big) \wedge \boldsymbol{s}' = \boldsymbol{s} \vee \\ \qquad \bigvee_j \big(H_j \wedge \boldsymbol{s}' = \boldsymbol{\Phi}_j\big)\Big) \end{array} \right.
$$

$$
V((\boldsymbol{q}, \boldsymbol{s}), \dot{\boldsymbol{x}}) = \left\{ \begin{array}{l} \bigvee_m V_m((\boldsymbol{q}, \boldsymbol{s}), \dot{\boldsymbol{x}}) \wedge \\ \quad \boxslash \big(\forall \boldsymbol{i} : \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \wedge \bigvee_j H^d_j\big) \end{array} \right.
$$

with $H_j = \varphi^Z_j[\forall m : up(z_m) \leftarrow G_m]$.

We obtain a hybrid automaton $\langle \{\ell_0\}, F, J, \Sigma^0 \rangle$ with
- $F(\ell_0) = V$,
- $J = \{(\ell_0, R, \ell_0)\}$, and
- $\Sigma^0(\ell_0) = \{(\boldsymbol{q}, \boldsymbol{s}) \mid \boldsymbol{q}^d = \mathbf{ff} \wedge \bigwedge_m q^c_m \in \{\mathsf{above}, \mathsf{below}\} \wedge \mathcal{I}(\boldsymbol{s})\}$

For the proofs of the complete translation we refer to [30].

8. DISCUSSION

We have presented the complete translation of a hybrid data-flow specification to a hybrid automaton. However, further preprocessing steps are necessary to enable verification using classical hybrid analysis methods.

Explicit representation. As explained in §5.3 we have chosen to present our translation by encoding the locations of the HA with N additional finite-state variables $\boldsymbol{q}$. This results in a HA with a single location and a single self-loop jump transition. Of course, it is possible to expand this "compressed" representation into a more explicit one, such as shown in Figs. 3 and 4. This is done by enumerating the valuations of these finite-state variables and by partitioning the system into these $\mathcal{O}(2^N)$ states. As already mentioned in §2, partial evaluation may be used to simplify expressions and to remove infeasible jump transitions.

Convexification of staying conditions and guards. The induced staying conditions $C(\boldsymbol{s})$ and guards $G(\boldsymbol{s})$ of jump transitions might be non-convex w.r.t. numerical constraints. For verification it is necessary to transform these conditions into the form $\bigvee_k \Delta_k$ with $\Delta_k = \phi_k(\boldsymbol{b}) \wedge \varphi^C_k(\boldsymbol{s})$, where $\phi_k(\boldsymbol{b})$ is an arbitrary formula over Boolean variables and $\varphi^C_k(\boldsymbol{s})$ is a conjunction of numerical constraints. For staying conditions, the system has to be partitioned according to these Δ_k; for guards, arcs are split w.r.t. the Δ_k.

Example 4. *The non-convex flow transition V*

$$
\begin{array}{l}
(b_1 \vee \neg b_2) \wedge \overbrace{(x \leq 0 \vee x \geq 5)}^{\textit{non-convex numerical condition}} \wedge (\dot{x} = 1 - x) \;\vee \\
(\neg b_1 \wedge b_2) \wedge (0 \leq x \leq 5) \wedge (\dot{x} = 1)
\end{array}
$$

has to be transformed into

$$
\begin{array}{l}
(b_1 \vee \neg b_2) \wedge (x \leq 0) \wedge (\dot{x} = 1 - x) \;\vee \\
(b_1 \vee \neg b_2) \wedge (x \geq 5) \wedge (\dot{x} = 1 - x) \;\vee \\
(\neg b_1 \wedge b_2) \wedge (0 \leq x \leq 5) \wedge (\dot{x} = 1)
\end{array}
$$

Then, the system is going to be partitioned into three locations, one for each line.

Approximations during analysis. In §5.1 we have explained that the translation to hybrid automata loses several properties, like determinism and urgency, which may result in an overapproximation in terms of reachable states. Moreover, hybrid reachability analysis methods further approximate the reachable states with (finite disjunctions of) convex sets, such as convex polyhedra.

The translation with "contact" semantics involves strict inequalities. Thus, the analysis may benefit from the ability of representing open sets. A suitable abstract domain might be in this case convex polyhedra with strict inequalities [5]. Otherwise, if the analysis can only handle closed sets, the translation with "contact" semantics (Fig. 3g) will behave like the one for "at-zero" semantics (Fig. 3d).

Preliminary experiments. We have implemented a prototype tool that makes use of the BddApron library [22] to handle logico-numerical formulas represented as MtBdds.

First experiments showed that – as expected – the major parameter affecting performance is the number of zero-crossings, which becomes apparent when making locations explicit as described at the beginning of this section. In the applications that we are targeting, *i.e.* synchronous controllers connected to their physical environment, zero-crossings are (1) those used for modeling the sampling of inputs and (2) those in the environment model. Since the number of (1) is usually small, the total number of zero-crossings inherently depends on the complexity of the environment model in practice.

9. CONCLUSION

We have presented a complete translation of a hybrid data-flow formalism to logico-numerical hybrid automata. In comparison to previously proposed translations, our translation handles zero-crossings.

To achieve this, we considered a simple yet expressive hybrid data-flow formalism to which large subsets of existing hybrid system languages can actually be reduced.

We discussed different choices of zero-crossing semantics and their possible translations to hybrid automata. Since hybrid automata are not as expressive as the source language, we can only provide sound over-approximations of the original semantics.

However, this is counterbalanced by the fact that existing hybrid verification tools such as HyTech [20], PHaver

[16] and SpaceEx [17] are all based on the standard hybrid automata model.

Though, these tools require to encode Boolean variables explicitly in locations. As this enumeration results in an exponential blow-up of the hybrid automaton size, we assume that this is a major bottleneck in verifying controllers with complex discrete state spaces jointly with their physical environment. Therefore future work will comprise the development of methods and tools for combining classical hybrid system analysis with implicit handling of Boolean variables in order to counter state space explosion. Our translation to logico-numerical hybrid automata lays the basis for such an approach.

10. REFERENCES

[1] V. Acary and B. Brogliato. *Numerical Methods for Nonsmooth Dynamical Systems: Applications in Mechanics and Electronics.* 2008.

[2] A. Agrawal, G. Simon, and G. Karsai. Semantic translation of Simulink/Stateflow models to hybrid automata using graph transformations. *ENTCS*, 109, 2004.

[3] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138, 1995.

[4] R. Alur, A. Kanade, S. Ramesh, and K. C. Shashidhar. Symbolic analysis for improving simulation coverage of Simulink/Stateflow models. In *Embedded Software*, 2008.

[5] R. Bagnara, P. M. Hill, and E. Zaffanella. Not necessarily closed convex polyhedra and the double description method. *Formal Aspects of Computing*, 17(2), 2005.

[6] A. Benveniste, T. Bourke, B. Caillaud, and M. Pouzet. Divide and recycle: types and compilation for a hybrid synchronous language. In *LCTES*, 2011.

[7] A. Benveniste, B. Caillaud, and M. Pouzet. The fundamentals of hyrid systems modelers. In *Conference on Decision and Control*, 2010.

[8] X. Briand and B. Jeannet. Combining control and data abstraction in the verification of hybrid systems. *Computer-Aided Design of Integrated Circuits and Systems*, 29(10), 2010.

[9] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: a declarative language for real-time programming. In *POPL*, 1987.

[10] A. Chutinan and B. H. Krogh. Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations. In *HSCC*, volume 1569 of *LNCS*, 1999.

[11] J.-L. Colaço, B. Pagano, and M. Pouzet. A conservative extension of synchronous data-flow with state machines. In *Int. Conf. On Embedded Software, EMSOFT'05*, 2005.

[12] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP'92*, volume 631 of *LNCS*, Jan. 1992.

[13] T. Dang and T. M. Gawlitza. Discretizing affine hybrid automata with uncertainty. In *Automated Technology for Verification and Analysis*, volume 6996 of *LNCS*, 2011.

[14] A. F. Filippov. Differential equations with discontinuous right-hand sides. *Mathematicheskii Sbornik*, 51(1), 1960.

[15] M. Fränzle and C. Herde. HySAT: An efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design*, 30(3), 2007.

[16] G. Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. In *HSCC*, volume 3414 of *LNCS*, 2005.

[17] G. Frehse, C. L. Guernic, A. Donzé, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. SpaceEx: Scalable verification of hybrid systems. In *Computer-Aided Verification*, volume 6806 of *LNCS*, 2011.

[18] N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2), 1997.

[19] T. A. Henzinger. The theory of hybrid automata. In *Logic in Computer Science*, 1996.

[20] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *Journal on Software Tools for Technology Transfer*, 1(1-2), 1997.

[21] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. Sundials: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.*, 31(3), 2005.

[22] B. Jeannet. Bddapron: A logico-numerical abstract domain library. `http://pop-art.inrialpes.fr/~bjeannet/bjeannet-forge/bddapron/`, 2009.

[23] T. Lindstrøm. An invitation to nonstandard analysis. In N. Cutland, editor, *Nonstandard Analysis and its Applications*, pages 1–105. Cambridge University Press, 1988.

[24] K. Manamcheri, S. Mitra, S. Bak, and M. Caccamo. A step towards verification and synthesis from Simulink/Stateflow models. In *HSCC*, 2011.

[25] M. Najafi and R. Nikoukhah. Implementation of hybrid automata in Scicos. In *Control Applications*, 2007.

[26] M. Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual.* Université Paris-Sud, LRI, 2006.

[27] S. Ratschan and Z. She. Safety verification of hybrid systems by constraint propagation-based abstraction refinement. *Transactions on Embedded Computing Systems*, 6(1), 2007.

[28] P. Raymond, Y. Roux, and E. Jahier. Lutin: A language for specifying and executing reactive scenarios. *EURASIP J. on Embedded Systems*, 2008.

[29] A. Robinson. *Non-Standard Analysis.* 1996.

[30] P. Schrammel and B. Jeannet. From hybrid system data-flow to hybrid automata: A complete translation. Technical Report 7859, INRIA, Jan 2012.

[31] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating discrete-time Simulink to Lustre. *Embedded Computing Systems*, 4(4), 2005.

[32] V. I. Utkin. *Sliding modes in Control and optimization.* 1992.