Abstract Domains for Constraint Programming with Differential Equations

Ghiles Ziat
ISAE-SUPAERO, Université de Toulouse
Toulouse, France
ziat@isae-supaero.fr

Olivier Mullier
ENSTA Paris, Institut Polytechnique de Paris
Palaiseau, France
mullier@ensta.fr

Julien Alexandre dit Sandretto
ENSTA Paris, Institut Polytechnique de Paris
Palaiseau, France
alexandre@ensta.fr

Christophe Garion
ISAE-SUPAERO, Université de Toulouse
Toulouse, France
garion@isae-supaero.fr

Alexandre Chapoutot
ENSTA Paris, Institut Polytechnique de Paris
Palaiseau, France
chapoutot@ensta.fr

Xavier Thirioux
ISAE-SUPAERO, Université de Toulouse
Toulouse, France
thirioux@isae-supaero.fr

Abstract

Cyber-physical systems (CPSs), as cruise control systems, involve life-critical or mission critical functions that must be validated. Formal verification techniques can bring high assurance level but have to be extended to embrace all the components of CPSs. Physical part models of CPSs are usually defined from ordinary differential equations (ODEs) and reachability methods can be used to compute safe over-approximation of the solution set of ODEs. However, additional constraints, as obstacle avoidance have also to be considered to validate CPSs. To meet this need, we propose in this paper a framework, based on abstract domains, for solving constraint satisfaction problems where the objects manipulated are described by ODEs. We use a form of disjunctive completion for which we provide a split operator and an efficient constraint filtering mechanism that takes advantage of the continuity aspect of ODEs. We illustrate the benefits of our method on a real-world application of trajectory validation of a swarm of drones, for which the main property we aim to prove is the absence of collisions between drone trajectories. Our work has been concretized in the form of a cooperation between the DynIbex library, used for the abstraction of ODEs, and the AbSolute constraint solver, used for the constraint resolution. Experiments show promising results.

NSAD 2020, November 2020, Chicago, Illinois

ACM ISBN 978-1-4503-8187-1/20/11…$15.00
https://doi.org/10.1145/3427762.3429453

ACM Reference Format:
Ghiles Ziat, Olivier Mullier, Julien Alexandre dit Sandretto, Christophe Garion, Alexandre Chapoutot, and Xavier Thirioux. 2020. Abstract Domains for Constraint Programming with Differential Equations. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Numerical and Symbolic Abstract Domains (NSAD '20), November 17, 2020, Virtual, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3427762.3429453

1 Introduction

CPSs often require the verification of dynamic properties linked either to their internal functioning or to the interaction with the physical environment in which they evolve. Previous work [6] has shown that the *Abstract Interpretation framework* [8] can be used for formal verification of hybrid systems and highlighted the interaction between the discrete and the continuous parts of a system. Verification of properties on such physical systems begins with their modeling and the study of their evolution. They are generally described by ODEs expressing the dynamics of the system and their interactions with their environment. Solving such equations is a hard task and a numerical analysis is generally necessary to compute an approximate solution (*e.g.*, using Euler or Runge-Kutta methods [5]). Furthermore, the numerical analysis obtained must be sufficiently precise to allow the verification of additional safety properties (*e.g.*, absence of collision between trajectories). In this paper, the formal verification of such properties is considered.

Classic approaches for approximating the continuous dynamics of hybrid systems use Taylor model-based flow-pipe constructions [7, 30] to compute over-approximation of the reachable states of hybrid systems starting from some initial states. Another technique is the use of step-wise functions [4] which consists in an abstraction of the continuous functions by considering small time steps. In both cases, over-approximations are generally being computed using some

numerical domains, such as intervals, octagons [21] or convex polyhedra [11]. However, when additional constraints on the systems must be respected, the problem becomes more difficult as not only the reachability space for each ODE must be computed, but we must also ensure that every point in these spaces respect the constraints.

In this work, we propose a two-step approach for the verification of these properties. The first step consists in the over-approximation of the physical systems described by ODE, and the second consists in the resolution of a constraint satisfaction problem about the solutions of these ODEs. The idea of mixing techniques from Constraint Programming and Abstract Interpretation is promising and has been exploited both for constraint solving [27, 28] and program analysis [22, 29]. The present work is based on the abstract solving method introduced in [27]. The challenge is twofold: first, we have to define a representation capable of taking into account the particularities of the ODE, *i.e.*, their topological and temporal aspects, in order to be able to efficiently solve the problems. Second, our techniques must be scalable: in order to have precise ODE solutions to avoid errors due to approximations, time variable is aggressively subdivided which results in solutions consisting of a large set of abstract elements. Several abstract domains exists for the handling of sets of abstract elements, namely powerset abstract domain [14] or disjunctive completion [9, 18], but, to our knowledge, none has been adapted yet to a constraint solving framework as in [27].

Previous work has focused on the resolution of constraints involving ODEs, as in [12] or in [13] in which the authors extend the iSAT algorithm with safe numerical integration of ODEs as constraint filtering mechanism. Also, in [15], the authors propose a slightly adapted filtering algorithms applied to constraints that handle ODEs. More recently, [19] focused on multi-physics dynamic problems and proposed a *Multiple-ODE* filtering algorithm, and [2] proposed a framework that is able to deal with a wide class of problems based on logical combination of high-level properties, involving ODEs. The present work differs from previous ones in several ways. First, these works are largely based on interval analysis, while our framework based on abstract domains allows the natural management of more complex representations (zonotopes, polyhedra, etc.). In addition, we incorporate within our structures properties of ODEs, such as their chronological and their contiguous aspects, which allows us to have more efficient operations and, to our knowledge, has never been done yet.

This paper is organized as follows: section 2 introduces physical systems and the formal definition of their abstractions as ODEs. Section 3 recalls definitions about constraint programming in general and the use of abstract domains to solve continuous problems in particular. Section 4 details two new abstract domains for constraint solving, namely the *sequence* abstract domain, and the *tree* abstract domain. Section 5 shows an example application of our method on a realistic example, and section 5.1 demonstrates the efficiency of our abstract domains on a benchmark. Finally, section 6 summarizes our work and discusses its perspectives.

2 Abstraction to handle physical systems

First, we need to define an abstract domain to handle physical systems. A physical system can be modeled in the sense of an ODE and the computation of their reachability set with the solution of an initial value problem with ODE (IVP-ODE). The domain of intervals is well suited for this purpose and has been studied extensively over the last years [6, 20, 26].

2.1 Abstraction with boxes by reachability

We now recall the definition of an IVP-ODE with a set of possible initial conditions:

$$\begin{cases} \dot{\mathrm{x}} = f(\mathrm{x}(t)) & \text{(ODE Constraint)} \\ \mathrm{x}(0) \in \mathcal{X}_0 \subseteq \mathbb{R}^n, & \text{(initial conditions)} \\ t \in [0, t_{\text{end}}] \quad . \end{cases} \tag{1}$$

The function $\mathrm{x}(t)$ is the state depending on time t. The solution of an ODE is a function $\mathrm{x}(t)$ which satisfies the constraint on $\mathrm{x}(t)$ and its derivative specified by function f. The solution of an IVP-ODE must also satisfy the initial condition. Classical hypotheses are considered to ensure the existence and uniqueness of solutions of eq. (1). For a given initial condition $\mathrm{x}_0 \in \mathcal{X}_0$, the solution when it exists is denoted $\mathrm{x}(t; \mathrm{x}_0)$. Instead of a single initial condition x_0, a set $\mathcal{X}_0$ of initial conditions is used, for example, to model some (bounded) uncertainties. The solution is then a set of functions, the concrete domain.

This solution cannot be computed in general. The goal of a validated (or rigorous) numerical integration method is to characterize the set of functions satisfying eq. (1), in the form of the values this set of functions can reach with their associated time instants: $\{\mathrm{x}(t; \mathrm{x}_0) : \forall \mathrm{x}_0 \in \mathcal{X}_0, \forall t \in [0, t_{\text{end}}]\}$, the abstract domain. A convenient way to access those values is using the abstract domain of intervals which uses interval analysis to compute an over approximation of this set [20, 24, 26].

Interval Analysis. When dealing with some computation involving sets of values, interval analysis [25] is a method designed to produce a sound over-approximation of the computation. Hereafter, an interval is denoted $[x] = [\underline{x}, \overline{x}]$ with $\underline{x} \leqslant \overline{x}$ and the set of intervals is $\mathbb{IR} = \{[x] = [\underline{x}, \overline{x}] \mid \underline{x}, \overline{x} \in \mathbb{R},\ \underline{x} \leqslant \overline{x}\}$. In order to deal with interval functions, an interval inclusion function also known as interval extension of a function can be defined.

Many interval extensions of functions can be defined when they verify the fundamental theorem (see [24]). We can cite the natural extension [25] which replaces the operations on reals by their interval counterparts using interval arithmetic.

Another interval extension is the mean value extension [25] which linearizes the function around its mean value.

2.2 Validated Numerical Integration of physical systems

The goal for a validated numerical integration method is then to compute the set of solutions of the IVP-ODE in eq. (1), *i.e.*, the set of possible solutions at time t given the initial condition in the set of initial conditions $\mathcal{X}_0$:

$$x(t;\mathcal{X}_0) = \{x(t;\mathrm{x}_0) \mid \mathrm{x}_0 \in \mathcal{X}_0\}. \tag{2}$$

A validated numerical integration scheme using set-membership framework aims at producing the solution of the IVP-ODE that is the set defined in (2). It results in the computation of an over-approximation of $x(t;\mathcal{X}_0)$.

When considering the set of initial conditions as a box $[\mathrm{x}_0]$, the use of the interval technique framework for eq. (1) makes possible the design of an inclusion function for the computation of an over approximation of $x(t;[\mathrm{x}_0])$ defined in eq. (2). We denote this inclusion function $[x]\,(t;[\mathrm{x}_0])$. To build it, a sequence of time instants $t_1,\ldots,t_s$ such that $t_1 < \cdots < t_s$ and a sequence of boxes $[\mathrm{x}_1],\ \ldots,[\mathrm{x}_s]$ such that $x(t_{i+1};[\mathrm{x}_i]) \subseteq [\mathrm{x}_{i+1}]$, $\forall i \in [0,s-1]$ are computed. From $[\mathrm{x}_i]$, computing the box $[\mathrm{x}_{i+1}]$ is a classical 2-step method (see [26]):

Phase 1 compute an *a priori* enclosure $[\tilde{\mathrm{x}}_i]$ of the set $\{x(t_k;\mathrm{x}_i) \mid t_k \in [t_i,t_{i+1}], x_i \in [\mathrm{x}_i]\}$ such that $x(t_k;[\mathrm{x}_i])$ is guaranteed to exist and is unique,

Phase 2 compute an enclosure of the solution $[\mathrm{x}_{i+1}]$ at time t_{i+1}.

A box $[\mathrm{X}_i]$ is the Cartesian product of the time interval $[t_{i-1},t_i]$ and the state interval $[\tilde{\mathrm{x}}_i]$ containing all the values the state can reach in the time interval $[t_{i-1},t_i]$.

The next section is dedicated to the handling of this set of trajectories.

2.3 Abstraction of boxes with disjunction of linear constraints

The solution of an IVP-ODE which is given as a set of timed boxes in the form $\{([t_1],[\tilde{\mathrm{x}}_1]),\ldots,([t_{\text{end}}],[\tilde{\mathrm{x}}_{\text{end}}])\}$ can easily be transcribed as a disjunction of constraints since each couple $([t_i],[\tilde{\mathrm{x}}_i])$ corresponds to a quantified proposition:

$$([t_i],[\tilde{\mathrm{x}}_i]) \equiv (\forall t \in [t_i])(\exists \mathrm{x} \in [\tilde{\mathrm{x}}_i])(\mathrm{x}(t) = \mathrm{x}) \tag{3}$$

or defined as a constraint over the variables t and x:

$$([t_i],[\mathrm{x}_i]) \equiv (t \in [t_i]) \wedge (\mathrm{x} \in [\mathrm{x}_i]) \tag{4}$$

where $t \in [t_i]$ means $\underline{[t_i]} \leqslant t \leqslant \overline{[t_i]}$ with $\underline{[t_i]}$ and $\overline{[t_i]}$ the lower and outer bound of $[t_i]$ respectively. Eventually, the abstraction of the set of trajectories can be modeled as a disjunction of all the constraints from eq. (4) since at each time $t \in [t_0,t_{\text{end}}]$ the state $\mathrm{x}(t)$ must verify exactly one of these constraints.

2.4 Tool: DynIbex

The tool used for abstraction of ODE's solutions is DynIbex [1], a library written in C++ using Ibex which is a library for constraint processing over real numbers. It adds a validated numerical integration method to compute an over approximation of the reachable set of an ODE in a time interval. It returns a set of timed boxes in the form $\{([t_1],[\tilde{\mathrm{x}}_1]),\ldots,([t_{\text{end}}],[\tilde{\mathrm{x}}_{\text{end}}])\}$ (and also tighter approximations at given time steps) using Runge-Kutta methods with interval analysis.

3 Constraint Programming

We now propose to incorporate ODEs solutions within a Constraint Programming framework. Constraint Programming [23] is a declarative programming paradigm in which a user specifies the constraints of a system, generally stated as first-order logic formulae, and then relies on a solver, which comes with constraint filtering mechanisms and choice heuristics, to establish its satisfiability.

3.1 Constraint satisfaction problems

A continuous constraint-satisfaction problem can be defined as a triplet $\langle\mathcal{V},\mathcal{D},C\rangle$, where $\mathcal{V} = \{v_1,\ldots,v_n\}$ is a set of variables, $\mathcal{D} = \{d_1,\ldots,d_n\}$ a set of interval domains, each one being associated to a variable, and $C = \{c_1,\ldots,c_m\}$ is a set of constraints over the variables.

Constraint Language. We consider a standard constraint language using a finite and fixed set $\mathcal{V}$ of real-valued variables, numeric and boolean expressions. Numeric expressions include real constants, variables and usual operators over arithmetic expressions. Boolean expressions are built using boolean operators ($\vee$, $\wedge$ and $\neg$) and usual comparison operators on arithmetic expressions. A constraint c is a boolean expression whose concrete semantic corresponds to the set of mappings, called instances, from variables to values i for which the evaluation, denoted $c(i)$, yields *true*. Solving a CSP usually means to find all the instances that satisfy every constraint of the problem. Because this is generally impossible when the domains of the variable are continuous, solvers generally compute a set of boxes (in our case any abstract element) that *covers* the solution space. In order to build this cover, such a solver alternates two main steps:

- *Filtering:* which reduces the domains of the variables by removing values that cannot be solutions.
- *Exploration:* when the domains cannot be reduced anymore, solvers then duplicate the problem, to create two (or more) complementary sub-problems.

As repeating these two steps in turn is not guaranteed to terminate, this procedure continues until the search space contains no solution, only solutions, or is smaller than some parameter according to a size metric. We base our work on the constraint solving framework introduced in [27] in

which the authors introduce a solving method based on abstract domains. Algorithm 1 illustrates this procedure and we explain it in the following.

Algorithm 1 Abstract solving

```
1:  function SOLVE(D, C, r)
2:      cover ← ∅
3:      explore ← ∅
4:      e = init(D)
5:      push e in explore
6:      while explore ≠ ∅ do
7:          e ← pop(explore)
8:          e ← ρ(e, C)
9:          if e ≠ ⊥ then
10:             if τ(e) ≤ r then
11:                 cover ← cover ∪ e
12:             else
13:                 push ⊕(e) in explore
14: return cover
```

This algorithm builds a set of abstract elements S that covers the solution space, *i.e.*, for all instances i that satisfy all the constraints C, we have $\exists e \in s, i \in \gamma(e)$. Firstly, `init` creates an abstract element from the initial domains $\mathcal{D}$ of the variables of the problem. Then, the obtained element is filtered using ρ for each constraint in turn. If the tightened abstract element is not empty, three cases are possible:

- if the element satisfies the constraint, it is added to the set of solutions `cover`.
- if it does not and if it is small enough with respect to a parameter r ($\tau(e) \leq r$), it is also added to the set of solutions (which makes the resolution method sound).
- otherwise, it is divided into sub-elements using the split operator $\oplus$ and the process is repeated with each of these sub-elements.

When `explore` is empty, all of the elements have been processed, and the union of the element in `cover` is a sound over-approximation of the solution space.

3.2 Abstract Domain for constraint solving

In [27], the authors define the operators and requirements over these operators an abstract domain must satisfy in order to be used in a constraint resolution scheme. The following definitions recall these.

Definition 3.1 (Abstract domains for constraint solving)**.** Abstract domains for constraint solving are given by

- a partial order $\langle \mathcal{D}^\sharp, \sqsubseteq \rangle$ and the usual abstract set operators and values $\langle \top_{\mathcal{D}^\sharp}, \bot_{\mathcal{D}^\sharp}, \sqcap^\sharp, \sqcup^\sharp \rangle$
- an abstraction α and a concretization function γ

Along with:

- a size function $\tau : \mathcal{D}^\sharp \to \mathbb{R}^+$
- a splitting operator on $\mathcal{D}^\sharp$, $\oplus : \mathcal{D}^\sharp \to P(\mathcal{D}^\sharp)$,
- a constraint filtering operator $\rho_{\mathcal{D}^\sharp} : \mathcal{D}^\sharp \times C \to \mathcal{D}^\sharp \cup \{\bot^\sharp\}$, which given an abstract value e and a constraint c computes the smallest abstract value (possibly empty) entailed by c and e.

where the split operator $\oplus$ should respect Definition 9 of [27], which we recall here:

Definition 3.2 (Split operator)**.**

1. $\forall d \in \mathcal{D}^\sharp, |\oplus(d)|$ is finite
2. $\forall d \in \mathcal{D}^\sharp, \forall d_i \in \oplus(d), d_i \sqsubset d$
3. $\forall d \in \mathcal{D}^\sharp, \gamma(d) = \bigcup\{\gamma(d_i) \mid d_i \in \oplus(d)\}$

The first property is necessary to guarantee the termination, the second ensures that the operator actually splits, *i.e.*, compute smaller elements than the original one, and the last one guarantees the soundness of the solving process, *i.e.*, the splitting does not lose instances. Also, in order to have a terminating process, no infinite sequence of split and constraint filtering must exists, and thus, every such finite sequence should yield a element smaller than a given parameter with respect to τ. More formally, $\oplus$ and τ must be compatible according to definition 10 of [27]:

Definition 3.3 (Compatibility of $\oplus$ and τ)**.** The split operator $\oplus$ and the size operator τ are compatible, for any reductive operator $\rho^\sharp$

$$\forall d \in \mathcal{D}^\sharp, \forall r \in \mathbb{R}^+, \exists k, \forall i \geq k, \tau((\oplus \circ \rho^\sharp)^i(d)) \leq r$$

In the remaining of the paper, we will define two abstract domains along with their split and measure operators, designed specifically for the handling of ODE solutions.

3.3 Tool: Absolute

The AbSolute solver [27] is an open source constraint solver based on abstract domains. It is built upon the Abstract Interpretation framework, and features several techniques and classical heuristic from Constraint Programming. The solver is written in *OCaml* and is usable with several numeric abstract domains (intervals, congruences, octagons, polyhedra), and domain combinator (products), some of which are based upon the Apron [17] library. We use AbSolute as an oracle for DynIbex to verify if the constraints over some dynamic objects described as ODEs hold. The abstract domains we present in this paper are implemented as plugins of AbSolute. They consist in domain combinators that allow the use of higher-level abstractions better suited for ODEs.

4 Specific abstract domains for constraint solving with ODEs

Solving a constraint satisfaction problem involving ODEs requires an efficient handling of disjunctive constraints as ODE solutions are represented using disjunctive constraints as

shown in section 2. Hence, it is crucial to use an abstract domain able to encode disjunctions precisely. Several abstract domains exist for this purpose (powerset [14, 31], binary decision trees or binary decision diagrams [10, 16], etc.) and are generally some kind of *disjunctive completion* [9]. Disjunctive completion is a way to augment the precision of an analysis by building, from a base abstract domain $\mathcal{D}_1^\sharp$, a more precise one $\mathcal{D}_2^\sharp$. Given that $\mathcal{D}_1^\sharp$ is able to represent exactly a certain set of properties, $\mathcal{D}_2^\sharp$ will be able to encode disjunctions of these properties with no loss of precision. This is very useful in practice as the $\sqcup^\sharp$ operator very often loses precision with convex representation (boxes, polyhedra, etc.). However, disjunctive completion can often be costly as the number of abstract element grows. This is problematic as in order to have a precise enough over-approximation of an ODE, it is crucial to split the sequence of time instants into a lot of very small frames. Moreover, ODEs feature a strong topological property: as they are abstractions of continuous functions, abstract elements are topologically close to each other, and in fact even touch, on two successive iteration steps. We exploit this idea in this section by proposing two abstractions: one that takes advantage of the sequential aspect of the ODEs (*i.e.*, the order in which the elements are built), and one that makes use of the continuity property to propose a fast pre-computation for the operations we need.

4.1 The sequence abstract domain

Using a disjunctive form, many operations can often be costly if made naively: for example, the meet operation between two abstract elements d_1 and d_2 would require $n \times m$ meet operations of the underlying abstract domain, where n and m are respectively the number of elements in d_1 and d_2. However, such an implementation does not take into account the structure of an ODE solution. To overcome this issue, we propose a first abstraction for ODEs: a disjunctive form in which we add an order between the disjunctions of an abstract element. As ODE solutions are built iteratively, we keep track of this structure by maintaining them sorted according to the total order defined as follows.

Definition 4.1 (Chronological order)**.** Given two (non-bottom) abstract elements $a, b \in \mathcal{D}^\sharp$, and a variable t, let $\leq_D$ be the total order defined as $a \leq_D b \triangleq min(\pi_t(a)) \leq min(\pi_t(b))$ where π_t is the interval hull of the projection of an abstract element over the variable t, and min is the lower bound of an interval.

Note that by construction, the resulting interval will always be non-empty, and hence will always have a minimum. By doing so, we suppose that the CSP is annotated with a special variable, denoting time, that should not be treated as other variables. Using this order we now define the *sequence* abstract domain $S(\mathcal{D}^\sharp)$ as follows.

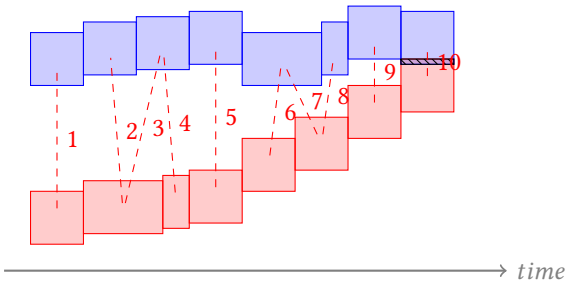

Figure 1. Order of meet operations (in red) of $\mathcal{D}^\sharp$ during the intersection of two sequences. The hatched zone is the resulting sequence of one element.

Definition 4.2 (Sequence abstract domain)**.** Given an abstract domain $\langle \mathcal{D}^\sharp, \sqsubseteq_D, \alpha_D, \gamma_D, \sqcup_D, \sqcap_D, \oplus_D \rangle$, an element $s \in S(\mathcal{D}^\sharp)$ is a sequence $\langle d_1, \ldots, d_n \rangle$ such that $\forall i, d_i \in \mathcal{D}^\sharp, d_i \leq_D d_{i+1}$.

This abstract domain can be seen as a sorted powerset, and its concretization is given by the union of the concretization of the elements of the sequence, $\gamma_S(\langle d_1, d_2, \ldots, d_n \rangle) = \bigcup_{i=1}^n \gamma_D(d_i)$. We define its partial order $\sqsubseteq_S$ as follows:

$$\langle s_1, \ldots, s_n \rangle \sqsubseteq_S \langle s'_1, \ldots, s'_m \rangle \triangleq \forall i \in [1, n], \exists j \in [1, m], s_i \sqsubseteq_D s'_j$$

This states that a sequence s_1 is smaller than another s_2 if all of the atoms of s_1 are included in at least one of the atoms of s_2. Also, the definition of a filtering operator ρ_S for this representation is natural as it is sufficient to call the filtering operator ρ_D for each element of the sequence.

Join for sequences. If we suppose that all ODEs are defined over the same time sequences, then we can define the $\sqcup_S$ join and $\sqcap_S$ meet operators over sequences as a point-to-point extension of their counterparts defined on $\mathcal{D}^\sharp$. However, as this is not necessarily the case, we use for the join operation a merge sort algorithm: given two sequences, this algorithm builds a sequence that is sorted according to the *starting times* of its abstract elements. Using this join operator allows us to merge two sequences in linear time.

Meet for sequences. Symmetrically, we define the meet operator for our sequence by taking into account the *ending times*, given by $max(\pi_t((e)))$ of each abstract element e. The algorithm for the meet operator maintains the ordering of the sequences, and fig. 1 illustrates the order of meet operations of $\mathcal{D}^\sharp$ using this algorithm.

Split and measure. As precised in section 3.2, the abstract domains require to have specific operators for the resolution of CSP in addition to usual operators, namely the *split* operator $\oplus_S$ and the *measure* operator τ_S.

Definition 4.3 (Split operator for $S(\mathcal{D}^\sharp)$)**.** Given a sequence $s = \langle s_1, \ldots, s_n \rangle \in S(\mathcal{D}^\sharp)$, the split operator $\oplus_S : S(\mathcal{D}^\sharp) \to$

$P(S(\mathcal{D}^\sharp))$ is defined as

$$\oplus_S(s) = \begin{cases} \{\langle s' \rangle \mid s' \in \oplus_D(s_1)\}, & \text{when } s = \langle s_1 \rangle \\ \{\langle s_1, \dots s_{\frac{n}{2}} \rangle, \langle s_{\frac{n}{2}+1}, \dots s_n \rangle\}, & \text{otherwise} \end{cases}$$

This operator cuts a sequence into two sub-sequences of equal cardinality, up to 1, corresponding to the first and second half of the initial sequence. In case the sequence contains only one abstract elements, we then use the splitting operator $\oplus_D$ of the underlying domain.

Proposition 4.4. *The operator $\oplus_S$ is a split operator according to definition 3.2.*

Proof. First, either s is a singleton $\langle s' \rangle$, and $|\oplus_S(s)| = |\oplus_D(s')|$ which is finite, or $|\oplus_S(s)| = 2$. Hence, $\forall s \in S(\mathcal{D}^\sharp), |\oplus_S(s)|$ is finite. Also, by definition of $\oplus_S$, we have $\forall \{s'_1, \dots, s'_n\} \in \oplus_S(\{s_1, \dots, s_m\}), \exists i \in [1,n], \exists j \in [1,m]$ such that $s'_i = s_j \implies \exists i \in [1,n], \exists j \in [1,m]$ such that $s'_i \sqsubseteq_D s_j$. Hence by definition of $\sqsubseteq_S$, item 2 is verified. Finally, we have $\forall s = \langle s_1, \dots, s_n \rangle$, $\gamma(s) = \bigcup_{i=1}^n \gamma_D(s_i)$ and $\bigcup\{\gamma(s'_i) \mid s'_i \in \oplus_S(s)\} = \gamma(\langle s_1 \dots s_{\frac{n}{2}} \rangle) \cup \gamma(\langle s_{\frac{n}{2}+1} \cdots s_n \rangle) = \gamma(s)$ hold, hence we have item 3 verified. Therefore, $\oplus_S$ is a split operator. □

The measure operator τ_S has to be designed in such a way that no infinite series of split $\oplus_S$ and reduction ρ are possible.

Definition 4.5 (Size operator for $S(\mathcal{D}^\sharp)$)**.** Given a sequence $s = \langle s_1, \dots, s_n \rangle \in S(\mathcal{D}^\sharp)$ the measure operator $\tau_S : S(\mathcal{D}^\sharp) \to \mathbb{R}^+$ is defined as follows:

$$\tau(s) = \begin{cases} \tau_D(s_1) & \text{when } s = \langle s_1 \rangle \\ +\infty & \text{otherwise} \end{cases}$$

To make sure not to lose precision and to treat each of the atoms in our sequence individually, the search process cannot stop when there are more than one element in a sequence s as $\tau_S(s) = +\infty$. When there is only one element, the exploration can continue as long as this element is large enough according to the measure operator $\mathcal{D}^\sharp$.

Proposition 4.6. *$\oplus_S$ and τ_S are compatible according to definition 3.3*

Proof. $\forall s = \langle s_1 \dots s_n \rangle$, we have at most $log_2(n)+1$ split operations using $\oplus_S$ before reaching a singleton. Hence, assuming the split operator $\oplus_D$ and the measure τ_D are themselves compatible, the split operator $\oplus_S$ and measure operator τ are compatible. □

The sequence abstraction we have defined avoids the combinatorial explosion resulting from using a finite powerset abstraction. For example, it allows us to have a linear time complexity for the join and meet operations which will prove very useful.

4.2 Tree abstraction

We now propose a second abstraction for ODEs. This abstraction is based on the following principle: the solution of an ODE is first approximated as the unions of all time frames, using the `join` operator, then is filtered using a potentially large number of constraints. Our idea is to incorporate additional information into the `join` operation that will speed up the filtering of constraints by proposing a *tree abstract domain* $\mathbb{T}(\mathcal{D}^\sharp)$. This domain can be viewed as a kd-tree in which leaves are defined using the numerical abstract domain $\mathcal{D}^\sharp$ and internal nodes, also called summaries, give information about their subtrees. Our use of summaries is similar to [3], but applied to a CSP framework, in particular, we show that filtering a constraint can be made more efficiently using the summaries information. Moreover, the fact that ODEs are continuous object greatly enhances the relevance of our summaries. Like the sequence abstract domain, the tree abstract domain will allow us to perform the `join` operation in a symbolic way. Definition 4.7 describes the tree abstraction.

Definition 4.7. Given an abstract domain $\langle \mathcal{D}^\sharp, \alpha_D, \gamma_D, \sqcup_D, \sqcap_D, \oplus_D \rangle$, an element $t \in \mathbb{T}(\mathcal{D}^\sharp)$ is either:

- a leaf: $\mathcal{D}^\sharp \to \mathbb{T}(\mathcal{D}^\sharp)$, noted $\mathbf{leaf}(d)$,
- or a union node: $\mathcal{D}^\sharp \times \mathbb{T}(\mathcal{D}^\sharp) \times \mathbb{T}(\mathcal{D}^\sharp) \to \mathbb{T}(\mathcal{D}^\sharp)$ noted $\mathbf{union}(u, t_1, t_2)$ with $u = \mathbf{envelope}(t_1) \sqcup_{\mathcal{D}^\sharp} \mathbf{envelope}(t_2)$

where **envelope** is a function $\mathbb{T}(\mathcal{D}^\sharp) \to \mathcal{D}^\sharp$ such that $\mathbf{envelope}(leaf(d)) = d$, and $\mathbf{envelope}(union(u, t_1, t_2)) = u$. The concretization for this representation is given by the recursive definition:

$$\gamma(t) = \begin{cases} \gamma_D(t'), & \text{when } t = \mathbf{leaf}(t') \\ \gamma(t_1) \cup \gamma(t_2), & \text{when } t = \mathbf{union}(u, t_1, t_2) \end{cases}$$

Here, we exploit the fact that $\gamma(d_1 \sqcup d_2)$ generally strictly contains $\gamma(d_1) \cup \gamma(d_2)$ to provide a summary u of what is contained in the sub-trees t_1 and t_2, in the sense that $\gamma(t_1) \subseteq \gamma_D(u) \wedge \gamma(t_2) \subseteq \gamma_D(u)$. This can be seen as a two-level abstraction, as u is an abstraction of both t_1 and t_2. This allow us to define a fast pre-computation for the meet and the filtering operation. Note that this abstract domain does not improve the precision of the analysis compared to $S(\mathcal{D}^\sharp)$, but is only defined to speed it up. We define also a partial order $\sqsubseteq_T$ for trees as following:

$$t_1 \sqsubseteq_T t_2 \triangleq \forall l \in \mathbf{leaves}(t_1), \exists l' \in \mathbf{leaves}(t_2), l \sqsubseteq_D l'$$

Similarly to sequences, we consider that a tree t_1 is smaller than an other t_2 if each of its leaves are smaller than one of the leaves of t_2. The auxiliary function **leaves** computes the set of all leaves of a tree.

Join operation for $\mathbb{T}(\mathcal{D}^\sharp)$**.** Our representation being able to encode exactly disjunctions, we define the join operator as

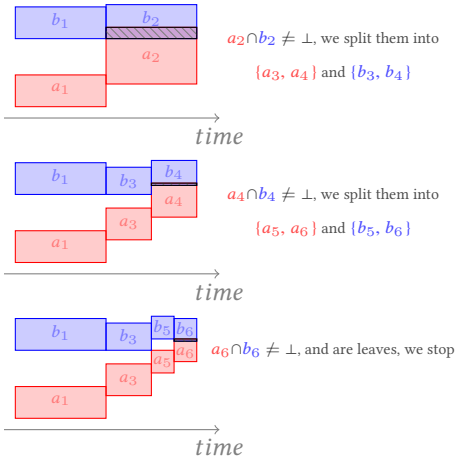

Figure 2. Order of meet operations of during the intersection of two abstract elements of $T(\mathcal{D}^\sharp)$

following:

$$t_1 \sqcup_T t_2 = \textbf{union}(\textbf{envelope}(t_1) \sqcup_D \textbf{envelope}(t_2), t_1, t_2)$$

The greater the intersection of two elements is, the more accurate the summary is. Because of the continuous nature of ODEs, this property is particularly interesting as two successive elements always intersect.

Meet operation for $\mathbb{T}(\mathcal{D}^\sharp)$**.** For the definition of the meet operator for trees, we now exploit the summaries to define a pre-computation. If two summaries do not intersect, then the whole corresponding trees do not either. Figure 2 illustrates the order of meet operations on an example.

4.2.1 Split and measure.

To embed our abstract domain in the constraint solving framework, we must define a split operation $\oplus$, along with a measure function τ and a constraint filtering operator ρ. The operator $\oplus$, given by definition 4.8, performs a symbolic cut when the tree is a **union** node, and otherwise uses $\oplus_D$.

Definition 4.8 (Split operator for $T(\mathcal{D}^\sharp)$)**.**

$$\oplus(t) = \begin{cases} \{\textbf{leaf}(d') \mid d' \in \oplus_D(d)\}, & \text{when } t = \textbf{leaf}(d) \\ \{t_1, t_2\} & \text{when } t = \textbf{union}(u, t_1, t_2) \end{cases}$$

Proposition 4.9. *The operator $\oplus$ is a split operator according to definition 3.2*

Proof. Given a tree t, assuming $\oplus_D$ respects definition 3.2, in case $t = \textbf{leaf}(l)$, $\oplus_T$ also trivially respects definition 3.2. in case $t = \textbf{union}(u, t_1, t_2)$, we have $|\oplus_T(t)| = 2$, and hence, is finite. Also, by definition of the function **leaves**, $\forall d \in \textbf{leaves}(t_1) \cup \textbf{leaves}(t_2), d \in \textbf{leaves}(t)$ holds, which implies that $\forall d \in \textbf{leaves}(t_1) \cup \textbf{leaves}(t_2), \exists d' \in \textbf{leaves}(t), d \sqsubseteq_D d'$. Hence, we have $\forall t' \in \oplus_T(t), t' \sqsubseteq_T t$. Finally, we have $\textbf{leaves}(t) = \bigcup\{\textbf{leaves}(t') \mid t' \in \oplus_T(t)\}$. Hence, by defintion of γ_T, $\gamma_T(t) = \bigcup\{\gamma_T(t') \mid t' \in \oplus_T(t)\}$. Therefore, $\oplus_T$ is a split operator according to definition 3.2. □

We now define the size function $\tau : \mathbb{T}(D) \to \mathbb{R}$ for trees, which is given in definition 4.10.

Definition 4.10 (Size operator)**.**

$$\tau(e) = \begin{cases} \tau_D(d), & \text{when } e = \textbf{leaf}(d) \\ +\infty & \text{when } e = \textbf{union}(u, t_1, t_2) \end{cases}$$

The splitting is done as long as there are disjunctions in the representation and when a leaf is reached, a branching to (τ_D) is performed.

Proposition 4.11. *The split operator $\oplus_T$ and the size operator τ_T are compatible for any reductive operator ρ, according to definition 3.3*

Proof. Given a leaf node $t = \textbf{leaf}(l)$, a reductive operator on trees $\rho_T : \mathbb{T}(\mathcal{D}^\sharp) \times \mathcal{C} \to \mathbb{T}(\mathcal{D}^\sharp)$, and a reductive operator on $\mathcal{D}^\sharp$, $\rho_D : \mathcal{D}^\sharp \times \mathcal{C} \to \mathcal{D}^\sharp$, by definition of $\oplus_T$ and τ_T: $\forall k, \tau_T((\oplus_T \circ \rho)^k(t)) = \tau_{\mathcal{D}^\sharp}((\oplus_{\mathcal{D}^\sharp} \circ \rho_D)^k(l))$. Hence, assuming $\oplus_{\mathcal{D}^\sharp}$ and $\tau_{\mathcal{D}^\sharp}$ are themselves compatibles, τ_T and $\oplus_T$ are compatible. Given $t = \textbf{union}(u, t_1, t_2)$, we have at most h split operations, where h is the height of the tree before reaching a leaf node. Hence, no infinite sequence of splits and constraint filtering can exist, and $\oplus_T$ and τ_T are compatible. □

4.2.2 Constraint filtering.

Finally, note that fast precomputation is not only available for the meet operation but also for the filtering of a constraint:

Definition 4.12. Given a tree t and a constraint c, the filtering operator $\rho_T : \mathbb{T}(\mathcal{D}^\sharp) \to \mathcal{C} \to \mathbb{T}(\mathcal{D}^\sharp)$ is such that:

$$\rho(t, c) = \begin{cases} \textbf{leaf}(\rho_D(t')) & \text{when } t = \textbf{leaf}(t') \\ \bot & \text{when } t = \textbf{union}(u, t_1, t_2) \wedge \\ & \quad \rho(u) = \bot_D \\ \sqcup_T(\rho(t_1), \rho(t_2)) & \text{when } t = \textbf{union}(u, t_1, t_2) \wedge \\ & \quad \rho(u) \neq \bot_D \end{cases}$$

Instead of filtering a constraint for each atom of a disjunction, we do it first for their summary. If we can prove that a summary u violates the constraint c, *i.e.*, $\forall i \in \gamma_D(u), \neg c(i)$ (resp. $\forall i \in \gamma_D(u), c(i)$), then the child elements will also violate it. In the case where we can not draw a definitive conclusion, we propagate the filtering towards the sub-trees.

Proposition 4.13. *The filtering operator ρ contracts and is complete, i.e:*

- $\rho_T(t, c) \sqsubseteq_T t$ *(contraction)*
- $\forall i \in \gamma_T(t), c(i) \implies i \in \gamma_T(\rho(t))$ *(completeness)*

The contraction property ensures that values are only removed from the abstract element, and the completeness property guarantees that no solution is removed from it.

Proof. Given a tree t, in case $t = \mathbf{leaf}(t')$, $\rho(t) = \mathbf{leaf}(\rho_D(t'))$, in which case both the contraction property and the completeness property are respected, assuming ρ_D respects these itself. In case $t = \mathbf{union}(u, t_1, t_2)$,

- either $\rho_D(u, c) = \bot_D$, in which case $\rho_T(t) = \bot_T$, and the contraction property is trivially verified. Also, as $\gamma(t_1) \subseteq \gamma(u)$ and $\gamma(t_1) \subseteq \gamma(u)$, then $\rho_D(u, c) = \bot_D \implies \rho_T(t_1, c) = \rho_T(t_2, c) = \bot_T$. Therefore no solution is lost and the completeness property is verified.
- either $\rho_D(u, c) \neq \bot_D$, then suppose that $\rho(t_1) \sqsubseteq_T t_1$ and $\rho(t_2) \sqsubseteq_T t_2$, then, by induction, we have $\sqcup_T(\rho(t_1), \rho(t_2)) \sqsubseteq_T t$, and the contraction property is respected. Similarly, suppose that $\forall i \in \gamma_T(t_1), c(i) \implies i \in \gamma_T(\rho(t_1))$ and $\forall j \in \gamma_T(t_2), c(j) \implies j \in \gamma_T(\rho(t_2))$ then, as $\gamma_T(t) = \gamma_T(t_1) \cup \gamma_T(t_2)$ the completeness property is verified by induction.

□

The tree abstraction that have been defined in this section exploits the continuity aspect of ODEs solution to propose a fast pre-computation for the filtering of a constraint and the meet operation. It can thus be seen as an incremental powerset, that gradually augments its precision, starting from the precision of a base abstract domain D to the precision of its powerset $P(D)$ if needed only. This will greatly speed-up the solving process in most cases.

5 Use case and benchmarks

We show the interest of our approach on a real-world application: the trajectory validation for a swarm of Unmanned Aerial Vehicles (UAVs). In this context, trajectory planning consists in associating to each UAV a trajectory by taking into account the dynamics of the vehicle, the environment the fleet evolves in, and uncertainties about informations such as the position or orientation of the UAV, which are bounded. We consider here that all UAVs are described by the same dynamics, explicited by the ODE presented in eq. (5):

$$\mathcal{S}_i = \begin{cases} \dot{\mathrm{X}}_i = \begin{pmatrix} \dot{x}_i \\ \dot{y}_i \\ \dot{z}_i \end{pmatrix} = \begin{pmatrix} v_i \cos\phi_i \cos\theta_i \\ v_i \sin\phi_i \cos\theta_i \\ v_i \sin\theta_i \end{pmatrix} \\ \mathrm{X}_i(0) \in [\mathrm{X}_0] \end{cases} \tag{5}$$

with $\mathrm{X}_i = (x_i, y_i, z_i)^T$ the state vector representing the position of the i^{th} UAV and $\mathrm{u}_i = (v_i, \phi_i, \theta_i)^T$ its control vector consisting in its velocity v_i, heading angle ϕ_i and track angle θ_i. Note that this physical model is not intended to be realistic, (*e.g.*, UAVs are considered as points) but to serve as a basic model for trajectory generation, the main interest of this application being the study of trajectory validation.

The safety property we aim to prove as we deal with UAV swarms is that trajectories do not intersect with each others. However, verifying this property can be costly, especially as the number of UAV grows as we have to prove for each UAV that it does not collide with all others. Consider a fleet of N UAVs, g a function associating to each UAV its trajectory, and d, the duration of the simulation, we want to prove that:

$$\forall i, j \in \{1, \ldots, N\}, \forall t \in [0, d], i = j \vee g_i(t) \cap g_j(t) = \emptyset$$

If a collision is detected, either there is indeed a collision between the concrete trajectories, or the collision on abstract trajectories is a false alarm due to the over-approximation of the solutions of the ODEs. In order to minimize the risk of obtaining such false alarms, it is crucial to aggressively sub-divide $[0, d]$ into small intervals. Our use case can be expressed as a CSP without loss of generality. We simply make the assumption that the space the fleet evolve in is bounded (although potentially big) and we limit the duration of the flight to a given time. These assumptions guarantee that the search-space is finite and that our solving method will always terminate.

A trajectory r can be as a disjunction of predicates $r = (p_1 \wedge m_1) \vee (p_2 \wedge m_2) \ldots \vee (p_n \wedge m_n)$ where each p_i represent the position (e.g. $x \in [10, 12] \wedge y \in [0, 2] \wedge z \in [0, 10]$) in space of a given UAV during a moment m_i. It is to be understood as the following property: *the UAV is either at point p_1 during the time frame m_1, or at position p_2 during the time frame m_2, etc.* This property is always true, as at least one of its atoms will be true. When dealing with a fleet of n UAVs, we have to define n trajectories $r_1, \ldots r_n$ and the constraint corresponding to the possible collisions for each UAV with the others, as expressed by Eq. 6.

$$\begin{aligned} (r_1 \wedge r_2) \vee (r_1 \wedge r_3) \vee \ldots (r_1 \wedge r_n) \vee \\ (r_2 \wedge r_3) \vee \ldots (r_2 \wedge r_n) \vee \\ \ldots \vee \\ (r_{n-1} \wedge r_n) \end{aligned} \tag{6}$$

There is a collision if two constraints representing two trajectories are true at the same time. If there is no solution to the CSP, then the trajectories do not collide with each other. If one or several solution is found, then the trajectories are invalid. The information about the collisions can then be used to replan part or all of the trajectories.

5.1 Experimental results

The two abstract domains presented in the previous sections have been implemented and integrated into the AbSolute constraint solver. We demonstrate their effectiveness on a set of CSPs corresponding to different instances of the use-case presented above. We modeled the physical system of each drone and carried out the flight simulation on different number of iterations from 1000 to 64000. We have repeated this protocol for swarms of different sizes N, from 2 to 20 UAVs per swarm. This makes the size of the constraints to be solved fairly large, which will be necessary to illustrate the good scaling of our approach. We then ran the solver on the

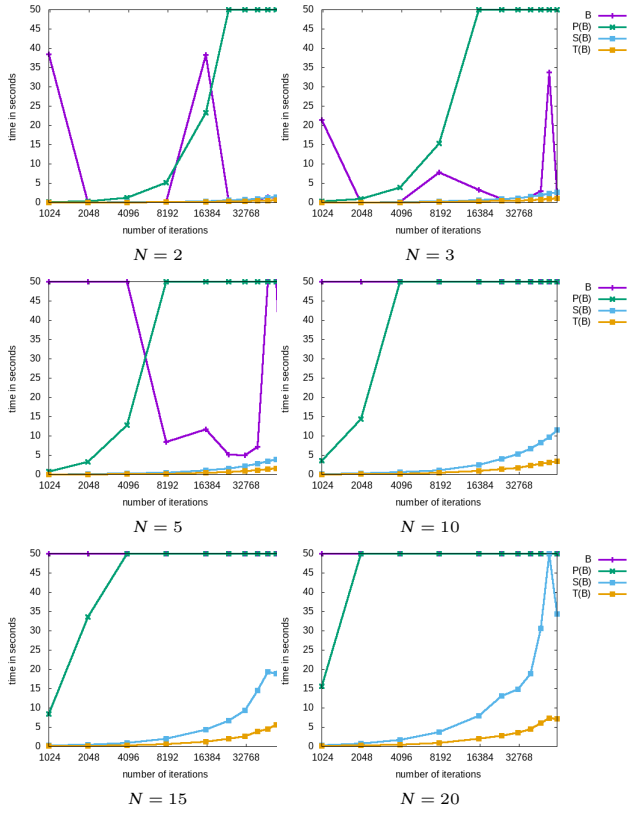

Figure 3. Solving time according to the number of iterations for swarms of 2, 3, 5, 10, 15 and 20 UAVs

generated examples with different configurations. Solver's output is either SAT, meaning the problem admits at least one solution (there is a collision), or UNSAT, meaning it does not (the trajectories are valid). All the domains compared lead to the same conclusions concerning the satisfiability of the problem being tested, some more rapidly than others, and this is why we exhibit here only their execution times. Experiments were run on a machine equipped with an Intel Core i7-4810MQ CPU at 2.80GHz and 16GB RAM.

Results. Figure 3 shows that the two abstractions we have defined outperform the default solving of AbSolute, which uses only boxes. The use of boxes allows the solver on some examples to resolve the CSP quite quickly, especially in the absence of collision. However, as soon as the trajectories are close to each other or worse, intertwined, a single box is no longer precise enough to draw a conclusion as to the satisfiability of the CSP. In this case the solver must resort to several split steps to regain precision, which ultimately becomes very expensive and makes it time out on most examples where the size of the swarm grows. Moreover, our abstract domains perform better than a simple powerset, which is able to encode precisely disjunctions but is outscaled by our methods. Indeed, using a powerset presents the opposite problem than using boxes: the abstract domain is precise enough to prove the satisfiability or the infeasibility of the problem in most cases, but too expensive to be able to do it before timing out so much so that it makes it slower than boxes in most cases. The two abstractions that we have developed bypass these two problems. The management of disjunctions is done symbolically, which permits a good precision and incorporating the assumptions specific to the nature of ODEs into our representations allows us to obtain more efficient operations in most cases. Our sequence abstraction exploits the chronological ordering of ODEs to perform a more efficient intersection operation than with a simple powerset, and our tree abstraction offers the possibility of having a quick pre-computation for the intersection and the filtering operations. This provides an incremental precision-cost ratio and gives good results in practice. Also, even though $T(\mathbb{B})$ always perform better than $S(\mathbb{B})$ on this benchmark, they are not comparable and $T(\mathbb{B})$ could behave less efficiently if the trajectories were very deeply intertwined, in which case the precomputation using the summary would be ineffective.

6 Conclusion

We have shown that it is possible to validate cyber-physical systems using techniques from different areas, such as Constraint Programming, Abstract Interpretation and Interval Analysis. Our work can be summarized in two parts: a) the correct over-approximation of the reachability states of some physical systems described by ODEs, and b) the resolution of constraint satisfaction problems implying such systems. For the latter purpose, we have defined two abstract domains able to take advantage of the characteristics of ODE solutions, namely their sequential and their continuous nature. We have demonstrated the usefulness of our techniques on realistic application examples implying the absence of collisions between the trajectories of a UAV swarm. The "no collision" property was expressed as a CSP over ODE, and its resolution highlighted the effectiveness of our abstract domain $\mathbb{T}(\mathcal{D}^\sharp)$. Both abstract domains were implemented as AbSolute plugins and benchmarked over real examples.

The work we have done may be deepened in several ways: we made the choice of using for elements of $\mathbb{T}(\mathcal{D}^\sharp)$ the same numeric representation for leaves and summary nodes. It could be effective to use some more expressive abstract domains for the nodes than for the leaves (*e.g.*, relational representation like polyhedra for nodes and intervals for leaves) to minimize the loss of precision due to join operation, and thus further improve the capabilities of our tree abstract domain. Another choice we have made is not to modify the partitioning of time defined by our numerical integration method. However, it might be interesting to divide certain time windows to *align* all the ODEs on the same partitioning, or to merge certain elements when an exact union can be made to improve performance. These points will be studied in the future. Moreover, the techniques we

have developed are intended for constraint satisfaction problems, and not optimization problems. It could be interesting to extend our methods on optimization problems, *e.g.* in our UAV swarm application, only the first collision is interesting, as the simulation is not valid anymore after it. Hence, this problem becomes an optimization problem, where the time is the value to minimize.

Acknowledgment

This work has been partially supported by a DGA AID project.

References

[1] Julien Alexandre dit Sandretto and Alexandre Chapoutot. 2016. Validated Explicit and Implicit Runge–Kutta Methods. *Reliable Computing* 22, 1 (07 2016), 79–103.

[2] Julien Alexandre dit Sandretto, Alexandre Chapoutot, and Olivier Mullier. 2018. *Constraint-Based Framework for Reasoning with Differential Equations.* Springer International Publishing, Cham, 23–41. https://doi.org/10.1007/978-3-319-98935-8_2

[3] Anna Becchi and Enea Zaffanella. 2019. Revisiting Polyhedral Analysis for Hybrid Systems. In *Static Analysis*, Bor-Yuh Evan Chang (Ed.). Springer International Publishing, Cham, 183–202.

[4] Olivier Bouissou and Matthieu Martel. 2005. Static Analysis by Abstract Interpretation of Hybrid Systems.

[5] Olivier Bouissou and Matthieu Martel. 2006. GRKLib: A guaranteed Runge Kutta library. 8–8. https://doi.org/10.1109/SCAN.2006.20

[6] Olivier Bouissou and Matthieu Martel. 2008. Abstract Interpretation of the Physical Inputs of Embedded Programs. 37–51. https://doi.org/10.1007/978-3-540-78163-9_8

[7] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. 2013. Flow*: An Analyzer for Non-linear Hybrid Systems. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 258–263.

[8] P. Cousot and R. Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, Los Angeles, California, 238–252.

[9] P. Cousot and R. Cousot. 1979. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, San Antonio, Texas, 269–282.

[10] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. 2010. A Scalable Segmented Decision Tree Abstract Domain. In *Time for Verification, Essays in Memory of Amir Pnueli (Lecture Notes in Computer Science)*, Z. Manna and D. Peled (Eds.), Vol. 6200. Springer-Verlag, New York, United States, 72–95. https://hal.inria.fr/inria-00543632

[11] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 84–96.

[12] Jorge Cruz and Pedro Barahona. 2003. Constraint Satisfaction Differential Problems. 259–273. https://doi.org/10.1007/978-3-540-45193-8_18

[13] Andreas Eggers, Martin Fränzle, and Christian Herde. 2009. Application of Constraint Solving and ODE-Enclosure Methods to the Analysis of Hybrid Systems. *AIP Conference Proceedings* 1168 (09 2009). https://doi.org/10.1063/1.3241327

[14] Gilberto Filé and Francesco Ranzato. 1999. The powerset operator on abstract interpretations. *Theoretical Computer Science* 222, 1 (1999), 77–111. https://doi.org/10.1016/S0304-3975(98)00007-3

[15] Alexandre Goldsztejn, Olivier Mullier, Damien Eveillard, and Hiroshi Hosobe. 2010. Including Ordinary Differential Equations Based Constraints in the Standard CP Framework. In *Principles and Practice of Constraint Programming – CP 2010*, David Cohen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 221–235.

[16] Arie Gurfinkel and Sagar Chaki. 2010. Boxes: A Symbolic Abstract Domain of Boxes. In *Static Analysis*, Radhia Cousot and Matthieu Martel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 287–303.

[17] Bertrand Jeannet and Antoine Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Proceedings of the 21th International Conference Computer Aided Verification (CAV 2009)*.

[18] T. P. Jensen. 1992. Disjunctive strictness analysis. In *[1992] Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*. 174–185. https://doi.org/10.1109/LICS.1992.185531

[19] Hugo Joudrier and Khaled Hadj-Hamou. 2015. Guaranteed Global Deterministic Optimization and Constraint Programming for Complex Dynamic Problems. In *21th International Conference on Principles and Practice of Constraint Programming (Doctoral Program Proceedings)*. Cork, Ireland. https://hal.archives-ouvertes.fr/hal-01244426

[20] Rudolf J Lohner. 1992. Computation of guaranteed enclosures for the solutions of ordinary initial and boundary value problems. In *Institute of mathematics and its applications conference series*, Vol. 39. Oxford University Press, 425–425.

[21] Antoine Miné. 2001. The Octagon Abstract Domain. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01) (WCRE '01)*. IEEE Computer Society, Washington, DC, USA, 310–. http://dl.acm.org/citation.cfm?id=832308.837141

[22] Antoine Miné, Jason Breck, and Thomas Reps. 2016. An Algorithm Inspired by Constraint Solvers to Infer Inductive Invariants in Numeric Programs. In *25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016 (Programming Languages and Systems)*, Peter Thiemann (Ed.), Vol. 9632. Springer, Eindhoven, Netherlands, 560–588. https://doi.org/10.1007/978-3-662-49498-1_22

[23] Ugo Montanari. 1974. Networks of Constraints: Fundamental Properties and Applications to Picture Processing. *Information Science* 7, 2 (1974), 95–132.

[24] R. E. Moore. 1966. *Interval Analysis.* Prentice Hall.

[25] Ramon E. Moore, R Baker Kearfott, and Michael J Cloud. 2009. *Introduction to interval analysis.* Siam.

[26] N. S. Nedialkov, K. R. Jackson, and G. F. Corliss. 1999. Validated solutions of initial value problems for ordinary differential equations. *Appl. Math. Comput.* 105, 1 (1999), 21–68.

[27] Marie Pelleau, Antoine Miné, Charlotte Truchet, and Frédéric Benhamou. 2013. A Constraint Solver Based on Abstract Domains. *JFPC 2013 - Neuviemes Journees Francophones de Programmation par Contraintes*. https://doi.org/10.1007/978-3-642-35873-9_26

[28] Marie Pelleau, Charlotte Truchet, and Frédéric Benhamou. 2014. The octagon abstract domain for continuous constraints. *Constraints* 19, 3 (Jan. 2014), 309–337. https://doi.org/10.1007/s10601-014-9162-x

[29] Olivier Ponsini, Claude Michel, and Michel Rueher. 2012. Refining Abstract Interpretation Based Value Analysis with Constraint Programming Techniques. In *Principles and Practice of Constraint Programming*, Michela Milano (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 593–607.

[30] Sriram Sankaranarayanan, Thao Dang, and Franjo Ivančić. 2008. Symbolic Model Checking of Hybrid Systems Using Template Polyhedra. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 188–202.

[31] Sriram Sankaranarayanan, Franjo Ivančić, Ilya Shlyakhter, and Aarti Gupta. 2006. Static Analysis in Disjunctive Numerical Domains. In *Static Analysis*, Kwangkeun Yi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 3–17.