HYDI: a language for symbolic hybrid systems with discrete interaction

Alessandro Cimatti
Fondazione Bruno Kessler
Email: cimatti@fbk.eu

Sergio Mover
Fondazione Bruno Kessler
Email: mover@fbk.eu

Stefano Tonetta
Fondazione Bruno Kessler
Email: tonettas@fbk.eu

***Abstract*—Complex embedded systems consist of software and hardware components that operate autonomous devices interacting with the physical environment. The complexity of such systems makes the design very challenging and demands for advanced validation techniques.**

Hybrid automata are a clean and consolidated formal language for modeling embedded systems which include discrete and continuous dynamics. They are based on a finite-state automaton structure enriched with invariant and flow conditions to model the continuous dynamics.

In this paper, we propose a new language, HYDI, for modeling Hybrid systems with Discrete Interaction. The purpose of the language is to apply state-of-the-art symbolic model checkers for infinite-state systems to the verification of complex embedded systems design. HYDI extends the standard symbolic language SMV with timing and synchronization aspects. The language distinguishes between discrete and continuous variables. Variables inside SMV modules evolve synchronously. Top-level modules represent the asynchronous components of a network and use explicit events to synchronize. The new language is automatically compiled into equivalent discrete-time infinite-state transition systems.

I. INTRODUCTION

Complex Embedded Systems (CES) consist of software and hardware components that operate autonomous devices interacting with the physical environment. They are now part of our daily life and are used in many industrial sectors including automotive, aerospace, consumer electronics, communications, medical and manufacturing. CES are used to carry out highly complex and often critical functions. They are used to monitor and control industrial plants, complex transportation equipment, and communication infrastructure. The development process of CES is widely recognized as a highly complex task. A thorough validation and verification activity is necessary to enhance the quality of the CES and, in particular, to fulfill the quality criteria mandated by the relevant standards.

CES are composed of many heterogeneous components, interacting with external environments, and deal with continuous and discrete dynamics. They may include multiple computation units, with possibly multiple cores per unit, and customizable hardware to implement computationally intensive functions. CES are often the result of a tight integration of hardware and software, with different families of applications for the same platform, communication ASICs specialized in the interaction with complex network protocols, and control ASICs for the acquisition of data from analog sensors.

Networks of communicating *Hybrid Automata* [26] (HAs) are increasingly used as a formal framework to model the interaction between discrete and continuous components. The architecture is typically *Globally Asynchronous/Locally Synchronous* (GALS), where the global asynchronicity takes into account the communication of shallowly connected modules, while the local synchronicity refers to the programming paradigm for each of the modules.

Historically, most of the formal verification tools have focused on either aspect. Some privilege an asynchronous semantics, with a language oriented to the representation of asynchronous components, and implementing variants of (extensions of) explicit-state search; examples are SPIN [28], UPPAAL [11], HYTECH [25], and PHAVER [22]. Others privilege the representation of synchronous modules, and rely on a synchronous engine, making substantial use of symbolic techniques. Examples are SMV and VIS.

In this paper we address the problem of symbolically representing complex hybrid systems for formal verification. Our work is motivated by the experience with the NUSMV model checker, that has been widely used as back-ends in many formal verification flows for GALS, either in its discrete version [3], [32], and in its timed/hybrid extension [1], [2]. All these translations to NUSMV have required similar activities, such as dealing with synchronization primitives and with timing aspects, that must be encoded into formulas over symbolic variables. Furthermore, the translational approach is often unsatisfactory, since the resulting models lost the structure of the original problem, and that can not be exploited by the model checker.

In this paper, we propose a novel language, called HYDI, that has been specifically designed to raise the abstraction level for modeling hybrid GALS in a fully symbolic manner. HYDI (HYbrid systems with Discrete Interaction) can be seen as an extension of the SMV language, able to represent networks of hybrid systems. In particular, the SMV language has been extended into two directions: on one hand, we introduced timing aspects such continuous variables and flow conditions; on the other hand we define a top-level structure to specify the network and the synchronization of the components.

HYDI has the following distinguishing features. First, it has a symbolic definition of flow conditions which are attached to symbolic predicates rather than states. This may allow for a compact representation of complex systems, where the same

flow condition applies to a large number of states. Second, it allows for the definition of symbolic transition relations, which allows an easy encoding of macro transition or parallel composition. Third, in the spirit of GALS, HYDI provides the generic primitives for specifying an application-specific scheduler definition, which allows the encoding of different scheduling strategies.

The HYDI language comes with four semantics. These are obtained by combining two orthogonal dimensions: the way time is treated (global- vs local-time), and asynchronicity (interleaving vs step). Although the semantics are equivalent from the reachability point of view, they induce different translations into the SMV language and enhance different verification techniques so that we can tailor the choice to the analyzed system. The semantics are obtained by combining two orthogonal dimensions: the way time is treated (global- vs local-time), and asynchronicity (interleaving vs step). In addition to providing a fully formal account of the language, these semantics induce different translations into the SMV language.

Support for HYDI has been fully implemented on top of NuSMV3, a "synchronous" extension of the publicly available NuSMV2, extended with continuous variables and SMT techniques. Two HYDI-based verification flows are supported. The first one is based on a translation to a monolithic SMV program and relies on the SMT-based verification engine that exploit the MathSAT SMT solver to deal with infinite-domain variables. The second, more advanced one, allows to exploit the structure of the network of the system under analysis [14]. The HYDI language was used to encode a comprehensive set of benchmarks (publicly available at http://es.fbk.eu/people/mover/hydi) and as back-end for translations of languages such as MatLab StateFlow/SymulLink and Altarica [2].

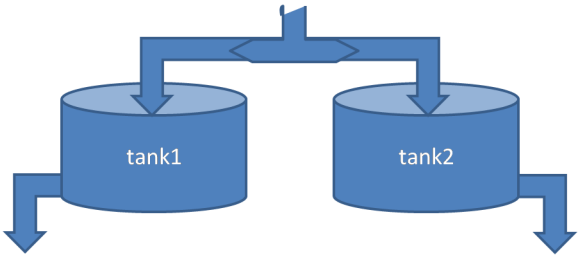
The rest of this paper is structured as follows. In Section II, we overview the features of HYDI. In Section III, we introduce some background notions, namely, syntax and semantics of SMV and hybrid automata. In Section IV, we define the abstract syntax and semantics of the language. In Section VI, we overview the HYDI-based verification techniques and flows. In Section VII, we discuss the related work, and in section VIII, we draw some conclusions.

II. Overview of the Language

A HYDI program is given by a set of *modules*, a set of *processes*, a set of *synchronization* constraints. Figure 1 shows a small example of communicating tanks specified in HYDI. Each tank has an input and output flow of water. When a tank is full, the input flow of the other tank may be doubled.

A. Modules

HYDI modules (e.g., `Tank`) extend SMV modules in order to specify explicitly the events used for synchronization and timing aspects such as continuous variables and flow conditions. The SMV language has been widely used to specify complex finite-state systems. The system description is typically decomposed into modules. Essentially, a module is




```
MODULE main
VAR
tank1: Tank;
tank2: Tank;

SYNC tank1, tank2 EVENTS filled, doubling;
SYNC tank1, tank2 EVENTS unfilled, halving;
SYNC tank2, tank1 EVENTS filled, doubling;
SYNC tank2, tank1 EVENTS unfilled, halving;

MODULE Tank
EVENT filled, unfilled, doubling, halving, tau;
VAR
state: {empty, emptying, filling, full};
flow: {single, double};
level: continuous;
INIT
state!=full & flow=single
INVAR
level>=0 & level<=100
INVAR
state=empty -> level=0
INVAR
state=full -> level=100
TRANS
EVENT=doubling <-> (flow=single & next(flow)=double)
TRANS
EVENT=halving <-> (flow=double & next(flow)=single)
TRANS
EVENT=filled <-> (state=filling & next(state)=full)
TRANS
EVENT=unfilled <-> (state=full & next(state)=emptying)
TRANS
state=emptying -> next(state)!=full
TRANS
state=filling -> next(state)!=empty
TRANS
next(level)=level
FLOW
flow=single -> (der(level)>=-10 & der(level)<=10)
FLOW
flow=double -> (der(level)>=-10 & der(level)<=20)
FLOW
state=emptying -> (der(level)<=0)
FLOW
state=filling -> (der(level)>=0)
URGENT
state=emptying & level=0
URGENT
state=filling & level=100
```

Fig. 1. A small HYDI example.

a set of declarations and constraints on the declared variables. Modules can be instantiated several times and nested to form a complex synchronous hierarchy.

In particular, modules may contain a `VAR` section with the declaration of variables; `INIT` constraints defining the initial states; `INVAR` constraints restricting the valid assignments to the variables; and `TRANS` constraints defining the valid transitions from one state to another.

HYDI modules inherits all the constructs of SMV modules and add three main new features:

- *events*, a list of symbols used in the synchronizations; these are introduced with the keyword `EVENT`; the keyword can be also used in the `TRANS` constraints as it was an input variable; intuitively, transitions are distinguished by the event which is being fired;
- *continuous* variables, a new type of variables declared with the keyword `continuous`; these are variables which are allowed to change in a timed transitions and evolve according to some function continuous in time;
- *flow* conditions, used to constrain the continuous evolution of continuous variables; the constraints are introduced with the keyword `FLOW` and may refer to the derivative of the continuous variables, denoted with `der`.

B. Processes

HYDI processes are instantiation of HYDI modules (in the example, `tank1` and `tank2` are processes). Differently from SMV processes, they can run both asynchronously or synchronize on shared events. Processes are declared in the *main* module of a HYDI program. They represent the components a network whose topology is defined by the synchronizations. The network is not hierarchical in that there is no further asynchronous decomposition of a process, although the modules may contain synchronous instantiation of other modules.

Variables and events of a module are renamed in the process by prefixing the name of the process itself (with the classic dot notation). In the example, the variable `state` of module `Tank` is renamed in `tank1.state` by the process `tank1`.

Processes can share variables through the passage of parameters in the instantiations. However, they are limited to reading the variables of other processes while writing is not allowed. This permits an easy identification of when the variables do not change even if the transitions are described with a generic relation (compared to a more restrictive functional description).

C. Synchronizations

Synchronizations specify if two events of two processes must happen at the same time. If two events are not synchronized, they must interleave. Such synchronization is quite standard in automata theory and process algebra. It has been generalized with guards to restrict when the synchronization can happen.

In order to capture the semantics of some design languages, it is necessary to enrich the synchronization with further constraints that specify a particular policy scheduling the interaction of the processes. For this reason, it is possible to enrich the main module of the HYDI program with a *scheduler* specified in terms of state variables, initial and transition conditions. These conditions may predicate over the events of the processes.

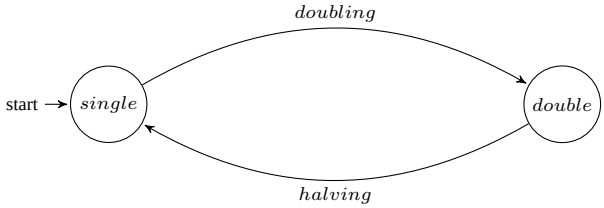


Fig. 2. LTS example.

III. BACKGROUND NOTIONS

A. Labeled Transition Systems

Labelled Transition Systems (LTSs) are a standard formalism to represent the semantics of languages, either based on automata, algebras or other higher-level description of computation. In particular, LTSs are used to define the semantics of Hybrid Automata.

An LTS is a tuple $\langle Q, A, Q_0, R\rangle$ where

- Q is the set of states,
- A is the set of actions/events (also called alphabet),
- $Q_0 \subseteq Q$ is the set of initial states,
- $R \subseteq Q \times A \times Q$ is the set of labeled transitions.

A *trace* is a sequence of events $w = a_1, \ldots, a_k \in A^*$. Given $A' \subseteq A$, the projection $w_{|A'}$ of w on A' is the sub-trace of w obtained by removing all events in w that are not in A'.

A *path* π of S over the trace $w = a_1, \ldots, a_k \in A^*$ is a sequence $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_k} q_k$ such that $q_0 \in Q_0$ and, $\langle q_{i-1}, a_i, q_i\rangle \in R$ for all i such that $1 \leq i \leq k$. We say that π accepts w. Given a predicate $p \subseteq Q$ we say that π terminates in p iff $q_k \in p$.

A *path* π of S is a sequence $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_k} q_k$ such that $q_0 \in Q_0$ and, $\langle q_{i-1}, a_i, q_i\rangle \in R$ for all i such that $1 \leq i \leq k$. Given a predicate $p \subseteq Q$ we say that π terminates in p iff $q_k \in p$.

The *language* $L(S)$ of an LTS S is the set of traces accepted by some run of S. Given a predicate $p \subseteq Q$, the language $L_p(S)$ of an LTS S is the set of traces accepted by some run of S terminating in p.

Example 1: Consider the simple LTS of Figure 2. The system has two states, $single$ and $double$, and two events, $doubling$ and $halving$. The system alternates between the two states. The only possible path loops over the events $doubling$ and $halving$.

The *parallel composition* $S_1||S_2$ of two LTSs $S_1 = \langle Q_1, A_1, Q_{01}, R_1\rangle$ and $S_2 = \langle Q_2, A_2, Q_{02}, R_2\rangle$ is the LTS $\langle Q, A, Q_0, R\rangle$ where

- $Q = Q_1 \times Q_2$,
- $A = A_1 \cup A_2$,

- $Q_0 = Q_{01} \times Q_{02}$,
- $R := \{\langle\langle q_1, q_2\rangle, a, \langle q_1', q_2'\rangle\rangle \mid \langle q_1, a, q_1'\rangle \in R_1, \langle q_2, a, q_2'\rangle \in R_2\}$
  $\cup\{\langle\langle q_1, q_2\rangle, a, \langle q_1', q_2\rangle\rangle \mid \langle q_1, a, q_1'\rangle \in R_1, a \notin A_2\}$
  $\cup\{\langle\langle q_1, q_2\rangle, a, \langle q_1, q_2'\rangle\rangle \mid \langle q_2, a, q_2'\rangle \in R_2, 1a \notin A_1\}$.

The parallel composition of two or more LTSs $S_1||\ldots||S_n$ is also called a *network*. If an event is shared by two or more components, we say that the event is a synchronization event; otherwise, we say that the event is local. The *reachability* problem for a network of LTSs consists of checking if there exists a path π reaching a given condition p.

Remark 1: The set of events is not restricted to be finite. Thus, it is possible to model also variable sharing where the two systems may synchronize the value of the variables on a common event. In fact, it is sufficient to encode the value of the variable in the event itself: for example, if on event a two systems synchronize the value of the real variable x, we use the set of events $\{\langle a, \underline{x}\rangle\}_{\underline{x}\in\mathbb{R}}$ and force that a transition $\langle q, \langle a, \underline{x}\rangle, q'\rangle$ exists only if the value of x in q is $\underline{x}$.

In the parallel composition presented above the local events of the components are interleaved. We refer to this semantics also as *interleaving* semantics of the composition. However, since the interaction among automata happens only on the shared events, the local actions are independent. Following [23], we now define an alternative equivalent semantics, called *step* semantics, where independent transition can be fired at the same time.

More formally, with the step semantics, the parallel composition $S_1||\ldots||S_n$ of n LTSs is defined as:

- $Q = Q_1 \times \ldots \times Q_n$,
- $A = A_1^\epsilon \times \ldots \times A_n^\epsilon$, where $A_i^\epsilon i = A_i \cup \{\epsilon\}$,
- $Q_0 = Q_{01} \times \ldots \times Q_{0n}$,
- $R := \{\langle \overline{q}, \overline{a}, \overline{q'}\rangle \in Q \times A \times Q' \mid$ for $1 \leq i \leq n \langle q_i, a_i, q_i'\rangle \in R_i$ and for all $1 \leq i < j \leq n$, if $a_i \neq a_j$ then $a_i \notin A_j, a_j \notin A_i\}$.

Theorem 1 (Step semantics [23]): A predicate p is reachable in the interleaving semantics iff it is reachable in the step semantics.

B. Symbolic representation

When the system is too large, instead of storing states and transitions explicitly, it is wiser to represent them symbolically by means of symbolic formulas. The system S is described with a set V of state variables. We use V' to denote the set of next state variables $\{v'\}_{v\in V}$, where v' represents the next value of v. A state of the system becomes an assignment to the variables V and a proposition is seen as a predicate over the variables V. A set of states is represented by a formula $\alpha(V)$ over the variables V: a state s belongs to the set if s, as an assignment, makes the formula true (namely, $s \models \alpha(V)$).

A Symbolic Transition System (STS) S is a tuple $\langle V, W, I, T, Z\rangle$ where:

- V is the set of symbolic variables representing the states,
- W is the set of symbolic variables representing the events,
- $I(V)$ is the initial formula,

```
MODULE main
VAR
flow: {single, double};
IVAR
event: {doubling, halving};
INIT
flow=single
TRANS
(event=doubling <-> (flow=single & next(flow)=double)) &
(event=halving <-> (flow=double & next(flow)=single))
```

Fig. 3. A small SMV example.

- $T(V, W, V')$ is the transition formula,
- $Z(V)$ is the invariant formula.

Every STS S corresponds to a LTS $C(S)$:

- Q is the set of assignments to the variables V,
- A is the set of assignments to the variables W,
- $Q_0 = \{s \in Q \mid s \models I\}$,
- $R = \{(s_1, a, s_2) \in Q \times A \times Q \mid s_1, a, s_2' \models T\}$.

Example 2: The LTS of Example 1 can be represented with one state variable $flow$ and one input variable $event$. The corresponding STS would be:

- $V := \{flow\}$,
- $W := \{event\}$,
- $I := (flow = single)$,
- $T := (event = doubling \leftrightarrow (flow = single \wedge flow' = double)) \wedge event = halving \leftrightarrow (flow = double \wedge flow' = single))$

C. SMV

The SMV language [30] is widely used to describe complex finite-state STSs. It is mostly known as the input language of two of the most famous symbolic model checkers, namely Cadence SMV http://www.kenmcmil.com/smv.html and NuSMV http://nusmv.fbk.eu.

The state of a system is described with a set of symbolic variables. Set of states and transitions are represented by formulas. Symbolic variables are grouped into *modules*. Modules can be instantiated several times and composed synchronously[1]. We refer to each instantiation as a component. The transition relations of the components are conjoined. Therefore, each step of the composition is equivalent to the conjunction of a step in each component.

We consider as baseline the version of SMV described in [15] (thus with input and frozen variables) extended with real variables. Figure 3 shows an SMV example representing part of the discrete component of the tank defined in the HYDI example of Figure 1.

An SMV program consists of a set of modules. Each module M can be seen as tuple $\langle$PARAM, VAR, IVAR, INIT, TRANS, INVAR$\rangle$, where:

- PARAM is a set P of formal parameters,
- VAR is a set of variable declaration defining a set V of variables and for each variable v a type $\tau(v)$,

[1]Also asynchronous composition without synchronization is possible but is deprecated and not considered here.

- IVAR is a set of input variable declaration defining a set W of variables and for each variable w a type $\tau(w)$,
- INIT is a set of initial condition declaration defining a formula I over the variables $V \cup P$,
- TRANS is a set of transition condition declaration defining a formula T over the variables $V \cup P \cup W \cup V' \cup P'$,
- INVAR is a set of invariant condition declaration defining a formula Z over the variables $V \cup P$.

The VAR declaration may contain also module instantiations, i.e., variables whose type is in the form $\langle M, \beta \rangle$, where M is a module and β associates each formal parameter to an actual parameter. For every parameter $p \in P$, the actual parameter $\beta(p)$ must evaluate to the same type of p.

A SMV program always has a *main* module, which is the root node of a hierarchy given by the instantiations.

The semantics of SMV is defined in terms of Symbolic Transition Systems (STSs). An STS S is a tuple $\langle V, W, I, T, Z \rangle$ where:

- V is the set of symbolic variables representing the states,
- W is the set of symbolic variables representing the events,
- $I(V)$ is the initial formula,
- $T(V, W, V')$ is the transition formula,
- $Z(V)$ is the invariant formula.

Every STS S corresponds to a Labelled Transition System (LTS) $C(S)$:

- Q is the set of assignments to the variables V,
- A is the set of assignments to the variables W,
- $Q_0 = \{s \in Q \mid s \models I\}$,
- $R = \{(s_1, a, s_2) \in Q \times A \times Q \mid s_1, a, s_2' \models T\}$.

The semantics of a module is defined recursively on the hierarchical structure assuming that there is no circular dependency in the hierarchy (see [30] for further details). If a module M does not contain any module instantiation, its semantics is given by the STS $\langle V \cup P, W, I, T, Z \rangle$. If the variable declaration contains the module instantiations $\mathcal{I}$ such that, for all $i \in \mathcal{I}$, $\tau(i) = \langle M_i, \beta_i \rangle$, let us consider the STS $S_{M_i} = \langle V_{M_i}, W_{M_i}, I_{M_i}, T_{M_i}, Z_{M_i} \rangle$ recursively defined for M_i. Let S_i be the STS associated to the instance i of type $\langle M_i, \beta \rangle$, defined as $\langle V_i, W_i, I_i, T_i, Z_i \rangle$, where:

- V_i is obtained by V_{M_i} by removing the formal parameters of M_i and renaming the other variables $v \in V_{M_i}$ with $i.v$,
- W_i is obtained by W_{M_i} by renaming each variable $w \in W_{M_i}$ with $i.w$,
- I_i is obtained by I_{M_i} renaming each variable v with $i.v$ and substituting each parameter p of M_i with $\beta(p)$,
- T_i is obtained by T_{M_i} renaming each variable v with $i.v$ and substituting each parameter p of M_i with $\beta(p)$,
- Z_i is obtained by Z_{M_i} renaming each variable v with $i.v$ and substituting each parameter p of M_i with $\beta(p)$.

The STS S of M is defined as the tuple $\langle V \setminus \mathcal{I} \cup \{i.v \mid i \in \mathcal{I}, v \in V_i\} \cup P, W, I \wedge \bigwedge_{i \in \mathcal{I}} I_i, T \wedge \bigwedge_{i \in \mathcal{I}} T_i, Z \wedge \bigwedge_{i \in \mathcal{I}} Z_i \rangle$. The semantics of an SMV program is given by the STS associated to the main module.

D. Hybrid automata

Hybrid automata are a mathematical model to represent hybrid systems such as embedded systems involving both continuous and discrete dynamics. There are several variants of hybrid automata. In this paper, we refer to the formalism introduced in [6] and surveyed in [26]. Intuitively, they are finite-state automata enriched with continuous variables whose dynamics change from node to node of the automata.

A *Hybrid Automaton* (HA) [26] is a tuple $\langle Q, A, Q_0, R, X, \mu, \iota, \xi, \theta \rangle$ where

- Q is the set of states,
- A is the set of events,
- $Q_0 \subseteq Q$ is the set of initial states,
- $R \subseteq Q \times A \times Q$ is the set of discrete transitions,
- X is the set of continuous variables,
- $\mu : Q \to P(X, \dot{X})$ is the flow condition,
- $\iota : Q \to P(X)$ is the initial condition,
- $\xi : Q \to P(X)$ is the invariant condition,
- $\theta : R \to P(X, X')$ is the jump condition,

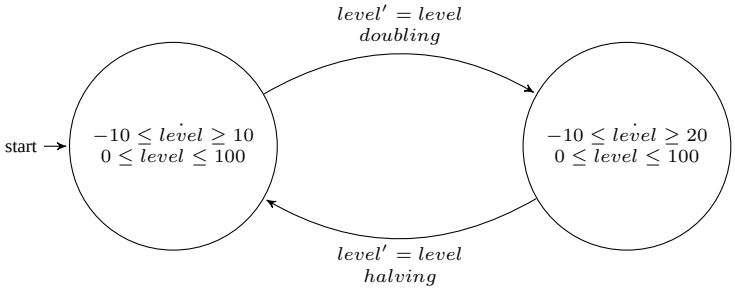
where P represents the set of predicates over the specified variables.

A Linear HA (LHA) is an HA such that:

- the initial, invariant, flow, and jump conditions are Boolean combinations of linear inequalities;
- the flow conditions contain variables in $\dot{X}$ only.

Moreover, we assume that the invariant conditions of linear HA contain only conjunctions of inequalities.

Example 3:



A *network* $\mathcal{H}$ of HAs is the parallel composition of two or more HAs. We consider two semantics for networks of HAs: the global-time semantics where all the components synchronize on timed events, and the local-time (or time-stamps) semantics where the timed events are local and components must synchronize the time on shared events.

Consider a network $\mathcal{H} = H_1 || \ldots || H_n$ of HAs with $H_i = \langle Q_i, A_i, Q_{0i}, R_i, X_i, \mu_i, \iota_i, \xi_i, \theta_i \rangle$. The *global-time semantics* (or time-action semantics) [26] of $\mathcal{H}$ is the network of LTSs $\mathcal{N}_{\text{GLTIME}}(\mathcal{H}) = S_1 || \ldots || S_n$ with $S_i = \langle Q_i', A_i', Q_{0i}', R_i' \rangle$ where

- $Q_i' = \{\langle q, \overline{x} \rangle \mid q \in Q_i, \overline{x} \in \mathbb{R}^{|X_i|}\}$,
- $A_i' = A_i \cup \{\langle \text{TIME}, \delta \rangle \mid \delta \in \mathbb{R}\}$,
- $Q_{0i}' = \{\langle q, \overline{x} \rangle \mid q \in Q_{0i}, \overline{x} \in \iota_i(q)\}$,
- $R_i' = \{\langle \langle q, \overline{x} \rangle, a, \langle q', \overline{x}' \rangle \rangle \mid \langle q, a, q' \rangle \in R_i, \langle \overline{x}, \overline{x}' \rangle \in$

$\theta_i(q,a,q'), \overline{x} \in \xi_i(q), \overline{x}' \in \xi_i(q')\} \cup$
$\{\langle\langle q,\overline{x}\rangle, \langle \text{TIME}, \delta\rangle, \langle q,\overline{x}'\rangle\rangle \mid$ there exists f satisfying $\mu_i(q)$ s.t. $f(0)=\overline{x}, f(\delta)=\overline{x}', f(\epsilon)\in\xi(q), \epsilon\in[0,\delta]\}$.

Consider a network $\mathcal{H} = H_1||\ldots||H_n$ of HAs with $H_i = \langle Q_i, A_i, Q_{0i}, R_i, X_i, \mu_i, \iota_i, \xi_i, \theta_i\rangle$. The *local-time semantics* (or time-stamps semantics) [10] of $\mathcal{H}$ is the network of LTSs $\mathcal{N}_{\text{LOCTIME}}(\mathcal{H}) = S_1||\ldots||S_n$ with $S_i = \langle Q'_i, A'_i, Q'_{0i}, R'_i\rangle$ where

- $Q'_i = \{\langle q,\overline{x},t\rangle \mid q\in Q_i, \overline{x}\in\mathbb{R}^{|X|}, t\in\mathbb{R}\}$,
- $A'_i = \{\langle a,t\rangle \mid a\in A_i, t\in\mathbb{R}\}\cup\{\text{TIME}_i\}$,
- $Q'_{0i} = \{\langle q,\overline{x},0\rangle \mid q\in Q_{0i}, \overline{x}\in\iota_i(q)\}$,
- $R'_i =$
  $\{\langle\langle q,\overline{x},t\rangle, \langle a,t\rangle, \langle q',\overline{x}',t\rangle\rangle \mid \langle q,a,q'\rangle \in R_i, \langle\overline{x},\overline{x}'\rangle \in \theta_i(q,a,q'), \overline{x}\in\xi_i(q), \overline{x}'\in\xi_i(q')\}\cup$
  $\{\langle\langle q,\overline{x},t\rangle, \text{TIME}_i, \langle q,\overline{x}',t'\rangle\rangle \mid$ there exists f satisfying $\mu_i(q)$ s.t. $f(t)=\overline{x}, f(t')=\overline{x}', f(\epsilon)\in\xi_i(q), \epsilon\in[t,t']\}$.

Given a state $\langle q,\overline{x},t\rangle\in Q_i$ we denote with $time(\langle q,\overline{x},t\rangle)$ the component t of the state.

Theorem 2 (Equivalence of two semantics [10]): Let p_t the set of states $\langle q_1, q_2, \ldots, q_n\rangle$ of a $\mathcal{N}_{\text{LOCTIME}}$ where the states of the components have the same time ($time(q_1) = time(q_2) = \ldots = time(q_n)$). Then, for every predicate p, p is reachable in $\mathcal{N}_{\text{GLTIME}}(\mathcal{H})$ iff $p \wedge p_t$ is reachable in $\mathcal{N}_{\text{LOCTIME}}(\mathcal{H})$.

IV. HYDI

A. Untimed HYDI programs

In this section we present the semantics of an HYDI program without continuous variables and flow conditions (*untimed*). In particular, we focus on the specification of processes and process synchronizations.

1) Syntax and semantics: A HYDI program is composed of a set of module declarations and a *main* module. The *main* module declares a set of process instances $\mathcal{I}$ and constraints over the variables of the processes. In particular, we use the constraints in the main module to encode the synchronization constraints of the processes and ad-hoc scheduler policies.

In HYDI, each process contains an enumerative event input variable ϵ (EVENT in the concrete syntax), which is used as guard for the transitions of the process. To model the asynchronous behavior we add the stutter action S to the domain of ϵ. Then, we force a process to not change its state when it performs the action S. The constraints in the *main* module can predicate over of the variables of an instance, thus forcing stuttering and synchronizations.

We identify the set of instances of an HYDI program with $\mathcal{I}$. We use IVAR_i^{S} when referring to the set of input variables IVAR_i of an instance i, where ϵ is enriched with the action S, namely $\text{IVAR}_i[\langle\epsilon,\tau(\epsilon)\rangle/\langle\epsilon,\tau(epsilon)\cup\{\text{S}\}\rangle]$. We identify the set of all the variables of a program with $\text{VAR}_H = (\text{VAR}\setminus\mathcal{I})\cup\bigcup_{i\in\mathcal{I}} i.\text{VAR}_i$, where VAR is the set of variables declared in the *main* module.

An untimed HYDI program H is a tuple $\langle\mathcal{M}, main\rangle$ where:

- $\mathcal{M} = \{M_1,\ldots,M_m\}$, such that each $M_i = \langle\text{PARAM}_i, \text{VAR}_i, \text{IVAR}_i, \text{INIT}_i, \text{TRANS}_i, \text{INVAR}_i\rangle$ is a module declaration,
- $main = \langle\text{PARAM}, \text{VAR}, \text{IVAR}, \text{INIT}, \text{TRANS}, \text{INVAR}\rangle$ is the *main* module such that:
  - PARAM is an empty set of parameters,
  - VAR is a set of declarations defining the set of variables V,
  - IVAR is a set of declarations defining the set of input variables W,
  - INIT is the set of initial conditions declarations which defines a formula I over VAR_H,
  - TRANS is a set of transition conditions declaration defining a formula T over variables declared in $\text{VAR}_H\cup\text{IVAR}_H\cup\text{VAR}'_H$,
  - INVAR is the set of invariant conditions declarations which defines a formula Z over VAR_H.
- for all the instances $i\in\mathcal{I} = \{i | i\in V$ and $\tau(i) = \langle M_i,\beta\rangle\}$:
  - $M_i\in\mathcal{M}$,
  - TRANS_i of M_i is a transition condition declaration defining the formula T_i over the variables $V\cup P\cup W\cup V'$ (i.e. all the input parameters can only be read by the process i).
  - $\epsilon\in W_i$,
  - $\tau(\epsilon) = \{a_1,\ldots,a_j\}$, such that $a_k\neq$ S is an enumerative value, for $1\le k\le j$,

Note that we allow to model *shared variables* between different instances, with the restriction that only the process which declares a variable can write its value, while the other processes can only read the value of the variable. The restriction is expressed in the constraints enforced on the TRANS_i formula for each process i, which can only change its next state variables V'.

As explained in III-C, we associate a STS to every instance $i\in\mathcal{I}$ of type $\langle M_i,\beta\rangle$. However, to enforce the stuttering condition, we first add to $\tau(\epsilon)$ the value S and then we add to the T_i formula the frame condition when the action is S. The frame condition forces each state variable $v\in V_i$ not to change its value during a transition. Thus, given the STS $S_i = \langle V_i, W_i, I_i, T_i, Z_i\rangle$ associated to a process instance $i\in\mathcal{I}$ of type $\langle M_i,\beta\rangle$ defined as in III-C, we define the STS $S_i^{\text{S}} = \langle V_i^{\text{S}}, W_i^{\text{S}}, I_i^{\text{S}}, T_i^{\text{S}}, Z_i^{\text{S}}\rangle$ as follows:

- $V_i^{\text{S}} := V_i$,
- $W_i^{\text{S}} := W_i$, where $\tau(\epsilon) = \{a_1,\ldots,a_k\}\cup\{\text{S}\}$,
- $I_i^{\text{S}}(V_i^{\text{S}}) := I(V_i)$,
- $T_i^{\text{S}}(V_i^{\text{S}}, W_i^{\text{S}}, V_i^{\text{S}'}) := T_i(V_i, W_i, V_i')\wedge\ (\epsilon = \text{S} \rightarrow \bigwedge_{v\in V_i} v' = v)$,
- $Z_i^{\text{S}}(V_i^{\text{S}}) := Z_i(V_i)$.

The STS associated to the *main* module of the HYDI program $H = \langle\mathcal{M}, main\rangle$ is defined as $STS_H = \langle V\setminus\mathcal{I}, W, I\wedge\bigwedge_{i\in\mathcal{I}} I_i^{\text{S}}, T\wedge\bigwedge_{i\in\mathcal{I}} T_i^{\text{S}}, Z\wedge\bigwedge_{i\in\mathcal{I}} Z_i^{\text{S}}\rangle$. The semantic of an HYDI program is given by the STS associated to its *main* module.

2) Synchronization constraints: The constraints in the *main* module of an HYDI program allows to express general synchronization policies between the processes. We enable the modeling of point-to-point synchronizations between pro-

cesses. We add the synchronization declarations SYNC to the *main* module of the HYDI program $\langle \mathcal{M}, main \rangle$. A synchronization in SYNC is a tuple $\langle i, j, a_i, a_j \rangle$, where $i, j \in \mathcal{I}$, $a_i \in \tau(i.\epsilon)$ and $a_j \in \tau(j.\epsilon)$. The synchronization enforces the instances i and j to perform a transition labelled with the event a_i and a_j at the same time.

Since the synchronization constraints are declared between couples of processes, we define the transitive synchronization relation SYNC^* from SYNC. The tuple $\langle i, j, a_i, a_j \rangle$ is in SYNC* iff there exists a sequence of instances $l_1, l_2, \ldots, l_n$ such that $\langle l_k, l_{k+1}, a_{l_k}, a_{l_{k+1}} \rangle \in \text{SYNC}$ for $1 \leq k \leq n$, $i = l_1$ and $j = l_n$.

Example 4: Consider three instances i, j, k with the synchronization constraints $\langle i, j, a_i, a_j \rangle$ and $\langle j, k, a_j, a_k \rangle$. When the instance i synchronizes with the instance j on the events a_i and a_j, also the instance k synchronizes with the instance j on the event a_k. Thus, in this example $\text{SYNC}^* = \{\langle i, j, a_i, a_j \rangle, \langle j, k, a_j, a_k \rangle, \langle i, k, a_i, a_k \rangle\}$.

To encode the synchronization declarations we define the following constraints:

$$\gamma = \bigwedge_{\langle i,j,a_i,a_j \rangle \in \text{SYNC}^*} i.\epsilon = a_i \leftrightarrow j.\epsilon = a_j, \quad \psi = \bigvee_{i \in \mathcal{I}} i.\epsilon \neq s,$$

$$\phi_{\text{INT}} = \bigwedge_{\substack{i,j \in \mathcal{I}, \\ i \neq j}} \bigwedge_{a_j \in \tau(j.\epsilon)} j.\epsilon = a_j \rightarrow \bigwedge_{\substack{a_i \in \tau(i.\epsilon), \\ \langle i,j,a_i,a_j \rangle \notin \text{SYNC}^*}} i.\epsilon = s.$$

The synchronization constraints γ enforces the synchronization between the processes. The interleaving constraint ϕ_{INT} enforces that a process i performs the stutter event if another process moves with a local event or with an event that does not synchronize with i. The ψ constraints ensures that at least one process does not perform the stutter action.

A HYDI program which contains the SYNC constraints is compiled in a correspondent program without SYNC, which are encoded in the constraints of the *main* module. We present two different compilation processes, which corresponds to the standard *interleaving* semantics and to the *step* semantics [23].

Given the HYDI program $H = \langle \mathcal{M}, main \rangle$, with $main = \langle \text{PARAM, VAR, IVAR, INIT, TRANS, INVAR, SYNC} \rangle$ we define the *interleaving* HYDI program H_{INT} $\langle \mathcal{M}, main_{\text{INT}} \rangle$ where $main_{\text{INT}} = \langle \text{PARAM, VAR, IVAR, INIT, TRANS}_{\text{INT}}, \text{INVAR} \rangle$ and $\text{TRANS}_{\text{INT}}$ = TRANS $\wedge \gamma \wedge \psi \wedge \phi_{\text{INT}}$.

Given the HYDI program $P = \langle \mathcal{M}, main \rangle$, with $main = \langle \text{PARAM, VAR, IVAR, INIT, TRANS, INVAR, SYNC} \rangle$ we define the step HYDI program H_{STEP} as the tuple $\langle \mathcal{M}, main_{\text{STEP}} \rangle$, where $main_{\text{STEP}} = \langle \text{PARAM, VAR, IVAR, INIT, TRANS}_{\text{STEP}}, \text{INVAR} \rangle$ and $\text{TRANS}_{\text{STEP}}$ = TRANS $\wedge \gamma \wedge \psi$.

The *interleaving* (*step*) semantics of a HYDI program with SYNC constraints is given by the STS associated to the module $main_{\text{INT}}$ ($main_{\text{STEP}}$).

Remark 2: Note that in the step semantics the $\text{TRANS}_{\text{STEP}}$ does not contain the ϕ_{INT} constraint, enabling independent local actions and different synchronizations to be executed by different processes at the same time. Moreover, the *step* semantic can be applied only if the processes do not share any variable (i.e. for all $i \in \mathcal{I}$, $\tau(i) = \langle M_i, \beta \rangle$, the module M_i is such that $\text{PARAM}_i = \emptyset$).

B. Hybrid processes

HYDI processes extend SMV processes with continuous variables and flow conditions. In particular, an HYDI module is a tuple $\langle$PARAM, VAR, IVAR, INIT, TRANS, INVAR, FLOW$\rangle$, where:

- PARAM,IVAR,INIT,TRANS,INVAR are defined as before,
- VAR is a set of variable declaration where an additional type, called *continuous*, is allowed; thus, the declaration defines a set V of discrete variables and a set X of continuous variables,
- FLOW is a set of flow condition declaration defining a formula F over the variables $V \cup W \cup \dot{X}$.

The semantics of HYDI modules is defined in terms of Symbolic Hybrid Systems (SHSs). An SHS S is a tuple $\langle V, X, W, I, T, Z, F \rangle$ where:

- V is the set of symbolic discrete variables representing the states,
- X is the set of symbolic continuous variables representing the states,
- W is the set of symbolic variables representing the events,
- $I(V, X)$ is the initial formula,
- $T(V, X, W, V', X')$ is the transition formula,
- $Z(V, X)$ is the invariant formula,
- $F(V, \dot{X})$ is the flow formula.

If the domain of the variables V is finite, a SHS H corresponds to the HA $\langle Q, A, Q_0, R, X, \mu, \iota, \xi, \theta \rangle$ where

- Q is the set of assignments to V,
- A is the set of assignments to W,
- $Q_0 = \{s \in Q \mid s, x \models I \text{ for some assignment } x \text{ to } X\}$,
- $R = \{(s_1, a, s_2) \in Q \times A \times Q \mid s_1, x_1, a, s_2', x_2' \models T \text{ for some assignment } x_1, x_2' \text{ to } X, X'\}$,
- $\mu(s) = F|_{s(V)}$,
- $\iota(s) = I|_{s(V)}$,
- $\xi(s) = Z|_{s(V)}$,
- $\theta(s_1, a, s_2) = T|_{s_1(V), a(W), s_2'(V')}$.

The semantics of an HYDI module is defined recursively on the hierarchical structure of the instantiation as for SMV modules (see Section III-C). If a module M does not contain module instantiations, its semantics is given by the SHS $\langle V \cup P, X, W, I, T, Z, F \rangle$. The semantics of a HYDI module instantiation extends straightforwardly the semantics of an SMV module instantiation, by renaming each variable v in the flow condition with $i.v$ and substituting each parameter p of M_i with $\beta(p)$.

If a module M contains the instantiations $\mathcal{I}$, its semantics is defined as the tuple $\langle V \setminus \mathcal{I} \cup P, W, I \wedge \bigwedge_{i \in \mathcal{I}} I_i, T \wedge \bigwedge_{i \in \mathcal{I}} T_i, Z \wedge \bigwedge_{i \in \mathcal{I}} Z_i, F \wedge \bigwedge_{i \in \mathcal{I}} F_i \rangle$.

The semantics of an HYDI program extends the definition of the un-timed case given in Section IV-A1 in the straightforward way.

C. Untiming compilation

We describe a compilation that maps a HYDI program into an equivalent un-timed program. The compilation depends on the semantics of the represented network of hybrid automata. Moreover we restrict the INIT, TRANS, INVAR and FLOW declarations of every module as follows:

- the corresponding formulas I, T, Z, and F are Boolean combinations of assignment to the discrete variables and linear constraints over the continuous variables (in the form $\sum_j c_j x_j \bowtie c$, where $\bowtie \in \{\leq, \geq, <, >, =\}$ where c_j, c are constants and x_j are variables, or next variables or derivatives),
- for all assignment s to the discrete variables V, the formulas Z and F restricted to s are conjunctions of linear constraints (thus, $v = d \rightarrow (\dot{x} \geq 0 \wedge \dot{x} \leq 1)$ is allowed, while $v = d \rightarrow (\dot{x} \geq 1 \vee \dot{x} \leq 0)$ or $v = d \rightarrow (\dot{x} \neq 0)$ are not allowed).

The restrictions is such that the constraints in the untimed model are linear, and thus we consider LHA. The untimed program may correspond either to the global-time or to the local-time semantics.

1) Global time: Let H be the HYDI program $\langle \mathcal{M}, main \rangle$ and let S_i be the SHS corresponding to the process instance i of the main. We define the un-timed HYDI program $U = \langle \mathcal{M}_U, main_U \rangle$ as follows:

- $\mathcal{M}_U = \{M_U\}_{M \in \mathcal{M}}$, where, if $M = \langle \text{PARAM}, \text{VAR}, \text{IVAR}, \text{INIT}, \text{TRANS}, \text{INVAR}, \text{FLOW} \rangle$, then $M_U = \langle \text{PARAM}_U, \text{VAR}_U, \text{IVAR}_U, \text{INIT}_U, \text{TRANS}_U, \text{INVAR}_U, \text{FLOW}_U \rangle$ is defined as follows:
  - PARAM_U is obtained from PARAM by adding a further parameter δ,
  - VAR_U is obtained from VAR by changing the type of continuous variables from *continuous* to *real*,
  - IVAR_U is obtained from IVAR by extending the domain of ϵ with an additional symbol T,
  - $\text{INIT}_U = \text{INIT}$,
  - TRANS_U is obtained from TRANS and FLOW as follows: if T and F are the corresponding formulas, TRANS_U defines the formula $((\epsilon \neq \text{T}) \rightarrow T) \wedge ((\epsilon = \text{T}) \rightarrow F_U)$ where F_U is obtained from F by replacing the predicates of the form $\sum_j \dot{x_j} \bowtie c$ with $\sum_j x'_j - x_j \bowtie c \times \delta$,
  - $\text{INVAR}_U = \text{INVAR}$.
- if $main = \langle \text{PARAM}, \text{VAR}, \text{IVAR}, \text{INIT}, \text{TRANS}, \text{INVAR}, \text{SYNC} \rangle$ then $main_U = \langle \text{PARAM}_U, \text{VAR}_U, \text{IVAR}_U, \text{INIT}_U, \text{TRANS}_U, \text{INVAR}_U, \text{SYNC}_U$ defined as follows:
  - $\text{PARAM}_U = \text{PARAM}$,
  - VAR_U is obtained from VAR by passing to each process instance the additional argument δ,
  - IVAR_U is obtained from IVAR by adding the declaration of δ of type $real$,
  - $\text{INIT}_U = \text{INIT}$, $\text{TRANS}_U = \text{TRANS}$, $\text{INVAR}_U = \text{INVAR}$,
  - SYNC_U is obtained from SYNC by adding the synchronizations $\langle i, j, \text{T}, \text{T} \rangle$, for every pair of process instances.

2) Local time: In case of local time, the HYDI program $U = \langle \mathcal{M}_U, main_U \rangle$ is defined as follows:

- $\mathcal{M}_U = \{M_U\}_{M \in \mathcal{M}}$, where, if $M = \langle \text{PARAM}, \text{VAR}, \text{IVAR}, \text{INIT}, \text{TRANS}, \text{INVAR}, \text{FLOW} \rangle$, then $M_U = \langle \text{PARAM}_U, \text{VAR}_U, \text{IVAR}_U, \text{INIT}_U, \text{TRANS}_U, \text{INVAR}_U \rangle$ is defined as follows:
  - $\text{PARAM}_U = \text{PARAM}$,
  - VAR_U is obtained from VAR by changing the type of continuous variables from *continuous* to *real* and by adding the declaration of t of type $real$,
  - IVAR_U is obtained from IVAR by extending the domain of ϵ with an additional symbol T and by adding the declaration of δ of type $real$,
  - $\text{INIT}_U = \text{INIT}$,
  - TRANS_U is obtained from TRANS and FLOW as follows: if T and F are the corresponding formulas, TRANS_U defines the formula $((\epsilon \neq \text{T}) \rightarrow (T \wedge t' = t)) \wedge ((\epsilon = \text{T}) \rightarrow (F_U \wedge t' = t + \delta)$, where F_U is obtained from F by replacing the predicates of the form $\sum_j \dot{x_j} \bowtie c$ with $\sum_j x'_j - x_j \bowtie c \times \delta$,
  - $\text{INVAR}_U = \text{INVAR}$.
- if $main = \langle \text{PARAM}, \text{VAR}, \text{IVAR}, \text{INIT}, \text{TRANS}, \text{INVAR}, \text{SYNC} \rangle$, then $main_U = \langle \text{PARAM}_U, \text{VAR}_U, \text{IVAR}_U, \text{INIT}_U, \text{TRANS}_U, \text{INVAR}_U, \text{SYNC}_U$ defined as follows:
  - $\text{PARAM}_U = \text{PARAM}$, $\text{VAR}_U = \text{VAR}$, $\text{IVAR}_U = \text{VAR}$, $\text{INIT}_U = \text{INIT}$, $\text{INVAR}_U = \text{INVAR}$, $\text{SYNC}_U = \text{SYNC}$.
  - TRANS_U is obtained from TRANS by adding the condition $i.\epsilon = a \rightarrow i.t = j.t$ for every synchronization $\langle i, j, a, b \rangle$ of $main$.

D. Urgency

When modeling complex timing constraints, it is usually useful to specify that a certain state is urgent, in the sense that we do not allow time elapsing in that state. HYDI has a syntactic sugar to specify such condition. The syntax is given by a new declaration URGENT followed by a predicate ϕ over variables and parameters. The semantics is given by a new condition $\phi \rightarrow (\epsilon \neq \text{T})$, implicitly conjoined with the transition condition.

V. VALIDATION OF THE LANGUAGE

We validated the use of the language modeling several hybrid automata benchmarks proposed in the literature [25], [27], [29], [33], [44]. The models exploit the HYDI features such as the symbolic representation, the discrete interaction among processes, the continuous variables and flow conditions and urgent transitions. Moreover, we generate benchmarks with an increasing number of components. In total, we modeled 12 different families of benchmarks, which are available at http://es.fbk.eu/people/mover/hydi.

To support the HYDI language we extended the state-of-the-art model checker NUSMV3. First, we added the support of parsing the HYDI language, then we added the support for compiling an HYDI program to a SMV program (with infinite

state variables). The compilation is divided in two step, the first performs the "untiming" of the model while the second performs the composition of the processes.

VI. Formal verification

A. SMT-based techniques

HyDI programs are compiled into STSs (as described in Section IV-C) and analyzed with techniques based on *Satisfiability Modulo Theory* (SMT) [37]. An SMT problem is the satisfiability problem for Boolean combination of predicates in particular decidable first-order theories. Given a formula ψ, the satisfiability problem consists of deciding whether there exists a model, an assignment to the free variables in ψ, which satisfies ψ. For example, the formula $x <= y \wedge x + 3 = z \vee z >= y$, with $x, y, z \in \mathbb{R}$, is in the theory of *Linear Rational Arithmetic* (LRA) and it is satisfiable (e.g. $x := 5, y := 6, z := 8$ is a model which satisfies the formula).

The theory used in the verification of hybrid systems is the theory of reals. In the restricted case of linear hybrid automata [26], the predicates lie in the theory of LRA, for which efficient solvers exist.

Several formal verification techniques for STSs relies on SMT solvers. Among them, one the most successful techniques is Bounded Model Checking (BMC), first proposed for finite-state systems [13], later extended to software [8] and hybrid systems [19]. BMC determines if the model of a system S violates a property ϕ up to a bounded number of steps k. In BMC the behavior of the system up to k steps and the formula ϕ are encoded in a Boolean formula. The satisfiability of the formula is checked using a SMT solver. If the formula is satisfiable then ϕ is violated by S, otherwise S does not violates ϕ up to k steps. The BMC approach was also extended in order to perform unbounded model checking, overcoming the drawback of the incompleteness. Methods used to perform unbounded model checking with a SMT solver are k-induction [18], [34], [38], interpolation-based model checking [31], [43], and abstraction refinement [16].

B. Exploiting the network

The HyDI input language enables the development of formal verification techniques which exploits the structure of the hybrid automata network.

In particular, we developed a new BMC encoding which exploit an efficient local encoding of the components and superimpose compatibility constraints resulting from the synchronizations [14]. This paradigm has been pushed forward by considering the structure mandated by scenarios specified by means of sequence message charts: we constructed an encoding which exploits the events of the scenario and enables the incremental use of the SMT solver; moreover, we simplify the encoding with invariants discovered applying discrete model checking on an abstraction of the network [4]. Finally, we conceived a new compositional algorithm, which combine over- and under-approximation of the components to effectively build a trace in the network [5].

VII. Related work

Several languages have been proposed to model Hybrid Systems. A first key difference is in the kind of representation, symbolic or explicit, used for specifying the discrete locations of the system.

Most languages that describe Hybrid Systems use an explicit representation of the discrete locations and transitions, which are explicitly enumerated in the model. For this reason they cannot specify flow and invariant conditions for a set of locations. Most of the model checker for timed and hybrid automata, have an input language with an explicit representation for the discrete evolution of the system. Uppaal [11] models a network of timed automata via message passing over communication channels. Relevant features are urgent channels and locations, commited locations and bounded integer variables. Instead, the input languages of HyTech [25] and Phaver [22] allow to specify a network of Linear Hybrid Automata. HyTech assumes the parallel composition of all automata, which synchronize on labels with the same name. In Phaver, the user can specify which automata synchronize, enabling compositional verification [21]. d/dt models affine continuous dynamics with inputs, while guards and invariants are convex polyhedra. The continuous dynamic for each location is defined using a matrix. Also HSolver enables non-linear constraints over continuous variables and it is not compositional. Uppaal, HyTech, Phaver, d/dt and HSolver do not allow rich types for discrete variables, such as Boolean, Enumeration, Word, Unbounded Integer and Unbounded Real. Both Charon [7] and Masaccio [24] have an explicit representation for the discrete state space. The first focuses on hierarchical specification, the latter on compositionality. CHECKMATE models *threshold event-driven hybrid systems* (TEHS) using a subset of the standard MATLAB STATEFLOW/SIMULINK blocks and a set of custom blocks. The *Hybrid Systems Interchange Format* (HSIF) [39] and the *Common Interchange Format* (CIF) [42] were proposed as standards to represent and interchange models of Hybrid Systems. HSIF does not allow hierarchical state machines, while CIF allows to write rich constraints (DAE, Differential Algebraic Equations), which mix variables and derivatives.

Other languages represent symbolically the entire Hybrid System.

The Hybrid System Description Language (HYSDEL) [41] uses a symbolic representation also for the discrete modes and switches. However, HYSDEL describes only discrete time Hybrid Systems and allows the communications of multiple components only via shared variables.

Hybrid SAL [40] extends the language of the SAL [12] model checker to model hybrid systems. It allows arbitrary polynomials over continuous variables and affine dynamics, but it forces to define flow conditions and invariants explicitly for each mode of the system. Moreover, since components in SAL communicates through shared variables, it not easy to express the synchronization of asynchronous components via

discrete interaction, as in the hybrid automata case.
The language HLANG [20] is very close to HYDI and was developed in the project area H of AVACS as intermediate input language for several verification tools (e.g. HYSAT [19], FOMC [17]). The language employs a symbolic representation for hybrid automata and enables very expressive constraints for continuous variables, also affine and non-linear. The language allows to express the composition of hybrid automata. However, the automata communicates through shared variables and automata are composed interleaving their transitions. Thus, there is no native support for event-based synchronizations.

Hybrid programs [9], [35] are similar to programs for discrete systems, but they add the description of the continuous evolution. [9] extends the synchronous language Quartz [36] with continuous variables. The user specifies in the program when the continuous evolution can happen. The semantic is such that the discrete statements in the program are istantaneous, while the statements over continuous variables allow the elapse of time. The continuous evolution is specified using ODE (Ordinary Differential Equations). Assignments in the program are expressed in a functional form, thus they are less expressive than the relational representation of HYDI. Two blocks of statements can run in parallel (synchronous composition), while there is no support for asynchronous composition. These hybrid programs are then translated into a monolithic extended finite state machine. Also the hybrid programs defined by Platzer [35] allow rich constraints over continuous variables, sequences, loops and non-deterministic choiches of statetements. However, they miss the possiblity of expressing the parallel execution of programs. Hybrid programs are very different in nature from the representation used in HYDI.

VIII. CONCLUSIONS

We described HYDI, a novel language that has been specifically designed in order to support the verification of hybrid synchronized processes with symbolic techniques. HYDI extends the SMV language into two directions: on one hand, we introduced timing aspects such as continuous variables and flow conditions; on the other hand we define a top-level structure to specify the network and the synchronization of the components. The language has been used as back-end for translations of languages such as MatLab StateFlow/SymulLink and Altarica [2]. We are working on new efficient verification techniques that scale up the analysis by taking into account the timing and synchronizing aspects [4], [5], [14].

REFERENCES

[1] http://compass.informatik.rwth-aachen.de/.
[2] http://www.missa-fp7.eu/.
[3] http://www.sti.uniurb.it/bernardo/twotowers/.
[4] Cimatti A., Mover S., and Tonetta S. Compositional reachability of hybrid systems. Technical report, Fondazione Bruno Kessler, 2010. https://es.fbk.eu/people/mover/hybrid_compositional.
[5] Cimatti A., Mover S., and Tonetta S. Efficient scenario-based verification of hybrid systems. Technical report, Fondazione Bruno Kessler, 2010. https://es.fbk.eu/people/mover/hybrid_scenario.
[6] R. Alur, C. Courcoubetis, T.A. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pages 209–229, 1992.
[7] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. Modular specification of hybrid systems in charon. In *HSCC*, pages 6–19. Springer, 2000.
[8] G. Audemard, M. Bozzano, A. Cimatti, and R. Sebastiani. Verifying Industrial Hybrid Systems with MathSAT. *ENTCS*, 119(2):17–32, 2005.
[9] K. Bauer and K. Schneider. From synchronous programs to symbolic representations of hybrid systems. In *HSCC*, pages 41–50, 2010.
[10] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi. Partial Order Reductions for Timed Systems. In *CONCUR*, pages 485–500, 1998.
[11] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In *Hybrid Systems*, pages 232–243. Springer–Verlag.
[12] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Muoz, S. Owre, H. Rue, J. Rushby, V. Rusu, H. Sadi, N. Shankar, E. Singerman, and A. Tiwari. An Overview of SAL. In *LFM*, pages 187–196, 2000.
[13] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *TACAS*, pages 193–207, 1999.
[14] L. Bu, A. Cimatti, X. Li, S. Mover, and S. Tonetta. Model checking of hybrid systems using shallow synchronization. In *FMOODS/FORTE*, pages 155–169, 2010.
[15] R. Cavada, A. Cimatti, E. Olivetti C.A. Jochim, G. Keighren, M. Pistore, M. Roveri, and A. Tchaltsev. *NuSMV 2.1 User Manual*, 2002.
[16] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *CAV*, pages 154–169, 2000.
[17] W. Damm, S. Disch, H. Hungar, S. Jacobs, J. Pang, F. Pigorsch, C. Scholl, U. Waldmann, and B. Wirtz. Exact state set representations in the verification of linear hybrid systems with large discrete state space. In *ATVA*, pages 425–440, 2007.
[18] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.*, 89(4), 2003.
[19] M. Fränzle and C. Herde. HySAT: An efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design*, 30(3):179–198, 2007.
[20] M. Fränzle, H. Hungar, C. Schmitt, and B. Wirtz. Hlang: Compositional representation of hybrid systems via predicates. Reports of SFB/TR 14 AVACS 20, July 2007. ISSN: 1860-9821, http://www.avacs.org.
[21] G. Frehse. Compositional verification of hybrid systems with discrete interaction using simulation relations. In *CACSD*, 2004.
[22] G. Frehse. PHAVer: Algorithmic Verification of Hybrid Systems Past HyTech. *International Journal on Software Tools for Technology Transfer (STTT)*, 10(3), June 2008.
[23] K. Heljanko and I. Niemelä. Bounded LTL model checking with stable models. *Theory and Practice of Logic Progr.*, 3(4-5):519–550, 2003.
[24] T. A. Henzinger. Masaccio: A formal model for embedded components. In *IFIP TCS*, pages 549–563. Springer-Verlag, 2000.
[25] T. A. Henzinger and P. Ho. Hytech: The cornell hybrid technology tool. In *Hybrid Systems II*, pages 265–293. Springer-Verlag, 1995.
[26] T.A. Henzinger. The Theory of Hybrid Automata. In *LICS*, pages 278–292. IEEE Computer Society, 1996.
[27] C. Herde, A. Eggers, M. Fränzle, and T. Teige. Analysis of Hybrid Systems Using HySAT. In *ICONS*, pages 196–201, 2008.
[28] G. Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, 2003.
[29] S.K. Jha, B.H. Krogh, J.E. Weimer, and E.M. Clarke. Reachability for Linear Hybrid Automata Using Iterative Relaxation Abstraction. In *HSCC*, pages 287–300, 2007.
[30] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.
[31] K. L. McMillan. Interpolation and sat-based model checking. In *CAV*, pages 1–13, 2003.
[32] S.P. Miller, M.W. Whalen, and D.D. Cofer. Software model checking takes off. *Commun. ACM*, 53(2):58–64, 2010.
[33] Olaf Mller, Olaf M Uller, and Thomas Stauner. Modelling and verification using linear hybrid automata - a case study. *Mathematical and Computer Modelling of Dynamical Systems: Methods, Tools and Applications in Engineering and Related Sciences*, 71, 2000.
[34] L. Pike. Real-Time System Verification by *k*-Induction. Technical Report TM-2005-213751, NASA Langley Research Center, 2005.

[35] A. Platzer. Differential-algebraic dynamic logic for differential-algebraic programs. *J. Log. Comput.*, 20(1):309–352, 2010. Advance Access published on November 18, 2008.
[36] K. Schneider. The synchronous programming language quartz. Technical report, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2009. Internal Report 375.
[37] R. Sebastiani. Lazy Satisability Modulo Theories. *JSAT*, 3(3-4):141–224, 2007.
[38] M. Sheeran, S. Singh, and G. Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In *FMCAD*, pages 108–125, 2000.
[39] MoBIES team. Hsif semantics. Technical report, University of Pennsylvania, 2002.
[40] A. Tiwari. Hybridsal: Modeling and abstracting hybrid systems. Technical report, Computer Science Laboratory, SRI International, 2003.
[41] F.D. Torrisi and A. Bemporad. Hysdel-a tool for generating computational hybrid models for analysis and synthesis problems. *Control Systems Technology, IEEE Transactions on*, 12(2):235 – 249, 2004.
[42] D.A. van Beek, M.A. Reniers, R.R.H. Schiffelers, and J.E. Rooda. Foundations of a Compositional Interchange Format for Hybrid Systems. In *HSCC*, pages 587–600, 2007.
[43] Y. Vizel and O. Grumberg. Interpolation-sequence based model checking. In *FMCAD*, pages 1–8, 2009.
[44] J. Zhao, X. Li, T. Zheng, and G. Zheng. Removing Irrelevant Atomic Formulas for Checking Timed Automata Efficiently. In *FORMATS*, pages 34–45, 2003.