# From control-command synchronous programs to hybrid automata

**Olivier Bouissou** *

* *CEA, LIST, Boîte courrier 94, 91191 Gif-sur-Yvette Cedex, France (e-mail: olivier.bouissou@cea.fr).*

**Abstract:** Hybrid automata are often cited as the best model to describe and analyze control-command systems. However, no formal link exists between this high-level description and the low-level implementation of the control law (in C for example). In this article, we propose a framework that computes from a synchronous program an equivalent hybrid automata. To do this, we use an intermediate model, namely *sampled hybrid automata* (or S-HA in short), that are a restriction of hybrid automata in which each location has a specific sampling rate. We show that these S-HA can be translated into equivalent hybrid automata, and show that we can automatically construct, from a control-command program, a S-HA. We also show that both models are equivalent w.r.t. the continuous trajectories.

Keywords: Hybrid automata; embedded programs.

## 1. INTRODUCTION

Due to the development of the embedded software industry, there is a growing need for verification techniques for systems that mix discrete and continuous evolutions. Real time control-command programs are good examples of such hybrid systems. A control-command software is a program that interacts with a continuous environment via sensors measuring some physical value and actuators modifying the dynamics of the physical values. Such programs often follow a loop "Read-Compute-Write", where "Read" means read values from the sensors, "Compute" means treat these values using numerical computations (often using floating point numbers) to chose the actuator to activate, and "Write" means actually changing the continuous dynamics using an actuator. Due to the real time constraint, this loop is executed at a specific frequency, which means that the "Read" and "Write" actions occur at specific, known time stamps. So, a real time control-command program implements a *synchronous, sampled based* controller that takes actions (i.e. changes actuators) at specific time instants and based on samples (i.e. inputs periodically read) of a continuous value.

As these programs are often safety critical, it is very important to verify their correct behavior. Abstract interpretation based techniques (Cousot and Cousot [1977, 1992]) are often applied to these programs but they forget about the "Read" and "Write" steps of the execution loop, i.e. they forget the hybrid nature of such systems (Cousot [2005]). On the other hand, hybrid automata (Henzinger [1996]) are often used to describe and analyze hybrid systems made of a continuous environment and a discrete controller (Müller and Stauner [2000]). However, there is no formal link between a hybrid automaton and the actual implementation of the controller: no semantics-preserving transformation exists. More important, the execution model of a hybrid automaton is very different to the "Read-Compute-Write" which is the standard in the embedded software industry: a hybrid automaton is an *event based* controller, i.e. it modifies the continuous evolution *as soon as an event occur*. These events (often named zero-crossing events) imply that there is no guarantee on when the change in the dynamics will occur, which is why this model is not well suited for real time applications.

To bypass this problem, we introduced in Bouissou and Martel [2008] a new model for hybrid systems, named H-Simple , which is an extension of imperative languages devoted to encode this "Read-Compute-Write" loop. We provided in Bouissou [2009] a method to analyze such models, but this method lacks of the efficiency and power of the most recent analysis methods for hybrid automata (Frehse et al. [2011]). In this article, we thus provide an automatic, semantics-preserving translation from H-Simple models to hybrid automata. This allows to easily integrate into an industrial development cycle the verification techniques developed for hybrid automata, and thus increase the confidence one can have in a control-command software. To do so, we introduce an intermediate model, named *sampled hybrid automata*, that helps for the translation from H-Simple to hybrid automata: we first translate sampled hybrid automata to HA, and then H-Simple models to sampled hybrid automata.

The rest of this article is organized as follows. In Section 2, we recall the definition of hybrid automata and H-Simple models, and give their semantics as transition systems. In Section 3, we present our new model of sampled hybrid automata and prove the semantics-preserving translation into hybrid automata. In Section 4, we present and prove the translation from H-Simple to S-HA, while Section 5 presents related work and conclusion.

*Notations* In this article, $\mathbb{R}$ (resp. $\mathbb{R}_+$) denotes the set of real (resp. non-negative real) numbers. The set of booleans is denoted $\mathbb{B}$ and the set of floating-point values is $\mathbb{F}$.

## 2. PRELIMINARIES

*Variables, valuations and predicates* Given a finite set $V$ of variables, we define the set of valuations by $\Sigma_V = V \to \mathbb{R}$, i.e. a valuation $\sigma \in \Sigma_V$ is a function associating to each variable a real value. An atomic predicate over a set of variable $V$ is of the form $e \bowtie 0$, where $e$ is an arithmetic expression involving variables of $V$ and $\bowtie \in \{\leq, \geq, =\}$ is a comparison operator. A predicate over $V$ is a conjunction of atomic predicates. We denote by $P(V)$ the set of all predicates over $V$. Given a valuation $\sigma \in \Sigma$ and a predicate $p \in P(V)$, we say that $\sigma$ verifies $p$, denoted by $\sigma \models p$, if and only if the evaluation of $p$ in $\sigma$ (where the value of a variable $v$ is given by $\sigma(v)$) verifies the predicate.

For a set of variables $V = \{x, y, \dots\}$, we denote $V'$ the set $\{x', y', ...\}$ and $\dot{V}$ the set $\{\dot{x}, \dot{y}, \dots\}$. Intuitively, a valuation over $V'$ represents the value of the variables after the firing of a discrete transition and a valuation over $\dot{V}$ represents the time derivative of the variables in $V$. We denote by $P(V, V')$ the set of predicates over $V \cup V'$.

### 2.1 Hybrid Automata

*Definition 1.* (Hybrid automata, Henzinger [1996]). A hybrid automata $\mathcal{A}$ is a 6-tuple $(L, V, Lbl, I, F, T)$ such that $L$ is a finite set of *locations*, $V$ is a finite set of *variables*, $Lbl$ is a finite set of *labels* and:

- $I : L \to P(V)$ maps locations to invariants;
- $F : L \to P(V, \dot{V})$ maps locations to a flow equations;
- $T \subseteq L \times Lbl \times L \times P(V) \times P(V, V')$ are transitions.

For an element $(s, l, o, g, u) \in T$, $s$ is the source, $o$ is the destination, $l$ is the label, $g$ is the guard (preventing the transition to fire) and $u$ is the update relation (linking the state before and after the transition is fired).

The execution of a hybrid automaton is a labeled transition system between states, see Definition 2. The state of a hybrid automaton $\mathcal{A}$ is a couple $\sigma = (s, \sigma_v)$ such that $s \in L$ is a location of $\mathcal{A}$ and $\sigma_v \in \Sigma_V$ is a valuation relating each variable of $\mathcal{A}$ to its value.

*Definition 2.* (Execution of a hybrid automaton). Let $\mathcal{A}$ be a hybrid automaton. We define the *labeled transition system* $\xrightarrow{\alpha}$ with $\alpha \in Lbl \cup \mathbb{R}_+$ that relates states of $\mathcal{A}$ by the following inference rules:

$$\frac{(s, l, o, g, u) \in T \qquad \sigma_v \models g \qquad (\sigma_v, \sigma'_v) \models u}{(s, \sigma_v) \xrightarrow{l} (o, \sigma'_v)} \text{ DISCRETE}$$

$$\frac{\begin{array}{c}\exists \tau > 0 \\ \exists \rho : [0, \tau[ \to \Sigma \qquad \rho(0) = \sigma_v \qquad \rho(\tau) = \sigma'_v \\ \forall t \in [0, \tau[,\ \rho(t) \models I(s),\ \rho(t), \dot{\rho}(t) \models F(s)\end{array}}{(s, \sigma_v) \xrightarrow{\tau} (s, \sigma'_v)} \text{ CONTINUOUS}$$

An execution of $\mathcal{A}$ is a sequence of states $\sigma_0,\ \sigma_1, \dots,\ \sigma_n$ such that $\forall i \in [0, n-1],\ \sigma_i \xrightarrow{\alpha} \sigma_i$ with $\alpha \in Lbl \cup \mathbb{R}_+^*$.

In the rule DISCRETE of Definition 2, the notation $(\sigma_v, \sigma'_v) \models u$ means that the predicate $u \in P(V, V')$ is verified by the valuation that assigns to each variable $v \in V$ the value $\sigma_v(v)$ and to each variable $v' \in V'$ the value $\sigma'_v(v)$. Equivalently, the notation $\rho(t), \dot{\rho}(t) \models F(s)$ means that the predicate $F(s) \in P(V, \dot{V})$ is verified by the valuation that assigns to each variable $v \in V$ the value $\rho(t)(v)$ and to each variable $\dot{v} \in \dot{V}$ the value $\dot{\rho}(t)(v)$. So the rule CONTINUOUS means that there must be a differentiable function $\rho$ that modifies the value of the variables according to the flow equation $F(s)$.

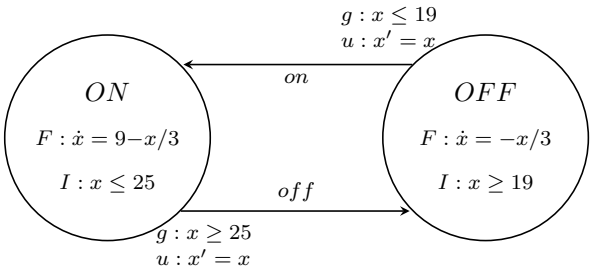


Fig. 1. Hybrid automata representing the heating system.

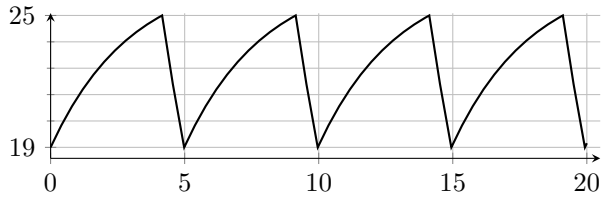


Fig. 2. Simulation of the execution of the hybrid automata for Figure 1.

*Example 3.* (Heating system). Figure 1 gives the hybrid automata representing a heating system: it is a thermostat that turns on or off a heater in a room in order to keep the room temperature $x$ between 19 and 25. Whenever the temperature reaches 19 (resp. 25), it turns the heater on (resp. off). When the heater is on, the temperature increases following the differential equation $\dot{x} = 9 - x/3$, and when the heater is off, it decreases with the differential equation $\dot{x} = -x/3$.
Figure 2 shows a simulation of the execution of this automata when the initial location is $ON$ and the initial value for $x$ is $x = 19$. We see that the room temperature stays within the acceptable range $[19, 25]$.

### 2.2 H-SIMPLE

We now present our model for control-command programs that allows to explicitly model sensors and actuators, as well as the continuous environment. This model was introduced in Bouissou and Martel [2008] together with a denotational semantics. We here present an operational semantics of the same model: this was informally presented in Bouissou [2009], we give more details in this paper. Our model is able to represent control-command systems that follow the "Read-Write-Compute" loop described in Section 1 and that have *binary* actuators (also named "Bang-bang" controllers).

*Syntax* We now consider three kind of variables that will be handled differently by the system: the discrete variables `DVar`, the continuous variables `CVar` and the shared variables `AVar` that model the actuators.

*Definition 4.* (H-SIMPLE model, Bouissou [2008]). A hybrid program $\mathcal{P}$ is a triple $(P, \kappa, m)$ such that $m \in \mathbb{N}$ is the number of actuators and:

$$f \in \mathbb{F} \qquad X \in \texttt{DVar} \cup \texttt{AVar}$$

**AExp :** $a ::= f \mid X \mid a + a \mid a - a \mid a \times a \mid a/a$

**BExp :** $b ::= \textbf{\textit{true}} \mid \textbf{\textit{false}} \mid a = a \mid a \leq a$

**Stmt :** $s ::= X := a \mid s; s$
$\mid \textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ s\ \textbf{\textit{else}}\ s$
$\mid \textbf{\textit{while}}\ b\ \textbf{\textit{do}}\ s$

---

$$u \in \mathbb{R}_+ \quad i \in \mathbb{N}_+$$
$$X \in \texttt{DVar} \quad y \in \texttt{CVar} \quad b \in \texttt{AVar}$$

$\mathbf{P} ::= \textbf{\textit{while}}\ 1\ \textbf{\textit{do}}\ \mathbf{P}_b$

$\mathbf{P}_b ::= (\textbf{\textit{sens}}.y?X)^*;\ s;\ (\textbf{\textit{act}}.i!b)^*;\ \textbf{\textit{wait}}\ u$

Fig. 3. Syntax of the H-SIMPLE language.

- $P$ is a program written in the syntax given by the rules of Figure 3, where the $\textbf{\textit{act}}.i!b$ statements are with $i \in [1, m]$;
- $\kappa = \{F_c : c \in \mathbb{B}^m\}$ is a set of continuous functions. □

The syntax of the program $P$ is an extension of a standard imperative programming language, as shown on Figure 3. We see that such a program (rule for **P** on Figure 3) is a loop whose body consists of four parts:

- a set of $sens.y?X$ statements, with $y \in \texttt{CVar}$ and $X \in \texttt{DVar}$, that let the program read values from the environment via sensors;
- a set of standard instruction that computes the new values for the actuators;
- a set of $act.i!c$ statements, with $i \in [1, m]$ and $c \in \texttt{AVar}$ that modify the value of the $i^{\text{th}}$ sensor to the value of the boolean variable $c$;
- a $wait\ u$ statement that simulates the passing of time.

*Example 5.* (Heating system in H-SIMPLE ). The listing of Figure 4 shows the program $P$ that encodes the heating system in H-SIMPLE . The full model is $(P, \kappa, 1)$ where $\kappa = \{F_0(t) = -t/3\ ,\ F_1(t) = 9 - t/3\}$.

```
// X is a discrete variable
// t is the continuous temperature
while (true) do
  sens.t?X;    // read value
  if (t>=26)
    c=0;
  if (t=<19)
    c=1;
  act.1!c;    // acts on actuators
  wait(0.1); // delay
```

Fig. 4. Encoding of the heating system in H-SIMPLE .

*Semantics* The state of a hybrid program $\mathcal{P}$ associates to each variable a value, and maintains a value $c \in \mathbb{B}^m$ for the configuration of the actuators. We denote $\Sigma_d : \texttt{DVar} \to \mathbb{F}$ the set of valuations of discrete variables, $\Sigma_c : \texttt{CVar} \to \mathbb{R}$ the set of all valuations of continuous variables and $\Sigma_a : \texttt{AVar} \to \mathbb{B}$ the set of valuations of the shared variables. The state of a hybrid program is thus a couple $\langle \sigma, c \rangle$ where $\sigma \in \Sigma_d \times \Sigma_c \times \Sigma_a$ and $c \in \mathbb{B}^m$. For a state $\langle \sigma, c \rangle$, we shall write $\sigma_d$ (resp. $\sigma_c$ and $\sigma_a$) the restriction of $\sigma$ to the discrete (resp. continuous and shared) variables. The valuations $\sigma_d$ and $\sigma_a$ will be changed by the discrete statements, $c$ will be modified by the ***act*** statements while ***sens*** modify $\sigma_d$ and ***wait*** modify $\sigma_c$. Given a discrete variable $X \in \texttt{DVar}$ and a value $v \in \mathbb{F}$, we write $\sigma[X \mapsto v]$ the state $\sigma'$ such that $\sigma'_c = \sigma_c$, $\sigma'_a = \sigma_a$ and $\sigma'_d = \sigma_d[X \mapsto v]$. We do the same for continuous and shared variables update. We also note $c' = c[i \mapsto v]$ the modification of the $i^{\text{th}}$ actuator in the configuration $c$.

The SOS rules of Figure 5 show the semantics of a hybrid program. We do not give the rules for the discrete statements as they are straightforward (see Hennessy [1990] for example). Let us just recall the rule for the ***while*** statement:

$$\frac{\sigma_d \models b}{(\textbf{\textit{while}}\ b\ \textbf{\textit{do}}\ P, \langle \sigma, c \rangle) \to (P; \textbf{\textit{while}}\ b\ \textbf{\textit{do}}\ P, \langle \sigma', c \rangle)}\ \text{WHILE}$$

This explains the LOOP rule of Figure 5: this loop simulates one execution of the main loop body, i.e. the execution of the *sens* statements, the discrete statements and finally the *act* and *wait* statements.

The execution of the model is as follows: the continuous and the discrete systems are processes that run in parallel and, from time to time, communicate. The time is governed by the program: we assume that all statements except ***wait*** are instantaneous so that the program computes the execution time. The communication from the environment to the program $P$ is done via the ***sens*** statement: $P$ reads the values of the continuous variables at the time the statement is executed (see rule SENSOR). The communication from $P$ to the environment is done via the ***act*** statement: $P$ changes the ODE governing the dynamics from the time the statement is executed (rule ACTUATOR). When the $\textbf{\textit{act}}.i!b$ statement is executed, the $i^{th}$ dimension of the configuration vector $c$ is changed to the value $\sigma_a(b)$, the new continuous mode is then $c' \in \mathbb{B}^m$ such that $c'$ is the vector $c$ where the $i^{th}$ coordinate is changed to $\sigma_a(b)$.

$$\frac{\mathcal{P} = (P, \kappa, m) \qquad (P, \langle \sigma, c \rangle) \to (P, \langle \sigma', c' \rangle)}{(\mathcal{P}, \langle \sigma, c \rangle) \to (\mathcal{P}, \langle \sigma', c' \rangle)}\ \text{LOOP}$$

$$\frac{\sigma' = \sigma[X \mapsto \sigma(y)]}{(\textbf{\textit{sens}}.y?X; P, \langle \sigma, c \rangle) \to (P, \langle \sigma', c \rangle)}\ \text{SENSOR}$$

$$\frac{c' = c[i \mapsto \sigma(b)]}{(\textbf{\textit{act}}.i!b; P, \langle \sigma, c \rangle) \to (P, \langle \sigma, c' \rangle)}\ \text{ACTUATOR}$$

$$\frac{y' = sol \qquad \sigma' = \sigma_c[y \mapsto y']}{(\textbf{\textit{wait}}\ u; P, \langle \sigma, c \rangle) \to (P, \langle \sigma', c \rangle)}\ \text{WAIT}$$

Fig. 5. SOS rules for the H-SIMPLE model.

In this article, we are interested in the evolution of the continuous variables: we will compute a hybrid automaton that makes them behave as the program does. We thus define next the continuous trajectory of a hybrid program $(P, \kappa, m)$.

*Definition 6.* (Continuous trajectory). Let $\mathcal{P}$ be a hybrid program, and let $\langle \sigma^0, c^0 \rangle$ be an initial state. We define the continuous trajectory of $\mathcal{P}$ starting from $\sigma^0$ as the sequence of continuous states $\left(\sigma_c^n\right)_{n\in\mathbb{N}}$ such that:

$$\forall i \in \mathbb{N},\ (\mathcal{P}, \langle \sigma^i, c^i \rangle) \to (\mathcal{P}, \langle \sigma^{i+1}, c^{i+1} \rangle)$$

In other words, the continuous trajectory is the projection on $\Sigma_c$ of the reachable states using the LOOP rule.

*Example 7.* Figure 6 shows the trajectory of the continuous variable $y$ of the system of Example 5. We see that because of the discretization of the controller, the temperature can go below 19.

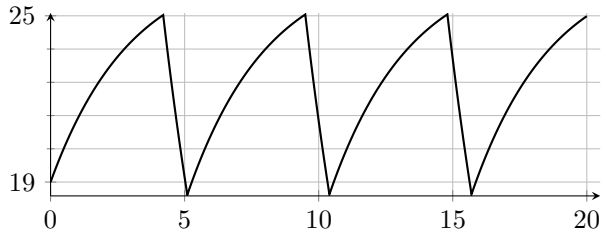


Fig. 6. Trajectory of the temperature for the H-SIMPLE encoding of the heating system.

### *2.3 Differences between HA and* H-SIMPLE

Control systems written in H-SIMPLE are sample based, which means that they perform discrete actions at specific time stamps and between such time stamps, no control is done on the continuous environment. This is the main difference with hybrid automata: in the semantics of hybrid automata, the controller (modeled by discrete states and transitions) permanently monitors the continuous variables and performs a discrete transition *as soon as it can*. Therefore, encoding a sampled based controller as a hybrid automaton is not trivial (it is the goal of this article) and we believe that there is a need for an intermediate language, that we call sampled hybrid automata, to make the link between H-SIMPLE and hybrid automata.

## 3. SAMPLED HYBRID AUTOMATA

In this section, we introduce a special kind of hybrid automata that are better suited to represent sampled based control systems.

### *3.1 Definition*

In a sampled hybrid automata, each each discrete location is associated with a sample time (i.e. a positive real number) that imposes the duration of the continuous transitions in this location. As a consequence, the discrete transitions can only be performed at specific time stamps, which is our goal. In the following, we formally define our notion of *sampled hybrid automata*, or S-HA in short, and the execution of the S-HA. Then, we show that an S-HA can be automatically translated into a hybrid automaton and prove the semantic equivalence of both.

*Definition 8.* (Sampled hybrid automata). A sampled hybrid automaton (S-HA) is a couple $(\mathcal{A}, \tau)$ where $\mathcal{A} = (L, V, Lbl, I, F, T)$ is a hybrid automaton and $\tau : L \to \mathbb{R}_+^*$ associates to each location of $\mathcal{A}$ a sampling rate.

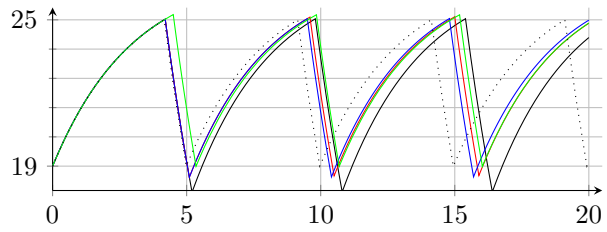


Fig. 7. Simulation of the sampled hybrid automata of Example 9 for various sample periods. In dotted line is the trajectory of the system without sampling (as in Example 3).

*Example 9.* Let $\mathcal{A}$ be the hybrid automaton of the heating system of Example 3. Let $\tau$ be such that $\tau(ON) = 0.2$ and $\tau(OFF) = 0.2$. Then $(\mathcal{A}, \tau)$ is a sampled hybrid automaton.

The execution of a sample hybrid automata $(\mathcal{A}, \tau)$ is a labeled transition system between the states $\sigma$ of the hybrid automata $\mathcal{A}$. Intuitively, a state $(s, \sigma_v)$ with $s \in L$ and $\sigma_v \in \Sigma$ can be modified either by a discrete transition if a guard is verified, or by a continuous transition of duration $\tau(s)$ if the invariant $I(s)$ is verified. This is formalized in Definition 10.

*Definition 10.* (Transition system $SA$). We define the *labeled transition system* $\xrightarrow{\alpha}_\tau$ with $\alpha \in L \cup \mathbb{R}_+^*$ by the inference rules:

$$\frac{(s, l, o, g, u) \in T \qquad \sigma_v \models g \qquad (\sigma_v, \sigma_v') \models u}{(s, \sigma_v) \xrightarrow{l}_\tau (o, \sigma_v')} \text{ DISCRETE}$$

$$\frac{\sigma \models I(s) \qquad \rho(0) = \sigma_v \qquad \rho(\tau(s)) = \sigma_v' \qquad \forall t \in [0, \tau(s)[,\ \rho(t), \dot{\rho}(t) \models F(s)}{(s, \sigma_v) \xrightarrow{\tau}_\tau (s, \sigma_v')} \text{ CONTINUOUS}$$

The only difference with the execution of a hybrid automata is in the CONTINUOUS transition in a location $s$: in an S-HA, this transition has a duration of $\tau(s)$, and the state $\sigma$ is only required to verify the invariant $I(s)$ at the beginning of the transition (we only check that $\sigma \models I(s)$ and not the $\rho(t) \models I(s)$ for all $t \in [0, \tau[$ as for HA).

*Example 11.* (Execution of the heating system). We show on Figure 7 the execution of the sampled version of the hybrid automata of the heating system, for various sample frequencies. We see that this can lead to very different trajectories, and that the invariant $x \in [19, 25]$ that we inferred for the hybrid automata is no longer valid for the sampled version. We also plot, in dotted line, the trajectory of the un-sampled hybrid automata.

### *3.2 Compiling S-HA into HA*

The formalism for hybrid automata is actually expressive enough to translate a S-HA into a HA. Intuitively, given a S-HA $(\mathcal{A}, \tau)$, we add to the variables of $\mathcal{A}$ a fresh variable $t$ that represents the time, whose continuous evolution is given by $\dot{t} = 1$. The transitions of the automata are modified so that the continuous evolution is forced to last $\tau(s)$ seconds in each location $s$. To do so, we modify the guard of each transition that originates in $s$ and add a constraint $t = \tau(s)$. We also add a discrete transition, labeled with $\tau$, that re-initializes $t$ to 0 in

case no discrete transition is activated. Then, only the continuous transition can fire and it will necessarily have a duration of $\tau(s)$.

*Definition 12.* (From S-HA to HA). Let $SA = (\mathcal{A}, \tau)$ be a S-HA, with $\mathcal{A} = (L, V, Lbl, I, F, T)$. We define the HA $\mathcal{A}' = \phi(SA)$ by $\mathcal{A}' = (L, V', Lbl', I', F', T')$ with :

- $V' = V \cup \{t\}$ where $t$ is a fresh variable representing the time;
- $Lbl' = Lbl \cup \{\tau\}$ where $\tau$ is a fresh label;
- $\forall s \in L,\ I'(s) = t \leq \tau(s)$ and $F'(s) = F(s) \wedge \dot{t} = 1$;
- $\forall (s, l, o, g, u) \in T,\ (s, l, o, g \wedge t = \tau(s), u \wedge t' = \tau(o)) \in T'$;
- $\forall s \in L,\ (s, \tau, s, I(s) \wedge t = \tau(l), t' = 0) \in T'$.

The initial state of $\phi(SA)$ is the initial state $(s_0, \sigma'_0)$ where $(s_0, \sigma_0)$ is the initial state of $SA$ and $\sigma'_0$ verifies:

$$\forall v \in V,\ \sigma'_0(v) = \sigma_0(v) \text{ and } \sigma'_0(t) = \tau(s_0)\ .$$

*Example 13.* Figure 8 show the hybrid automata obtained from the translation of the S-HA $(\mathcal{A}, \tau)$ of Example 9.

### *3.3 Correctness of the transformation*

We now prove the equivalence between a sampled hybrid automaton $(\mathcal{A}, \tau)$ and the hybrid automaton $\mathcal{A}' = \phi(\mathcal{A}, \tau)$ defined by Definition 12. Both the S-HA $(\mathcal{A}, \tau)$ and the HA $\mathcal{A}'$ define labeled transition systems and we shall prove that the reachable set for both systems are the same.

The reachable set for a hybrid automata $\mathcal{A}$ from an initial state $\sigma_0$ is defined by:

$$Reach_{\mathcal{A}} = \bigcup_{n \in \mathbb{N}} Reach^n_{\mathcal{A}}(\sigma_0)$$

with

$$Reach^0_{\mathcal{A}}(\sigma_0) = \sigma_0$$
$$Reach^{n+1}_{\mathcal{A}}(\sigma_0) = \left\{ \sigma' \mid \exists \sigma \in Reach^n_{\mathcal{A}}(\sigma_0) \text{ s.t. } \sigma \xrightarrow{\alpha} \sigma' \right\}.$$

We define equivalently the reachable set $Reach_{SA}$ for a sampled hybrid automaton $SA = (\mathcal{A}, \tau)$. If $\mathcal{A}' = \phi(SA)$, we want to compare $Reach_{\mathcal{A}'}$ with $Reach_{SA}$. Clearly, $Reach_{\mathcal{A}'}$ has more states as the continuous transitions in a given state $s$ of $\mathcal{A}'$ can last any duration between 0 and $\tau(s)$, while the continuous transition of $SA$ must last $\tau(s)$. Therefore, we shall prove that $Reach_{\mathcal{A}'}$ and $Reach_{SA}$ contains the same state when $t = \tau(s)$. Formally, we will prove that (see Theorem 16):

$$Reach_{SA} = \left\{ (s, \sigma_v) \;\middle|\; \begin{array}{c} \exists (s, \sigma'_v) \in Reach_{\mathcal{A}'} : \sigma'_v(t) = \tau(s) \\ \text{and } \forall v \in V,\ \sigma'_v(v) = \sigma_v(v) \end{array} \right\}$$

Let us introduce two helpful notations: for a state $\sigma = (s, \sigma_v)$ of $SA$, we write $\uparrow \sigma = (s, \sigma_v[t \mapsto \tau(s)])$ the state of $\mathcal{A}'$ and for a state $\sigma' = (s, \sigma'_v)$ of $\mathcal{A}'$ with $\sigma'_v(t) = \tau(s)$, we write $\downarrow \sigma' = (s, \sigma_v)$ the state of $SA$ such that $\forall v \in V,\ \sigma_v(v) = \sigma'_v(v)$. Remark that we have $\uparrow (\downarrow \sigma) = \sigma$.

Before stating our main theorem, we prove two propositions that are helpful for the proof. They state that the transitions of $\mathcal{A}'$ and the transitions of $SA$ are equivalent. From now on, we consider a S-HA $SA = (\mathcal{A}, \tau)$ with $\mathcal{A} = (L, V, Lbl, I, F, T)$ and we note $\mathcal{A}' = \phi(SA)$, with $\mathcal{A}' = (L', V', Lbl', I', F', T')$.

*Proposition 14.* (Equivalence of $l$-transitions). Let $\sigma_1,\ \sigma_2$ be two states of $SA$. For all labels $l \in Lbl$, we have:

$$\sigma_1 \xrightarrow{l}_\tau \sigma_2 \Leftrightarrow \uparrow \sigma_1 \xrightarrow{l} \uparrow \sigma_2\ .$$

**Proof.** Let $\sigma_1 = (s_1, \sigma^1_v)$ and $\sigma_2 = (s_2, \sigma^2_v)$. We have:

$$\begin{aligned} (s_1, \sigma^1_v) \xrightarrow{l}_\tau (s_2, \sigma^2_v) \Leftrightarrow & (s_1, l, s_2, g, u) \in T, \\ & \sigma^1_v \models g \text{ and } \sigma^1_v, \sigma^2_v \models u \\ \Leftrightarrow & (s_1, l, s_2, g', u') \in T', \\ & g' = g \wedge \big(t = \tau(s_1)\big), \\ & u' = u \wedge \big(t = \tau(s_2)\big), \\ & \sigma^1_v \models g \text{ and } \sigma^1_v, \sigma^2_v \models u \\ \Leftrightarrow & \uparrow \sigma_1 \xrightarrow{l} \uparrow \sigma_2 \end{aligned}$$ □

*Proposition 15.* (Equivalence of $\tau$-transitions). Let $\sigma_1,\ \sigma_2$ be states of $SA$, with $\sigma_1 = (s_1, \sigma^1_v)$ and $\sigma_2 = (s_1, \sigma^2_v)$. We have:

$$\sigma_1 \xrightarrow{\tau(s)}_\tau \sigma_2 \Leftrightarrow \uparrow \sigma_1 \xrightarrow{\tau} (s_1, \sigma_v[t \mapsto 0]) \xrightarrow{\tau(s)} \uparrow \sigma_2\,.$$

**Proof.** We know that:

$$\sigma_1 \xrightarrow{\tau(s_1)}_\tau \sigma_2 \Leftrightarrow \begin{cases} \sigma^1_v \models I(s_1) \\ \exists \rho : [0, \tau(s)[ \to \Sigma \;\Big|\; \begin{array}{l} \rho(0) = \sigma^1_v, \\ \rho(\tau(s_1)) = \sigma^2_v, \\ \text{and } \rho \text{ verifies } F(s_1) \end{array} \end{cases}$$

However, $\sigma^1_v \models I(s_1) \Leftrightarrow \uparrow \sigma_1 \xrightarrow{\tau} (s_1, \sigma^1_v[t \mapsto 0])$. Let $\sigma^{1b}_v = \sigma^1_v[t \mapsto 0]$. The conditions on $\rho$ and the invariant $I'(s_1) = t \leq \tau(s_1)$ implies that $(s_1, \sigma^{1b}_v) \xrightarrow{\tau(s_1)} \uparrow \sigma_2$. □

These two propositions are useful to prove that $SA$ and $\phi(SA)$ compute the same reachable set. This is stated in the main theorem of this section.

*Theorem 16.* Let $SA$ be a S-HA and $\mathcal{A}' = \phi(SA)$. Then, we have:

$$Reach_{SA} = \left\{ (s, \sigma_v) \;\middle|\; \begin{array}{c} \exists (s, \sigma'_v) \in Reach_{\mathcal{A}'} : \sigma'_v(t) = \tau(s) \\ \text{and } \forall v \in V,\ \sigma'_v(v) = \sigma_v(v) \end{array} \right\}$$

**Proof.** Given a set $S$ of states for $\mathcal{A}'$, let us write

$$\downarrow S = \left\{ (s, \sigma_v) \;\middle|\; \begin{array}{c} \exists (s, \sigma'_v) \in S : \sigma'_v(t) = \tau(s) \\ \text{and } \forall v \in V,\ \sigma'_v(v) = \sigma_v(v) \end{array} \right\}\ .$$

We thus want to prove that $Reach_{SA} = \downarrow Reach_{\mathcal{A}'}$. To do so, we will prove by induction that, for all $n \in \mathbb{N}$, $Reach^n_{SA} = \downarrow \left(Reach^n_{\mathcal{A}'}\right)$. The case for $n = 0$ follows immediately from the definition of the initial state of $\mathcal{A}'$, see Definition 12.

Assume now that $Reach^n_{SA} = \downarrow \left(Reach^n_{\mathcal{A}'}\right)$ for some $n \in \mathbb{N}$. Then,

$$\begin{aligned} \sigma \in Reach^{n+1}_{SA} \Leftrightarrow & \exists \sigma' \in Reach^n_{SA},\ \sigma' \xrightarrow{\alpha}_\tau \sigma \\ \Leftrightarrow & \exists \sigma_a \in Reach^n_{\mathcal{A}'},\ \sigma' = \downarrow \sigma_a,\ \sigma' \xrightarrow{\alpha}_\tau \sigma \end{aligned}$$

According to propositions 14 and 15, we have

$$\sigma' \xrightarrow{\alpha}_\tau \sigma \Leftrightarrow \uparrow \sigma' \xrightarrow{\tau}{}^* \xrightarrow{\alpha} \uparrow \sigma$$

So, we have:

$$\sigma \in Reach^{n+1}_{SA} \Leftrightarrow \uparrow \sigma \in Reach^n_{\mathcal{A}'}$$

which proves that $Reach^{n+1}_{SA} = \downarrow \left(Reach^{n+1}_{\mathcal{A}'}\right)$. □

In this section, we introduced an extension of hybrid automata, namely sampled hybrid automata, and showed

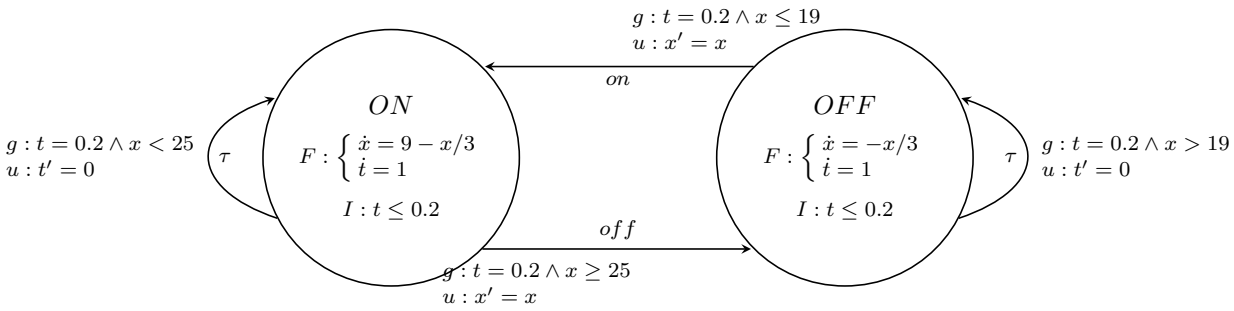


Fig. 8. Hybrid automata encoding the SHA of Example 9.

that they can be expressed in the model of hybrid automata. This model is better suited to encode the "Read-Compute-Write" loop of control-command softwares, as the actions an S-HA can take are driven by the sample times rather than by events, as in HA. In the next section, we show how to encode a H-SIMPLE system into a S-HA and prove the correctness of this translation.

## 4. FROM HSIMPLE TO SHA

We now examine the relations between a hybrid program written in H-SIMPLE and the S-HA model. From a hybrid program $\mathcal{P}$, we build a S-HA named $\mathcal{A}_\mathcal{P}$ such that $\mathcal{A}_\mathcal{P}$ contains all continuous trajectories of $\mathcal{P}$. We thus still have a correctness result, but we are losing the completeness that we had for the translation from S-HA to HA. The main reason is that we want to obtain an automaton which is less complex than the original program so that it can be analyzed using existing tools such as SpaceEx (Frehse et al. [2011]). Therefore we will build an abstraction (in the sense of the abstract interpretation theory) of the impact of the program on the continuous environment and use this abstraction to build the automata. We develop this idea in the rest of this section.

The impact of the program $P = \textbf{\textit{while}}\ 1\ \textbf{\textit{do}}\ P_b$ on its environment $\kappa$ is as follows: at each loop cycle, the program reads values through sensors and modify the continuous dynamics through the binary actuators. This is summed up by the following sequence of transitions of the SOS rules of Figure 5, where we denoted $P_b = P_b'; \textbf{\textit{wait}}\ u$:

$$\begin{aligned}(P, \langle \sigma, c\rangle) &\rightarrow (P_b'; \textbf{\textit{wait}}\ u; P, \langle \sigma, c\rangle) \\ &\rightarrow^* (\textbf{\textit{wait}}\ u; P, \langle \sigma', c'\rangle)\end{aligned} \tag{1}$$

Note that $\sigma_c' = \sigma_c$, as only the ***wait*** transition may modify the continuous variables. We sum up these transitions under the "transition function" $F_P : \Sigma \times \mathbb{B}^m \rightarrow \mathbb{B}^m$ that is defined by $F_P(\sigma, c) = c'$ where $c'$ is defined by equation (1). The function $F_P$ tells how the program modifies the actuators at a given loop cycle. We shall now make an assumption on the function $F_P$ that facilitate the definition of the S-HA that models the hybrid program. We assume that $F_P$ verifies equation (2):

$$\begin{aligned}\forall \sigma_c \in \Sigma_c, c \in \mathbb{B}^m, \forall \sigma_d, \sigma_d' \in \Sigma_d, \forall \sigma_a, \sigma_a' \in \Sigma_a, \\ F_P(\langle \sigma_d, \sigma_c, \sigma_a\rangle, c) = F_P(\langle \sigma_d', \sigma_c, \sigma_a'\rangle, c)\end{aligned} \tag{2}$$

This means that $F_P$ only depends on the values of the continuous variables (that are read through the ***sens*** statements), and not on the values of the discrete variables. In other terms, this means that the program has no memory and that the change in the actuators only depends on the value $\sigma_c$ of the continuous environment, the discrete variables being only used to perform numerical computations on $\sigma_c$.

In this case, we can consider $F_P$ as a function $F_P : \Sigma_c \times \mathbb{B}^m \rightarrow \mathbb{B}^m$ and we shall now write $F_P(\sigma_c, c)$ to be the value of $F_P(\sigma, c)$ for any $\sigma \in \Sigma$ such that $\forall v \in \Sigma_c,\ \sigma(v) = \sigma_c(v)$.

One possibility to transform a hybrid program into a sample hybrid automata is to create an automata with one location and a flow equation parametrized by $c \in \mathbb{B}^m$, with then one transition with no guard and the reset predicate define as $F_P$. This however is not our solution, as such an automata would not really represent the control law. Instead, we translate the hybrid program into a sampled hybrid automata that has one location for each possible configuration of the actuators, i.e. $2^m$ locations. There will be a transition between two locations $c$ and $c'$ if $F_p(\sigma, c) = c'$. This idea is developed in the rest of this section.

### *4.1 Construction of the concrete sampled hybrid automata*

We start by expressing $F_P$ as a conjunction of simpler functions that detect when *one* actuator is activated. To do that, let us first rewrite the program and change each ***act***.$i$!$b$ statement, with $i \in [0, m]$ and $b \in$ `AVar` by the statements

***if*** $b$ ***then***
  ***act***.$i$!1;
***else***
  ***act***.$i$!0;

The semantics of a ***act***.$i$!$v$ statement, with $v \in \mathbb{B}$, is obvious, and this transformation does not change the behavior of the program. We thus now have a set $\{\textbf{\textit{act}}.i_j!v_j : j \in [1, 2p]\}$ of ***act*** statements, with $v_j \in \mathbb{B}$, where $p$ is the number of ***act*** statements in the original program $P$. For each $j \in [1, 2p]$, we define the function $F_j : \Sigma_c \times \mathbb{B}^m \rightarrow \mathbb{B}$ that returns 1 if the statement ***act***.$i_j$!$v_j$ is reached under input $\sigma_c$ and $c$, i.e. if there is a sequence of transitions such that:

$$\begin{aligned}(P, \langle \sigma, c\rangle) &\rightarrow (P_b'; \textbf{\textit{wait}}\ u; P, \langle \sigma, c\rangle) \\ &\rightarrow^* (\textbf{\textit{act}}.i_j!v_j; P_b''; \textbf{\textit{wait}}\ u; P, \langle \sigma', c'\rangle)\end{aligned}$$

Then, the function $F_P$ is defined by, for all $\sigma_c, c \in \Sigma_c \times \mathbb{B}^m$, $F_P(\sigma_c, c) = c'$ with :

$$\forall j \in [1, 2p],\ c'[i_j] = \begin{cases} c[i_j] & \text{if } F_j(\sigma, c) = 0 \\ v_j & \text{if } F_j(\sigma, c) = 1 \end{cases}.$$

We can now define a S-HA that behaves as the hybrid program $\mathcal{P} = (P, \kappa, m)$, with $\kappa = \{F_c : c \in \mathbb{B}^m\}$. We denote it $A_{\mathcal{P}} = (\mathcal{A}, U)$ with $\mathcal{A} = (L, V, Lbl, I, F, T)$, $\forall l \in L,\ U(l) = u$ (remember that $U$ must associate to each location a sampling time, here it is the same for all locations) and:

- $V = \texttt{CVar}$, $L = \mathbb{B}^m$, $Lbl = \{\tau\}$;
- for each $c \in \mathbb{B}^m$, $F(c) = \big(\dot{V} = F_c(V)\big)$;
- for each $c \in \mathbb{B}^m$, $I(c) = \bigwedge_{j\in[0,2p],c\in\mathbb{B}^m} F_j(V, c) \neq 1$;
- $\forall c, c' \in \mathbb{B}^m$, there is a transition $(c, \tau, c', g, u)$ with $u = true$ and
$$g = \bigwedge_{j\in[1,2p]} \{F_j(V, c) : c'[i_j] \neq c[i_j],\ c'[i_j] = v_j\} .$$

The flow equation in each location is thus given by the configuration of the actuator represented by this location. The invariant in a location $c$ says that the system may stay in this location if no ***act*** statement can be reached: this is the meaning of each predicate $F_j(V, c) \neq 1$. Equivalently, there is a transition from $c$ to $c'$ if it possible the reach the ***act*** statements that change $c$ to $c'$. This is the meaning of the guard predicates in each transition.

*Theorem 17.* Let $\mathcal{P}$ be a hybrid program, let $\mathcal{A}_{\mathcal{P}}$ be the corresponding S-HA. Let $Reach_{\mathcal{P}}$ be the set of reachable continuous states from $\mathcal{P}$, and $Reach_{\mathcal{A}_{\mathcal{P}}}$ be the set of reachable states from $\mathcal{A}_{\mathcal{P}}$. Then, we have:
$$Reach_{\mathcal{A}_{\mathcal{P}}} = Reach_{\mathcal{P}} .$$

The proof of this theorem is not difficult but long, so we omit it for the sake of conciseness. Theorem 17 shows that we can encode a hybrid program into a sample hybrid automata. Then, using Theorem 16, we can construct an equivalent general hybrid automata and use model-checking techniques (for example) to verify safety property on the high level description of the program. However, the methods for hybrid systems model checking almost always assume that the guards, invariants and flow equations are linear inequalities (Halbwachs et al. [1994], Sankaranarayanan et al. [2008]), with recent works that can handle polynomial systems (Matringe et al. [2010]). In our case, the invariants and guards are defined using the functions $F_j$, which are in general not linear. Even more, as $F_j$ must check whether some line in the program is reachable for a given state, this can make the computation of reachable sets using polyhedra or zonotopes very difficult. Thus, we provide in the next section a method to replace each function $F_j$ by an abstract function $\boldsymbol{F_j}$ that transforms the condition "is the line $\boldsymbol{act}.i_j!v_j$ reached" by a geometrical condition "does $\sigma_c$ belong to some set". This makes the produced automata better suited for model checking.

### *4.2 Abstracting the transition function*

The abstract functions $\boldsymbol{F_j}$ will be constructed as $k$-trees, see Definition 18. To compute then, we must manipulate abstract states $\boldsymbol{\sigma_c} : \texttt{CVar} \to \mathbb{I}_{\mathbb{R}}$ that maps each continuous variable to an interval of real values. We denote by $\boldsymbol{\Sigma_c}$ the set of abstract states and by $\boldsymbol{B}$ the abstract boolean domain $\boldsymbol{B} = \{0, 1, \top\}$, where $\top$ means that we do not know the value of the boolean value.

*Definition 18.* ($k$-trees ( Bouissou [2009])). A $n$-dimension $k$-tree is a directed, acyclic graph with a unique root and two kinds of nodes: non terminal nodes $N(\boldsymbol{v})$, with $\boldsymbol{v} \in \boldsymbol{\Sigma_c}$ and terminal nodes $V(\boldsymbol{v}, \boldsymbol{b})$ with $\boldsymbol{v} \in \boldsymbol{\Sigma_c}$ and $\boldsymbol{b} \in \boldsymbol{B}$. Each non terminal node $N(\boldsymbol{v})$ has either one terminal child $V(\boldsymbol{v}, \boldsymbol{b})$ with for some $\boldsymbol{b}$, or $2^n$ non terminal children $N_i(\boldsymbol{v_i})$, $i \in [1, 2^n]$, with: $\forall i, j \in [1, 2^n]$, $\boldsymbol{v_i} \cap \boldsymbol{v_j} = \emptyset$ and $\bigcup_{i\in[1,2^n]} \boldsymbol{v_i} = \boldsymbol{v}$.

A $k$-tree $T$ can be seen as a function $T : \Sigma_c \to \boldsymbol{B}$: $\forall \sigma_c \in \Sigma_c,\ T(\sigma_c) = \boldsymbol{b}$ such that $\exists V(\boldsymbol{v}, \boldsymbol{b})$ in $T$ with $\sigma_c \in \boldsymbol{v}$. So a $k$-tree can be seen as a partitioning of the space into three regions: the ones labeled with 1, the ones labeled with 0 and the ones labeled with $\top$.

Now, for each $j \in [1, 2p]$ and each $c \in \mathbb{B}^m$, we compute the $k$-tree $T_{c,j}$ such that:
$$T_{c,j}(\sigma_c) = 1 \Rightarrow F_j(\sigma_c, c) = 1$$
$$T_{c,j}(\sigma_c) = 0 \Rightarrow F_j(\sigma_c, c) = 0$$
However, when $T_{c,j}(\sigma_c) = \top$, we have no information on the value of $F_j(\sigma_c, c)$. This $k$-tree is defined using interval analysis techniques (branch and bound, see Jaulin et al. [2001]) and reachability analysis of program statements, using abstract interpretation for example. The complete algorithm for computing $T_{c,j}$ was given in Bouissou [2009].

We can now modify the sample hybrid automata $\mathcal{A}_{\mathcal{P}}$ generated from a hybrid program $\mathcal{P}$ (see Section 4.1). We define the abstract S-HA $\boldsymbol{\mathcal{A}_{\mathcal{P}}}$ which is the same as $\mathcal{A}_{\mathcal{P}}$, but in which:

- for each $c \in \mathbb{B}^m$, we have the invariant $I(c) = \bigwedge_{j\in[0,2p],c\in\mathbb{B}^m} T_{j,c}(V) \neq 1$;
- the guard of a transition between $c$ and $c'$ is now
$$g = \bigwedge_{j\in[1,2p]} \{T_{j,c}(V) \neq 0 : c'[i_j] \neq c[i_j],\ c'[i_j] = v_j\} .$$

Let us remark that, as $T_{j,c}$ may return $\top$, the automata $\boldsymbol{\mathcal{A}_{\mathcal{P}}}$ is non-deterministic as $T_{j,c}(\sigma_c)$ may return $\top$ for some $\sigma$, making both the invariant and a transition guard verified at the same time.

Let us now prove the correction of $\boldsymbol{\mathcal{A}_{\mathcal{P}}}$, stated by Proposition 19.

*Theorem 19.* Let $\mathcal{P}$ be a hybrid program and $\boldsymbol{\mathcal{A}_{\mathcal{P}}}$ be the abstract S-HA. Then, we have:
$$Reach_{\mathcal{P}} \subseteq Reach_{\boldsymbol{\mathcal{A}_{\mathcal{P}}}} .$$

**Proof.** This is easily proved as the abstract functions $T_{c,j}$ are abstraction of the functions $F_j$: for each state $\sigma_c \in \Sigma_c$ and $c \in \mathbb{B}^m$, we have $F_j(\sigma_c, c) \sqsubseteq T_{c,j}(\sigma_c)$, where $\sqsubseteq$ is the partial order in $\boldsymbol{B}$ defined by $0 \sqsubseteq \top$ and $1 \sqsubseteq \top$. The safety of $\boldsymbol{\mathcal{A}_{\mathcal{P}}}$ derives from it. □

Remark that once the $k$-trees $T_{c,j}$ are built, we can check very efficiently that $T_{c,j}(\sigma) \neq 0$ (for example), the complexity for it is $O\big(\log_2(n)\big)$, where $n$ is the number of continuous variables. So this abstraction techniques shifts the computation of $F_j(\sigma_c, c)$ to the construction of $T_{j,c}$.

## 5. CONCLUSION

In this article, we showed the equivalence between synchronous, sample based control-command programs and hybrid automata. To do this, we introduced a special kind

of hybrid automata, namely sampled hybrid automata, that are better suited for representing control-command programs. We then showed that the three models (hybrid automata, sampled hybrid automata and H-Simple programs) are equivalent in the following sense: there is a semantic preserving transformation from each to hybrid automata. In this way, we filled the gap between the high-level description of hybrid systems in hybrid automata and the low-level implementation of the control-command laws. We believe that this work can the be used to verify, using model-based techniques, safety properties on control-command programs.

Other worked proposed an extension of hybrid automata or imperative programs to handle better sampled based control-command programs. The I/O-Automata framework of Lynch et al. [1996] allows to better represent sensors and actuators, in particular it is possible to impose a sampling rate on sensors using a special encoding ( Fehnker et al. [2003]). However, we believe that this model is still too complex to fully embrace the "Read-Compute-Write" behavior of sample based control-command programs. In Briand and Jeannet [2010], the authors also want to encode a hybrid system as a discrete controller, written in Lustre, connected to a continuous environment. They then apply dynamic partitioning techniques to analyze the whole model. The main difference with our method is in the input language: they consider a Lustre program which is simpler than H-Simple programs (actually, we only presented basics imperative statements, but we can integrate very easily more complex constructs like pointers or arrays). Moreover, we believe that their dynamic partitioning technique can be used in collaboration with our $k$-tree abstraction to reduce the number of actuator configuration we must consider.

This should greatly reduce the complexity of our abstraction technique and thus help for the verification of the control-command program, and the use of more abstract-interpretation based techniques for the verification of the program are part of our future work. Another direction that we must investigate is the abstraction of the program: actually, for now, we use $k$-trees while hybrid systems verification tools as SpaceEx take as input linear hybrid automata, i.e. an automata where guards are linear inequalities between variables and flows are linear. In our setting guards are $k$-trees, they can be seen as the conjunction of many linear inequalities. This however is probably not very efficient, and we will look into building other abstractions of the program as a collection of polyhedral constraints instead of interval based constraints as in $k$-trees.

## REFERENCES


O. Bouissou. *Analyse statique par interpretation abstraite de systèmes hybrides.* PhD thesis, Ecole Polytechnique, 2008.

O. Bouissou. Proving the correctness of the implementation of a control-command algorithm. In Jens Palsberg and Zhendong Su, editors, *SAS*, volume 5673 of *Lecture Notes in Computer Science*, pages 102–119. Springer, 2009. ISBN 978-3-642-03236-3.

O. Bouissou and M. Martel. A hybrid denotational semantics of hybrid systems. In *Proceedings of the 17th European Symposium on Programming Languages and Systems (ESOP'08)*, volume 4960 of *Lecture Notes in Computer Science*, pages 63–77. Springer, 2008.

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

P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *VMCAI*, volume 3385 of *LNCS*, pages 1–24, 2005.

P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM Press, 1977.

P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

Ansgar Fehnker, Frits Vaandrager, and Miaomiao Zhang. Modeling and verifying a lego car using hybrid i/o automata. In *Proceedings of the Third International Conference on Quality Software*, QSIC '03, pages 280–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2015-4.

Goran Frehse, Colas Le Guernic, Alexandre Donzé, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. Spaceex: Scalable verification of hybrid systems. In *CAV*, 2011.

N. Halbwachs, Y. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In *SAS*, pages 223–237, 1994.

Matthew Hennessy. *Semantics of programming languages - an elementary introduction using structural operational semantics.* Wiley, 1990. ISBN 978-0-471-92772-3.

T. A. Henzinger. The theory of hybrid automata. In *Symposium on Logic in Computer Science*, pages 278–292. IEEE Computer Society Press, 1996.

L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied Interval Analysis.* Springer, 2001.

N. Lynch, R. Segala, F. Vaandrager, and H.B. Weinberg. Hybrid I/O automata. In *Proceedings of Hybrid systems III : verification and control*, pages 496–510. Springer, 1996.

Nadir Matringe, Arnaldo Vieira Moura, and Rachid Rebiha. Generating invariants for non-linear hybrid systems by linear algebraic methods. In *Proceedings of the 17th international conference on Static analysis*, SAS'10, pages 373–389, Berlin, Heidelberg, 2010. Springer-Verlag.

Olaf Müller and Thomas Stauner. Modelling and verification using linear hybrid automata – a case study. *Mathematical and Computer Modelling of Dynamical Systems*, 6(1):71–89, 2000.

Sriram Sankaranarayanan, Thao Dang, and Franjo Ivancic. Symbolic model checking of hybrid systems using template polyhedra. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2008. ISBN 978-3-540-78799-0.