Towards a Generic Interval Solver for Differential-Algebraic CSP

Simon Rohou[1], Abderahmane Bedouhene[2], Gilles Chabert[3], Alexandre Goldsztejn[4], Luc Jaulin[1], Bertrand Neveu[2], Victor Reyes[5], and Gilles Trombettoni[5]

[1] Lab-STICC, ENSTA Bretagne, CNRS, France
[2] LIGM, Ecole des Ponts, Univ. Gustave Eiffel, CNRS, Marne-la-Vallée, France
[3] IRT Jules Verne, LS2N, France
[4] LS2N, CNRS, France
[5] LIRMM, University of Montpellier, CNRS, France

Abstract. In this paper, we propose an interval constraint programming approach that can handle the *differential-algebraic CSP* (DACSP), where an instance is composed of real and functional variables (also called dynamic variables or trajectories) together, and differential and/or "static" numerical constraints among those variables. Differential-Algebraic CSP systems can model numerous real-life problems occurring in physics, biology or robotics. We introduce a solver, built upon the `Tubex` and IBEX interval libraries, that can rigorously approximate the set of solutions of a DACSP system. The solver achieves temporal slicing and a tree search by splitting trajectories domains. Our approach provides a significant step towards a generic interval CP solver for DACSP that has the potential to handle a large variety of constraints. First experiments highlight that this solver can tackle interval Initial Value Problems (IVP), Boundary Value Problems (BVP) and integro-differential equations.

1 Introduction

Differential-Algebraic CSP (DACSP) systems include real variables, functional variables (also called trajectories or dynamical variables) describing the dynamics of the system, and differential and/or "static" numerical constraints among those variables. Because they are at the core of so many applications, such as biological systems, mechanism dynamics, astronomy, robotics, control, a lot of work has been dedicated to solving specific subclasses of the DACSP. Most of the approaches follow a probabilistic approach and are limited to linear constraints with Gaussian errors [17, 35].

There are a number of advantages to using interval methods for handling dynamical and/or static systems. They can manage nonlinear constraints and approximate the solutions rigorously, whatever the uncertainties on the parameters (*e.g.*, uncertainties related to measurements or to inaccurate physical models). Bounded intervals are used to characterize these uncertainties, as well as the errors due to operations over floating-point numbers.

With Jaulin *et al.* works [15, 16], a significant step has been made towards a declarative constraint programming approach for dynamical systems. Contrary to dominant constraint approaches dealing with differential equations such as [8, 30, 37], the trajectories are viewed as variables of the CSP. They consider trajectories as variables, differential equations as constraints and *tubes* as domains. The solution and the domain are given by a tube representing a set of possible trajectories (see Fig. 1). This increases the level of abstraction and simplifies the formalization of the problem. For estimating a trajectory, a set of *contractors*, similar to *propagators* in CP, are applied to filter (contract) the bounds of the domain/tube. A benefit of the contractor programming approach [3] is the variety of dynamical systems that can be handled. More recently, the `Tubex` library [32] has provided data structures for representing tubes and a catalogue of contractors, similarly to the catalogue of propagators available in CP solvers [29]. The user defines a sequence of contractors for modeling his problem. The set of contractors is applied iteratively until a quasi fixpoint in terms of contraction is reached. For instance, the resulting framework has been recently applied to actual data for autonomous robot localization. [31] describes the problem of localizing an underwater robot. Its evolution is depicted by an Ordinary Differential Equation (ODE) and bounded measurements, but the initial condition (position) of the system is unknown. During the mission, its sonar detects indistinguishable rocks on the seabed, that all look alike but are known to belong to a discrete point map embedded beforehand. The map of rocks positions is also modeled as a constraint. The constraint propagation approach is able to merge all data coming from the robot evolution and rocks observations. The identity of the rocks is finally associated to items in the map, and the trajectory of the robot is accurately estimated. This application highlights how a complex problem involving discrete, continuous and differential constraints can be solved easily following a CP/contractor approach.

Contractor programming is relevant when the problem is defined by heterogeneous constraints, provided they are redundant and numerous enough to enable the contraction phase alone to solve the problem.

Contributions. In this paper, we introduce a generic solver using the `Tubex` library that can handle a DACSP instance made of differential constraints and/or static continuous constraints. The numerical constraints relate real variables that either represent states of the trajectories at given instants or are independent from the dynamics (*e.g.*, the rocks positions in the previous example). Compared to the `Tubex` approach alone, the solver is endowed with an operator that can perform a choice point by splitting (bisecting) a tube in two at a chosen time. Thus, a tree search can accurately estimate distinct trajectories (for problems with several solutions) and can better handle several hard problems. To our knowledge, no other tool for solving dynamical systems performs a tree search.

Our solver can manage most of the contractors available in the literature. On the one hand, several contractors coming from CP, *e.g.*, 3B [22] and CID [36], are included naturally to improve the pruning of domains of functional variables, *i.e.*, tubes. On the other hand, our generic framework enables to wrap existing solvers dedicated to specific types of (differential) constraints into contractors. These contractors efficiently reduce domains by taking into account space and time dependencies. In particular, we show in the experiments the benefits of using the contractor `CtcVnode` built upon the state-of-the-art VNODE guaranteed integration IVP solver [27, 28].

This generic CP framework allows separating the problem description, which includes here all combinations of differential and algebraic problems, from its resolution. We also wanted to solve difficult standard problems coming from numerical analysis, *e.g.*, BVP for integro-differential equations, that are out of the scope of previous CP/ODE frameworks and remain today difficult to solve in the numerical analysis framework.

Related Work. VNODE [28], CAPD [18], COSY [30] and `DynIbex` [4] are state of the art interval analysis solvers dedicated to IVPs.[6] They are fully relevant for determining the guaranteed solution of a system at a final time, that has crucial applications such as the position determination of celestial bodies in astronomy [39] or to characterize chaotic attractors [37]. They use different algorithms to reliably simulate the initial information over time. In particular, the VNODE tool used in our solver combines a high-order interval Taylor form to integrate the state from an instant to a next one, and a step limiting the wrapping effect implied by interval calculation: it encloses the solution at the discrete times by an envelope sharper than a box, such as rotated boxes, zonotopes or polygons [23].

A BVP is generally defined by an ODE, but the trajectory is not entirely determined at the initial or final times, which prevents a solving algorithm from propagating (integrating) the information from the known state to the rest of the trajectory. Instead, static constraints are defined on specific states (at specified instants). To deal with BVPs, the shooting method [12] is a sampling-and-optimization algorithm that runs several integration processes from the initial state while trying to minimize the distance to a solution satisfying the ODE. This makes the approach incomplete and unable to determine several solutions, if any. Instead, our rigorous and deterministic solver can explore the whole search space and isolate distinct solutions.

The constraint programming community has contributed to dynamical systems. Janssen *et al.* propose a strategy [8] dedicated to interval IVPs that has similarities with VNODE or CAPD. The integration step between two consecutive states (instants) used a box consistency algorithm [1] to better contract the output state. A first attempt to create a constraint language extended to ODEs was made a few years later [6, 7]. The language considers the entire ODE as a whole, but the unknown function has a special status that prevents direct manipulations: any constraint involving the unknown trajectory is managed by a specific operator. The fact that the function is bounded in an interval requires the introduction of an ad-hoc constraint called minimum/maximum restriction.

[6] An Initial Value Problem is composed of an ODE and an initial condition. Numerical integration propagates the initial value through the whole trajectory by integrating the evolution function of the ODE.

The link with real variables is also achieved via an ad-hoc constraint called value restriction. This language does not have the level of genericity targeted by our solver, where the concept of trajectories appears at the same level as other variables and ODEs are syntactic constructions among others. In our solver, we will rather use the $\mathcal{C}_{\text{eval}}$ contractor [33] to handle value restriction constraints.

Although [11] improves [6] in both modeling possibilities and solving efficiency, it is still restricted to ODE constraints relating solutions values at specified times. Finally, our model can accept various differential constraints (like an integro-differential equation) and static constraints such as distance relations between states at different instants, as long as the corresponding contractors exist.

Outline. Section 2 introduces the notations used in the paper and the background useful to understand the following sections. Sections 3 and 4 detail our DACSP solver and its different procedures and parameters: contractors, choice point heuristics, *etc.* Section 5 reports first experimental results obtained on interval IVPs, but also integro-differential equations and BVPs.

2 Background and Notations

We first provide some background about intervals, inclusion functions and contraction. We then present several differential constraints and interval techniques adapted to dynamical systems.

2.1 Intervals

Contrary to numerical analysis methods that work with single values, interval methods can manage sets of values enclosed in intervals. Interval methods are known to be particularly useful for handling nonlinear constraint systems.

Definition 1 (Interval, box, box width, box hull)
An interval $[x_i] = [\underline{x_i}, \overline{x_i}]$ defines the set of reals x_i such that $\underline{x_i} \leq x_i \leq \overline{x_i}$. $\mathbb{IR}$ denotes the set of all intervals. A box $[\boldsymbol{x}]$ denotes a Cartesian product of intervals $[\boldsymbol{x}] = [x_1] \times ... \times [x_n]$. The size, width or diameter of a box $[\boldsymbol{x}]$ is given by $w([\boldsymbol{x}]) \equiv \max_i(w([x_i]))$ where $w([x_i]) \equiv \overline{x_i} - \underline{x_i}$. The hull of boxes approximates the union operator. It returns the smallest box enclosing all the boxes hulled.

Interval arithmetic [26] has been defined to extend to $\mathbb{IR}$ the usual mathematical operators over $\mathbb{R}$ For instance, the interval sum is defined by $[x_1] + [x_2] = [\underline{x_1} + \underline{x_2}, \overline{x_1} + \overline{x_2}]$. When a function $\boldsymbol{f}$ is a composition of elementary functions, an *inclusion function* $[\boldsymbol{f}]$ of $\boldsymbol{f}$ must be defined to ensure a conservative image computation. There are several inclusion functions. The *natural* inclusion function of a real function f corresponds to the mapping of f to intervals using interval arithmetic. For instance, the natural inclusion function $[f]_N$ of $f(x) = x(x+1)$ in the domain $[x] = [0,1]$ computes $[f]_N([0,1]) = [0,1] \cdot [1,2] = [0,2]$. Another inclusion function is based on an interval Taylor form [14].

Interval arithmetics can be used for solving the *numerical CSP* (NCSP), *i.e.* finding solutions to an NCSP instance $P = (\boldsymbol{x}, [\boldsymbol{x}], \boldsymbol{c})$, where $\boldsymbol{x}$ is an n-set of variables taking their real values in the domain $[\boldsymbol{x}]$ and $\boldsymbol{c}$ is an m-set of numerical constraints using operators like $+$, $-$, $\times$, a^b, exp, log, sin, *etc.* NCSP solvers, like GlobSol [19], Gloptlab [9], RealPaver [13] or Ibex [2] to name a few, follow a Branch and Contract method to solve an NCSP. The branching operation subdivides the search space by recursively bisecting variable intervals into two subintervals and exploring both sub-boxes independently. The combinatorial nature of this tree search is not always observed thanks to the *contraction* (filtering) operations applied at each node of the search tree. Informally, a contraction applied to an NCSP instance can reduce the domain without losing any solution. A contractor used in this paper is the well-known `HC4-revise` [1, 25], also called *forward-backward*. This contractor handles a single numerical constraint and obtains a (generally non optimal [5]) contracted box including all the solutions of that constraint. To contract a box w.r.t. an NCSP instance, the HC4 algorithm performs a (generalized) AC3-like propagation loop applying iteratively the HC4-Revise procedure on each constraint individually until a quasi fixpoint is obtained in terms of contraction.

3B-consistency [22] and CID-consistency [36] are two other stronger consistencies, enforced on an NCSP, that are exploited by our solver. The corresponding 3B and CID algorithms should call their Var3B or VarCID procedure on all the NCSP variables for enforcing the 3B or CID consistency. In practice however the algorithms implemented apply these procedures on a subset of the variables to get a better tradeoff between contraction and performance. VarCID splits a variable interval in k subintervals, and runs a contractor, such as HC4, on the corresponding sub-boxes. The k sub-boxes contracted are finally hulled. Var3B is somehow a dual operator that tries to remove subintervals at the bounds of a variable interval. If a contraction, like HC4, applied to a sub-box, where the interval is replaced by a subinterval at a bound, leads to an empty domain, then it proves that the subinterval contains no solution and can be removed safely from the variable domain.

2.2 Trajectories and Tubes

A trajectory, denoted $\boldsymbol{x}(\cdot) = (x_1(\cdot), .., x_n(\cdot))$, is a function from $[t_0, t_f] \subset \mathbb{R}$ to $\mathbb{R}^n$. The input (argument) of $\boldsymbol{x}(\cdot)$ is named *time* in this article (and denoted $\cdot$ or t) while the output (image) is called *state*.

A tube $[\boldsymbol{x}](\cdot)$ is the interval counterpart of a trajectory and is defined as an envelope over the same temporal domain $[t_0, t_f]$. The concept appeared in [10,20] in the context of ellipsoidal estimations. In our solver, it is used as a domain on which we apply operations of contractions and bisections. We employ them as intervals of trajectories, which is consistent with the aforementioned tools.

Hence, we will use the definition given in [21] where a tube $[\boldsymbol{x}](\cdot) : [t_0, t_f] \to \mathbb{IR}^n$ is an interval of two trajectories $[\underline{\boldsymbol{x}}(\cdot), \overline{\boldsymbol{x}}(\cdot)]$ such that $\forall t \in [t_0, t_f],\ \underline{\boldsymbol{x}}(t) \leqslant \overline{\boldsymbol{x}}(t)$. We also consider empty tubes that depict an absence of solutions. A trajectory $\boldsymbol{x}(\cdot)$ belongs to the tube $[\boldsymbol{x}](\cdot)$ if $\forall t \in [t_0, t_f],\ \boldsymbol{x}(t) \in [\boldsymbol{x}](t)$. Fig. 1 illustrates a one-dimensional tube enclosing a trajectory $x(\cdot)$.

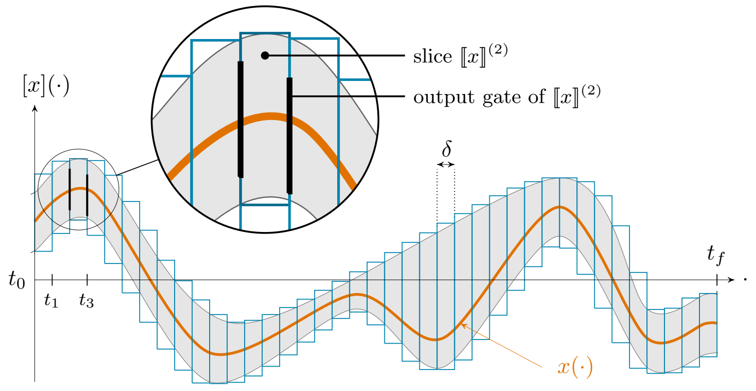

Fig. 1. A one-dimensional tube $[x](\cdot)$, interval of two functions $[\underline{x}(\cdot), \overline{x}(\cdot)]$, enclosing a random trajectory $x(\cdot)$ depicted in orange. The tube is numerically represented by a set of δ-width slices illustrated by blue boxes.

Our choice is to represent numerically a tube as a set of boxes corresponding to temporal slices. More precisely, an n-dimensional tube $[\boldsymbol{x}](\cdot)$ with a sampling time $\delta > 0$ is implemented as a box-valued function which is constant for all t inside intervals $[k\delta, k\delta + \delta]$, $k \in \mathbb{N}$. The box $[k\delta, k\delta + \delta] \times [\boldsymbol{x}](t_k)$, with $t_k \in [k\delta, k\delta + \delta]$, is called the k^{th} *slice* of the tube $[\boldsymbol{x}](\cdot)$ and is denoted by $[\![\boldsymbol{x}]\!]^{(k)}$. This implementation takes rigorously into account floating-point precision when building a tube: computations involving $[\boldsymbol{x}](\cdot)$ will be based on its slices, thus giving a reliable outer approximation of the solution set. The slices may be of same width as depicted in Fig. 1, but the tube can also be implemented with a customized temporal *slicing*. Finally, we endow the definition of a slice $[\![\boldsymbol{x}]\!]^{(k)}$ with the *slice (box) envelope* (blue painted in Fig. 1) and two input/output *gates* $[\boldsymbol{x}](t_k)$ and $[\boldsymbol{x}](t_{k+1})$ (black painted) that are intervals of $\mathbb{IR}^n$ through which trajectories are entering/leaving the slice.

Once a tube is defined, it can be handled in the same way as an interval. We can for instance use arithmetic operations as well as function evaluations. If f is an elementary function such as sin, cos or exp, we define $f([x](\cdot))$ as the smallest tube containing all feasible values: $f([x](\cdot)) = \left[\,\{f(x(\cdot)) \mid x(\cdot) \in [x](\cdot)\}\,\right]$.

2.3 Dynamical Systems and Differential-Algebraic CSP

A differential constraint relates one or several trajectories and/or real variables. Numerous types of differential constraints can be considered in our approach, including:

1. $\dot{\boldsymbol{x}}(\cdot) = \boldsymbol{f}\big(\boldsymbol{x}(\cdot)\big)$ (ODE)
2. $\dot{\boldsymbol{x}}(t) = \boldsymbol{f}\big(\boldsymbol{x}(t)\big) + \int_{t_0}^{t} \boldsymbol{x}(\tau)d\tau$ (integro-differential equation)
3. $\boldsymbol{x}(t_k) = \boldsymbol{y}$, $\dot{\boldsymbol{x}}(\cdot) = \boldsymbol{v}(\cdot)$ (evaluation constraint)
4. $\forall t \in [t]$, $\boldsymbol{x}(t) \notin [\boldsymbol{y}]$
5. $\boldsymbol{x}(t) = \boldsymbol{y}(t+\delta)$ (delay constraint)

The first one is the most widespread differential constraint (see Def. 2). The second constraint is a little more complicated in that the state at a given time depends on the sum (integral) of the previous states. The third evaluation constraint $\boldsymbol{y} = \boldsymbol{x}(t_k)$ states that the trajectory goes through an uncertain real value in $[\boldsymbol{y}]$ at an uncertain time in $[t_k]$. The fourth constraint is the complementary, although more complicated, constraint of the evaluation. The fifth one imposes a delay constraint of unknown real value δ between two trajectories and is particularly useful for clock calibration purposes in autonomous systems [38].

The idea behind our approach is to decompose a differential-algebraic system into a set of such primitive constraints associated to contractors (similar to propagators in CSP solvers [29]) that belong, or can be added to, the `Tubex` library. We formally define below the following differential constraints used in the experimental part.

Definition 2 (ODE and integro-differential equation)
Consider $\boldsymbol{x}(\cdot) : [t_0, t_f] \to \mathbb{R}^n$*, its derivative* $\dot{\boldsymbol{x}}(\cdot) : [t_0, t_f] \to \mathbb{R}^n$*, and an* evolution function $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^n$*, possibly non-linear. An ODE*[7] *is defined by:*

$$\dot{\boldsymbol{x}}(\cdot) = \boldsymbol{f}\big(\boldsymbol{x}(\cdot)\big)$$

An integro-differential equation is defined by:

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}\big(\boldsymbol{x}(t)\big) + \int_{t_0}^{t} \boldsymbol{x}(\tau)d\tau.$$

The ODEs considered are *explicit*, *i.e.* the evolution function $\boldsymbol{f}$ computes $\dot{\boldsymbol{x}}$ directly. These differential constraints can define dynamical systems.

Definition 3 (IVP, interval IVP, BVP)
The initial value problem (IVP) is defined by an ODE $\dot{\boldsymbol{x}}(\cdot) = \boldsymbol{f}\big(\boldsymbol{x}(\cdot)\big)$ *and an initial condition* $\boldsymbol{x}(t_0) = \boldsymbol{x}_0$*, where* $\boldsymbol{x}_0$ *is a constant in* $\mathbb{R}^n$*. In an interval IVP, the initial condition is bounded by an interval, i.e.* $\boldsymbol{x}(t_0) \in [\boldsymbol{x}_0]$*. A boundary value problem (BVP) is defined by an ODE and a numerical constraint* $\boldsymbol{c}\big(\boldsymbol{x}(t_1)..\boldsymbol{x}(t_n)\big) = \boldsymbol{0}$*, where* $\boldsymbol{c} : \mathbb{R}^n \to \mathbb{R}^n$ *and* $\forall i \in \{1..n\},\ t_i \in [t_0, t_f]$*.*

A BVP generalizes an IVP in that no initial condition fully determines the trajectory at a unique instant. Instead, n algebraic constraints relate several states

[7] Note that a high-order problem can be transformed automatically into a first-order ODE shown in Def. 2 by introducing auxiliary variables. Also note that non autonomous ODEs of the form $\dot{\boldsymbol{x}}(t) = \boldsymbol{f}\big(\boldsymbol{x}(t), t\big)$ can also be transformed into autonomous ODEs $\dot{\boldsymbol{x}}(t) = \boldsymbol{f}\big(\boldsymbol{x}(t)\big)$ whose derivative depends only on the state.

at times $t_1..t_n$ and enable the trajectory determination. Note that a condition known at any instant is equivalent to knowing the state at time t_0. Indeed, numerical integration can propagate the information forward or backward in time indifferently. We are now in position to define the DACSP.

Definition 4 (Differential-algebraic CSP – DACSP)
A DACSP network is defined by a quintuplet $(\boldsymbol{x}(\cdot), [\boldsymbol{x}](\cdot), \boldsymbol{y}, [\boldsymbol{y}], \boldsymbol{c})$*, where* $\boldsymbol{x}(\cdot)$ *is a trajectory variable of domain* $[\boldsymbol{x}](\cdot)$ *(a tube* $\mathbb{R} \rightarrow \mathbb{IR}^{n_1}$*, as defined in the previous section),* $\boldsymbol{y} \in \mathbb{R}^{n_2}$ *denotes the static numerical variables with a domain/box* $[\boldsymbol{y}]$ *and* $\boldsymbol{c}$ *denotes the set of static or differential constraints. Solving a DACSP instance consists in finding the set of values in* $[\boldsymbol{x}](\cdot)$ *and* $[\boldsymbol{y}]$ *satisfying* $\boldsymbol{c}$*.*

3 A Generic Solver for Differential-Algebraic CSP

In Algorithm 1, we give a description of a first generic solver for DACSP. The solver works on a network $P = (\boldsymbol{x}(\cdot), [\boldsymbol{x}](\cdot), \boldsymbol{y}, [\boldsymbol{y}], \boldsymbol{c})$ and returns a set of trajectories satisfying $\boldsymbol{c}$. The input tube $[\boldsymbol{x}](\cdot)$ is defined generally with one single slice $[t_0, t_f] \times [-\infty, \infty]^n$. We attempt to tackle a wide class of problems with pos-

```
Algorithm DACSP-Solver ($P = (\boldsymbol{x}(\cdot), [\boldsymbol{x}](\cdot), \boldsymbol{y}, [\boldsymbol{y}], \boldsymbol{c})$, specialTimes,
  maxTubeDiam, #slicesMax, Integration, $\boldsymbol{\varepsilon} = (\varepsilon_{fpt}, \varepsilon_{integr}, \varepsilon_{3B})$,
  slicingPolicy, bisectionPolicy)
    do
        /* Slicing loop: */
        $([\boldsymbol{x}](\cdot), [\boldsymbol{y}]) \leftarrow$ Contraction($P$, specialTimes, Integration, $\boldsymbol{\varepsilon}$, $false$)
        $[\boldsymbol{x}](\cdot) \leftarrow$ Slicing($[\boldsymbol{x}](\cdot)$, slicingPolicy)
    while MaxDiam($[\boldsymbol{x}](\cdot)$) > maxTubeDiam and #Slices(tube) < #slicesMax
    if MaxDiam($[\boldsymbol{x}](\cdot)$) $\leq$ maxTubeDiam then return $[\boldsymbol{x}](\cdot)$
    $L \leftarrow \{ ([\boldsymbol{x}](\cdot), null) \}$
    while $L \neq \emptyset$ /* Depth-first tree search */ do
        $([\boldsymbol{x}](\cdot), gate) \leftarrow$ Pop($L$)
        $([\boldsymbol{x}](\cdot), [\boldsymbol{y}]) \leftarrow$ Contraction($P$, specialTimes, Integration, $\boldsymbol{\varepsilon}$, $true$,
          gate)
        if MaxDiam($[\boldsymbol{x}](\cdot)$) $\leq$ maxTubeDiam then
            solutionsList $\leftarrow$ solutionsList $\cup \{[\boldsymbol{x}](\cdot)\}$
        else
            $([\boldsymbol{x}_1](\cdot), [\boldsymbol{x}_2](\cdot), gate) \leftarrow$ Bisect ($[\boldsymbol{x}](\cdot)$, bisectionPolicy)
            $L \leftarrow \{([\boldsymbol{x}_1](\cdot), gate)\} \cup \{([\boldsymbol{x}_2](\cdot), gate)\} \cup L$
    solutionsList $\leftarrow$ TubeMerge(solutionsList, $[\boldsymbol{x}](\cdot)$)
    return solutionsList
```

Algorithm 1: The DACSP solver.

sibly different behaviors. This may impair the effectiveness of a unique generic algorithm. In practice however, the user may already have an intuition of some instants from which things should propagate. As a consequence, in addition to P, we allow the user to provide a set of *special times*, *i.e.* elements of the temporal domain that involve states of the trajectory $\boldsymbol{x}(\cdot)$ and other static constraints. It allows this first solver to perform contraction more incrementally.

The solver works in two main phases: a so-called *slicing* step splitting the temporal domain into time slices, followed by a tree search subdividing the vectorial tube $[\boldsymbol{x}](\cdot)$. The last `TubeMerge` function compensates a potential clustering effect and merges together pairs of solution tubes that intersect along the temporal domain (on all the slices) in all the $(\boldsymbol{x})$ dimensions. It is necessary when bisection is not used for identifying different solutions, but helps the solver to compute accurate trajectories.

The main precision parameter of the solver is *maxTubeDiam*, a size expressed as the maximum width over all the slices envelopes of the tube. The solver can indifferently compute "thin" trajectories of theoretical null volume (*e.g.*, when dealing with pure IVPs) or "thick" trajectories (*i.e.*, continua of trajectories, *e.g.* when dealing with interval IVPs). In the latter case, the user has to tune this precision parameter to get a good approximation of thick trajectories. A second user-defined $\#slicesMax$ parameter is a maximum number of slices created during the solving, especially during the slicing phase. A large slice number leads to a better trajectory accuracy at the cost of worse performance. Note that the CPU time generally grows linearly in the number of slices.

The first slicing phase is performed by the first `do..while` loop interleaving contraction of P and time slicing ("discretization"). The latter splits several slices of $[\boldsymbol{x}](\cdot)$ into two slices of equal temporal size. Three main slicing policies have been tested:

– (*all*) Split *all* the slices in two.
– (*median*) Compute for all the bounded slices a dx difference between the middle points of 2 consecutive gates, maximum over all the dimensions, *i.e.* $dx = \max_i |m([x_i^k]) - m([x_i^{k-1}])|$, where $m([x_i])$ denotes the middle of $[x_i]$. Split half of the slices with the largest dx and all the unbounded slices.
– (*average*) Split the slices having a dx greater than the average value and all the unbounded slices.

If the loop terminates because the number of slices reaches $\#slicesMax$, the tree search will be in charge to get the $maxTubeDiam$ precision. This slicing phase seems to contradict the principles of most numerical algorithms that decide to subdivide a given time step adaptively. However, the `Integration` procedure called by the `Contraction` method carries out these adaptive time discretization steps, so that both mechanisms, *i.e.* integration and slicing phases, perform time discretization in a complementary manner.

The second phase performed by the second `while` loop is combinatorial. It implements a tree search branching on the domains of the trajectory variables, *i.e.* tubes. Although depth-first search is well-known in the CP community, to our knowledge, no prior work proposed to make choice points on tubes, defined as follows.

Definition 5 (Tube bisection)
Let $[\boldsymbol{x}](\cdot)$ be a tube of a trajectory $\boldsymbol{x}(\cdot)$ defined over $[t_0, t_f]$.
Let t_k be an instant in $[t_0, t_f]$, i a dimension in $\{1..n\}$, and $[x_i]$ the interval value of $[x_i](\cdot)$ at t_k. Let $mid(x_i)$ be $\frac{\overline{x_i}+\underline{x_i}}{2}$.
The tube bisection (t_k, i) of $[\boldsymbol{x}](\cdot)$ produces two tubes $[\boldsymbol{x}^L](\cdot)$ and $[\boldsymbol{x}^R](\cdot)$ equal to $[\boldsymbol{x}](\cdot)$ except at time t_k, where $[x_i^L] = [\underline{x_i}, mid(x_i)]$ and $[x_i^R] = [mid(x_i), \overline{x_i}]$.

In practice, a bisection (t_k, i) is applied only to a gate of the tube. Two heuristics are proposed to the user for selecting the instant t_k. The first one picks randomly one instant among the "special times" specified by the user. The second one selects the t_k having the largest box $[\boldsymbol{x}](t_k)$. The dimension $i \in \{1..n\}$, on which the bisection is performed, is decided according to the largest component $[x_i]$.

Note that the DACSP solver is sound because no operator used in Algorithm 1 can eliminate a solution: `Contraction`, `Slicing`, `Bisect`, `TubeMerge`.

4 Contractions in the Solver

The `Contraction` function consists of a simple propagation loop that calls the contractors corresponding to constraints in $\boldsymbol{c}$ until the relative gain in contraction volume is less than ε_{fp}. Contractors can be of any type: HC4-Revise for a numerical constraint or the "map" contractor mentioned in introduction for the robotic application. The propagation loop is followed by a call to a contractor `Dyn3B` enforcing a strong consistency on the tube (see Algorithm 3).

Let us detail in Algorithm 2 an important contractor, called `ExplicitDE`, that carries out tube contractions based on ODEs or integro-differential equations. The procedure is mainly responsible for launching integration steps forward and backward in time through the tube. The actual integration method used is a parameter of Algorithm 2. Note that a guaranteed integration algorithm inferring a new information, like a value of the state known at a specific instant, is incremental in that it may contract only a subset of a tube if no more contraction is obtained at a given gate. The tube contraction is not incremental in the first slicing phase or at the top of the search tree ($gateBis = null$) because the `Slicing` procedure can subdivide numerous slices everywhere in the tube (parameter *isIncremental* set to false). Therefore integration is run from t_0 to t_f (forward) and from t_f to t_0 (backward). Conversely, during the tree search, integration is triggered by a tube bisection or a domain modification at a special time (whose state is related with a static variable). That is why incremental integrations start from each of these instants.[8] Our solver is endowed with two possible `Integration` procedures. The first one is an "internal" generic integration algorithm that will be incorporated in the Tubex library. Its signature is close to the procedure `Integration` shown in Algorithm 2. It can be triggered from any specified time in the tube, forward or backward, and with the

[8] The actual code is a little bit more complicated. An instant is skipped if it is handled by the previous integration step.

```
Algorithm ExplicitDE($\boldsymbol{f}$, $[\boldsymbol{x}](\cdot)$, specialTimes, Integration, $\boldsymbol{\varepsilon}$,
 isIncremental, gateBis)
    if gateBis = null or not isIncremental then
        $[\boldsymbol{x}](\cdot) \leftarrow$ Integration($\boldsymbol{f}$, $[\boldsymbol{x}](\cdot)$, $t_0$, FORWARD, $\boldsymbol{\varepsilon}$, $false$)
        $[\boldsymbol{x}](\cdot) \leftarrow$ Integration($\boldsymbol{f}$, $[\boldsymbol{x}](\cdot)$, $t_f$, BACKWARD, $\boldsymbol{\varepsilon}$, $false$)
    else
        gates $\leftarrow$ Sort($\{gateBis\} \cup specialTimes$)
        forall $gate \in gates$ do
            // forward and incremental simulation:
            $[\boldsymbol{x}](\cdot) \leftarrow$ Integration($\boldsymbol{f}$, $[\boldsymbol{x}](\cdot)$, $gate$, FORWARD, $\boldsymbol{\varepsilon}$, $true$)
        forall  $gate \in gates$, in reverse order do
            // backward and incremental simulation:
            $[\boldsymbol{x}](\cdot) \leftarrow$ Integration($\boldsymbol{f}$, $[\boldsymbol{x}](\cdot)$, $gate$, BACKWARD, $\boldsymbol{\varepsilon}$, $true$)
```
Algorithm 2: A generic contractor for ODE and integro-diff equation

possibility of running the simulation incrementally, *i.e.* stopping it if no sufficient contraction volume gain has been obtained in a gate box. This procedure `Integration` is generic in that it can accept an evolution function $\boldsymbol{f}$ describing either an ODE or an integro-differential equation (see Definition 2). It can also be specialized by a "slice integration contractor" called at each time step of the simulation. Two slice contractors are highlighted in this paper. The first one, called `DynBasic` hereafter, wraps at the slice level two simple contractors available since the very first version of `Tubex`: `CtcDeriv` and an evaluation of the evolution function $\boldsymbol{f}$ called iteratively. `CtcDeriv` (denoted $\mathcal{C}_{\frac{d}{dt}}$ in the literature [34]) is a tube contractor treating the constraint $\dot{\boldsymbol{x}}(\cdot) = \boldsymbol{v}(\cdot)$, where $\boldsymbol{x}(\cdot)$ and $\boldsymbol{v}(\cdot)$ are two trajectories and $\boldsymbol{v}(\cdot)$ is the derivative of $\boldsymbol{x}(\cdot)$ over time. The fundamental theorem of calculus that relates differentiation and integration, is used by `CtcDeriv` for contracting the tube $[\boldsymbol{x}](\cdot)$. The second slice contractor, called `DynCIDGuess` hereafter, generates for each integration step a "slice" contractor graph, where the variables correspond to the two gates and the slice envelope (see Sec. 2.1). Based on the input gate box and the envelope, `DynCIDGuess` can improve the output gate box using sophisticated singleton consistencies based on 3B and CID (see Section 2.1). This contractor will be detailed in another article. This generic integration contractor starts by calling a Picard operator that allows one to set non-infinite initial bounds on some tube $[\boldsymbol{x}](\cdot)$, which is required for engaging contraction, and can create new slices adaptively [27].

A second `Integration` procedure wraps directly the state-of-the-art VNODE [28] guaranteed integration solver into a `CtcVnode` contractor. During the slicing phase, it calls VNODE simulations forward and backward from the smallest gate. After each bisection, it calls VNODE simulations forward and backward starting from the bisected gate. To make `CtcVnode` a contractor, the results obtained by the VNODE simulations are intersected with the current tube. VNODE performs its own slicing, especially in the first iterations, and the slices produced are added to those from the slicing phase. Finally, as we will see in the experiments, the contractor `CtcVnode`, and a slice integration contractor as `DynBasic` or `DynCIDGuess`, can be called successively inside the contraction loop performed by the `Contraction` procedure.

Another new and useful dynamic contractor is the `Dyn3B` contractor described in Algorithm 3. This is a dynamic adaption of the 3B algorithm described in Section 2.1. It selects iteratively the instant (gate) t_k with the largest interval (the tube is thus not contracted at all instants) and applies a `VarDyn3B` shaving procedure to all the $[x_i]$ intervals at t_k. `VarDyn3B` is a straightforward adaptation of the standard `Var3B` shaving procedure (see Section 2.1) to tubes. Subintervals at the bounds of $[x_i]$ can be safely eliminated if an integration starting from the corresponding sub-tube leads to an empty domain. This integration procedure can be achieved by `DynBasic`, or `CtcVnode` followed by `DynBasic`.

```
Algorithm Dyn3B ($P$, $\varepsilon$)
    do
        $volumeSave \leftarrow$ volume($[\boldsymbol{x}](\cdot)$)
        $t_k \leftarrow$ SelectGate($[\boldsymbol{x}](\cdot)$)
        forall $i \in \{1..n\}$ do
            $[\boldsymbol{x}](\cdot) \leftarrow$ VarDyn3B($P$, $t_k$, $i$)
    while $VolumeGain([\boldsymbol{x}](\cdot), volumeSave) > \varepsilon_{3B}$
```

Algorithm 3: The Dynamic 3B algorithm

5 Experiments

The goal of this section is to highlight that the DACSP model, the contractors available via `Tubex` and our DACSP solver can handle a large variety of systems that no competitor or a few ones can deal with. All the results have been obtained on a CPU computer using an `x86-64` processor (1.6 GHz).

5.1 BVP for Integro-Differential Equation

Let us illustrate the versatility of our DACSP solver on the following problem. It combines an integro-differential equation defined on the domain $[0, 1]$ and a constraint between the initial and final values, as follows:

$$\begin{cases} \dot{x}(t) = 1 - 2x(t) - 5\int_0^t x(\tau)d\tau;\ t \in [0,1] \\ x(0)^2 + x(1)^2 = 1 \end{cases} \tag{1}$$

Our solver can find both solutions in 8.35 seconds and needs to resort to 66 bisections (and a search tree depth of 25) to isolate them at a good accuracy. For both solutions, Table 1 reports some details. Note that only our generic `Integration` algorithm can be used in the solver for this particular problem

Table 1. Solutions obtained on the integro-differential based system. The table reports the diameters of the initial and final gates, the tube volume and the slices number.

Solution	Diam. of gate $t_0 = 0$	Diam. of gate $t_f = 1$	Tube volume	#slices
1	0.015	0.030	0.018	400
2	0.034	0.022	0.024	400

since there is no ODE, contrarily to the following DACSP systems. It has been run with $maxTubeDiam = 0.02$ and $\#slicesMax = 400$.

For the next two DACSP categories tested, we show the best combination of the `CtcVnode` (refered by `vnode` in the tables), `DynBasic` (`basic`), `DynCIDGuess` (`CIDG`) and `Dyn3B` (`3Bvnode` or `3Bbasic`) contractors.

5.2 BVPs and Cruz & Barahona System

We have tested and reported in Fig. 2 five BVPs and the Cruz system close to a BVP because no state is fully determined at a given instant. However, note that this system has a thick tube solution.

$$\begin{cases} \dot{x}(\cdot) = x(\cdot) \\ x(0)^2 + x(1)^2 = 1 \\ [t_0, t_f] = [0, 1] \\ maxTubeDiam = 0.0005 \end{cases} \quad (2)$$

$$\begin{cases} \ddot{x}(\cdot) = -x(\cdot) \\ x(0) = 0; x(\pi/2) = 2 \\ [t_0, t_f] = [0, \pi/2] \\ maxTubeDiam = 0.0005 \end{cases} \quad (3)$$

$$\begin{cases} \ddot{x}(\cdot) = 5\dot{x}(\cdot) \\ x(0) = 1;\ x(1) = 0 \\ \dot{x}(0) \in [-10, 10];\ \dot{x}(1) \in [-10, 10] \\ [t_0, t_f] = [0, 1] \\ maxTubeDiam = 0.02 \end{cases} \quad (4)$$

$$\begin{cases} \ddot{x}(\cdot) = -10(\dot{x}(\cdot) + x(\cdot)^2) \\ x(0) = 0; x(1) = 0.5 \\ \dot{x}(0) \in [-20, 20]; \dot{x}(1) \in [-20, 20] \\ [t_0, t_f] = [0, 1] \\ maxTubeDiam = 0.05 \end{cases} \quad (5)$$

$$\begin{cases} \ddot{x}(\cdot) = -\exp(x(\cdot)) \\ x(0) = 0;\ x(1) = 0 \\ \dot{x}(0) \in [-20, 20] \\ \dot{x}(1) \in [-20, 20] \\ [t_0, t_f] = [0, 1] \\ maxTubeDiam = 0.05 \end{cases} \quad (6)$$

$$\begin{cases} \dot{x}_1(\cdot) = -0.7x_1(\cdot) \\ \dot{x}_2(\cdot) = 0.7x_1(\cdot) - (\ln(2)/5)x_2(\cdot) \\ x_1(0) = 1.25 \\ x_2 \in [1.1, 1.3] \text{ during } [1, 3] \\ [t_0, t_f] = [0, 6] \\ maxTubeDiam = 0.04 \end{cases} \quad (7)$$

Fig. 2. Five BVPs and the Cruz system. (2) A one-dimensional problem with an algebraic constraint between the initial and final states; (3) Classical linear example cited in Wikipedia; (4) and (5) denote resp. Systems 2 and 23 in the BVPSolve benchmark [24]; (6) the Bratu system, the only one with two solutions in the BVPSolve benchmark, (7) the Cruz system with a partial information in the middle of the temporal domain.

5.3 Interval IVPs

Although solving interval IVPs is not the primary purpose of the DACSP solver, we present results obtained on three interval IVPs (see Fig. 3).

Table 2. Solutions obtained on BVP systems. For each system, strategy and solution (s_1 and/or s_2), we report the diameters of the two unknown states in t_0 and t_f (most of the systems tested are 2-dimensional, but 2 of the 4 bounds are provided as initial conditions), the volume of the solution tubes, the number of slices, the computational time and the number of choice points required (#bis.).

Sys.	Best strategy	#sol	t_0 diam.	t_f diam.	Tube vol.	#slices	Time	#bis.
(2)	vnode+basic	s_1	2e-8	5e-8	2.e-4	5,000	7.63s	1
		s_2	2e-8	5e-8	2.e-4	5,000		
(3)	vnode+basic	s_1	7e-15	7e-15	6e-4	12,288	12.7s	0
(4)	vnode+CIDG+3Bvnode	s_1	2e-9	4e-7	5e-3	1,216	3.05s	0
(5)	vnode+CIDG+3Bbasic	s_1	3.e-2	2.e-4	0.012	5,000	81s	6
(6)	vnode+basic	s_1	3.e-6	2.e-6	7.e-4	2,000	75s	62
		s_2	5.e-3	5.e-3	0.025	2,000		
(7)	CIDG	s_1	0.0644	0.0282	0.2637	10,000	6.85s	1

$$\begin{cases} \dot{x} = -x^2 \\ x(0) \in [0.1, 0.4] \\ [t_0, t_f] = [0, 5] \\ eps = 0.2 \end{cases} \quad (8) \qquad \begin{cases} \dot{x}_1 = -x_1 - 2x_2 \\ \dot{x}_2 = -3x_1 - 2x_2 \\ x_1(0) \in [5.9, 6.1] \\ x_2(0) \in [3.9, 4.1] \\ [t_0, t_f] = [0, 1] \\ eps = 0.5 \end{cases} \quad (9) \qquad \begin{cases} \dot{x}_1 = -x_2 + 0.1\, x_1\, (1 - x_1^2 - x_2^2) \\ \dot{x}_2 = x_1 + 0.1\, x_2\, (1 - x_1^2 - x_2^2) \\ x_1(0) \in [0.7, 1.3] \\ x_2(0) = 0.0 \\ [t_0, t_f] = [0, 5] \\ eps = 0.15 \end{cases} \quad (10)$$

Fig. 3. Three interval IVPs tested. (8) and (9) were introduced in [8]. (10) describes a limit cycle and is particularly sensitive to the wrapping effect caused by interval computation.

For the 2 examples from [8], the exact solution is known, so we also report the relative error (column Gap in Table 3) on interval width of the final gates.

5.4 Discussion on Experiments

Note first that the `VNODE` solver alone (outside the DACSP solver) cannot cope with BVPs and is not efficient on the interval IVPs selected. `CtcVnode` (inside the DACSP solver) often provides a very good accuracy on the gates. The good performance is probably due to the high-order interval Taylor form used (order 11 has been set for the experiments). However, `CtcVnode` does generally not obtain good contraction on the whole tube (slice envelopes). Additional work is required to envisage obtaining a better tube volume accuracy using `CtcVnode`.

Table 3. Solutions obtained on interval IVPs systems. For each system, we report the best strategy, the diameters of the state at t_f, the volume of the solution tube, the number of slices, the computational time and the number of bisections.

Sys.	**Strategy**	**t_f diam.**	**Tube vol.**	**Gap**	**#slices**	**Time**	**#bis.**
(8)	CIDG	0.06668	0.6934	0.02%	40,000	9.35 s	1
(9)	vnode+CIDG	(0.544;0.544)	0.700	(0.01%;0.01%)	2,000	3.93 s	1
(10)	vnode+basic	(0.0695;0.2273)	2.54		1,000	13.3 s	7

`DynCIDGuess` alone is generally not efficient, except on Systems (7) and (8), because it requires too many slices to reach the precision (recall that the CPU time generally grows linearly in the number of slices). Finally, the best option for the `Integration` procedure is generally to call first `CtcVnode` and then `DynBasic` or `DynCIDGuess`. A final call to `Dyn3B` is useful for Systems (4) (using `CtcVnode` as subcontractor) and (5) (using `DynBasic`).

Overall, different solver strategies provide the best results on the different systems tested. All the devices offered by the solver can be useful on different instances: contractors, slicing, choice points. When the best strategy includes a number of bisections, this means that the $\#slicesMax$ has been reached and the solver resorts to choice points to better approximate the solution tube. An issue for future work is to better study the interplay between slicing and bisection in order to obtain a more generic DACSP solver that can work without the $\#slicesMax$ parameter.

6 Conclusion and Future Work

We have presented a new generic solver that can handle together differential and static numerical constraints. The originality of the approach lies both in the underlying model considering trajectories as variables and in a novel backtracking mechanism applicable to DACSP. Our DACSP solver is endowed with an exploration operator that enables to bisect a tube at a chosen time. This allows the DACSP solver to better handle hard DACSP systems and accurately estimate distinct trajectories of problems having several solutions. We have shown on first experimental results that our solver is versatile enough to solve DACSP instances for which no or a few algorithmic solutions currently exist. We have also demonstrated the benefits of wrapping the state-of-the-art VNODE in a `CtcVnode` contractor implemented in `Tubex`.

With regard to future work, we will first try to limit the number of user-defined parameters, in particular remove $\#slicesMax$. Also, we want to propose ideally only one combination of contractors in the DACSP solver for every DACSP subclass. Second, we will study a more general search tree branching static and dynamical variables domains indifferently, though we need to explore new ideas on large-scale problems. Finally, in the current solver, the propagation between functional and real variables domains is somewhat naive and is partly ensured by the “special times” specified by the user (see Algorithm 2). The quite recent Tubex 3.0 accepts bi-level slice/tube variables, which will enable a fully incremental contraction achieved by a propagation engine.

Supplementary materials including the sources of the solver and the experiments are available on http://simon-rohou.fr/research/dacsp-solve/.

Acknowledgements. This work was supported by the French Agence Nationale de la Recherche (ANR) [grant number ANR-16-CE33-0024].

References

1. F. Benhamou, F. Goualard, L. Granvilliers, and J.-F. Puget. Revising Hull and Box Consistency. In *Proc. of International Conference on Logic Programming (ICLP)*, pages 230–244, 1999.
2. G. Chabert. IBEX – an Interval-Based EXplorer, 2020. http://www.ibex-lib.org/.
3. G. Chabert and L. Jaulin. Contractor Programming. *Artificial Intelligence*, 173:1079–1100, 2009.
4. A. Chapoutot, J. Alexandre dit Sandretto, and O. Mullier. *Dynibex*. 2015. http://perso.ensta-paristech.fr/ chapoutot/dynibex/.
5. H. Collavizza, F. Delobel, and M. Rueher. Comparing Partial Consistencies. *Reliable Computing*, 5(3):213–228, 1999.
6. J. Cruz and P. Barahona. Constraint Satisfaction Differential Problems. In *Principles and Practice of Constraint Programming - CP 2003.*, pages 259–273, 2003.
7. J. Cruz and P. Barahona. Constraint Reasoning with Differential Equations. *Applied Numerical Analysis & Computational Mathematics*, 1(1):140–154, 2004.
8. Y. Deville, M. Janssen, and P. VanHentenryck. Consistency Techniques in Ordinary Differential Equations. In *Proc. of CP98*, pages 162–176, 1998.
9. F. Domes. GLOPTLAB: A configurable framework for the rigorous global solution of quadratic constraint satisfaction problems. *Optimization Methods & Software*, 24:727–747, 10 2009.
10. T. F. Filippova, A. B. Kurzhanski, K. Sugimoto, and I. Vályi. Ellipsoidal State Estimation for Uncertain Dynamical Systems. In *Bounding Approaches to System Identification*, pages 213–238. Springer US, Boston, MA, 1996.
11. A. Goldsztejn, O. Mullier, D. Eveillard, and H. Hosobe. Including Ordinary Differential Equations Based Constraints in the Standard CP Framework. In *Proc of CP 2010*, pages 221–235. Springer Berlin, Heidelberg, 2010.
12. T.R Goodman and G.N. Lance. The Numerical Solution of Two-Point Boundary Value Problems. *Mathematical Tables and Other Aids to Computation*, 10:82–86, 1956.
13. L. Granvilliers and F. Benhamou. RealPaver: An Interval Solver using Constraint Satisfaction Techniques. *ACM Transactions on Mathematical Software - TOMS*, 32:138–156, 2006.
14. E. R. Hansen. *Global Optimization using Interval Analysis.* Marcel Dekker, New York, NY, 1992.
15. L. Jaulin. Nonlinear Bounded-error State Estimation of Continuous-Time Systems. *Automatica*, 38:1079–1082, 2002.
16. L. Jaulin. Range-Only SLAM with Indistinguishable Landmarks: A Constraint Programming Approach. *Constraints*, 21(4):557–576, 2016.
17. R.E. Kalman. Contributions to the Theory of Optimal Control. *Bol. Soc. Mat. Mex.*, 5:102–119, 1960.
18. T. Kapela, M. Mrozek, P. Pilarczyk, D. Wilczak, and P. Zgliczynski. CAPD – a rigorous toolbox for Computer Assisted Proofs in Dynamics. `http://capd.ii.uj.edu.pl/`, 2010.
19. R. Kearfott. GlobSol: History, Composition, and Advice on Use. In *Proc of COCOS2002*, LNCS 2861, pages 17–31. Springer, 10 2002.
20. A. B. Kurzhanski and T. F. Filippova. On the Theory of Trajectory Tubes - A Mathematical Formalism for Uncertain Dynamics, Viability and Control. In *Advances in Nonlinear Dynamics and Control: A Report from Russia*, pages 122–188. Birkhäuser, Boston, MA, 1993.

21. F. Le Bars, J. Sliwka, L. Jaulin, and O. Reynet. Set-membership state estimation with fleeting data. *Automatica*, 48(2):381–387, 2012.
22. O. Lhomme. Consistency Techniques for Numeric CSPs. In *IJCAI*, pages 232–238, 1993.
23. R. Lohner. Enclosing the solutions of ordinary initial and boundary value problems. In E. Kaucher, U. Kulisch, and Ch. Ullrich, editors, *Computer Arithmetic: Scientific Computation and Programming Languages*, pages 255–286. BG Teubner, Stuttgart, Germany, 1987.
24. F. Mazzia, J.R. Cash, and K. Soetaert. Solving boundary value problems in the open source software R: Package bvpSolve. *Opuscula mathematica*, 34(2):387–403, 2014.
25. F. Messine. *Méthodes d'optimisation globale basées sur l'analyse d'intervalle pour la résolution des problèmes avec contraintes*. PhD thesis, LIMA-IRIT-ENSEEIHT-INPT, Toulouse, 1997.
26. R. E. Moore. *Interval Analysis*, volume 4. Prentice-Hall Englewood Cliffs, 1966.
27. N. Nedialkov, K. Jackson, and G. Corliss. Validated Solutions of Initial Value Problem for Ordinary Differential Equations. *Applied Mathematics and Applications*, 105(1):21–68, 1999.
28. N.S. Nedialkov. VNODE-LP, A Validated Solver for Initial Value Problems in Ordinary Differential Equations. Technical Report CAS-06-06-NN, Department of Computing and Software, McMaster University,Hamilton, Ontario, Canada, 2006.
29. C. Prud'homme, J.-G. Fages, and X. Lorca. Choco documentation. 2014. http://www. choco-solver. org.
30. N. Revol, K. Makino, and M. Berz. Taylor models and floating-point arithmetic: proof that arithmetic operations are validated in COSY. *Journal of Logic and Algebraic Programming*, 64:135–154, 2005.
31. S. Rohou, B. Desrochers, and L. Jaulin. Set-membership State Estimation by Solving Data Association. In *IEEE International Conference on Robotics and Automation*, 2020.
32. S. Rohou et al. The Tubex library – Constraint-programming for robotics, 2020. http://simon-rohou.fr/research/tubex-lib/.
33. S. Rohou, L. Jaulin, L. Mihaylova, F. Le Bars, and S. M. Veres. Reliable Nonlinear State Estimation Involving Time Uncertainties. *Automatica*, 93:379–388, 2018.
34. S. Rohou, L. Jaulin, L. Mihaylova, F. Le Bars, and S. M. Veres. Guaranteed computation of robot trajectories. *Robotics and Autonomous Systems*, 93:76–84, 2017.
35. S. Thrun, W. Bugard, and D. Fox. *Probabilistic Robotics*. MIT Press, Cambridge, M.A., 2005.
36. G. Trombettoni and G. Chabert. Constructive Interval Disjunction. In *Proc. CP, Constraint Programming, LNCS 4741*, pages 635–650. Springer, 2007.
37. W. Tucker. A Rigorous ODE Solver and Smale's 14th Problem. *Foundations of Computational Mathematics*, 2(1):53–117, 2002.
38. R. Voges and B. Wagner. Timestamp offset calibration for an IMU-camera system under interval uncertainty. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 377–384, 2018.
39. D. Wilczak and P. Zgliczynski. Heteroclinic connections between periodic orbits in planar restricted circular three-body problem–a computer assisted proof. *Communications in mathematical physics*, 234(1):37–75, 2003.