Abstract Interpretation of the Physical Inputs of Embedded Programs

Olivier Bouissou and Matthieu Martel

[1] CEA LIST
Laboratoire MeASI
F-91191 Gif-sur-Yvette Cedex, France
Olivier.Bouissou@cea.fr
[2] ELIAU-DALI Laboratory
Université de Perpignan Via Domitia
66860 Perpignan Cedex
Matthieu.Martel@univ-perp.fr

Abstract. We define an abstraction of the continuous variables that serve as inputs to embedded software. In existing static analyzers, these variables are most often abstracted by a constant interval, and this approach has shown its limits. We propose a different method that analyzes in a more precise way the continuous environment. This environment is first expressed as the semantics of a special continuous program, and we define a safe abstract semantics. We introduce the abstract domain of interval valued step functions and show that it safely over-approximates the set of continuous functions. The theory of guaranteed integration is then used to effectively compute an abstract semantics and we prove that this abstract semantics is safe. An example inspired by well known difficult problems shows the interest of our approach.

1 Introduction

The behavior of an embedded system depends on both a discrete system (the program) and a continuous system (the physical environment). The program continuously interacts with the environment, picking up physical values by means of sensors and modifying them via actuators. Thus, static analyzers for critical embedded software [4, 15] usually face the discrete part of a wider, hybrid system [19] but they often poorly abstract the physical environment in which, in practice, the embedded systems are run. To take an extreme example (more reasonable examples abound in articles dedicated to hybrid systems [1, 26]), the static analysis of avionic codes should abstract the plane environment, that is, the engines, the wings, and the atmosphere itself.

In practice, the sensors correspond to volatile variables in C programs and, at analysis time, the user must assign to these variables a range given by the minimal and maximal values the sensor can send. In this case, a static analyzer assumes that the value sent by the sensor may switch from its minimum to its maximum in an arbitrary short laps of time, while, in practice, it follows a continuous evolution. As a consequence, the results of the static analysis are significantly over-approximated, and our experience with Fluctuat [15] has shown that this naive abstraction of the continuous variables is the main source of loss of precision. The abstraction of the physical environment is even more crucial for embedded systems that cannot be physically tested in their real environment, like space crafts whose safety only relies on verification tools.

In this article, we consider a special case of hybrid systems: the continuous environment serves as input for the discrete system which is represented by the embedded program. We present an analysis of the continuous part of the system using the abstract interpretation framework [8, 9]. Our analysis permits a better over-approximation of the continuous variables than an abstraction with intervals and can be seen as the first step in the process of introducing hybrid components to existing static analyzers. A first approach to the abstraction of a continuous function could be done as follows: first partition the time line into (not necessarily regular) steps and then chose for each step an over- and under-approximation of the function on this step. This technique defines a family of Galois connections between various domains (one for each choice of the partitioning) [25], but is not compatible with efficient implementations. We indeed compute the over- and under-approximations using validated ODE solvers (see Section 4); modern algorithms [6, 27] dynamically change the step size, and thus dynamically partition the time line, in order to reach a user defined precision. We must thus consider an abstract domain for which the step sizes are not statically defined (see Section 3). This article is focused on the definition and correction of the abstract domain and we omit for the sake of conciseness the purely numerical aspects of the computation. For more details on Section 5, see [6]. In addition, a guaranteed extrapolation algorithm that safely bounds a function on an interval $[t, \infty[$ would be necessary to find a non naive widening. These purely numerical aspects are out of the scope of this article.

In Section 2, we describe the continuous environment as the collecting semantics of a continuous program, described by an interval valued ODE. As most collecting semantics of usual programs, this semantics is neither representable in machine nor computable, so we present in Section 3 an abstract domain which can be effectively used for the over-approximation of continuous functions. We also give criteria in order to build a safe abstraction of elements of the concrete domain. In Section 4, we show that guaranteed integration algorithms compute an abstract semantics of the ODE and we prove that this semantics is a safe over-approximation of the collecting one in Section 5. Finally, we show (Section 6) using a basic example that our approach gives better results than a naive abstraction of the environment using intervals.

To our knowledge, this is the first formalism that allows for the integration of the continuous environment in an abstract interpretation of embedded software. Edalat et al. defined a domain theoretic characterization of continuous functions [12, 14, 13] and showed that the solution of ODEs can be obtained by successive approximations. Their work is located at the concrete level, as they describe the continuous functions, and it does not provide an abstraction in the sense of the abstract interpretation theory, which is the main result of the present article. On the other hand, the analysis of non-linear hybrid automata using guaranteed ODE solvers was implemented in HyperTech [21], but the continuous dynamics is not defined by its own, which is necessary for the analysis of embedded software where the discrete and the continuous subsystems are clearly disjoint. Previous works on abstract interpretation strategies for hybrid systems mainly involved the analysis of hybrid automata [18, 20].

1.1 Notation

The set of real numbers is $\mathbb{R}$, while the set of non negative real numbers is $\mathbb{R}_+$. The set of natural numbers is $\mathbb{N}$. We will also consider the set of floating point numbers $\mathbb{F}$ [28]. The domain of continuous functions defined on $\mathbb{R}_+$ with values in $\mathbb{R}$ is $\mathcal{C}^0_+$ and the set of differentiable functions from $\mathbb{R}_+$ to $\mathbb{R}$ is $\mathcal{C}^1_+$. For a function $f \in \mathcal{C}^1_+$, we note $\dot{f} \in \mathcal{C}^0_+$ its first derivative. We use bold symbols to represent intervals: given a domain $\mathcal{D}$ with an order $\leq_\mathcal{D}$, the set of *intervals* on $\mathcal{D}$ is $\boldsymbol{D}$. Elements of $\boldsymbol{D}$ are denoted $\boldsymbol{x}$; for an interval $\boldsymbol{x} \in \boldsymbol{D}$, we note its lower bound $\underline{\boldsymbol{x}}$ and its upper bound $\overline{\boldsymbol{x}}$, such that $\boldsymbol{x} = \{x \in \mathcal{D} \mid \underline{\boldsymbol{x}} \leq_\mathcal{D} x \leq_\mathcal{D} \overline{\boldsymbol{x}}\}$. In particular, we will consider the set of intervals on real numbers $\boldsymbol{R}$, intervals on positive real numbers $\boldsymbol{R}_+$ and intervals on floating point numbers $\boldsymbol{F}$. Finally, we use arrows to represent vectors: given a domain $\mathcal{D}$, the set of vectors of dimension n is $\mathcal{D}^n$ and elements of $\mathcal{D}^n$ are denoted $\overrightarrow{x}$. Vectors of intervals are denoted $\overrightarrow{\boldsymbol{x}}$.

2 Syntax and Semantics of Continuous Processes

Hybrid systems are composed of two intrinsically different processes that run in parallel: a discrete program and a continuous environment. In order to reason about and analyze the whole system, one needs to find a unified representation of both parts. As for computer programs, we define a syntax, a collecting and an abstract semantics of the continuous environment. This section is dedicated to the definition of the concrete part.

2.1 Syntax

The environment represents physical quantities such as the temperature of the air, the speed of the wind or the deceleration of a car. Such quantities evolve continuously with time (i.e. their value cannot instantaneously jump from a to b), and thus follow a function from $\mathcal{C}^0_+$. Most often, this function is not explicitly known, but is defined as the solution of an *ordinary differential equation* (ODE). An ODE is a relation between a function $y \in \mathcal{C}^1_+$ and its first derivative $\dot{y}$ via a continuous function F: $\dot{y} = F\big(y; \overrightarrow{p}\big)$. $\overrightarrow{p}$ is a set of constant parameters (e.g. the gravitational constant, the length of the plane, ...). This representation as an autonomous ODE of order 1 (i.e. F only depends on the spatial value of y, and not on the time t) is expressive enough to capture other forms of ODE (non-autonomous and higher order ODEs are easily converted into higher dimensional autonomous ODEs of order one). An ODE links the value of the system at time $t + dt$ with the value of the system at time t, which is the continuous equivalent to any discrete dynamical system. It consequently forms the *syntax* of the continuous process. In order to achieve more expressiveness, we allow the parameters of the function F to be intervals, leading to the notion of *interval ODE.*

Definition 1. *Interval ODE*
Let F be a continuous function with a set of parameters $\overrightarrow{p} \in \overrightarrow{\mathbb{R}}$. An interval ordinary differential equation *(interval ODE) is given by the relation: $\dot{y} = F\big(y; \overrightarrow{\boldsymbol{p}}\big), \quad \overrightarrow{\boldsymbol{p}} \subseteq \overrightarrow{\boldsymbol{R}}$.*

This formalism is expressive enough to capture most dynamical systems and the introduction of interval parameters makes it suitable to express uncertainties on the system. We extend the uncertainty to the initial conditions of the ODE, and define the notion of interval initial value problems.

Definition 2. *Interval IVP.*
Let F be a continuous function with a set of parameters $\overrightarrow{p}$. An interval initial value problem *is given by an interval ODE and an interval initial condition:*

$$\dot{y} = F(y, \overrightarrow{\boldsymbol{p}}) \quad y(0) \in \boldsymbol{y_0} \tag{1}$$

An interval IVP gives a complete characterizations of a set of continuous environments using only three terms: a continuous function, a set of parameters and an initial interval value. We will thus write the physical environment `P:=(F,p,y)`, where `F` is the function, `p` its parameters and `y` the initial value. Example 1 shows how this compact notation is used to define a set of functions.

Example 1. The continuous process `P=(F,p,y)` with `F(y)=p*y`, `p=[-2,-1]` and `y=[0.5,3]` corresponds to the IV $\dot{y} = p.y,\ y(0) \in [0.5, 3],\ p \in [-2, -1]$. It defines the functions $y(x) = q.e^{p.x}$ with $q \in [0.5, 3]$, $p \in [-2, -1]$ (see Figure 1).

2.2 Collecting Semantics

Just like the collecting semantics of a discrete program is the set of all the (discrete) execution traces corresponding to a set of input parameters, the collecting semantics of the continuous process `(F,p,y)` is the solution of the corresponding interval IVP, that is, the set of all possible dynamics (i.e. continuous traces) of the system. The solution of a (real valued) ODE $\dot{y} = F\big(y; \overrightarrow{p}\big)$ is a function $y \in \mathcal{C}^1_+$ such that for every time t, it holds that $\dot{y}(t) = F\big(y(t); \overrightarrow{p}\big)$. The solution of a real valued initial value problem is a solution of the ODE that additionally verifies the initial condition. The existence and/or uniqueness of this solution depends mainly on the function F, and this question is not relevant for this article. On the contrary, we will always assume that F is smooth enough so that there exists a solution y defined on $\mathbb{R}_+$ for any initial condition and any parameter. The notion of solution to an IVP is then extended to interval IVP:

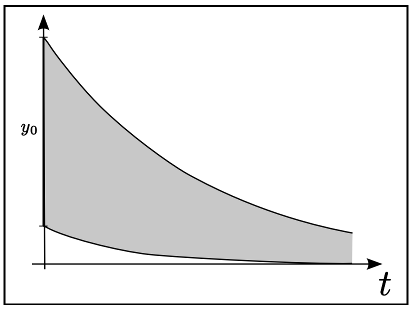


Fig. 1. Solutions of the interval ODE of Example 1.

Definition 3. *Solution of an interval IVP.*
The solution of the interval IVP (1) *is a set of functions* $\mathcal{Y} \subseteq \mathcal{C}^1_+$ *such that* $y \in \mathcal{Y}$ *if and only if there exists* $p \in \boldsymbol{p}$ *and* $y_0 \in \boldsymbol{y_0}$ *such that* y *is a solution of the (real valued) IVP* $\dot{y} = F(y,p),\ y(0) = y_0$.

The semantics $[\![\mathtt{P}]\!]$ of `P=(F,p,y)` is the solution of the interval initial value problem $\dot{y} = \mathtt{F}\big(y,\mathtt{p}\big),\ y(0) \in \mathtt{y}$. It is thus an element of the concrete domain $\mathcal{D} = \mathcal{P}\big(\mathcal{C}^1_+\big)$, the power set of $\mathcal{C}^1_+$. The operations inclusion $\subseteq$, union $\cup$ and intersection $\cap$ give a lattice structure to $\mathcal{D}$. Each element of $[\![\mathtt{P}]\!]$ is a continuous function which characterizes *one* particular evolution of the continuous system under *one* set of parameters and *one* input.

3 Abstract Domain

The concrete domain for the continuous processes is thus the powerset of the set of continuous functions $\mathcal{C}^0_+$. In this section, we present an abstract domain that collects elements from $\mathcal{C}^0_+$.

3.1 Interval valued step functions

Continuous functions are not representable as they assign to an infinite, uncountable number of elements (every $t \in \mathbb{R}_+$) a value that is itself not representable (as a real number) on a finite precision machine. Thus, an abstraction of a set of continuous functions must abstract the values reached by the functions as well as the instants at which these values are obtained. The former is done by using intervals instead of sets while the latter is done by considering step functions, i.e. functions that are almost always constant.

Definition 4. *Interval valued step functions*
$\mathcal{D}^\sharp$ *is the set of all step functions from* $\mathbb{R}_+$ *to* $\boldsymbol{R}$. *We recall that given a domain* D, *a function* $f : \mathbb{R}_+ \to D$ *is a step function if and only if there exist* $t_0 = 0 < t_1 < \cdots < t_n < t_{n+1} < \ldots$ *such that* $\forall n \in \mathbb{N}$, f *is constant between* t_n *and* t_{n+1}.

Representation of step functions.
Following the notations used by Julien Bertrane [2], we represent the step functions as a conjunction of constraints of the form "$t_i : \boldsymbol{x_i}$", which means that the function switches to $\boldsymbol{x_i}$ at time t_i. The switching times t_i do not need to be ordered, nor different; the infinite conjunction $f = t_0 : \boldsymbol{x_0} \wedge t_1 : \boldsymbol{x_1} \wedge \dots \wedge t_n : \boldsymbol{x_n} \wedge \dots$ represents the function f such that $\forall t \in \mathbb{R}_+, f(t) = \boldsymbol{x_i}$ with $i = \max\{j \in \mathbb{N} | t_j \leq t\}$. A finite sequence of constraints $f = t_0 : \boldsymbol{x_0} \wedge t_1 : \boldsymbol{x_1} \wedge \dots \wedge t_N : \boldsymbol{x_N}$ represents the step function f such that $\forall t \in \mathbb{R}_+,\ f(t) = \boldsymbol{x_i}$ with $i = \max\{j \in [0, N] | t_j \leq t\}$. We use the more compact notation $f = \bigwedge_{0 \leq i \leq N} t_i : \boldsymbol{x_i}$, with $N \in \mathbb{N} \cup \{\infty\}$. Let us remark that this notation is however not unique. For example, the conjunctions $3 : [1, 2] \wedge 0 : [1, 2]$ and $0 : [1, 2] \wedge 1 : [1, 2]$ define the same constant function with value $[1, 2]$. This makes the equality of functions difficult to define (we can say that $f = g \Leftrightarrow \forall t \in \mathbb{R}_+, f(t) = g(t)$ but this is not satisfying as it cannot be used for an implementation). To solve this problem, we define a normal form for the conjunctions of constraints characterized by:

1. the switching times are sorted and all different, i.e. if $f = \bigwedge_{0 \leq i \leq N} t_i : \boldsymbol{x_i}$, then $0 = t_0 < t_1 < \dots < t_n < \dots$;
2. two consecutive constraints cannot have equal values: $\forall i \in [0, N],\ \boldsymbol{x_i} \neq \boldsymbol{x_{i+1}}$.

With these conditions, the representation is unique. It is moreover easy to compute the normalized form $Norm(f)$ of a given conjunction of constraints f. First we sort the constraints by ascending switching time, with the convention that if two constraints have the same time, then we only keep the one with the highest index. This makes the conjunction to fulfill the first normalization condition. Then, we remove any constraint $t_i : \boldsymbol{x_i}$ such that $\boldsymbol{x_{i-1}} = \boldsymbol{x_i}$. This way, we only keep the longest possible steps, which satisfies the second condition. It is easy to see that the normalization process does not change the meaning of the representation: for a conjunction f, then it holds that $\forall t \in \mathbb{R}_+,\ f(t) = Norm(f)(t)$. Given two normalized conjunctions, we define an equality test:

$$\bigwedge_{0 \leq i \leq N} t_i : \boldsymbol{x_i} = \bigwedge_{0 \leq j \leq M} u_j : \boldsymbol{y_j} \iff N = M \text{ and } \forall i \in [0, N],\ t_i = u_i \text{ and } \boldsymbol{x_i} = \boldsymbol{y_i}. \quad (2)$$

The normalization process induces an equivalence relation ($f \equiv g \Leftrightarrow Norm(f) = Norm(g)$). Thus, from now on we work in the domain $\mathcal{D}^\sharp{}_{/\equiv}$, i.e. we always consider that the conjunctions are normalized. We will however keep the notation $D^\sharp$ for $D^\sharp{}_{/\equiv}$ when it is clear from the context.

Proposition 1. *Let $f, g \in D^\sharp$. Then it holds that:*

$$f = g \iff \forall t \in \mathbb{R}_+,\ f(t) = g(t) \quad (3)$$

Proof. Clearly, we have $f = g \Rightarrow \forall t \in \mathbb{R}_+,\ f(t) = g(t)$. Let us prove the other direction. Let $f, g \in \mathcal{D}^\sharp$ such that $\forall t \in \mathbb{R}_+,\ f(t) = g(t)$, with $f = \bigwedge_{0 \leq i \leq N} t_i : \boldsymbol{x_i}$ and $g = \bigwedge_{0 \leq j \leq M} u_j : \boldsymbol{y_j}$. We have $N = M$: suppose that $N \neq M$, then we can suppose that $N < M$ and $N \neq \infty$, so $\forall t \geq t_N,\ f(t) = g(t) = \boldsymbol{x_N}$; however, g has at least on last step in $[t_n, \infty]$, and thus its value changes at least once, hence the contradiction. Now, let us suppose that $A = \{i \in \mathbb{N} | t_i \neq u_i\} \neq \emptyset$, and let $k = \min A$, with $t_k < u_k$. Then we have $t_{k-1} = u_{k-1} < t_k < u_k$, so $f(t_{k-1}) = g(t_{k-1}) = \boldsymbol{x_{k-1}}$ and $f(t_k) = \boldsymbol{x_k}$, $g(t_k) = g(t_{k-1}) = \boldsymbol{x_{k-1}}$, so $\boldsymbol{x_k} = \boldsymbol{x_{k-1}}$, hence the contradiction. So, $\forall i \in [0, N],\ t_i = u_i$, and $\boldsymbol{x_i} = \boldsymbol{y_i}$. □

3.2 Concretisation and abstraction

The function $f = \bigwedge_{0 \le i \le N} t_i : \boldsymbol{x_i}$ represents the set of continuous, differentiable functions that remain within $\boldsymbol{x_i}$ for any time $t \in [t_i, t_{i+1}]$. The concretisation function $\gamma : \mathcal{D}^\sharp \to \mathcal{D}$ is thus defined by:

$$\gamma\Big(\bigwedge_{0 \le i \le N} t_i : \boldsymbol{x_i}\Big) = \{y \in \mathcal{C}^1_+ \mid \forall i \le N,\ \forall t \in [t_i, t_{i+1}],\ y(t) \in \boldsymbol{x_i}\} \tag{4}$$

If $N < \infty$, the last constraint transforms into $\forall t \ge t_N,\ y(t) \in \boldsymbol{x_N}$.
For example, Figure 2(a) shows a step function (represented by the black bold steps) and a function within its concretisation (the dashed curve). Among others, the solutions of Example 1 are contained in the concretisation (gray surface).
The definition of an abstraction is not as direct as for the concretisation. As in the case of the polyhedra domain [10], we cannot define the best one: it is always possible to increase the quality of the abstraction by selecting smaller steps. Thus, we only give a criteria for a function to be a safe abstraction. Let us first define the lower- and upper-functions for a given set of continuous real functions. Let $\mathcal{Y} \in \mathcal{D}$, we define the two functions $\underline{\mathcal{Y}}$ and $\overline{\mathcal{Y}}$ to be the inf- and sup-functions of $\mathcal{Y}$: $\underline{\mathcal{Y}} = \lambda t.inf\,\{y(t) \mid y \in \mathcal{Y}\}$ and $\overline{\mathcal{Y}} = \lambda t.sup\,\{y(t) \mid y \in \mathcal{Y}\}$. Equivalently, we define the lower- and upper-functions of an interval valued step function. Let $f \in \mathcal{D}^\sharp$, the real valued step functions $\underline{f}$ and $\overline{f}$ are: $\underline{f} = \lambda t.\underline{\boldsymbol{f(t)}}$ and $\overline{f} = \lambda t.\overline{\boldsymbol{f(t)}}$. These four functions are the basis of the *Validity condition*:

Definition 5. *A function $\alpha : \mathcal{D} \to \mathcal{D}^\sharp$ satisfy the* Validity condition *(V.C.) if and only if for all $\mathcal{Y} \in \mathcal{D}$, it holds that:*

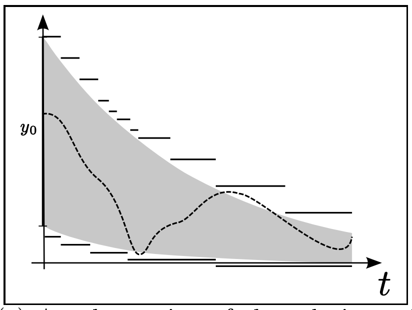
$$\forall t \in \mathbb{R}_+,\ \underline{\alpha(\mathcal{Y})}(t) \le \underline{\mathcal{Y}}(t) \le \overline{\mathcal{Y}}(t) \le \overline{\alpha(\mathcal{Y})}(t) \tag{5}$$

This property states that the computed interval valued step function indeed encloses the set $\{y(t) \mid y \in \mathcal{Y}\}$ for all $t \in \mathbb{R}_+$. The V.C. is a necessary and sufficient condition for the abstraction α to be sound (see Theorem 1).

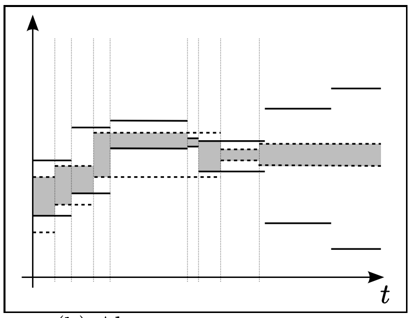
3.3 Structure of the abstract domain

Let us now show that $\mathcal{D}^\sharp$ can be given a lattice structure and that, under the V.C., the abstraction $\alpha : \mathcal{D} \to \mathcal{D}^\sharp$ is sound. Intuitively, we want to define the order $\subseteq^\sharp$ pointwise (i.e. $f \subseteq^\sharp g \Leftrightarrow \forall t \in \mathbb{R}_+, f(t) \subseteq g(t)$). We give a condition (6) on the constraints that allows for the effective testing of whether $f \subseteq^\sharp g$. Let $f = \bigwedge_{0 \le i \le N} t_i : \boldsymbol{x_i}$ and $g = \bigwedge_{0 \le j \le M} u_j : \boldsymbol{y_j}$, then

$$f \subseteq^\sharp g \iff \forall (i,j) \in [0,N] \times [0,M],\ [t_i, t_{i+1}] \cap [u_j, u_{j+1}] \ne \emptyset \Rightarrow \boldsymbol{x_i} \subseteq \boldsymbol{y_j} \tag{6}$$



(a) An abstraction of the solutions of Example 1.



(b) Abstract meet operator.

Fig. 2. Abstract domain.

Proposition 2. *If $f, g \in \mathcal{D}^\sharp$ are in a normalized form, then it holds that:*

$$f \subseteq^\sharp g \Longleftrightarrow \forall t \in \mathbb{R}_+,\ f(t) \subseteq g(t) \tag{7}$$

Proof. Let $f = \bigwedge_{0\le i\le N} t_i : \boldsymbol{x_i}$ and $g = \bigwedge_{0\le j\le M} u_j : \boldsymbol{y_j}$ be such that $f \subseteq^\sharp g$, and let $t \in \mathbb{R}_+$. There exist $i \in [0, N]$ and $j \in [0, M]$ such that $t \in [t_i, t_{i+1}]$ and $t \in [u_j, u_{j+1}]$. Thus, $[t_i, t_{i+1}] \cap [u_j, u_{j+1}] \neq \emptyset$, so $f(t) = \boldsymbol{x_i} \subseteq \boldsymbol{y_j} = g(t)$.
Now, let $f, g \in \mathcal{D}^\sharp$ such that $\forall t \in \mathbb{R}_+,\ f(t) \subseteq g(t)$, f and g written as above. Let $i, j \in [0, N] \times [0, M]$ such that $[t_i, t_{i+1}] \cap [u_j, u_{j+1}] \neq \emptyset$, and let $t \in [t_i, t_{i+1}] \cap [u_j, u_{j+1}]$. Then, $f(t) = \boldsymbol{x_i}$ and $g(t) = \boldsymbol{y_j}$, so $\boldsymbol{x_i} \subseteq \boldsymbol{y_j}$. □

The equality (Equation (2)) and the order (Equation (6)) we defined on $\mathcal{D}^\sharp$ are equivalent to the usual equality and order on functions, but the characterizations we provide permits an efficient implementation of them (in linear time). The meet operator $\cap^\sharp$ on $\mathcal{D}^\sharp$ is defined as follows. If $f = \bigwedge_{0\le i\le N} t_i : \boldsymbol{x_i}$ and $g = \bigwedge_{0\le j\le M} u_j : \boldsymbol{y_j}$, then

$$f \cap^\sharp g = Norm\left(\bigwedge_{0\le i\le N} t_i : \tilde{\boldsymbol{x_i}} \wedge \bigwedge_{0\le j\le M} u_j : \tilde{\boldsymbol{y_j}}\right) \text{ where} \tag{8}$$

$$\tilde{\boldsymbol{x_i}} = \boldsymbol{x_i} \cap \boldsymbol{y_k} \text{ where } k = \max\{j | u_j \le t_i\} \tag{9}$$

$$\tilde{\boldsymbol{y_j}} = \boldsymbol{y_j} \cap \boldsymbol{x_k} \text{ where } k = \max\{i | t_i \le u_j\} \tag{10}$$

The intersection $f \cap^\sharp g$ creates a new step function whose value is at every time t the intersection $f(t) \cap g(t)$. If this intersection is empty (i.e. $\tilde{\boldsymbol{x_i}} = \emptyset$ for some i or $\tilde{\boldsymbol{y_j}} = \emptyset$ for some j), we define $f \cap^\sharp g$ as $\bot^\sharp$, the bottom element of $\mathcal{D}^\sharp$. A graphical representation of the effect of $\cap^\sharp$ is shown in Figure 2(b): the intersection of two step functions (bold and dashed steps) is computed. The result is the gray area, and the vertical dashed lines represent the switching times.
The abstract join operator $\cup^\sharp$ is defined in the same way. Let $h = f \cup^\sharp g$, h is given as for the meet $\cap^\sharp$, except that Equations (9) and (10) are changed into:

$$\tilde{\boldsymbol{x_i}} = \boldsymbol{x_i} \cup \boldsymbol{y_k} \text{ where } k = \max\{j | u_j \le t_i\} \tag{11}$$

$$\tilde{\boldsymbol{y_j}} = \boldsymbol{y_j} \cup \boldsymbol{x_k} \text{ where } k = \max\{i | t_i \le u_j\} \tag{12}$$

The only difference is that we set the value of h at any time t to be $f(t) \cup g(t)$.

Proposition 3. *Let $\top^\sharp = 0 : [-\infty, \infty]$ be the step function with only one step with value $\mathbb{R}$. We define a special element $\bot^\sharp$ such that $\gamma(\bot^\sharp) = \emptyset$ and $\forall f \in \mathcal{D}^\sharp,\ \bot^\sharp \subseteq^\sharp f$. Then $(\mathcal{D}^\sharp, \top^\sharp, \bot^\sharp, \subseteq^\sharp, \cap^\sharp, \cup^\sharp)$ is a lattice.*

Proof. Clearly, $\forall f \in \mathcal{D}^\sharp,\ \bot^\sharp \subseteq^\sharp f \subseteq^\sharp \top^\sharp$. We still need to prove that:

1. $\cap^\sharp$ is a meet operator. Let $f, g \in \mathcal{D}^\sharp$ and $h = f \cap^\sharp g$, with $f = \bigwedge_{0 \le i \le N} t_i : \boldsymbol{x_i}$ and $g = \bigwedge_{0 \le j \le M} u_j : \boldsymbol{y_j}$. We first show that $h \subseteq^\sharp f$ by showing that $\forall t \in \mathbb{R}_+,\ h(t) \subseteq f(t)$. Let $t \in \mathbb{R}_+$, and $i, j \in [0, N] \times [0, M]$ be such that $t \in [t_i, t_{i+1}]$ and $t \in [u_j, u_{j+1}]$. Then, depending on the relative positions of $t_i,\ t_{i+1},\ u_j$ and u_{j+1}, the computation of h (Equation (8)) defines $h(t)$ to be $\tilde{\boldsymbol{x_i}}$ or $\tilde{\boldsymbol{y_j}}$, with $\tilde{\boldsymbol{x_i}} = \boldsymbol{x_i} \cap \boldsymbol{y_j}$ and $\tilde{\boldsymbol{y_j}} = \boldsymbol{y_j} \cap \boldsymbol{x_i}$. Thus, we have $h(t) \in \boldsymbol{x_i} = f(t)$. So, $h \subseteq^\sharp f$. Equivalently, we have $h \subseteq^\sharp g$. Now, let $H \in \mathcal{D}^\sharp$ such that $H \subseteq^\sharp f$ and $H \subseteq^\sharp g$. Let $t \in \mathbb{R}_+$ and i, j such that $t \in [t_i, t_{i+1}]$ and $t \in [u_j, u_{j+1}]$. Then, $H(t) \subseteq f(t) = \boldsymbol{x_i}$ and $H(t) \subseteq g(t) = \boldsymbol{y_j}$, so $H(t) \subseteq \boldsymbol{x_i} \cap \boldsymbol{y_j}$, i.e. $H(t) \subseteq h(t)$. So, $H \subseteq^\sharp h$.
2. $\cup^\sharp$ is a join operator. The proof runs as the one for $\cap^\sharp$. □

We now formulate the main theorem of this section that guarantees the soundness of the abstraction.

Theorem 1. *If α satisfies the V.C., then for every $\mathcal{Y} \in \mathcal{D}$, $\mathcal{Y} \subseteq \gamma\big(\alpha(\mathcal{Y})\big)$.*

Proof. Let $\mathcal{Y} \in \mathcal{D}$ and $f = \alpha(\mathcal{Y}) \in D^\sharp$. We want to prove that $\mathcal{Y} \subseteq \gamma\big(f\big)$. As α satisfies the V.C., we know that $\forall t \in \mathbb{R}_+,\ \forall y \in \mathcal{Y},\ y(t) \in f(t)$. Let now $y \in \mathcal{Y}$; y is a continuous function that verifies $\forall t \in \mathbb{R}_+,\ y(t) \in f(t)$, thus $y \in \gamma\big(f\big)$. So, it holds that $\mathcal{Y} \subseteq \gamma\big(f\big)$. □

4 Guaranteed Integration

In this section, we present a technique called *guaranteed* (or validated) *integration* of ODEs that, as shown in Section 5, enables one to compute the abstract semantics of the continuous processes. Guaranteed integration of ODEs tries to answer the following question: given an ODE (possibly with interval parameters), an initial value (possibly an interval) and a final time T, can we compute bounds on the value of the solution of the IVP at T? There are basically two kinds of methods for computing such bounds on the solution of the IVP. On the one side, classical methods use the Taylor series decomposition of the solution and then interval arithmetics. Advanced techniques are used in order to limit the wrapping effect inherent to the interval computations, and the first tools (e.g. VNODE [27] or AWA [23]) use such techniques. On the other side, new methods have recently been proposed [6, 29] that compute the bounds as the sum of a non-validated approximation point and a guaranteed error, i.e. an interval that is proved to contain the distance between the real solution and the approximation point. We give the main ideas of how the GRKLib [6] method works in the proof of Theorem 2. The reader can find more detailed explanations about GRKLib and a complete proof in [6].

Theorem 2. *Given an interval ODE $\dot{y} = F(y, \boldsymbol{p})$ and an interval initial value denoted $y(0) \in y_n + \boldsymbol{e_n}$, where $\boldsymbol{e_n}$ is the error and y_n the approximation point, it is possible to find a step size h, a global enclosure $\tilde{\boldsymbol{y}}$, an approximation point y_{n+1} and a local enclosure $\boldsymbol{e_{n+1}}$ of the error such that $\forall t \in [0, h]$, $y(t) \subseteq \tilde{\boldsymbol{y}}$ and $y(h) \subseteq y_{n+1} + \boldsymbol{e_{n+1}}$, where y is any solution of the interval IVP.*

Proof sketch. Let us first assume that the step size h is given. The next point y_{n+1} is computed by the classical RK4 algorithm [17] that uses four evaluations of F to approximate the mean derivative between t and $t+h$. y_{n+1} is a function of y_n and h only. So, $y_{n+1} = \psi(y_n, h)$, where ψ is expressed using F only.
The computation of $\boldsymbol{e_{n+1}}$ requires a two steps process: first we compute the a priori bound $\tilde{\boldsymbol{y}}$ and then we use it to compute a tighter bound on the global error at $t+h$. The computation of $\tilde{\boldsymbol{y}}$ uses the Picard interval operator and a Banach fix-point argument as in [23]. Using results from [3, 22], we compute $\boldsymbol{e_{n+1}}$ as $\boldsymbol{e_{n+1}} = \boldsymbol{\eta} + \boldsymbol{\chi} + \boldsymbol{\mu}$. The three terms are computed as follows:

- $\boldsymbol{\eta}$ represents the discretization error and is computed as the distance between the flows of the real valued solution of the IVP and the real valued function ψ. Both functions are equal at time t and so are their first 4 derivatives. As a consequence, $\boldsymbol{\eta}$ can be expressed as a function of their fifth derivative only (we use Taylor expansion to prove it).
- $\boldsymbol{\chi}$ represents the propagation of the error $\boldsymbol{e_n}$ into $\boldsymbol{e_{n+1}}$. In other words, it is the distance between the images by ψ of two points inside $y_n + \boldsymbol{e_n}$. This is computed using the Jacobian matrix of ψ and this is mainly where the wrapping effect occurs (if the matrix is a rotation matrix, then big over-approximations arise).
- $\boldsymbol{\mu}$ represents the implementation error, i.e. the distance between the computed floating point number y_{n+1} and the real value that would have been obtained on an infinite precision computer. We use the global error domain [24] to compute y_{n+1} so that we obtain both the floating point number and an over-approximation of its distance to the real number, i.e. an over-approximation of $\boldsymbol{\mu}$.

By combining this three computations, we obtain an over-approximation of the global error at time $t+h$ based on the error at time t. We leave the problem of finding an appropriate step size h open for now and show in Section 5 how to deal with it, and that it can be seen as some kind of dynamic partitioning [7] of the set of control points of the continuous semantics. □

5 Abstract Semantics

In this section, we show that the guaranteed integration methods provide a safe abstract semantics for the continuous processes. An abstract semantics $[\![\texttt{P}]\!]^\sharp$ of a continuous process `P=(F,p,y)` is an interval valued step functions (i.e. an element of $\mathcal{D}^\sharp$) that provides two things. On the one side, we have an abstraction of the values that represents as an interval the set $\{y(t) | y \in [\![\texttt{P}]\!]\}$ at all time

$t \in \mathbb{R}_+$. On the other side, we have an abstraction of the time line that collects the instants whose values are abstracted by the same interval. In Section 4, we showed that these abstractions are provided by the guaranteed integration algorithms: given an abstraction of the values at one time t ($y(t) \in \boldsymbol{y_n}$), a function $GRK(\mathtt{F}, \mathtt{P}, \boldsymbol{y_n})$ exists that computes h (i.e. the abstraction on the instants), $\boldsymbol{\tilde{y}}$ (i.e. the abstraction on the values) and a new interval $\boldsymbol{y}$ such that $\forall u \in [t, t+h]$, $y(u) \in \boldsymbol{\tilde{y}}$ and $y(t+h) \in \boldsymbol{y}$. Let us briefly explain how the step size h is chosen: first, during the computation of the a priori approximation $\tilde{y}$, we use a Banach fix-point argument, and thus compute $\tilde{y}$ as the limit of the iterates of a contracting function. On a computer, this can loop forever either due to rounding errors or because the fix-point is reached after an infinite iteration. Thus, we use a limit on the number of iterations and we use a smaller step size if this limit is reached. Secondly, after each step, the width of $\boldsymbol{y}$ is compared to the user specified tolerance and control theoretic techniques are used in order to adjust the next step-size and avoid the error to grow. Thus, the partitioning of the time line is dynamically computed at each step.

Definition 6. *Abstract semantics*
Let `P=(F,p,y)` *be a continuous process. The abstract semantics* $[\![P]\!]^\sharp$ *of* `P` *is the result of Algorithm 1.*

The abstract semantics is computed by iterating the guaranteed integration process. Let us remark that the implementation of the GRK function can fail to find h, $\boldsymbol{\tilde{y}}$ and $\boldsymbol{y}$ if the selected step size becomes smaller than the machine precision. Whenever this happens, a sort of widening is performed as we end the constraint conjunction by $t : \mathbb{R}_+$.

Theorem 3. *Let* `P=(F,p,y)` *be a continuous process. Then the abstract semantics* $[\![P]\!]^\sharp$ *is a safe abstraction of the concrete semantics, i.e.:*

$$[\![P]\!] \subseteq \gamma\big([\![P]\!]^\sharp\big) \tag{13}$$

Proof. Let $\alpha : \mathcal{D} \to \mathcal{D}^\sharp$ be the abstraction function defined by $\alpha\big([\![\mathtt{P}]\!]\big) = [\![\mathtt{P}]\!]^\sharp$ and $\forall \mathcal{Y} \in \mathcal{D}$, $\mathcal{Y} \neq [\![\mathtt{P}]\!]$, $\alpha(\mathcal{Y}) = \top^\sharp$. If we show that α verifies the V.C., then Equation (13) holds. Let $\mathcal{Y} \in \mathcal{D}$, we show that $\alpha(\mathcal{Y})$ verifies Equation (5). This is clearly the case if $\mathcal{Y} \neq [\![\mathtt{P}]\!]$. Let us suppose that $\mathcal{Y} = [\![\mathtt{P}]\!]$, and let $[\![\mathtt{P}]\!]^\sharp = \bigwedge_{0 \le i \le N} t_i : \boldsymbol{x_i}$. We need to prove that $\forall i \in [0, N]$, $\forall t \in [t_i, t_{i+1}]$ and $\forall y \in \mathcal{Y}$, $y(t) \in \boldsymbol{x_i}$. We

```
Input: P=(F,p,y)
Output: $[\![P]\!]^\sharp$
$t = 0$; $h =$ InitialGuess(F);
$\boldsymbol{yn} =$ y;
while $(h, \boldsymbol{\tilde{y}}, \boldsymbol{y}) = GRK(F, \boldsymbol{p}, \boldsymbol{yn})$ do
    $res = res \wedge t : \boldsymbol{\tilde{y}}$;   $\boldsymbol{yn} = \boldsymbol{y}$;
end
return $res \wedge t : \mathbb{R}$
```
Algorithm 1: Abstract semantics computation.

prove this by induction on i.
For $i = 0$, we have $\forall y \in \mathcal{Y},\ y(0) \in \mathtt{y}$ and the *GRK* function gives h, $\tilde{y}$ and y_1 such that $\forall t \in [0, h],\ \forall y \in \mathcal{Y},\ y(t) \in \tilde{y}$. The algorithm 1 sets x_0 to $\tilde{y}$, which proves the case $i = 0$.
Let now $i \in [0, N]$ be such that $\forall y \in \mathcal{Y},\ \forall t \in [t_i, t_{i+1}],\ y(t) \in \boldsymbol{x_i}$. Clearly, the algorithm 1 also gives an interval $\boldsymbol{y_i}$ such that $\forall y \in \mathcal{Y},\ y(t_{i+1}) \in \boldsymbol{y_i}$. Let `P'=(F,p,`$\boldsymbol{y_i}$`)` be the interval IVP which differs from `P` only for the initial value. Then, for every $H > 0$, it holds that $\{y(t) | t \in [0, H],\ y \in [\![\mathtt{P'}]\!]\} = \{y(t) | t \in [t_{i+1}, t_{i+1} + H],\ y \in [\![\mathtt{P}]\!]\}$, i.e. the solutions of the initial IVP for time between t_{i+1} and $t_{i+1} + H$ are the same as the solutions of the IVP `P'` for time between 0 and H. The *GRK* function gives $\tilde{y}$ and h such that $\forall t \in [0, H],\ \forall y \in [\![\mathtt{P'}]\!],\ y(t) \in \tilde{y}$. The algorithm 1 sets $\boldsymbol{x_{i+1}}$ to be $\tilde{y}$ and t_{i+2} to $t_{i+1} + h$, so we have $\forall t \in [t_{i+1}, t_{i+2}],\ \forall y \in \mathcal{Y},\ y(t) \in \boldsymbol{x_{i+1}}$. □

6 Example of use

We present an example that illustrates how we intend to include our work into existing static analyzers. We consider a code that is often used in embedded programs: an integrator. The program in Listing 1 (inspired from [16]) integrates using the rectangle method the input data. The integration is carried out up to some threshold defined by the interval `[INF,SUP]`. The input data are given by a sensor (hence the `volatile` variable `x`) at a frequency of 8KHz. The integrator is a well known difficult problem for the analysis of numerical precision [11]. Its behavior is extremely depending of the input data (i.e. the physical environment) of the frequency of the integration process (i.e. the sampling rate) and of the precision of the sensor.

```
 1  #define SUP 4
 2  #define INF -4
 3  // assume x'=2*Pi*y and y'=-2*Pi*x
 4  volatile float x;
 5  static float intgrx=0.0,h=1.0/8;
 6  void main() {
 7     while (true) { // assume frequency = 8 KHz
 8       xi = x; intgrx += xi*h;
 9       if (intgrx > SUP)
10         intgrx = SUP;
11       if (intgrx < INF)
12         intgrx = INF;
13     }}
```

Listing 1. Simple integrator.

The comments on the code in Listing 1 indicate how we give the analyzers hints on the physical environment. The first one (Line 5) gives the differential equation followed by `x` and `y` while the second one (Line 8) indicates the frequency of the main loop. Such comments could be understood by the static analyzer and are often already present (although not in this form) in embedded programs (it is very frequent to find a comment such as “this loop runs at 8KHz” in the code usually given to static analyzers). In this example, the input signal is $x(t) = sin(2\pi t)$. In theory (i.e. with a perfect knowledge of the environment and an infinite precision computer), the value of `intgrx` remains bounded by $[0, 2]$. As explained in the introduction, a naive abstraction of the continuous environment approximates `x` by the interval $[-1, 1]$. In this case, the analyzer binds the variable `intgrx` with the value $[-n \cdot h, n \cdot h]$ after unrolling the main loop n times. We implemented a prototype analyzer that uses the abstraction of the continuous environment of Section 5 to improve this result. The analyzer uses the GRKLib library [6] as guaranteed integration tool and is implemented in OCAML. The analyzer takes as input language a subset of C and the comments are changed into specific assertions that the analyzer understands. Figure 3 shows the results obtained by this analyzer and by an abstraction using intervals. After 100 iterations, the value of `intgrx` is $[-4, 4]$ with intervals because of the thresholds and $[-4.353 \cdot 10^{-2}, 4.192 \cdot 10^{-2}]$ with our analyzer.

7 Conclusion

In this article, we provided a formalization and an abstraction of the physical environment of embedded software which is coherent with the analysis of the discrete program itself. Like the collecting semantics of a program describes all the possible executions for any input data, our description of the continuous environment describes all possible continuous evolutions for any initial condition and parameter of the system. We then defined an abstract domain that allows for the sound over-approximation of continuous functions: the domain of interval valued step functions. A major difficulty in the definition of this domain was to

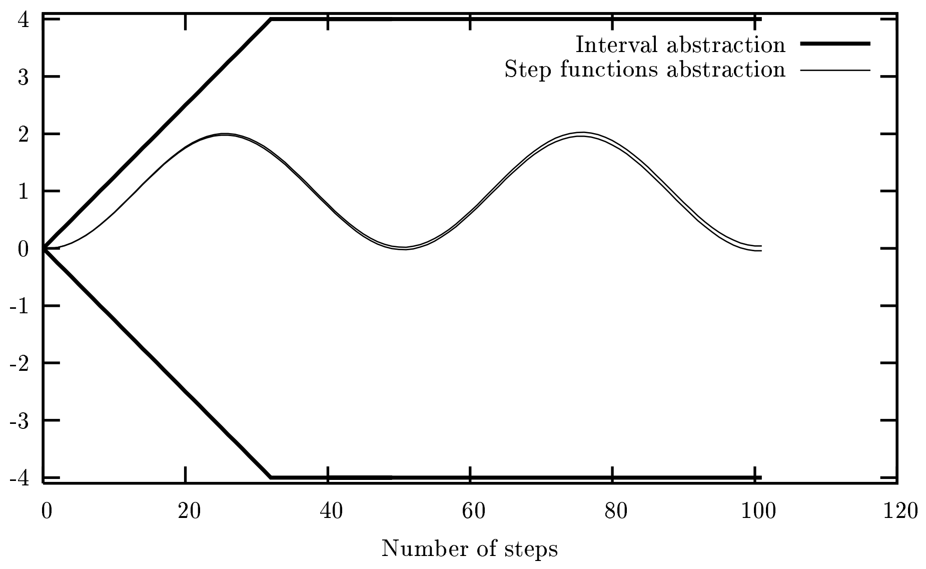


Fig. 3. Result of the analysis of the integrator.

deal with dynamic step sizes in order to cope with the most efficient numerical algorithms. Our representation of such functions as a conjunction of constraints allows for an elegant definition of the abstract operators and their efficient implementation. Finally, we showed that the guaranteed integration methods provide an abstract semantics for the continuous process that is sound with respect to the collecting one. A simple example derived from a well known, difficult problem shows that our approach considerably improves the analysis.

The analysis of the complete hybrid system still needs some extensions. First of all, we do not consider yet feedback from the program, i.e. we do not mention actuators. Previous work of by Olivier Bouissou [5] dealt with this problem and a merge of both results is necessary. Secondly, our formalism only abstracts the environment and does not consider the action of the sensors. These latter introduce some noise inside the system as their measurements as well as their sampling rate are imprecise. In a sense, our model supposes that we have perfect sensors, i.e. that the values passed to the program are the exact values of the continuous environment. Clearly, a better modeling of sensors will be necessary. For example, we can add comments and/or assertions in the program that describes the inaccuracy of the sensors. Thus, our abstraction of the environment remains valid, and it is only when the values are passed to the program that they are modified in order to match the specification of the sensor. Finally, the addition of an extrapolation method as a widening operator will complete the abstract interpretation of continuous functions. This is a purely numerical problem that does not change anything to our domain.

References

1. R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.
2. J. Bertrane. Static analysis by abstract interpretation of the quasi-synchronous composition of synchronous programs. In *VMCAI*, volume 3385 of *LNCS*, pages 97–112. Springer, 2005.
3. Ludwig Bieberbach. On the remainder of the runge-kutta formula. *Z.A.M.P.*, 2:233–248, 1951.
4. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation: Complexity, Analysis, Transformation.*, volume 2566 of *LNCS*. Springer-Verlag, October 2002.
5. O. Bouissou. Analyse statique par interpretation abstraite de système hybrides discrets-continus. Technical Report 05-301, CEA-LIST, 2005.
6. O. Bouissou and M. Martel. GRKLib: a guaranteed runge-kutta library. In *SCAN*, 2006.
7. F. Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 2(4):407–423, 1992.
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*, 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. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–97. ACM Press, 1978.
11. M. Daumas and D. Lester. Stochastic formal methods: An application to accuracy of numeric software. In *Proceedings of the 40th IEEE Annual Hawaii International Conference on System Sciences*, 2007.
12. A. Edalat and A. Lieutier. Domain theory and differential calculus. *Mathematical Structures in Computer Science*, 14(06):771–802, 2002.
13. A. Edalat, A. Lieutier, and D. Pattinson. A computational model for multi-variable differential calculus. In *FOSSACS*, volume 3441 of *LNCS*, 2005.
14. A. Edalat and D. Pattinson. A domain theoretic account of Picard's theorem. In *ICALP*, volume 3142 of *LNCS*, pages 494–505, 2004.
15. E. Goubault, M. Martel, and S. Putot. Asserting the precision of floating-point computations: A simple abstract interpreter. In *ESOP*, volume 2305 of *LNCS*, pages 209–212. Springer, 2002.
16. E. Goubault, M. Martel, and S. Putot. Some future challenges in the validation of control systems. In *ERTS*, 2006.
17. E. Hairer, S. P. Norsett, and G. Wanner. *Solving ordinary differential equations I (2nd revised. ed.): nonstiff problems.* Springer-Verlag New York, Inc., New York, NY, USA, 1993.
18. N. Halbwachs, Y. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In *SAS*, volume 864 of *LNCS*, pages 223–237, 1994.
19. T. A. Henzinger. The theory of hybrid automata. In *Symposium on Logic in Computer Science*, pages 278–292. IEEE Computer Society Press, 1996.
20. T. A. Henzinger and P. Ho. A note on abstract interpretation strategies for hybrid automata. In *Hybrid Systems II*, volume 999 of *LNCS*, pages 252–264, London, UK, 1995. Springer-Verlag.
21. T. A. Henzinger, B. Horowitz, R. Majumdar, and H. Wong-Toi. Beyond HYTECH: Hybrid systems analysis using interval numerical methods. In *HSCC*, volume 1790 of *LNCS*, pages 130–144, 2000.
22. J. W. Carr III. Error bounds for the runge-kutta single-step integration process. *JACM*, 5(1):39–44, 1958.
23. R. Lohner. *Einschließung der Lösung gewöhnlicher Anfangsund Randwertaufgaben und Anwendungen.* PhD thesis, Universität Karlsruhe, 1988.
24. M. Martel. An overview of semantics for the validation of numerical programs. In *VMCAI*, volume 3385 of *LNCS*, pages 59–77. Springer, 2005.
25. M. Martel. Towards an abstraction of the physical environment of embedded systems. In *NSAD*, 2005.
26. P. J. Mosterman. An overview of hybrid simulation phenomena and their support by simulation packages. In *HSCC*, volume 1569 of *LNCS*, pages 165–177. Springer Verlag, 1999.
27. N. S. Nedialkov and K. R. Jackson. An interval Hermite-Obreschkoff method for computing rigorous bounds on the solution of an initial value problem for an ordinary differential equation. In *Developments in Reliable Computing*, pages 289–310. Kluwer, Dordrecht, Netherlands, 1999.
28. IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic.* IEEE, New York, August 12 1985.
29. A. Rauh, E. Auer, and E. Hofer. ValEncIA-IVP: A case study of validated solvers for initial value problems. In *SCAN*, 2006.