Proving the Correctness of the Implementation of a Control-Command Algorithm

Olivier Bouissou

CEA LIST, Laboratory of Modelling and Analysis of Systems in Interaction,
Point Courrier 94, Gif-sur-Yvette, F-91191 France
Olivier.Bouissou@cea.fr

Abstract. In this article, we study the interactions between a control-command program and its physical environment via sensors and actuators. We are interested in finding invariants on the continuous trajectories of the physical values that the program is supposed to control. The invariants we are looking for are periodic sequences of intervals that are abstractions of the values read by the program. To compute them, we first build octrees that abstract the impact of the program on its environment. Then, we compute a period of the abstract periodic sequence and we finally define the values of this sequence as the fixpoint of a monotone map. We present a prototype analyzer that computes such invariants for C programs using a simple specification language for describing the continuous environment. It shows good results on classical benchmarks for hybrid systems verification.

1 Introduction.

The behavior of an embedded, control-command program depends on both a discrete system (the program) and a continuous system (the physical environment). The program constantly interacts with the environment, picking up physical values by means of sensors and modifying them via actuators. The goal of the program is usually to control its environment, i.e. it must ensure that some physical values remain stable by activating the right actuator at the right time. The correctness of a control-command program thus relies on two properties. First, we must prove that the program does not induce any bad behaviors, i.e. that no bugs were introduced by the developer. Formal methods, and in particular abstract interpretation techniques [8, 9], are widely used to prove this for some kinds of bugs (either run-time error [1], errors due to the precision of floating point computations [25] or execution time validation [12]). Then, we must prove that the program correctly controls its environment. This is usually done on high level models like Simulink, via numerical simulations. Sometimes, formal methods are used on equivalent models like hybrid automata [10, 17, 18], but this has not been done on the program itself. We leave the notion of "a program correctly controlling its environment" voluntarily vague as it may cover many properties. For example, one may want to prove the stability of the continuous trajectories under the influence of the program (which was mainly studied for switched dynamical systems [19, 29]), or one may try to prove that some region of the continuous state space is reached (which is the main goal of the existing analysis techniques on hybrid automata or equivalent models [18, 23]).

In this article, we focus on proving the correctness of the control-command program, without having a particular property in mind. Actually, we provide an abstract interpretation based method to compute invariants on the trajectories of the continuous environment that are sufficient to check the properties of interest. These invariants, that hold for every execution of the program, are defined in Section 1.2. They allow us to prove that the implementation choices (frequency of the sensors, format of floating point numbers, ...) do not modify the behavior of the control algorithm. We extend the work from [3], where we considered open loop programs, as we now take into account the action of the program on its environment. In the rest of this introduction, we present an example of the kind of programs that we consider (Section 1.1) and we recall the concrete model that we developed in [4] for such hybrid discrete-continuous systems (Section 1.2).

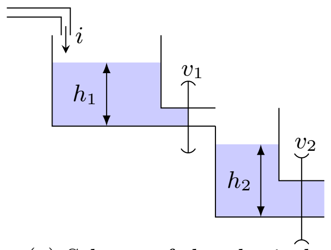
1.1 Introductory Example.

We consider the so-called "two tanks system", well known from the hybrid system community [21]. It consists of two water tanks linked together by an horizontal tube (see Figure 1(a)). There is a constant input of water in the first tank, and a tube at the bottom of the second tank lets the water leave the system. At each instant, a controller monitors the water levels in both tanks and must keep them between safe bounds. To do so, it may act on two valves (v_1 and v_2 on Figure 1(a)) that control the flow of water in the horizontal tubes. The evolution of the water levels is governed by the differential equations of Figure 1(b), where $v_i = 0$ if the valve i is closed, and $v_i = 1$ otherwise.

A controller for this system is a synchronous, control-command program: at each time stamp, it reads values from sensors (the water levels), computes an answer (open or close each valve) and writes this answer to actually modify the physical system. This read-compute-write loop is well cadenced as the program is synchronous. An execution of this system typically runs as follows: the initial state consists of initial water levels in both tanks and an initial mode (open or close) for each valve. After an initialization period in which the system reaches its stable state, we observe a cyclic behavior: the water level in each tank follows a periodic evolution while the decisions made by the program (open or close the valves) are the same from one period to the other. This kind of behavior (an initialization phase followed by a periodic evolution) is typical of control-command systems and it is what we try to automatically exhibit in this article.

1.2 Concrete Model.

We consider hybrid systems made of a pair (P, κ) where P is a program written in an extension of an imperative programming language and κ contains a set $\{F_c \,:\, c \in \mathbb{B}^m\}$ of ordinary differential equations (ODEs). Here $\mathbb{B} = \{0, 1\}$ is the concrete domain of boolean values and m is the number of (binary) actuators of



(a) Scheme of the physical system

$$\dot{h1} = i - v_1 \times k_1 \times \sqrt{h_1}$$
$$\dot{h2} = v_1 \times k_1 \times \sqrt{h_1} - v_2 \times k_2 \times \sqrt{h_2}$$

(b) The differential equations

```
int main() {
  double x,y;
  double l1=2.5,L1=8;
  double l2=2,L2=9;
  while(1) {
    sens.x?h1; sens.y?h2;
    if (h1 > L1)
      act.1!1;
    else if (h1 < l1)
      act.1!0;
    if (h2 > L2)
      act.2!1;
    else if (h2 < l2)
      act.2!0;
    wait(0.1);
}}
```

(c) The controller

Fig. 1. The two tanks system.

the system. Each $c \in \mathbb{B}^m$ is a continuous mode, one dimension of c indicates the state of one actuator of the system. For each $c \in \mathbb{B}^m$, the function F_c defines an ODE that can govern the way the continuous environment behaves: the program chooses between these continuous modes via the actuators. At each instant, the state of the system thus consists of a state of the program P, a value for each continuous variable and a continuous mode $c \in \mathbb{B}^m$.

The extension of imperative programming languages that is used for P contains standard imperative statements plus three hybrid statements. The *sens.x?y* statement lets the program read values from the environment via sensors: it binds the value of the discrete variable x with the current value of the continuous variable y. The *act.i!b* statement (with $i \in [1, m]$, m being the number of actuators of the system, and $b \in \mathbb{B}$) lets the program change the upcoming continuous mode: it sets the i^{th} dimension of the current continuous mode to b. Finally, the *wait* statement lets the program measure the continuous time.

The execution model is the following. 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 $v \in \mathbb{R}^n$ of the continuous variables[1] at the time t the statement is executed. We write it $v \xrightarrow{t} P$. 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. When the *act.i!b* statement is executed, the i^{th} dimension of the mode is changed to b, we write that: $P \xrightarrow{t} (i, b)$. 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 b. We write that $c' = c \oplus (i, b)$. If we apply various

[1] In this article, $\mathbb{R}$ (resp. $\mathbb{R}_+$) denotes the set of real (resp. non-negative real) numbers.

changes at once, we write $c \oplus \{(i_1, b_1), \ldots, (i_j, b_j)\}$. For example, if $c = [0; 1]$ and $c' = c \oplus \big\{(1, 1), (2, 0)\big\}$, then $c' = [1; 0]$. This corresponds to executing, at the same time stamp, the lines 8 and 14 of the program of Figure 1(c).

In this article, we are interested in the semantics of the continuous part of a hybrid system (P, κ), we will compute abstractions of it. We note $[\![P]\!]_c$ its continuous semantics: $[\![P]\!]_c$ is a function from $\mathbb{R}_+$ to $\mathbb{B}^m \times \mathbb{R}^n$ associating at each instant a continuous mode and a value such that:

$$\forall t \in \mathbb{R}_+,\ [\![P]\!]_c(t) = \big(c(t), y(t)\big) \text{ with } \dot{y}(t) = F_{c(t)}\big((y(t)\big)\ .$$

We only consider synchronous control-command programs that read values at the beginning of each cycle and act on its environment at the end of this cycle. If h is the duration of one cycle, we note $F_h : \mathbb{B}^m \times \mathbb{R}^n \to \mathbb{B}^m \times \mathbb{R}^n$ the concrete transition function defined as, with $(c, v) \in \mathbb{B}^m \times \mathbb{R}^n$:

$$F_h(c, v) = (c', v')\ :\ \exists t \in \mathbb{R}_+,\ [\![P]\!]_c(t) = (c, v) \text{ and } [\![P]\!]_c(t + h) = (c', v')\ . \qquad (1)$$

In other words, the concrete transition function maps the state of the continuous system at the beginning of a cycle with the state of system at the beginning of the next cycle. On such hybrid systems, we look for invariants that overapproximate the continuous semantics $[\![P]\!]_c$, as defined in Definition 1. In this definition, as in the rest of this article, $\mathbf{I}$ denotes the complete lattice of n-dimensional boxes.

Definition 1 (Trajectory invariants). *A* trajectory invariant *of a control-command system* (P, κ) *is function* $\phi : \mathbb{R}_+ \to \mathbf{I}$*, such that:*
$\forall t \in \mathbb{R}_+,\ [\![P]\!]_c(t) = \big(c(t), y(t)\big)$ *with* $y(t) \in \phi(t)$*.*
A discrete trajectory invariant *is a sequence* $\tilde{\phi} : \mathbb{N} \to \mathbf{I}$ *such that, if* h *is the duration of one cycle of* P*:* $\forall k \in \mathbb{N},\ [\![P]\!]_c(k \times h) = \big(c(k), y(k)\big)$ *with* $y(k) \in \tilde{\phi}(k)$*.*

The main contribution of this paper is to provide a method for computing discrete trajectory invariants for control-command program given in the formalism presented in this section. While existing analysis techniques for hybrid systems are specialized for proving a certain kind of property on high-level models, this work focuses on inferring the most general invariants on code-level models.

Example 1. Figure 1(c) shows a program P for the two-tanks system written in this language. The whole system is then $\big(P, \{F_{0,0}, F_{0,1}, F_{1,0}, F_{1,1}\}\big)$ where $F_{i,j}$ corresponds to the ODE of Figure 1(b) with $v_1 = i$ and $v_2 = j$. For example, line 8 in the program represents the action of opening the valve v_1.

Outline of the paper. In Section 2, we informally explain the main ideas of our analysis. In Section 3, we present the domains that we use and we study their properties. In Sections 4 and 5, we present our three-steps analysis. Section 6 presents our implementation while Section 7 presents concluding remarks.

2 Informal Presentation of the Analysis.

We thus consider a control-command program together with a specification of its environment encoded in the model presented in Section 1.2, and try to derive discrete trajectory invariants of it. Our analysis proceeds in three steps. First, we compute an abstraction of F_h. To do that, we use an abstraction of the program (see Section 4) as a set of boolean functions that answer the following questions: given an input v, will the j^{th} *act* statement be activated ? Then, we look for an overapproximation of the sequence of values read by the program. In order to finitely represent this infinite sequence, we try to exhibit an abstract periodic behavior, i.e. an ultimately periodic sequence of intervals that contains all the values read by the sensors. We first look for a period in this abstract sequence (see Section 5.2). Note that if there is no periodic behavior in the concrete sequence, we will obtain a period of 1 in the abstract sequence, meaning that we only bind the values by an interval. So, our analysis always terminates, even if there is no periodic evolution in the first place. On the other hand, if there is a periodic behavior in the system, we will exhibit it. Once we have the period for the abstract sequence, we compute the values of that sequence (see Section 5.3) as the fixpoint of a monotone map on a complete lattice.

In the rest of the article, n will denote the number of sensors, m the number of binary actuators and h the period of one cycle of the system, i.e. the program reads values at time $t_k = k \times h$, $\forall k \in \mathbb{N}$. We also note m_a the number of *act* statements in the program, and for each $j \in [1, m_a]$, we note $i_j \in [1, m]$ the dimension and $b_j \in \mathbb{B}$ the boolean value such that the j^{th} *act* statement is $act.i_j!b_j$. So, for each $j \in [1, m_a]$, if the j^{th} statement is executed at time t_k, we have $P \xrightarrow{t_k} (i_j, b_j)$. For example, if the second *act* statement in Figure 1(c) is executed, we have $P \xrightarrow{t_k} (1, 0)$. If this comes in response to an input $v \in \mathbb{R}^n$ (i.e. if at time t_k, the program also read v), we write: $\big(v \xrightarrow{t_k} P\big) \Rightarrow \big(P \xrightarrow{t_k} (i_j, b_j)\big)$.

In order to reduce the complexity of defining an abstraction of the transition function F_h, we will assume in the rest of this article that the program is *time-invariant*, i.e. the action it takes in response to a read value v does not depend on the time this value was read. In other words, if at some time t_k the program reads the value v and then activates an *act* statement, it will activate it again if the same value v is read at time $t_{k'} \neq t_k$. This is formally stated in Definition 2.

Definition 2 (Time-invariant program). *We say that a program P is* time-invariant *if it holds, for all $v \in \mathbb{R}^n$ and for all $j \in [1, m_a]$, that:*

$$\big(\exists k \in \mathbb{N} :\ (v \xrightarrow{t_k} P \Rightarrow P \xrightarrow{t_k} (i_j, b_j))\big) \Longrightarrow \big(\forall k' \in \mathbb{N} :\ (v \xrightarrow{t_{k'}} P \Rightarrow P \xrightarrow{t_{k'}} (i_j, b_j))\big) .$$

We recall that m_a is the number of act *statements in the program and that for $j \in [1, m_a]$, the j^{th} act statement is $act.i_j!b_j$.*

This assumption may be restrictive as it excludes, for example, a program that integrates its input values and activates the actuators based on this computation. However, many safety critical control-command programs are time-invariant.

3 Domains.

The abstract boolean lattice is $\mathbf{B} = \{\bot_{\mathbf{B}}, \mathbf{0}, \mathbf{1}, \top_{\mathbf{B}}\}$, with $\gamma_b : \mathbf{B} \to \mathcal{P}(\mathbb{B})$ being the canonical concretization. More generally, bold typed symbols represent abstract values while normal symbols represent concrete values. The symbol v (resp. $\mathbf{v}$) refers to the concrete (resp. abstract) state for the continuous variables, i.e. $v \in \mathbb{R}^n$ and $\mathbf{v} \in \mathbf{I}$. The symbol c (resp. $\mathbf{c}$) refers to the concrete (resp. abstract) continuous mode, i.e. $c \in \mathbb{B}^m$ and $\mathbf{c} \in \mathbf{B}^m$.

3.1 Abstract Continuous State.

The state of the continuous environment is given by a vector of real values (for the continuous variables) and the continuous mode that governs its evolution. Continuous variables are abstracted by boxes. Continuous modes are elements of $\mathbb{B}^m$, we abstract them by elements of $\mathbf{B}^m$ and define $\mathbf{M} = \mathbf{B}^m$. We note $\gamma_{\mathbf{M}} : \mathbf{M} \to \mathcal{P}(\mathbb{B}^m)$ the canonical concretization function. The set $\mathbf{M}$ is a complete lattice with order $\sqsubseteq_{\mathbf{M}}$, supremum $\top_{\mathbf{M}}$, infimum $\bot_{\mathbf{M}}$, join $\sqcup_{\mathbf{M}}$ and meet $\sqcap_{\mathbf{M}}$.

Definition 3 (Abstract continuous states). *The domain of abstract continuous states is* $\mathbf{S} = \mathbf{M} \times \mathbf{I}$. *The concretization function* $\gamma_{\mathbf{S}}$ *is defined componentwise and the order* $\sqsubseteq_{\mathbf{S}}$ *is defined as:*

$$\forall (\mathbf{c}, \mathbf{v}), (\mathbf{c}', \mathbf{v}') \in \mathbf{S}, (\mathbf{c}, \mathbf{v}) \sqsubseteq_{\mathbf{S}} (\mathbf{c}', \mathbf{v}') \Leftrightarrow \big((\mathbf{c} \sqsubseteq_{\mathbf{M}} \mathbf{c}') \wedge (\mathbf{v} \subseteq \mathbf{v}') \big) .$$

We also introduce a notion of distance between abstract states, based on the Hausdorff distance $d_H : \mathbf{I} \times \mathbf{I} \to \mathbb{R}_+$ defined by:

$$d_H(\mathbf{v}, \mathbf{v}') = \max \big(\max_{x \in \mathbf{v}} \big\{ \min_{y \in \mathbf{v}'} |x - y| \big\}, \max_{x \in \mathbf{v}'} \big\{ \min_{y \in \mathbf{v}} |x - y| \big\} \big) .$$

Definition 4 (Distance on continuous states). *The distance between two abstract states* $(\mathbf{c}, \mathbf{v})$ *and* $(\mathbf{c}', \mathbf{v}')$ *is:*

$$d_{\mathbf{S}}\big((\mathbf{c}, \mathbf{v}), (\mathbf{c}', \mathbf{v}')\big) = \begin{cases} \infty & \text{if } \mathbf{c} \not\sqsubseteq_{\mathbf{M}} \mathbf{c}' \text{ and } \mathbf{c}' \not\sqsubseteq_{\mathbf{M}} \mathbf{c} \\ d_H(\mathbf{v}, \mathbf{v}') & \text{otherwise} \end{cases} .$$

Intuitively, if two states $(\mathbf{c_1}, \mathbf{v_1})$ and $(\mathbf{c_2}, \mathbf{v_2})$ are close enough, then the possible evolutions of the dynamical system starting from two points within $\gamma_{\mathbf{S}}(\mathbf{c_1}, \mathbf{v_1})$ and $\gamma_{\mathbf{S}}(\mathbf{c_2}, \mathbf{v_2})$ remain close. This is the reason for the first condition in the definition of $d_{\mathbf{S}}$: if two states have incomparable modes, then their future evolution may be completely different, as different modes represent significantly different dynamics (for example, in Figure 1(b), the various modes represent the fact that the two valves are closed or open, leading to an increase or a decrease in the water levels). Their distance is thus set to ∞ to notice that they may lead to significantly different trajectories.

3.2 Cyclic Sequences of States.

We abstract infinite sequences of states by *cyclic sequences.* A cyclic sequence is a pair made of a *lasso shaped graph* and a function linking each vertex of the graph to an abstract continuous state. A lasso shaped graph (also used in [16] to prove non-termination of programs) is a finite, directed graph G composed of two linked subgraphs: a *stem* and a *loop*. The *size* of the graph G, noted $|G|$, is the number of vertices. The *loop size* (resp. *stem size*) of the graph is the number of vertices in the loop (resp. stem): we note it $|G|_l$ (resp. $|G|_s$). For a graph G, V_G denotes its vertices and E_g its edges.

Definition 5 (Cyclic sequences). *A cyclic sequence over the domain* $\mathbf{S}$ *is a pair* $\mathbf{s} = (G, f)$ *where* G *is a lasso shaped graph and* $f : V_G \to \mathbf{S}$ *maps vertices of* G *with abstract states. We note* $\mathbf{S}^{\circlearrowleft}$ *the set of all cyclic sequences over* $\mathbf{S}$.

Cyclic sequences are finite representations of infinite sequences of abstract states. Let $\mathbf{s} = (G, f) \in \mathbf{S}^{\circlearrowleft}$, with $V_G = \{w_0, \ldots, w_{p-1}\}$, and let i be the index of the first vertex within the loop: $i = |G|_s$. We note $\hat{\mathbf{s}} : \mathbb{N} \to \mathbf{S}$ the function defined by:

$$\forall k \in \mathbb{N},\ \hat{\mathbf{s}}(k) = \begin{cases} f(w_k) & \text{if } k < |G| \\ \bot_{\mathbf{S}} & \text{if } k \geq |G| \text{ and } |G|_l = 0 \\ f\big(w_{(p+(k-i \mod |G|_l)}\big) & \text{otherwise} \end{cases} .$$

Example 2. Figure 4 in Section 6 shows an example of a cyclic sequence: it is the invariant computed on the trajectories of the heater problem (see Section 6).

Definition 6 (Order on cyclic sequences). *The order* $\sqsubseteq^{\circlearrowleft}$ *is the point-wise extension of the order on abstract states, if the sequences have the same graph. Formally, we have, for* $\mathbf{s_1} = (G_1, f_1) \in \mathbf{S}^{\circlearrowleft}$ *and* $\mathbf{s_2} = (G_2, f_2) \in \mathbf{S}^{\circlearrowleft}$*:*

$$\mathbf{s_1} \sqsubseteq^{\circlearrowleft} \mathbf{s_2} \Leftrightarrow G_1 = G_2 \textit{ and } \forall w \in G_1,\ f_1(w) \sqsubseteq_{\mathbf{S}} f_2(w) .$$

Note that this definition implies, in particular, that $\forall k \in \mathbb{N},\ \hat{\mathbf{s}}_\mathbf{1}(k) \sqsubseteq_{\mathbf{S}} \hat{\mathbf{s}}_\mathbf{2}(k)$. To define the join of two cyclic sequences, we introduce a new element $\top^{\circlearrowleft} \in \mathbf{S}^{\circlearrowleft}$ such that $\forall \mathbf{s} \in \mathbf{S},\ \mathbf{s} \sqsubseteq^{\circlearrowleft} \top^{\circlearrowleft}$. The join of two cyclic sequences $\mathbf{s_1}, \mathbf{s_2} \in \mathbf{S}^{\circlearrowleft}$ is then defined as follows: if $\mathbf{s_1}$ and $\mathbf{s_2}$ share the same graph, we make the union of the states associated to each vertex, otherwise, we return $\top^{\circlearrowleft}$. Formally we have:

$$\forall \mathbf{s_1}, \mathbf{s_2} \in \mathbf{S}^{\circlearrowleft},\ \mathbf{s_1} \sqcup^{\circlearrowleft} \mathbf{s_2} = \begin{cases} \top^{\circlearrowleft} & \text{if } G_1 \neq G_2 \\ (G_1, f) : \forall w \in G_1, f(w) = f_1(w) \sqcup_{\mathbf{S}} f_2(w) & \text{otherwise} \end{cases} . \quad (2)$$

Let us remark that if $\mathbf{s} = \mathbf{s_1} \sqcup^{\circlearrowleft} \mathbf{s_2}$, $\mathbf{s} \neq \top^{\circlearrowleft}$, then $\forall k \in \mathbb{N},\ \hat{\mathbf{s}}(k) = \hat{\mathbf{s}}_\mathbf{1}(k) \sqcup_{\mathbf{S}} \hat{\mathbf{s}}_\mathbf{1}(k)$. We define in the same way the intersection of $\mathbf{s_1}$ and $\mathbf{s_2}$: if they share the same graph, we compute the meet of the values associated to each vertex. Otherwise, we set the meet to be the $\bot^{\circlearrowleft}$, a new element of $\mathbf{S}^{\circlearrowleft}$ such that $\forall \mathbf{s} \in \mathbf{S}^{\circlearrowleft},\ \bot^{\circlearrowleft} \sqsubseteq^{\circlearrowleft} \mathbf{s}$.

Property 1. The domain $\big(\mathbf{S}^{\circlearrowleft}, \sqsubseteq^{\circlearrowleft}, \sqcup^{\circlearrowleft}, \sqcap^{\circlearrowleft}\big)$ is a complete lattice.

Proof. As $\mathbf{S}$ is a complete lattice, so is the domain of functions $V_G \to \mathbf{S}$ for some lasso graph $G = (V_G, E_G)$. As two elements of $\mathbf{S}^{\circlearrowleft}$ are comparable if and only if they share the same graph, this proves that $\mathbf{S}^{\circlearrowleft}$ is a complete lattice. □

Definition 7 (Concretization $\gamma^{\circlearrowleft}$). *The concretization* $\gamma^{\circlearrowleft} : \mathbf{S}^{\circlearrowleft} \to \mathcal{P}(\mathbb{N} \to \mathbb{R}^n)$ *maps a cyclic sequence with infinitely many concrete sequences:*

$$\forall \mathbf{s} \in \mathbf{S}^{\circlearrowleft},\ \gamma^{\circlearrowleft}(\mathbf{s}) = \big\{ s : \mathbb{N} \to \mathbb{R}^n \ :\ \forall k \in \mathbb{N},\ \exists b \in \mathbb{B}^m, (b, s(k)) \in \gamma_{\mathbf{S}}(\hat{\mathbf{s}}(k)) \big\} .$$

Property 2. The function $\gamma^{\circlearrowleft}$ is monotone.

4 Abstraction of the Program.

We now explain how, with the assumption of Section 2, we compute an abstraction of the program that is sufficient for computing the abstraction of the iteration function. In this way, we split our analysis of the hybrid system in two steps: first (in this section) we focus on the discrete program, then (in Section 5) we use this program abstraction and we focus on the continuous environment.

4.1 Definition of a Program Abstraction.

To build the program abstraction, we see a program as a decision function that, given an input (the value read by the sensor), decides whether or not each actuator must be activated. In other words, we are only interested in the impact a program has on its environment and not in its intern evolution. For the j^{th} *act* statement in the program, we consider the boolean function $\Phi_j : \mathbb{R}^n \to \mathbb{B}$ defined by $\Phi_j(v) = 1$ if $\big((v \xrightarrow{t_k} P) \Rightarrow (P \xrightarrow{t_k} (i_j, b_j))\big)$, and $\Phi_j(v) = 0$ otherwise. In other words, Φ_j returns 1 if, when the program reads the value v, it activates the j^{th} *act* statement, and 0 otherwise. A program abstraction (see Definition 8) is a collection of boolean inclusion functions that safely abstract these functions.

Definition 8 (Program abstraction). *Let (P, κ) be a system of our concrete model with m_a binary actuators. An abstraction of the program consists of m_a inclusion boolean functions $[\mathbf{\Phi_1}], \ldots, [\mathbf{\Phi_{m_a}}] : \mathbf{I} \to \mathbf{B}$, such that:*

$$\forall \mathbf{v} \in \mathbf{I},\ j \in [1, m_a],\ \begin{cases} [\mathbf{\Phi_j}](\mathbf{v}) = 1 & \textit{if } \big(\forall v \in \mathbf{v},\ v \xrightarrow{t_k} P \Rightarrow P \xrightarrow{t_k} (i_j, b_j)\big) \\ [\mathbf{\Phi_j}](\mathbf{v}) = 0 & \textit{if } \big(\forall v \in \mathbf{v},\ v \xrightarrow{t_k} P \Rightarrow P \not\xrightarrow{t_k} (i_j, b_j)\big) \\ [\mathbf{\Phi_j}](\mathbf{v}) = \top_B & \textit{otherwise} \end{cases} .$$

With the assumption that the program is time-invariant (see Section 2), we can compute such program abstractions using standard reachability analysis for imperative programs. It suffices to check if, with an input $\mathbf{v} \in \mathbf{I}$, the program reaches the j^{th} *act* statement (for example, [6, 7] or even [5] can be used to do that). We can thus compute, for each *act* statement of the program, the function $[\mathbf{\Phi_j}]$. However, the computation of $[\mathbf{\Phi_j}](\mathbf{v})$ can be computationally expensive for large programs, and we will need to compute that often. Thus, we chose to partition, for each $j \in [1, m_a]$, the input state space into three regions $R^j_{\mathbf{0}}$, $R^j_{\mathbf{1}}$ and $R^j_{\top}$ in such a way that, if $\mathbf{v} \subseteq R^j_{\mathbf{b}}$, then $[\mathbf{\Phi_j}](\mathbf{v}) = \mathbf{b}$ for all $\mathbf{b} \in \mathbf{B}$. These regions serve as a program abstraction: to decide whether an actuator is activated, we just need to check in which region the input lies.

4.2 Representation as Octrees.

A n-dimension *octree*[2] is a directed, acyclic graph with a unique root and two kinds of nodes: non terminal nodes $N(\mathbf{v})$, with $\mathbf{v} \in \mathbf{I}$ and terminal nodes $V(\mathbf{b})$ with $\mathbf{b} \in \mathbf{B}$. Each non terminal node $N(\mathbf{v})$ has either one terminal child or 2^n non terminal children $N_i(\mathbf{v_i})$, $i \in [1, 2^n]$, with: $\forall i, j \in [1, 2^n]$, $\mathbf{v_i} \cap \mathbf{v_j} = \emptyset$ and $\bigsqcup_{i \in [1, 2^n]} \mathbf{v_i} = \mathbf{v}$. We note $\mathbf{T}$ the set of all n-dimension octrees. The root of an octree T must be a non terminal node $N(\mathbf{v})$, and we say that $\mathbf{v}$ is the domain of T, denoted $\mathbf{v} = dom(T)$. Figure 2 shows an example of a 2-dimension octree.

An octree $T \in \mathbf{T}$ represents a partition of the state space into the three regions $R_{\mathbf{0}}$, $R_{\mathbf{1}}$ and $R_{\top}$ mentioned before. Now, for an input $\mathbf{v} \in dom(T)$, we

[2] The term *octree* is generally used for 3-dimension trees only (i.e. trees with 8 children for each node). We here use it for the more general case of n-dimension trees.

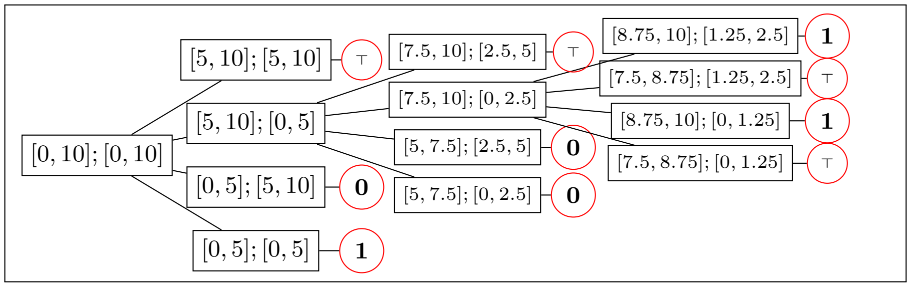


Fig. 2. A 2-dimension octree T with domain $[0, 10] \times [0, 10]$. It verifies $T\big([1, 3]; [2; 4]\big) = \mathbf{1}$ and $T\big([1, 3]; [4; 6]\big)\big) = \top_{\mathbf{B}}$.

check efficiently in which region it is included by computing $T(v)$ defined by $T(\mathbf{v}) = \bigsqcup\big\{\mathbf{b}\ :\ \exists\ N(\mathbf{v}') \in T,\ \mathbf{v} \cap \mathbf{v}' \neq \emptyset$ and $N(\mathbf{v}')$ has a child $V(\mathbf{b})\big\}$. The value $T(v)$ is thus the join of all $\mathbf{b} \in \mathbf{B}$ such that $\mathbf{v}$ intersects the region $R_{\mathbf{b}}$.

Property 3. For every octree T, the function $F_T : dom(T) \to \mathbf{B}$ defined by $\forall \mathbf{v} \in dom(T),\ F_T(\mathbf{v}) = T(\mathbf{v})$ is monotone.

Proof. Let $\mathbf{v_1},\ \mathbf{v_2} \in dom(T)$ with $\mathbf{v_1} \subseteq \mathbf{v_2}$. Let S be the set of non terminal nodes $N(\mathbf{v_k})$ of T with a child $V(\mathbf{b_k})$ such that $\mathbf{v_k} \cap \mathbf{v_1} \neq \emptyset$. Then, $T(\mathbf{v_1}) = \sqcup\{\mathbf{b_k}\ :\ N(\mathbf{v_k}) \in S\}$. Let now $N(\mathbf{v_k}) \in S$. As $\mathbf{v_1} \subseteq \mathbf{v_2}$, it holds that $\mathbf{v_k} \cap \mathbf{v_2} \neq \emptyset$, so that $\mathbf{b_k} \sqsubseteq_{\mathbf{B}} T(\mathbf{v_2})$. This implies that $T(\mathbf{v_1}) \sqsubseteq_{\mathbf{B}} T(\mathbf{v_2})$. □

We use Algorithm 1 to compute an octree that represents the regions $R_{\mathbf{0}}$, $R_{\mathbf{1}}$ and $R_\top$ for one *act* statement of the program. Theorem 1 shows that these octrees are sound program abstractions.

Theorem 1. *For every time-invariant program P with m_a act statement, and for every $j \in [1, m_a]$, the octree $T_j =$ `BuildOctree`(P, j) verifies that:*

$$\forall \mathbf{v} \subseteq dom(T_j),\ \forall v \in \mathbf{v},\ \Phi_j(v) \in \gamma_b\big(T_j(\mathbf{v})\big)\ .$$

We recall that Φ_j is the concrete function deciding whether the j^{th} act statement is activated given some input values, see Section 4.1.

Remark 1. The maximal depth of the octrees computed in Algorithm 1 is a time-precision trade-off: deeper octrees are more precise but the computation time is exponential in this maximal depth.

5 Abstraction of the Continuous Evolution.

In this section, we explain how, given an abstraction of the program as a set of octrees, we compute an abstraction of the evolution of the continuous environment as a cyclic sequence (see Section 3). This cyclic sequence is computed in two steps: first, we compute the lasso shaped graph and an initial guess for the values of the vertices, then we compute the values of each vertex of the graph. Both steps are based on an iteration of an *abstract transition function* that overapproximates the concrete transition function (see Section 1.2).

```
Input: $P$;                                    /* A time-invariant program */
Input: $\mathbf{D} \in \mathbf{I}$;                    /* Range for the input variables of $P$ */
Input: $j, N_{max} \in \mathbb{N}$;   /* The actuator to reach and the maximal depth */
Result: An octree $T$ of maximal depth $N_{max}$
start
    $T = N(\mathbf{D})$;               /* Initialize the octree and set its root */
    $\mathbf{b} = [\mathbf{\Phi_j}](\mathbf{D})$;
    if $\mathbf{b} = \top_{\mathbf{B}} \wedge N_{max} \neq 0$ then
        Find $\mathbf{v_1}, \dots, \mathbf{v_{2^n}}$ such that $\mathbf{D} = \mathbf{v_1} \sqcup \mathbf{v_2} \sqcup \dots \sqcup \mathbf{v_{2^n}}$;
        for every $\mathbf{v} \in \{\mathbf{v_1}, \dots, \mathbf{v_{2^n}}\}$ do
            $aux$ = BuildOctree($P, \mathbf{v}, j, N_{max} - 1$); Add $aux$ as a child of $T$;
        done
    else
        Add $V(\mathbf{b})$ as a child of $T$;
    endif
    return $T$;
end
```

Algorithm 1: `BuildOctree`(P,$\mathbf{D}$,j,N_{max}): construction of an octree of domain $\mathbf{D}$ and depth N_{max} that represents the function $[\mathbf{\Phi_j}]$.

5.1 The Abstract Transition Function.

For each continuous mode $c \in \mathbb{B}^m$, the environment evolves according to an ODE $\dot{y} = F_c(y)$ (see Section 1.2). So, if at some time t we have $[\![P]\!]_c(t) = (c, y_t)$ for $c \in \mathbb{B}^m$ and $y_t \in \mathbb{R}^n$, and if the program does not change the continuous mode between t and $t+h$, then we have $[\![P]\!]_c(t+h) = (c, y_\infty(h))$, i.e. $F_h(c, y_t) = (c, y_\infty(h))$, where y_∞ is the solution of the ODE $\dot{y} = F_c(y)$ with initial state $y_\infty(0) = y_t$. The theory of guaranteed integration [24] shows that it is possible to overapproximate y_∞. In particular, the library GRKLib [2] proposes a systematic way to compute a monotone function $[F_c] : \mathbf{I} \times \mathbb{R}_+ \to \mathbf{I}$ such that[3]:

$$\forall \mathbf{v} \in \mathbf{I},\ \forall t, h \in \mathbb{R}_+,\ y_\infty(t) \in \mathbf{v} \Rightarrow y_\infty(t+h) \in [F_c](\mathbf{v}, h)\ .$$

Let us now define the abstract transition function $\mathbf{F_h}$ that overapproximates the concrete transition function F_h defined Section 1.2.

Definition 9 (Abstract transition function). *Let (P, κ) be a system of our concrete model, with $\kappa = \{F_c\ :\ c \in \mathbb{B}^m\}$, and let m_a be the number of act statements in P. Let $\{T_i\ :\ i \in [1, m_a]\}$ be a program abstraction as defined in Section 4. The abstract transition function $\mathbf{F_h}$ is a function between abstract states, $\mathbf{F_h} : \mathbf{S} \to \mathbf{S}$, defined by $\forall (\mathbf{c}, \mathbf{v}) \in \mathbf{S},\ \mathbf{F_h}(\mathbf{c}, \mathbf{v}) = (\mathbf{c}', \mathbf{v}')$ with:*

$$\begin{cases} \mathbf{v}' = \bigsqcup_I \left\{ [F_c](\mathbf{v}, h)\ :\ c \in \gamma_m(\mathbf{c}) \right\} \\ \mathbf{c}' = \mathbf{c} \oplus \left\{ (i_j, b_j)\ :\ j \in [1, m_a] \wedge \mathbf{1} \sqsubseteq_{\mathbf{B}} T_j(\mathbf{v}') \right\} \end{cases} .$$

We recall (see Section 1.2) that for a continuous mode $c \in \mathbb{B}^m$, an integer $i \in [1, m]$ and a boolean value $b \in \mathbb{B}$, $c \oplus (i, b)$ is the continuous mode c where the i^{th} dimension is set to b. We use the same notation for abstract modes $\mathbf{c} \in \mathbf{M}$.

[3] Remark that there is no limitation on F_c: we consider linear and non-linear dynamics.

```
Input: $(G_0, f_0)$,  $\alpha$ ;      /* Initial sequence and distance threshold */
Input: $\mathbf{F_h}$ ;                           /* Abstract transition function */
Result: A cyclic sequence $(G, f)$
start
    $(G, f) = (G_0, f_0)$; $w = w_0$ ; // $w$ is the last added vertex
    while $|G|_l = 0$ do
        $\mathbf{s} = \mathbf{F_h}(f(v))$;
        if $\exists w' \in V_G \ : \ s \sqsubseteq_{\mathbf{S}} f(w')$ then
            $E_g = E_G \cup \{(w, w')\}$; // The graph now contains a loop
        else
            $d = \min\{d_{\mathbf{S}}\big(\mathbf{s}, f(w')\big) \ : \ w' \in V_g\}$;
            if $d > \alpha$ then
                $V_G = V_G \cup \{w^*\}$ ; // $w^*$ is a new, fresh state
                $f(w^*) = \mathbf{s}$; $E_g = E_G \cup \{(w, w^*)\}$; $w = w^*$;
            else
                Choose $w' \in V_G$ such that $d_{\mathbf{S}}(f(w'), \mathbf{s}) < \alpha$;
                $E_g = E_G \cup \{(w, w')\}$; // The graph now contains a loop
            endif
        endif
    done
    return $(G, f)$;
end
```

Algorithm 2: `BuildLassoGraph`: construction of the initial lasso graph.

The abstract transition function $\mathbf{F_h}$ thus transforms an abstract state $(\mathbf{c}, \mathbf{v}) \in \mathbf{S}$ as follows: we compute the effect of every possible continuous modes $c \in \gamma_m(\mathbf{c})$ on $\mathbf{v}$ using the guaranteed integration function $[F_c]$, and then compute the join of all these evolutions to form the new abstract continuous state $\mathbf{v'}$. Then, for each *act* statement (i.e. for each $j \in [1, m_a]$) we check if it is activated under the input $\mathbf{v'}$ by computing $T_j(\mathbf{v'})$. For all those statements that are potentially activated (i.e. such that $\mathbf{1} \sqsubseteq_{\mathbf{B}} T_j(\mathbf{v'})$), we compute their effect on the continuous mode $\mathbf{c}$ by computing $\mathbf{c} \oplus (i_j, b_j)$.

Property 4. The abstract continuous function $\mathbf{F_h}$ is monotone over the complete lattice of continuous states (its monotonicity relies on the monotonicity of all the functions $[F_c]$ and T_j for $c \in \mathbb{B}^m$ and $j \in [1, m_a]$, see Property 3).

We can now state our main theorem for this section that proves that the abstract transition function is an abstraction of the concrete transition function.

Theorem 2. *Let (P, κ) be a control-command system, and let $\mathbf{F_h}$ be the function given at Definition 9. For all $(\mathbf{c}, \mathbf{v}) \in \mathbf{S}$, it holds that $F_h \circ \gamma_{\mathbf{S}}(\mathbf{c}, \mathbf{v}) \subseteq \gamma_{\mathbf{S}} \circ \mathbf{F_h}(\mathbf{c}, \mathbf{v})$.*

5.2 Constructing the Graph.

The computation of the cyclic sequence that serves as a discrete trajectory invariant (see Definition 1) requires two steps: first, we need to compute the lasso-shaped graph, and then we need to compute the abstract values associated to each vertex. In this section, we focus on the first task. To build the graph, we use Algorithm 2 that we here explain. Starting from an initial cyclic sequence (G_0, f_0) whose graph contains only one vertex w with $f(w) = \mathbf{s_0}$, where $\mathbf{s_0}$ is the initial abstract continuous state, we iterate the following process. We first compute $\mathbf{s} = \mathbf{F_h}\big(f(v)\big)$, i.e. the abstract continuous state at the next time stamp. If we find a vertex w' in the graph such that $\mathbf{s} \sqsubseteq_{\mathbf{S}} f(w')$, then we make a loop between w and w' – this case means that the we already overapproximated the evolution of the environment up to $t = \infty$. If there is no such vertex w', then we compute the minimal distance between $\mathbf{s}$ and $f(w')$ for all vertices w'. If this distance is smaller than some threshold α, we make a loop between w and w' – this case means that we have computed the evolution of the continuous system from a state "close to" w. Otherwise, we add a new vertex to the graph and start over from this vertex.

Of course, this algorithm may not terminate or take a very long time before creating a loop. We enforce the termination by using a widening strategy on the threshold α: at each iteration (or every K iterations, where K is a predefined number), we increase α to capture more abstract periodic behaviors. The cyclic sequence we compute is always a safe abstraction of the beginning of the continuous evolution, as stated by Theorem 3 (a direct consequence of Theorem 2).

Theorem 3. *Let* $\mathbf{s} = (G, f)$ *be the cyclic sequence returned by the algorithm* `BuildLassoGraph`. *Then for all* $k \in [0, |G|-1]$, *we have* $[\![P]\!]_c(k \times h) \in \gamma_{\mathbf{S}}\big(\hat{\mathbf{s}}(k)\big)$.

Remark 2. At this point, we only have an overapproximation of $[\![P]\!]_c(k \times h)$ for all $k < |G|$ as, when we created the loop, we may have changed the value of one vertex without propagating it to its successors. This is done in Section 5.3.

Remark 3. Parameter α in Algorithm 2 is a time-precision trade-off: the larger α, the faster the algorithm terminates, and the larger the overapproximation is.

5.3 Computing the Values.

We now explain how, starting from the initial cyclic sequence $\mathbf{s_o}$ computed by the Algorithm 2, we compute an overapproximation of the continuous evolution up to time ∞. To do so, we see the graph of the cyclic sequence as a control flow graph as used for the static analysis of programs. The transition function between two vertices of the graph is the abstract transition function $\mathbf{F_h}$. Thus, we define the function $\mathbf{F_S} : \mathbf{S}^{\circlearrowleft} \to \mathbf{S}^{\circlearrowleft}$ that updates the cyclic sequence as:

$$\mathbf{F_S} : \begin{cases} \quad \mathbf{S}^{\circlearrowleft} \to \mathbf{S}^{\circlearrowleft} \\ (G, f) \mapsto (G, f') \text{ with } \forall w \in V_G,\ f'(w) = \bigsqcup_{(w', w) \in E_G} \mathbf{F_h}(f(w')) \end{cases} .$$

Property 5. The function $\mathbf{F_S}$ is monotone over the complete lattice $\mathbf{S}^{\circlearrowleft}$.

As $\mathbf{F_S}$ is a monotone function over a complete lattice, it has a least fixpoint greater than $\mathbf{s_0}$ (the cyclic sequence computed by Algorithm 2). We can compute it using Kleene's iteration. Theorem 4 shows that this fixpoint is an overapproximation of the concrete sequence of values read by the program.

Theorem 4. *Let* $\mathbf{s} \in \mathbf{S}^{\circlearrowleft}$ *be the least fixpoint of* $\mathbf{F_S}$ *greater than* $\mathbf{s_0}$*, and let* $[\![P]\!]_c(t) = (c(t), y(t))$ *for all* $t \in \mathbb{R}_+$*. Then, the sequence* $\big(y(k \times h)\big)_{k\in\mathbb{N}}$ *verifies* $\big(y(k \times h)\big)_{k\in\mathbb{N}} \in \gamma^{\circlearrowleft}(\mathbf{s})$*, i.e. we have:* $\forall k \in \mathbb{N},\ [\![P]\!]_c(k \times h) \in \gamma_{\mathbf{S}}\big(\hat{\mathbf{s}}(k)\big)$.

Theorem 4 states that the computed cyclic sequence is a discrete trajectory invariant of the program P, as defined in Definition 1. It may thus be used to prove more complex properties like stability or reachability.

Remark 4. To accelerate Kleene's iteration for computing the fixpoint of $\mathbf{F_S}$, we use standard widening techniques on abstract continuous states. The widening naturally occurs on the first vertex of the loop of the cyclic sequence.

5.4 Discussion about the Method.

During the analysis, several operations may lead to a coarse approximation of the continuous trajectory. It is particularly the case if, during the iterations of $\mathbf{F_h}$, we obtain a state with the abstract mode $\top_{\mathbf{M}}$ (or $\top_{\mathbf{B}}$ in some direction). Then, as we don't know the continuous mode of the future evolution, we need to follow several modes. In the case of a control-command program, the different modes correspond to significantly different dynamics, so that following several of them lead to a huge overapproximation. There are various reasons for obtaining an imprecise abstract mode, we now explain them and we give some solutions on how to avoid them. Let thus $(\mathbf{c}, \mathbf{v}) \in \mathbf{S}$ be such that $\mathbf{F_h}(\mathbf{c}, \mathbf{v}) = (\mathbf{c}', \mathbf{v}')$, where $\mathbf{c}'$ contains $\top_{\mathbb{B}}$ in at least one dimension.

Problems due to the program abstraction. We may obtain $\top_{\mathbb{B}}$ because the box $\mathbf{v}'$ lies within the region $R^j_{\top}$ for some $j \in [1, m_a]$. This means that the program abstraction is too coarse. We can then locally increase the precision of the program abstractions by increasing the maximal depth we allow for the octrees. In this way, we will eventually enter the $R^j_{\mathbf{1}}$ or $R^j_{\mathbf{2}}$ regions. However, we might also enter both, which gives us another issue: if we have $\mathbf{v}' \cap R^j_{\mathbf{1}} \neq \emptyset$ and $\mathbf{v}' \cap R^j_{\mathbf{0}} \neq \emptyset$, then $T_j(v) = \top_{\mathbb{B}}$, so $\mathbf{c}'$ contains $\top_{\mathbb{B}}$ in the j^{th} coordinate. To reduce the loss of precision in this case, we can follow separately the possible dynamics, and try to collect them later. This is very similar to the disjunctive analysis used in the static analysis of programs [27]. We use our notion of distance for joining the different paths. Of course, this technique may lead to a combinatorial explosion in the number of states of the system, so in practice we limit the number of steps we make within each dynamics.

Problems due to the guaranteed integration. Another source of imprecision comes from the overapproximation of the continuous dynamics due to the guaranteed integration, that may compute a too large box $\mathbf{v}'$. To reduce it, a solution consists in using local subdivisions: we split $\mathbf{v}$ into smaller boxes, compute $\mathbf{F_h}$ on each of them and then join the results. This is not guaranteed to reduce the imprecision, but it has good results when the continuous dynamics is stiff.

Some remarks on our notion of distance. The distance $d_{\mathbf{S}}$ indicates how close the evolutions of the system starting from two abstract states will be. Of course, it is difficult to know if two states will remain close with only the information of their distance at some instant t_k. If the states are close to a region where the mode must change, their evolutions might diverge, and we should not merge them. Another heuristic considering more than one time stamp could detect such situations. However, on contracting systems (i.e. systems in which the distance between two trajectories decreases with time), our method has good results.

Comparison with other techniques. Finally, we would like to make an informal comparison between our method for computing invariants on control-command systems and the existing analysis techniques for hybrid systems [13, 15, 18, 26]. First, we consider linear and non-linear dynamics, which is almost never the case for the reachability analysis on hybrid automata. We also consider complicated discrete dynamics, as the only limitation on the program is that it must be time-invariant. Thus, the regions where there is a mode change may be very complicated: for example, in some control-command programs we analyzed, the decision for activating the actuator depends on the result of a Runge-Kutta integration predicting the value of the continuous variables one second later. The regions represented by the octree were then non convex and very irregular. Modeling such systems (frequent in industrial systems) with hybrid automata would be a very difficult task in its own. Finally, as we consider code-level models, we can focus on computing discrete trajectory invariants as the mode changes only occur at time t_k for some $k \in \mathbb{N}$, while analysis techniques on high-level models need to enclose the trajectories for all $t \in \mathbb{R}_+$ to detect the mode changes. Our task is thus simplified because we consider more realistic models. Let us remark that this approach is also used in [28] where the authors start from the very general model of hybrid automata and impose a periodicity condition for the control actions. We, at the opposite, start from a new model which is especially designed for such perdiodically controlled hybrid systems.

6 Experimentation

We implemented the techniques presented in this article in a prototype analyzer named HyPrA (for Hybrid Programs Analyzer). HyPrA is based on Newspeak [20] for parsing the C files and it uses a specification language for specifying the actuators, sensors and continuous modes of the system. The specifications are special C comments understood by Newspeak, so our framework can be easily integrated into a development cycle. The user may provide plugins that compute the overapproximations of the continuous dynamics, or HyPrA may build them automatically using the specifications and an OCAML version of GRKLib.

We tested our analyzer on various benchmarks from the hybrid systems literature: the two-tanks system, the heater and the navigation problems [11]. In the heater problem, the dynamics are strongly contracting thus making our tool converge quickly. The navigation problem shows that HyPrA can be used to prove reachability of some region: to do that, we associated to this region a special continuous mode where all derivatives are equal to 0: if the trajectory enters this mode, it stops here, and we immediately obtain a loop of size 1 and a fixpoint.

Problem	Initial state	Width in the loop	Graph size/ Loop size	Computation time (s)
Heater	$[0, 5]$	0.19	11/3	0.071
Navigation	$[3.5, 3.6] \times [3.5, 3.6]$	0.2	16/1	5.456
Two-tanks	$[5, 5] \times [6, 6]$	0.8	96/96	32.9

Fig. 3. Results of HyPrA on classical hybrid systems benchmarks.

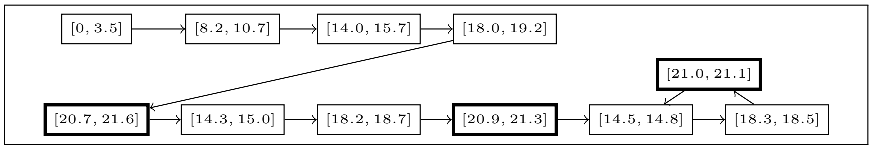


Fig. 4. Result of HyPrA on the heater problem. The figure shows the computed invariant as a cyclic sequence. In the bolded vertices, the continuous mode is **0**, meaning the heater is off. In the normal vertices, the continuous mode is **1**, meaning the heater is on. The labels represent the value of the continuous variable associated to the vertices of the graph.

Finally the two-tanks problem shows that our tool can deal with non-linear dynamics. The table on Figure 3 shows for each problem the chosen initial state, the size of the computed cyclic sequence, the size of its loop and the computation time. Figure 4 shows the cyclic sequence computed for the heater problem. We also indicate in the table the width of the abstract state of the first vertex within the loop (column "Width in the loop"). Actually, it should be noted that when we find such a loop, not only do we prove that the trajectories starting from the initial state remain within the loop, but also that any trajectory starting from a point within a vertex of the loop remains in the loop. For example, in the two tanks system, we chose a point as initial state, but the width of the first loop vertex is large, thus proving the correctness of the control-command program for a set of initial configurations.

7 Conclusion and Future Works.

In this article, we presented a method for the static analysis, using the abstraction interpretation framework, of the physical environment of embedded control-command programs. Using the fact that the program is time-invariant, we transform the control-command system into a system of *switched ordinary differential equations* [22] where the switching function is given by a program abstraction. Then, we look for a discretized overapproximation of the solution of this system as a ultimately periodic sequence of intervals. We use a notion of distance between abstract states to detect the periodic behavior. Our implementation shows that our method is efficient, in particular for non-linear dynamics: we can derive precise invariants on such systems.

Our analysis relies on the domain of intervals: we use them as the abstraction of the physical environment and we use interval analysis techniques for computing the octrees that serve as a program abstraction. Of course, using relational domains can improve the precision of the analysis: we plan to investigate the guaranteed integration of ODEs using zonotopes. They are already used for the verification of hybrid systems [15] and the static analysis of numerical programs [14], we believe they are well suited for our framework. We will also investigate methods to improve the precision of our analysis. For now, we did not implement any of the ideas mentioned in Section 5.4. The disjunctive analysis is probably the best way to improve the precision, and we can control the combinatorial explosion inherent by choosing a large threshold on the minimum distance for joining two states. Finally, removing the time invariant assumption will be a more complex task: defining a safe program abstraction is more complex as the action of the program on its environment depends on all the sensed values. To do so, we would probably need to analyze both the programs and the environment at the same time, and not one after the other as it is the case now.

References

1. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI'03*, pages 196–207. ACM, 2003.
2. O. Bouissou and M. Martel. GRKLib: a guaranteed runge-kutta library. In *Follow-up of International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*. IEEE Press, 2007.
3. O. Bouissou and M. Martel. Abstract interpretation of the physical inputs of embedded programs. In *VMCAI'08*, volume 4905 of *LNCS*, pages 37–51. Springer, 2008.
4. O. Bouissou and M. Martel. A hybrid denotational semantics of hybrid systems. In *ESOP'08*, volume 4960 of *LNCS*, pages 63–77. Springer, 2008.
5. Y. Chen, E. Gansner, and E. Koutsofios. A C++ data model supporting reachability analysis and dead code detection. In *ESEC/FSE'97*, volume 1301 of *LNCS*, pages 414–431. Springer, 1997.
6. B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Vardi. Proving that programs eventually do something good. *SIGPLAN Notices*, 42(1):265–276, 2007.
7. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI'06*, pages 415–426. ACM, 2006.
8. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, pages 238–252. ACM Press, 1977.
9. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
10. I.R. de Oliveira and P.S. Cugnasca. Checking safe trajectories of aircraft using hybrid automata. In *SAFECOMPK'02*, pages 224–235. Springer-Verlag, 2002.
11. A. Fehnker and F. Ivancic. Benchmarks for hybrid systems verification. In *HSCC'04*, volume 2993 of *LNCS*, pages 326–341. Springer Verlag, 2004.
12. C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *EMSOFT'01*, volume 2211 of *LNCS*, pages 469–485. Springer, 2001.

13. G. Frehse. Phaver: Algorithmic verification of hybrid systems past hytech. In *HSCC'05*, volume 3414 of *Lecture Notes in Computer Science*, pages 258–273. Springer, 2005.
14. E. Goubault and S. Putot. Static analysis of numerical algorithms. In *SAS'06*, volume 4134 of *LNCS*, pages 18–34. Springer, 2006.
15. C. Le Guernic and A. Girard. Zonotope-hyperplane intersection for hybrid systems reachability analysis. In *HSCC'08*, volume 4981 of *LNCS*, pages 215–228. Springer, 2008.
16. A. Gupta, T. Henzinger, R. Majumdar, A. Rybalchenko, and R. Xu. Proving non-termination. In *POPL'08*, pages 147–158. ACM Press, 2008.
17. T. A. Henzinger. The theory of hybrid automata. In *Symposium on Logic in Computer Science*, pages 278–292. IEEE Computer Society Press, 1996.
18. T. A. Henzinger and V. Rusu. Reachability verification for hybrid automata. In *HSCC'98*, volume 1386 of *LNCS*, pages 190–204. Springer-Verlag, 1998.
19. J. Hespanha. Uniform stability of switched linear systems: Extensions of LaSalle's invariance principle. *IEEETAC*, 49(4):470–482, 2004.
20. C. Hymans and O. Levillain. Newspeak, Doubleplussimple Minilang for Goodthinkful Static Analysis of C. Technical Note 2008-IW-SE-00010-1, EADS IW/SE, 2008.
21. S. Kowalewski, O. Stursberg, M. Fritz, H. Graf, I. H., J. Preuß, and et al. A case study in tool-aided analysis of discretely controlled continuous systems: the two tanks problem. In *Hybrid Systems V*, volume 1567 of *LNCS*. Springer, 1999.
22. D. Liberzon. *Switching in Systems and Control.* Birkhäuser, Boston, MA, 2003.
23. I. Ben Makhlouf and S. Kowalewski. An evaluation of two recent reachability analysis tools for hybrid systems. In *Second IFAC Conference on Analysis and Design of Hybrid Systems*, pages 377–382. ELSEVIER, 2006.
24. N.S. Nedialkov, K.R. Jackson, and G.F. Corliss. Validated solutions of initial value problems for ordinary differential equations. *Applied Mathematics and Computation*, 105(1):21–68, 1999.
25. S. Putot, E. Goubault, and M. Martel. Static analysis-based validation of floating-point computations. In *Numerical Software with Result Verification*, volume 2991 of *LNCS*, pages 306–313. Springer, 2003.
26. Nacim Ramdani, Nacim Meslem, and Yves Candau. Reachability of uncertain nonlinear systems using a nonlinear hybridization. In *HSCC'08*, volume 4981 of *LNCS*, pages 415–428, 2008.
27. S. Sankaranarayanan, F. Ivančić, I. Shlyakhter, and A. Gupta. Static analysis in disjunctive numerical domains. In *SAS'06*, volume 4134 of *LNCS*, pages 3–17, 2006.
28. Tichakorn Wongpiromsarn, Sayan Mitra, Richard M. Murray, and Andrew G. Lamperski. Periodically controlled hybrid systems. In *HSCC'09*, volume 5469 of *LNCS*, pages 396–410. Springer, 2009.
29. C. Yfoulis and R. Shorten. A numerical technique for stability analysis of linear switched systems. In *HSCC'04*, volume 2993 of *LNCS*, pages 631–645, 2004.