Contractor Programming

Gilles Chabert, Luc Jaulin

▶ **To cite this version:**

Gilles Chabert, Luc Jaulin. Contractor Programming. Artificial Intelligence, 2009, 173, pp.1079-1100. 10.1016/j.artint.2009.03.002 . hal-00428957

HAL Id: hal-00428957
https://hal.science/hal-00428957

Submitted on 30 Oct 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Contractor Programming

Gilles Chabert[1] and Luc Jaulin[2]

[1] Ecole des Mines de Nantes LINA CNRS UMR 6241,
4, rue Alfred Kastler 44300 Nantes, France
gilles.chabert@emn.fr
[2] ENSIETA, 2, rue Fran cois Verny 29806 Brest Cedex 9, France
luc.jaulin@ensieta.fr

Abstract. This paper describes a solver programming method, called *contractor programming*, that copes with two issues related to constraint processing over the reals. First, continuous constraints involve an inevitable step of solver design. Existing softwares provide an insufficient answer by restricting users to choose among a list of fixed strategies. Our first contribution is to give more freedom in solver design by introducing programming concepts where only configuration parameters were previously available. Programming consists in applying operators (intersection, composition, etc.) on algorithms called *contractors* that are somehow similar to propagators.
Second, many problems with real variables cannot be cast as the search for vectors simultaneously satisfying the set of constraints, but a large variety of different outputs may be demanded from a set of constraints (e.g., a paving with boxes inside and outside of the solution set). These outputs can actually be viewed as the result of different *contractors* working concurrently on the same search space, with a bisection procedure intervening in case of deadlock. Such algorithms (which are not strictly speaking solvers) will be made easy to build thanks to a new branch & prune system, called *paver*.
Thus, this paper gives a way to deal harmoniously with a larger set of problems while giving a fine control on the solving mechanisms. The contractor formalism and the paver system are the two contributions. The approach is motivated and justified through different cases of study. An implementation of this framework named `Quimper` is also presented.

1 Introduction

Constraint programming is a simple and efficient paradigm to handle a large class of combinatorial problems [40, 10, 44]. In the presence of real-valued variables, constraint propagation algorithms combined with interval analysis [16, 22, 27, 19, 9] are also particularly well-suited, included for real-world applications (see, e.g., [33, 21]). We shall refer to this interval variant of constraint programming as *interval programming*. Even then, interval programming has had only

moderate success. In our opinion, the reason is a lack of clear and unified formalism describing how solvers and derived programs are built. This paper is an attempt to fill this gap.

We propose a framework that allows one to build a continuous solver with a few lines, in a high-level syntax. More than just another tuning language, a *programming framework* is proposed.

Motivation

Three reasons justify the introduction of a solver programming framework in presence of real variables.

1. Constraint programming is a declarative paradigm which means that a programmer should spend most of the effort in modeling conveniently the problem. This effort may involve breaking symmetries, introducing global or soft constraints, etc.. All these concepts are related to modeling and represent, by the way, very active fields of research. In theory, the solver is a black box of which a programmer could ignore the details.
   Despite of this, there is always a need at some point to control the solver, as it is with all declarative languages (consider for instance `Prolog` *cuts* that allow ruling out choices in the search). We can say that the overall efficiency one gets is the combined result of efforts made on both aspects: modeling and solver control.
   With real variables, modeling languages are limited[1]. Constraints can usually be nothing but equations which means that mathematics is de facto the ultimate modeling language[2]. The consequence is that solver control becomes an inevitable step if one is to improve efficiency.
2. Continuous solvers have a two-layered structure, namely *interval analysis* and *constraint programming*, The lower layer includes interval arithmetics and interval numerical algorithms (e.g., an interval variant of the Newton iteration) with round-off considerations. The upper layer includes branch & prune algorithms for describing sets of reals defined by constraints (or optimizing a criterion under constraints). Since these layers correspond to quite different scientific communities, there is a need more than ever for an interface between them. In concrete words, it would be useful to give a

[1] This limitation holds for numerical systems involving analytic expressions, which cover most of the mathematical models of physics problems. But modeling languages as such are not limited with real variables any more than with discrete ones (one may introduce table of constraints, piecewise constraints, etc.).

[2] There is still a notable exception: geometrical constraints. Geometry represents a semantic level above algebra; as an example, the "intersection of three spheres" can be introduced as a global constraints instead of three equivalent distance equations [1]. But, except in such cases, no improvement has to be expected from the modeling side.

constraint programmer the ability to develop a continuous solver without digging into details of interval analysis.

3. The last and possibly main point is related to the output of constraint solvers. With discrete domains, the output is always the set of solutions (or a subset optimizing some criteria). But in continuous domains, there may be a large variety of different outputs. First, one may look for a sub-paving (a set of boxes) encompassing the solutions and this is precisely what most of the existing solvers provide. They act as *root finders*. Next, in case of a solution set with a non-null volume, several sub-pavings are expected, each satisfying a different property, basically: "may contain a solution", "does contain a solution" or "contains only solutions". Such solvers rather act as *set describers*. Actually, we will see through the examples of Section 2 that the semantics behind the sub-pavings may completely change from one problem to the other[3]. As we will show, neither a *root finder* nor a *set describer* is adapted for solving these problems. Of course, ad-hoc solutions always exist, but the purpose is precisely to avoid a multiplicity of programs where a single one would be enough. In practice, when people are facing a specific constraint problem that requires a specific algorithm, they have to reverse-engineer the code of an existing solver. Often, they redevelop it from scratch.

Contribution

We propose a formalism and an algorithm, called *paver*.

In the formalism, the different interval routines (evaluations, projections, existence tests, etc.) are all wrapped in the very same object called *contractor* (see Section 3). Of course, the concept of *contractor* is not a novelty on its own. Our (first) contribution is to redefine various constraint programming techniques (propagation, shaving, parameter splitting, etc.) as operations over contractors that yield new contractors (Section 4). Syntactically, the contractor is then the unique atom, whence a certain simplicity. A solver can then be programed, rather than configured, by combining different contractors (examples are given in §4.2, §4.3, §5.1 and §5.5).

The paver algorithm is a generic solver. It takes a list of contractors, an initial box and follows a classical recursion: the contractors are successively called on the current box until either it gets empty or no more contraction could be done. In the latter case, the box is bisected and contractors are called back again.

The fact that different contractors work concurrently allows solving problems of quite different nature (see next section). This is our second contribution.

[3] We can say that no modeling language dedicated to the output exists so far, and this is another important distinction with discrete problems where this modeling aspect is not as ubiquitous (solvers using explanations are counter-examples).

Hence, contractor programming consists in two distinct steps: *contractor design* and *paver design*. The former refers to the design of the most possible efficient contractor for a given set (constraint) and shall be discussed in Section 4. The latter refers to the selection of contractors that yield the desired output, regardless of efficiency.

This framework is already supported by a real system named `Quimper` that will be introduced in Section 6.

Thinking of contractor programming as an extension of constraint programming is valid to the extent that contractors help in modeling the output of a problem. But, fundamentally, there is not such an extension since constraints basically tell the "what" whereas contractors tell the "how".

Note that branching will not be covered in this paper but one has to keep in mind that this part of the paver should be customizable as well. Only plain bisection will be used in the examples (variable are selected with a round-robin heuristic).

Power of Contractor programming

As for every programming language, the power of contractor programming, i.e., the class of problems that can be solved using this paradigm would require the setting of computability theory to be described formally. Pavings obtained in our framework are the results of algorithms that recursively contract boxes to smaller boxes and this general definition does not even involve the concept of constraint. It can also fit well, e.g., in the context of computer graphics to draw fractals. This means that some pavings generated by our system cannot be characterized by constraints.

However, one may wonder which problems contractor programming is the most aimed at. The contractor paradigm is mostly aimed at building *set computation tools* over the reals. These tools typically calculate disjunction of equidimensional sets and sets are usually described by constraints. There are however very few restriction on their form: constraints can be first-order logic formulas involving numerical equations or inequalities and quantified parameters. They can also be handled implicitly by their associated contractor.

The main point is that the representation of the sets and the way it is calculated are both entirely controllable.

Related works

In discrete constraint programming, existing tools already give control mechanisms with sometimes an associated formalism. In constrained-based local search, the `COMET` system [46] offers a language allowing one to give his own definition

of moves and neighborhoods, the incremental computation of the impact after a move being done automatically.

In branch & prune systems, choice points and propagation can usually be controlled. The `Choco` [25], `ILOG Solver` [20] and `Gecode` [43] libraries allow to specify the instantiation order for both variables and values. It can also be decided how to split domains (either instantiate or bisect domains, etc.).

Propagation has given rise to more sophisticated concepts. The aforementioned libraries allow the programmer to associate his own propagator, or filtering algorithm, to a constraint. In `Gecode`, propagators can be automatically generated from a formula. It is possible with `Choco` to control when propagators are launched thanks to a system of events (e.g., *a bound is reduced*, *the variable is instantiated*, etc.). This allows, as a side effect, to schedule constraints in the propagation loop (some costly propagators may ignore events of minor importance). Events handling have to be implemented by the propagators in `Gecode`, albeit recent works about *advisors* [29] facilitate this task. These mechanisms are all related to modeling (a propagator is associated to a constraint), preserving declarativity. As far as we know, there is not a way to build new propagators, say, by composing different propagators, at the modeling phase. Only programmers with a good knowledge of the target library are able to do so. This is a difference with *contractor programming* that aims at building solvers rather than operational constraints.

In contrast, efficiency has always taken precedence over flexibility in continuous constraint programming. Perhaps because of the prevailing numerical culture, solvers are indeed usually compared on a performance basis. As a consequence, none of the different existing implementations (Numerica [47], RealPaver [15], Alias [32], Rsolver [38], GlobSol [26], Baron [41], etc.) supply a good level of openness or extensibility. These libraries contain solvers or global optimization algorithms with a very high-level interface, basically restricting users to enter a system of equations, set up some parameters and press enter. Of course, there may be *many* parameters. For instance, it can be decided whether `Alias` has to resort to the computation of the Hessian matrix or to ask `RealPaver` for the global consistency instead of a solution set. But there is no way to venture off the beaten tracks.

A side contribution of this paper is a new representation for solution spaces with non-empty interiors. More suitable representations have been proposed in [42] using *quadtrees/octrees* or in [48] using *extreme vertices*. However, quadtrees consist in discretizing constraints and reasoning using this auxiliary representation. This otherwise appealing method is therefore quite distant from *interval programming*. Results of this paper are unlikely to be applicable in the context of *quadtrees*.

Computing extreme vertices can be viewed as an independent operation to obtain a compact representation of a set of boxes. The dissertation [48] contains also smart branching heuristics and a concept of *active variables* to preserve the

alignment of small boxes. All these concepts are fully compatible with our framework and somehow orthogonal to *contractors* since they are related to branching issues. They can be viewed as complementary to ours.

1.1 Notations and vocabulary

The set of all intervals is denoted by $\mathbb{IR}$.

A Cartesian product of intervals is called a *box*. Intervals and boxes will be surrounded by brackets, e.g., $[x]$. The symbol $[x]$ will always denote a n-dimensional box, except otherwise stated. If $[x]$ is a box, x_i^- and x_i^+ will stand for the lower and upper bound of $[x]_i$ respectively.

The *width* of a box $[x]$ is the largest diameter of its components. The smallest box enclosing a set $\mathcal{S}$ in denoted by $\Box\mathcal{S}$.

A *sub-paving* of $[x]$ is a set of non-overlapping boxes included in $[x]$. A sub-paving will either be considered as a collection of boxes $\{[x]^{(1)}, \ldots, \}$ or as a union $[x]^{(1)} \cup \ldots$ depending on the context. Hence, a sub-paving can either be viewed as a discrete subset of $\mathbb{IR}^n$ or as a compact subset of $\mathbb{R}^n$.

A *paving* of $[x]$ is a collection of *sub-pavings* $\mathbb{K}_1, \ldots, \mathbb{K}_N$ such that

$$[x] = \bigcup_{1 \leq k \leq N} \bigcup_{[b] \in \mathbb{K}_k} [b].$$

Finally, we will temporarily adopt the following definition. Let c be a constraint on the reals, i.e. $c(x)$ is either true or false. A *contractor for* c is a function $C : \mathbb{IR}^n \to \mathbb{IR}^n$, usually given by a polynomial-time algorithm, such that:

$$\forall [b] \in \mathbb{IR}^n, \begin{cases} C([b]) \subseteq [b] \text{ and} \\ \forall x \in [b] \backslash C([b])\ c(x) \text{ is not satisfied.} \end{cases}$$

Note that the concept of *contractor* will be generalized in our formalism (see Section 3). Thanks to interval arithmetics, many contractors can be built with respect to an equation $f(x) = 0$, where $f : \mathbb{R}^n \to \mathbb{R}^m$.

Example 1. Consider $f : \mathbb{R}^2 \to \mathbb{R}$ with $f(x) = x_1 - \exp(x_2)$ and define $C : \mathbb{IR}^2 \to \mathbb{IR}^2$ as follows: $C([x]) := ([x_1] \cap \exp([x_2]), [x_2] \cap \log[x_1])$. C is a contractor with respect to $f(x) = 0$.

Contractor versus propagator. In spite of appearance, *contractor* is not just a new name for *propagator*. A propagator is associated to a constraint and destined to be called in a propagation loop. As we will see, a contractor is not necessarily associated to a constraint and may have nothing to do with propagation. Furthermore, we will add a *continuity* condition in the definition of a contractor (see §3.2) that does not exist for propagators. Finally, the term *filtering* was not chosen because of a conflict with the everyday meaning of *filtering* in many areas dealing with continuous systems (e.g., *Kalman filtering*).

1.2 The ABC of interval programming

Let us end this introduction with a short description of what interval programming is all about.

The basic algorithm of interval programming is the *root finder*. Consider a system of equations $f(x) = 0$, where x is a variable in $\mathbb{R}^n$ and f a mapping from $\mathbb{R}^n$ to $\mathbb{R}^m$. A root finder wraps solutions into boxes of any desired precision (leaving aside floating-point considerations) by using a branch & prune process, where the *prune* operation is performed by a contractor.

Here is the sketch of the algorithm. Note that a stack is used for the depth-first search and that the precision criterion chosen here is width$[x] < \varepsilon$.

```
algorithm root-finder(function f, box [x]^(0))
output: a sub-paving
 push [x]^(0) on the stack
 while the stack is not empty
 | pop [x] from the stack
 | [x]' ← C([x]) where C is a contractor w.r.t. f(x) = 0
 | if [x]' is not empty
 | | if width [x]' < ε
 | | | insert [x]' into the output sub-paving
 | | else
 | | | bisect [x]' into two subboxes [x]^(1) and [x]^(2)
 | | | push [x]^(1) and [x]^(2) on the stack
```

Of course, the existing implementations are much more sophisticated. But this simplified description is enough from a language perspective. In the ideal case of a square system ($m = n$) without singularity, solutions are punctual (and certification can be demanded). In the general case, a $n-m$ dimensional surface is described. This is illustrated in Figure 1.

Consider now $f : \mathbb{R}^n \to \mathbb{R}^m$, and $[x] \times [y] \in \mathbb{R}^{n+m}$. We may look for the following *set inverse*:

$$f^{-1}([y]) \cap [x] := \{x \in [x] \mid \exists y \in [y]\ y = f(x)\}.$$

Note that set inversion is a particular subclass of quantified constraints (see, e.g., [12, 39]). The `root-finder` algorithm will split the whole solution set into small boxes, as depicted in Figure 2.a.

In this case (i.e., when the solution set has a non-null n-dimensional volume) we might expect the solver to dissociate boxes that may contain solutions (*boundary boxes*) from boxes that only contain solutions (*inner boxes*). Clearly, precision is only required for boundary boxes, splitting an inner box being useless and even counter-productive.

A more appropriate output is given in Figure 2.b. Hence, an enhanced algorithm `set-describer` can be derived from `root-finder` by inserting an inner test

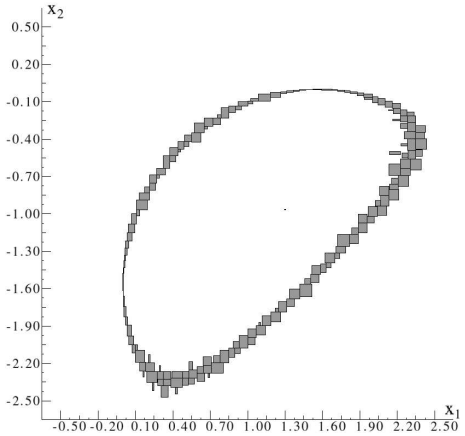

Fig. 1. An enclosure of $\{x \in [-0.5, 2.5] \times [-2.5, 0.5],\ f(x) = 0\}$ with $f(x) = \exp(x_1 x_2) - \sin(x_1 - x_2)$.

before the call to the contractor. The design of inner tests has been a matter of study in several publications (see, e.g., [13, 18, 14]).

2 Motivating examples

The purpose of this section is to present five different problems tractable by interval programming and to show that neither `root-finder` nor `set-describer` are appropriate. Once again, we do not claim that no algorithm exists for these problems. We just claim that no algorithm exist that solves them all. Note that `root-finder` and `set-describer` will not be two different algorithms in our formalism.

2.1 A decomposable problem

In several applications, systems are sufficiently sparse to be decomposed by equational or geometric techniques [24, 36, 6]. Consider for example the following system of distance constraints in two dimensions. There are 5 points connected by 6 distance equations. Points A and B are fixed and the three others are the unknowns. Instead of directly solving the 6x6 overall system, we can process progressively in three steps. We can solve the first triangle $\{A, B, C\}$, which is a 2×2 system. Because of rigidity, there is only a finite number of solutions (two flips). Then, for each solution, the second triangle $\{B, C, D\}$ can be solved because C is fixed. Finally, the triangle $\{C, D, E\}$ can be solved in turn once C and D are fixed.

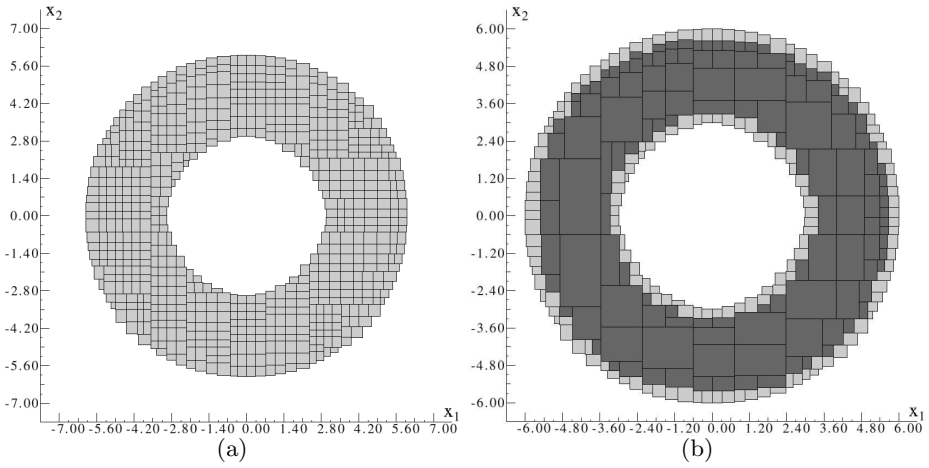

Fig. 2.
A set inversion problem $f(x) \in [y]$ with $f(x) = \sqrt{x_1^2 + x_2^2}$ and $[y] = [3, 6]$ in $[x] = [-7, 7] \times [-7, 7]$. (a) The output of the set inversion problem as defined in §2.1 (i.e., small boxes enclosing all the solutions). (b) A more adapted output for the same set inversion problem, including inner test.

In general, solving the decomposed problem (here, seven 2×2 systems including flips) is faster than solving the global problem (here, one 6×6 system). No standard root finder supports the implementation of such strategies. The state of the art in decomposition-based solving methods relies on dedicated algorithms.

2.2 Enhanced description of the ring

Let us consider again the set inversion problem introduced in §1.2. We mentioned that an inner test was necessary (whence the `set-describer` algorithm) but an *inner contractor* could also be introduced in this case and would actually be much more adapted.

Indeed, the set inversion problem can be easily formulated as a conjunction of two inequalities. It has been shown (see [8, 2]) in the case of inequalities that, by a negation trick, the problem can be reversed so that any contracted region of the reversed problem lies inside the feasible region of the original one.

Remember that $[y] \in \mathbb{IR}$. We have

$$c(x) \iff f(x) \in [y] \iff y^- \leq f(x) \text{ and } f(x) \leq y^+$$

Hence, a point is unfeasible if

$$y^- > f(x) \text{ or } f(x) > y^+.$$

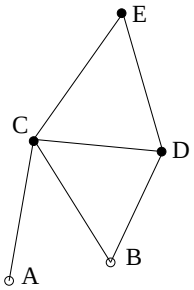

Fig. 3. A decomposable system.

Now, we can contract a box $[x]$ with respect to $y^- > f(x)$ and $f(x) > y^+$. If $[x]^{(1)}$ and $[x]^{(2)}$ are the two resulting boxes, then $[x]\backslash([x^{(1)}] \cup [x^{(2)}])$ is inside the solution set $\{x \in [x] \mid c(x)\}$. In such situation, it would be fruitful to replace the current search space $[x]$ by $[x^{(1)}] \cup [x^{(2)}]$ and to memorize that the complementary space is inner. The `set-describer` algorithm is therefore not really appropriate.

2.3 Level surfaces

Level surfaces is a generalization of the previous problem, and can be stated as follows. Consider $f : \mathbb{R}^n \to \mathbb{R}$ and N decreasing values $y_1 > \ldots > y_N$. The goal is to classify all the points of an initial domain $[x]$ according to the constraints:

$$\begin{array}{r} f(x) \geq y_1 \\ \text{or } f(x) \geq y_2 \\ \vdots \\ \text{or } f(x) \geq y_N, \end{array} \tag{1}$$

with a **priority** corresponding to the intuitive idea that we are more interested in the *highest* values: the surface level y_1, i.e., the number of points satisfying $f(x) \geq y_1$ must be maximized first. Then, the surface level y_2 must be maximized and so on. Figure 4.a shows an example of 4 surface levels. Note that this problem could be easily generalized to a MAX-SAT problem.

Let us now focus on input/output. Of course, using directly (1) as an input constraint makes no sense, since (1) is equivalent to $f(x) \geq y_N$. One could rather solve N times the set inversion problem $f(x) \in [y_k, y_{k+1}]$ and superimpose the different outputs (see Figure 4.b). However, this is not satisfactory for at least two reasons.

First, the boundaries around each level surface (the little white boxes in Figure 4.b) should not appear, except for the lowest one. This undesirable boundaries will be cleared off in Figure 11. One could circumvent this effect by slightly enlarging the intervals $[y_k, y_{k+1}]$ before computing the inversions, making level

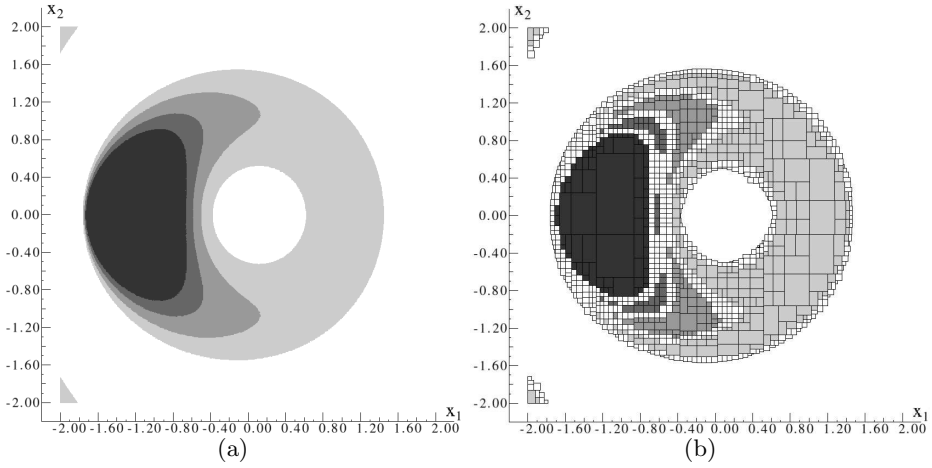

Fig. 4. Surfaces levels $f(x) \geq y_i$. (a) An example with $y_1 = 0.8$, $y_2 = 0.6$, $y_3 = 0.4$, $y_4 = 0.2$ and $f(x) = \sin(x_1^2 + x_2^2)/(\exp(x_1) + x_2^2)$, in $[x] = [-2, 2] \times [-2, 2]$. Darkness increases with f. (b) The same example decomposed into set inversion problems.

surfaces overlap. Another trick would simply be to move the boundary sub-paving of the k^{th} surface level into the inner sub-paving of the $(k-1)^{th}$ level surface. However, in both cases, we "cheat" and lose reliability of the result.

Second, the decomposition of the input problem into N sub-problems and the aggregation of the intermediate results requires extra and undesirable manipulations from the user.

2.4 Set inclusion

Given two sets defined by constraints c_1 and c_2, the *set inclusion* problem consists in proving or refuting that one set is included in the other. This is illustrated in Figure 5.a.

We shall denote by set(c) the set associated to a constraint:

$$\text{set}(c) = \{x \in \mathbb{R}^n \mid c(x) \text{ is true}\}.$$

Using the conjunction of c_1 and c_2 as an input in algorithm `set-describer` may address the problem in the following situations: if no solution is found, then no set can be included in the other since these sets do not intersect at all. In the other way around, if the whole initial box is included in the inner sub-paving then the two sets coincide. Except in these extreme cases, nothing can be said.

As before, one may solve separately c_1 and c_2, and check for the inclusion of both the inner & boundary sub-pavings of c_1 into the inner sub-paving of c_2 (see Figure 5.b). Again, this requires undesirable manipulations by the user; but, worse, this

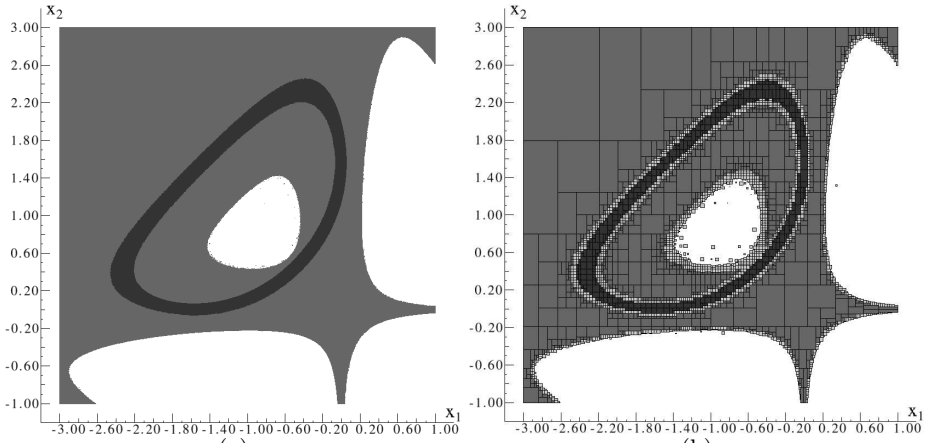

Fig. 5. Set inclusion. (a). Two subsets of $[x] = [-3, 1] \times [-1, 3]$ defined by constraints: $c_1(x) \iff \exp(x_1 x_2) - \sin(x_2 - x_1) \in [-0.1, 0.1]$ (in dark gray) and $c_2(x) \iff \exp(x_1 - x_2) \times \sin(x_1 x_2) \in [-0.1, 0.1]$ (in light gray). We have $\text{set}(c_1) \subseteq \text{set}(c_2)$. (b) The same set inclusion problem decomposed into set inversion problems.

process is extremely inefficient. Assume that a box B in Figure 5.b is detected as inner in the paving of $\text{set}(c_2)$ and recursively bisected and contracted in the paving of $\text{set}(c_1)$ until either boundary or inner boxes are found. All this work could be spare since the inclusion $\text{set}(c_1) \cap B \subseteq \text{set}(c_2) \cap B$ is already established. Hence, this decomposition involves a huge number of useless computations.

Of course, one could legitimately object that in the example given in Figure 5, c_1 can be reversed as in §2.2 to $\bar{c_1}$ so that running a standard solver with $\bar{c_1}$ and c_2 will do the job as expected. Indeed, if no solution is found, no point satisfy c_2 and $\bar{c_1}$, i.e., $\text{set}(c_2) \subseteq \text{set}(c_1)$. However, in general, a constraint cannot be reversed while an inner test is still possible. As an example, a constraint $c_1(x)$ of the following form:

$$c_1(x) \iff \exists y \in [0, 1] \begin{cases} \text{f(x,y)=0} \\ \text{g(x,y)=0} \end{cases}$$

cannot be reversed if y has multiple occurrences while testing for an inner box is possible (an algorithm is given in [12]).

2.5 Bounded-error Parameter estimation

This last example is rather devoted to solver design: the problem is not new in terms of input/output but requires a specific strategy.

Consider the following model of a real-time system [23]:

$$f(p, t) = 20 \exp(-p_1 t) - 8 \exp(-p_2 t)$$

where p_1 and p_2 are two unknown parameters. Assume that a vector of 10 measurements $y_1, \ldots, y_{10}$ is available, these measurements corresponding to times $t_1, \ldots, t_{10}$ respectively.

The question is to compute a rigorous enclosure of all the feasible p_1 and p_2 from the experimental data, taking into account uncertainties on both y_i and t_i. Hence, given intervals $[t_i]$ and $[y_i]$, we have to describe the following set:

$$\{(p_1, p_2) \mid \forall i \in [1..10],\ \exists y_i \in [y_i],\ \exists t_i \in [t_i]\ y_i = f(p, t_i)\}.$$

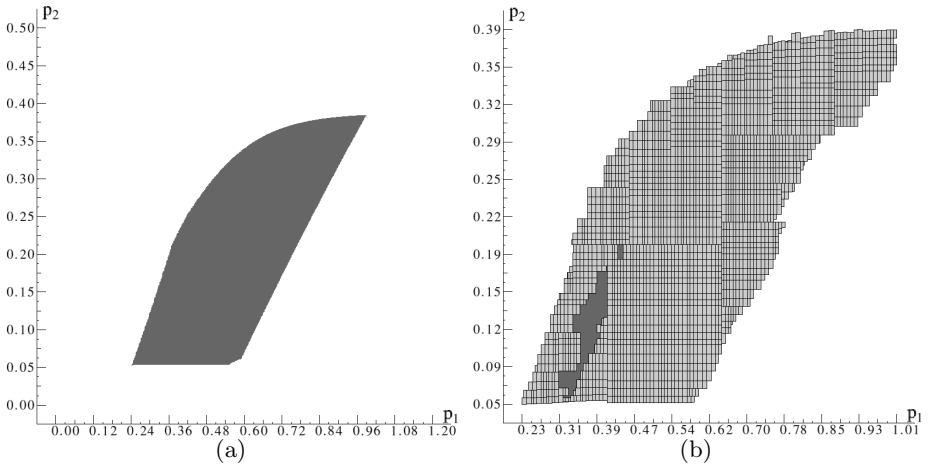

Fig. 6. The parameter estimation problem. (a) The set of feasible points (p_1, p_2). (b) The result using a standard contraction and inner test without splitting the domains of t_i (bisection precision is 0.01). First, a very few of inner boxes could be found. Second, the outer enclosure is not sharp at all.

If we restrict ourselves to a single measurement (instead of 10), the problem boils down to a simple set inversion, with a *thick* constant $[t_i]$. Hence, the set to be computed seems to be nothing but the intersection of 10 set inversions. The `set-describer` algorithm seems to suit well.

In fact, the multi-incidence of t in the expression of f makes this strategy inefficient. Indeed, whatever is the underlying method used for the contractor or the inner test, the domain of t has to be small for the resulting operator to be sharp. Hence, the domain of t has to be split by the solver. This is illustrated in Figure 6.

Second, introducing the vector t as variables makes the search space dimension jumps from 2 to 12 unless one wants to solve the 10 problems separately. This leads to a combinatorial explosion.

In conclusion, there is a need for splitting the component t_i (and only this one) inside the contractor and inner test associated to the i^{th} measure. To our knowledge, no available solver is generic enough to be tuned in this way.

3 A new formalism

We show now that the different examples of Section 2 can be solved as suggested by the very same algorithm, a generic solver called *paver*, when expressed in the contractor formalism.

We describe the paver and the formalism in this section. We will provide concrete examples of contractors in Section 4 and subsequently revisit the problems of Section 2.

3.1 The "contract and classify" paver

To help intuition, let us consider the problem in §2.3. It turns out from this example that several sets have to be described simultaneously, each set corresponding to a different constraint.

Let us assume that sets are disjoint. We will see below how to circumvent the case of overlapping sets.

Each point of the initial space has to be classified with respect to these sets. For a given point x, this means that the membership of x to the different sets can be tested successively. If one test succeeds, then x can be marked as treated and associated to the corresponding set.

A generic interval solver can directly be derived from this "test and classify" principle. Remember first that a contractor C can be easily built from any constraint c. The contractor C has the following property: if $C([x]) = [y]$ (i.e., $[y]$ is the contraction of $[x]$) then every eliminated point in $[y]\backslash[x]$ does not satisfy c. Thus, with a simple negation trick, a contractor $\bar{C}$ such that all eliminated points satisfy c is as easy to build.

Example 2. Consider the set inversion problem, in §2.2. A point inside the ring satisfies $f(x) \in [y]$. Consider a contractor $\bar{C}$ built from $f(x) \notin [y]$. If $\bar{C}([x]) = [y]$ then $[x]\backslash[y]$ is inside the ring.

Given a box $[x]$ and a constraint c, if the complementary contractor $\bar{C}$ reduces $[x]$ to $[y]$, then $[x]\backslash[y]$ can be classified, i.e., mark as "inside c". Then, the paver proceeds as follows (the algorithm is detailed below). The solver takes as input a list of contractors and an initial box $[x]^{(0)}$. The first contractor is enforced on $[x] := [x]^{(0)}$. If the contraction is effective, a smaller box $[x]'$ is returned and the difference[4] $[x]\backslash[x]'$ is stored in a sub-paving associated to the first contractor.

[4] Note that resorting to a simple tree structure avoids to explicitly describe the difference between $[x]$ and $[x]'$: the latter simply becomes a subnode of the former.

Only $[x]'$ is left to be treated, and the second contractor is called. When all the contractors become ineffective on a box $[x]$, we can say that the common fixpoint of all contractors is reached and $[x]$ is bisected. The whole process is then repeated until the list of boxes becomes empty. We will see in the next section how the termination of the algorithm can be guaranteed.

```
algorithm paver(contractors C_1, ..., C_k, box [x]^(0))
output: a paving of [x]^(0)
 create k empty subpavings K_1, ..., K_k
 push [x]^(0) on the stack
 while the stack is not empty
 | pop [x] from the stack
 | do
 | | fixpoint ← true
 | | i ← 1
 | | while [x] ≠ ∅ and i≤k
 | | | [x]' ← C_i([x])
 | | | if [x]' ≠ [x]
 | | | | K_i ← K_i ∪([x]\[x]')
 | | | | fixpoint ← false
 | | | | [x] ← [x]'
 | | | i ← i + 1
 | while [x] ≠ ∅ and fixpoint=false
 | if [x] ≠ ∅
 | | bisect [x] into two subboxes [x]^(1) and [x]^(2)
 | | push [x]^(1) and [x]^(2) on the stack
```

Note that `paver` is not an AC3-like algorithm, for essentially two reasons. First, contractors given to the paver algorithm usually involve all the variables merely because they all correspond to sets (or more generally, to sub-pavings) which have the same dimension as the initial box. This is not a rule but a "user advice". One can put different constraints at this level in order to trace their contraction power. But, the paver level is intended to deal with sets while all the constraints related to a given set should be encapsulated in the same *outer* or *inner* contractor, as explained further. Second, because the semantics of contractors goes beyond that of constraints, the order in which they are called may have an influence on the quality of the result (the number of boxes in a subpaving). For instance, a contractor designed to detect a very weak condition should preferably be called on last resort (we might lose boxes satisfying a stronger condition).

3.2 Contractors

Definition 1 (Contractor). *A contractor is a mapping C from $\mathbb{IR}^n$ to $\mathbb{IR}^n$ such that*

$$\begin{array}{ll} \textit{(i)} \quad \forall [x] \in \mathbb{IR}^n, \mathcal{C}([x]) \subseteq [x] & \textit{(contraction)} \\ \textit{(ii)} \quad (x \in [x], \mathcal{C}(\{x\}) = \{x\}) \Rightarrow x \in \mathcal{C}([x]) & \textit{(consistency)} \\ \textit{(iii)} \; C(\{x\}) = \emptyset \Leftrightarrow (\exists \varepsilon > 0, \forall [x] \subseteq B\,(x, \varepsilon)\,, \mathcal{C}([x]) = \emptyset) & \textit{(continuity)} \end{array}$$

where $B\,(x, \varepsilon)$ is the ball centered on x with radius ε.

A box $[x]$ is said to be *insensitive* to C if $C([x]) = [x]$ and *sensitive* otherwise. By extension, a point x is said to be sensitive or insensitive whether $\{x\}$ is sensitive or not. Property (i) states that a box can only be reduced by a contractor. Property (ii) states that no insensitive points can be removed. Finally, property (iii) is required for properties on the paver as we will explain further.

The set associated to a contractor C is the union of all of its insensitive points:

$$\text{set}(C) = \{x \in \mathbb{R}^n, C(x) = x\}.$$

The *continuity* property of C implies that $\text{set}(C)$ is closed.

If C is a contractor for a constraint c, in the classical sense of the word (see §1.1), then $\text{set}(C) \supseteq \text{set}(c)$. An important novelty in our formalism is to consider sets associated to contractors rather than to constraints. This allows a rigorous description of the output of our paver (see Proposition 2). A contractor is not only an algorithm, it can also be interpreted as a subset of $\mathbb{R}^n$, and all the standard operations on sets can be extended to contractors. We define:

$$\begin{array}{rll} (C_1 \cap C_2)([x]) & := C_1([x]) \cap C_2([x]) & \text{(intersection)} \\ (C_1 \cup C_2)([x]) & := \square\Big(C_1([x]) \cup C_2([x])\Big) & \text{(union)} \\ (C_1 \circ C_2)([x]) & := C_1(C_2([x])) & \text{(composition)} \\ C_1^{\infty} & := C_1 \circ C_1 \circ C_1 \circ \ldots & \text{(iterated composition)} \\ C_1 \sqcap C_2 & := (C_1 \cap C_2)^{\infty} & \text{(iterated intersection)} \\ C_1 \sqcup C_2 & := (C_1 \cup C_2)^{\infty} & \text{(iterated union).} \end{array} \tag{2}$$

All these operations are *stable*, i.e., they only yield contractors.

We also introduce the following definition:

Definition 2. *Let $C_1, \ldots, C_k$ be a collection of contractors.*

$$\begin{array}{l} C_1, \ldots, C_k \textit{ are } \textbf{complementary} \textit{ if } \text{set}(C_1) \cap \ldots \cap \text{set}(C_k) = \emptyset, \\ C_1, \ldots, C_k \textit{ are } \textbf{independent} \textit{ if } \forall i \neq j, \textit{ set}(C_i)^C \cap \text{set}(C_j)^C = \emptyset. \end{array}$$

Additional properties of contractors play a significant role:

$$\begin{array}{ll} C \text{ is } \textit{monotonous} \text{ if } & [x] \subseteq [y] \Rightarrow C([x]) \subseteq C([y]), \\ C \text{ is } \textit{minimal} \text{ if } & \forall [x] \in \mathbb{IR}^n,\ C([x]) = \square([x] \cap \text{set}(C)), \\ C \text{ is } \textit{idempotent} \text{ if } & \forall [x] \in \mathbb{IR}^n, C(C([x])) = \mathcal{C}([x]). \end{array} \tag{3}$$

Given a contractor C on $\mathbb{R}^n \times \mathbb{R}^m$, we also define the two following contractors on $\mathbb{R}^n$:

$$C^{\cup[y]}([x]) := \bigcup_{y\in[y]} \pi_x(C([x], y)$$
$$C^{\cap[y]}([x]) := \bigcap_{y\in[y]} \pi_x(C([x], y)$$

with $\pi_x([x], [y]) := [x]$. The first operation is called *proj-union* and the second *proj-intersection*. Once again, these operations are stable in the set of contractors. We have:

$$\text{set}(C^{\cup[y]}) = \{x,\ \exists y \in [y],\ (x, y) \in \text{set}(C)\},$$
$$\text{set}(C^{\cap[y]}) = \{x,\ \forall y \in [y],\ (x, y) \in \text{set}(C)\}.$$

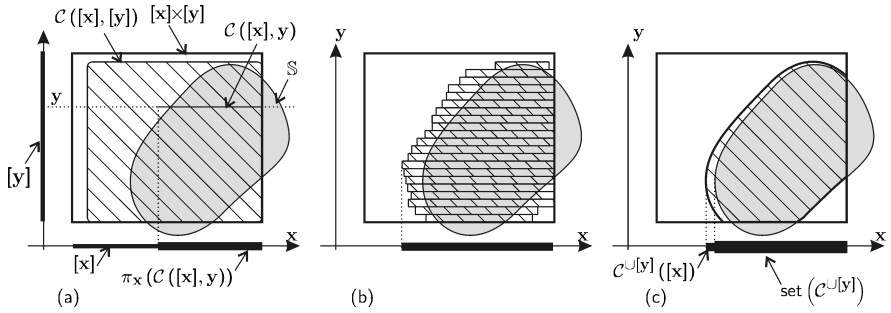

Fig. 7. Proj-Union. We consider a contractor C on $\mathbb{IR}^2$ with $\mathbb{S} := \text{set}(C)$. (a) Representation of $C^{\cup}[y]$ when $[y]$ is degenerated (a real y). The contraction on $[x]$ results in an enclosure of $\{x \in [x] \mid (x, y) \in \mathbb{S}\}$. Since C is continuous, when both $[x]$ and $[y]$ are degenerated, the enclosure is minimal, i.e., $C^{\cup y}(x) = \{x\}$ iff $(x, y) \in \mathbb{S}$. (b) Principle of an implementation resorting to parameter splitting and union of sub-contractions (the interval $[y]$ is decomposed as finely as possible). (c) Result of the proj-intersection in practice: only an outer approximation of $\text{set}(C^{\cup[y]})$ is obtained, after numerous repetitions of the splitting operation presented in (b).

This result being rather intuitive, the proof is not given (see Figures 7 and 8). In practice, these operators are implemented as follows. For the *proj-union*, $[y]$ is split into a sequence of small subintervals $[y]_i$. Then, the contractor C is enforced on each sub-domain $[x] \times [y_i]$ and the hull of the results is returned. For the *proj-intersection*, a sequence of points y_i are sampled from $[y]$, C is enforce on $[x] \times \{y_i\}$ and the intersection of the results is returned.

Although examples of contractors will be the topic of Section 4, we shall introduce right here the *precision contractor* C_ε that has a special status. This contractor is aimed at controlling the precision of the paver or scheduling contractors (see §5.1 and §5.3) and will be implicitly referred to in the subsequent propositions.

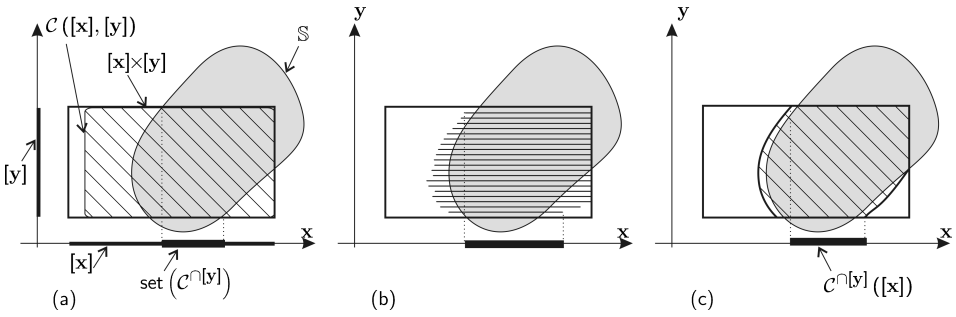

Fig. 8. Proj-intersection. (a) In a similar way as in Figure 8, given a fixed interval $[y]$, $C^{\cap[y]}$ allows to constrain $x \in \mathbb{R}$ with $\forall y \in [y]$, $(x, y) \in \mathbb{S}$. (b) Principle of an implementation resorting to parameter splitting and intersection. (c) Result of the proj-intersection in practice: only an outer approximation of set$(C^{\cap[y]})$ is obtained.

Definition 3 (Precision contractor). *Given $\epsilon > 0$, the precision contractor C_ε is defined as follows:*

$$C_\varepsilon([x]) := \begin{cases} [x] \text{ if } \text{width}([x]) > \epsilon \\ \emptyset \quad \text{otherwise.} \end{cases}$$

Many properties could be stated on contractors. As an example, the set of idempotent and monotonous contractors is a complete lattice for the inclusion. Listing these properties would be out of the scope of this paper, which is rather aimed at introducing the concepts. However, the fundamental properties of the paver given in the previous section are based on the properties of contractors and we are now in position for proving them.

3.3 Paver properties

Proposition 1 (Termination). *Let $\{C_1, \ldots, C_k\}$ be a list of complementary contractors. The* `paver` *terminates for any initial box $[x]^{(0)}$.*

Proof. Assume by contradiction that `paver` does not terminate. Hence, there is a sequence of non-empty boxes $[x]^{(i)}$ and $\tilde{x} \in [x]^{(0)}$ such that for all $i > 0$:

$$\begin{array}{ll} (i) & [x]^{(i)} \subset [x]^{(i-1)} \text{ (strict inclusion, i.e., } [x]^{(i)} \neq [x]^{(i+1)}) \\ (ii) & C_1([x]^{(i)}) = \ldots = C_k([x]^{(i)}) = [x]^{(i)} \\ (iii) & \tilde{x} \in [x]^{(i)} \end{array}$$

Indeed, (i) is a direct consequence of the properties of contractors and bisections and (ii) is due to the fixpoint postcondition of the inner `while` loop inside the

`paver` algorithm. Finally, for all i, pick a point $\tilde{x}^{(i)} \in [x]^{(i)}$ (which is nonempty). The sequence is bounded so it admits at least one accumulation point $\tilde{x}$. Since for all i, $[x]^{(i)}$ is a closed set that contains the whole subsequence $(\tilde{x}^{(j)})_{j \geq i}$, it also contains this accumulation point $\tilde{x}$. Hence, (iii) holds.

Now, each box $[x]^{(i)}$ can be identified to a $2n$-tuple $u^{(i)}$ of its vertices coordinates. This identification can be made isometric with appropriate distances (the Hausdorff distance on $\mathbb{IR}^n$ and its counterpart on $\mathbb{R}^{2n}$). For all component j, $1 \leq j \leq 2n$, the sequence $u_j^{(i)}$ is either increasing or decreasing, and bounded by a component of $\tilde{x}$. Hence, it converges to a point $\bar{u}_j$. By the inverse isometry, it follows that the sequence of boxes $[x]^{(i)}$ converges then to a box $[\bar{x}]$. Furthermore, the width of this box is necessarily null (i.e., $[\bar{x}]$ is a degenerated box $\{\bar{x}\}$) since

$$\forall i \geq 0, \quad \mathrm{rad}([x]^{(i+n)}) \leq \frac{1}{2}\mathrm{rad}([x]^{(i)}).$$

Next, we have $\mathrm{set}(C_1) \cap \ldots \cap \mathrm{set}(C_k) = \emptyset$ which means that at least one contractor C_l satisfies $\bar{x} \notin \mathrm{set}(C_l)$. By applying the *continuity* property of C_l (see Definition 1) we obtain that there exists ϵ and i such that $[x]^{(i)} \subseteq B(\bar{x}, \epsilon)$ and $C_l([x]^{(i)}) = \emptyset$, i.e., a contradiction. ■

To ensure the complementarity between contractors, we usually resort to the precision contractor because $\mathrm{set}(C_\varepsilon) = \emptyset$.

Informally, the following proposition states that, up to a given precision ϵ, the sub-paving associated to a contractor matches the intersection of the other contractors sets. Since the set of a contractor usually approximates the set related to an initial constraint, we have now a clear semantics for the `paver`.

Proposition 2 (Sub-pavings characterization). *Let $\{C_1, \ldots, C_k\}$ be a list of independent contractors and C_ε such that $\mathrm{set}(C_\varepsilon) = \emptyset$. Let us denote by $\mathbb{K}_1, \ldots, \mathbb{K}_k$ and $\mathbb{K}_\epsilon$ the sub-paving returned by* `paver` *for $C_1, \ldots, C_k$ and C_ε respectively. We have:*

$$\forall i,\ 1 \leq i \leq k, \quad \bigcap_{j \neq i} \mathrm{set}(C_j) \setminus \mathbb{K}_i \subseteq \mathbb{K}_\epsilon\,.$$

Proof. First of all, `paver` terminates thanks to the previous proposition. Let i be an index, $1 \leq i \leq k$. For all $j \neq i$ we have by hypothesis $\mathrm{set}(C_i)^C \cap \mathrm{set}(C_j)^C = \emptyset$ hence $\mathrm{set}(C_i)^C \subseteq \mathrm{set}(C_j)$ which implies $\mathrm{set}(C_i)^C \subseteq \cap_{j \neq i} \mathrm{set}(C_j)$. Since $\forall [x] \in \mathbb{K}_i$ we have $[x] \subseteq \mathrm{set}(C_i)^c$ (by applying the *consistency* of C_i), then $\mathbb{K}_i \subset \mathrm{set}(C_i)^C \subset \cap_{j \neq i} \mathrm{set}(C_j)$. The two last inclusions are strict because $\mathrm{set}(C_i)^C$ is open while the other sets are closed. It follows that $\bigcap_{j \neq i} \mathrm{set}(C_j) \setminus \mathbb{K}_i$ is nonempty.

Take $[x]$ in this set and assume $[x] \notin \mathbb{K}_\epsilon$. The box $[x]$ has been removed from the search by C_l with $1 \leq l \leq k$, i.e. $[x] \subseteq \mathrm{set}(C_l)^C$. Since $[x] \in \bigcap_{j \neq i} \mathrm{set}(C_j)$ then $l = i$ which leads to a contradiction since $[x] \notin \mathbb{K}_i$. ■

4 Contractor design

In this section, we present some contractors and operations that do not help in modeling problems but in improving the efficiency of a paver. We basically explain how to manage a constraint c, i.e., the different ways to build a contractor C satisfying $set(C) = set(c)$. Likewise, all the operations *op* will preserve associated sets. E.g., an unary operator *op* will satisfy for all contractor C, $set(op(C)) = set(C)$.

Hence, contractors and operations in this section will not have an influence on the result, in terms of sub-paving characterization. On the other hand, they may decrease the time complexity or the space complexity (i.e., the number of boxes in a sub-paving).

We will overlook technical details on purpose here, sticking to generalities. The interested reader may refer to Section 6 where important implementation choices we made for our own system are disclosed.

We shall distinguish *numerical contractors*, those which are directly built from a numerical constraint, and those which are the result of an operation (such as composition) between other contractors.

Numerical contractors can be related to equations, inequalities or systems of equations. The point is that a large part of interval analysis routines can be wrapped into these contractors. As announced above, a good separation can then be obtained between numerical skills (the design of numerical contractors) and constraint programming skills (the design of compound contractors).

In this layered framework, constraint propagation will be generalized to *contractor propagation*. A possible implementation of the corresponding operator (`propag`) will be described in detail in §6.3.

4.1 Numerical contractors

Several contractors can be associated to an equation $f(x) = 0$.

The simplest one consists in evaluating with interval arithmetics $f([x])$ and checking whether 0 belongs to the image range or not. If $0 \notin f([x])$ then $[x]$ can be contracted to the empty set. Otherwise, it is left unchanged. This contractor can therefore be qualified as *binary* (in the sense that it keeps all or nothing). This test can be easily extended to inequalities. This contractor can be given many variants, based on various symbolic or numerical processing as the two following examples show.

Example 3. Assume that there exist several equivalent expressions for f, say $f_1, \ldots, f_k$, each of them minimizing the multi-incidence of a different variable. Since the overestimation of interval evaluation grows with the multi-incidence of variables (see [34]), we get to sharper results by computing $f_1([x]) \cap \ldots \cap f_k([x])$ instead of $f([x])$, whence a more accurate (but slower) test.

Example 4. On the numerical side, we can resort to the centered form, i.e., compute $f(x_0) + f'([x])([x] - x_0)$ instead of $f([x])$ where x_0 is an arbitrary point inside $[x]$. Higher-order Taylor formulas may also be payfull. One may also introduce intermediate *monotonicity* tests. Consider a two-dimensional vector x. If $\frac{\partial f}{\partial x_1}([x])$ turns to be a positive interval then f is increasing with respect to x_1 so that $f([x])$ can be replaced by $[f(\{x_1^-\} \times [x_2]), f(\{x_1^+\} \times [x_2])]$, a smaller interval.

More sophisticated contractors associated to a constraint c (equation or inequality) compute an outer estimation of the feasible set, i.e., $C([x])$ is a nontrivial superset of $\{x \in [x] \mid c(x)\}$. Three important techniques can be found in the literature:

1. Forward-backward traversals of the syntax tree with intermediate interval computations, such as `HC4Revise` [3].
2. Univariate interval Newton iterations (as performed by the *narrowing* operator of box consistency [47]).
3. Linear relaxations (see [30] and references therein). In such techniques, a nonlinear constraint is cast into a linear program which feasible region encompasses the original solution set.

Given an equation $f(x) = 0$, we will denote by C_f in the sequel an arbitrary contractor among those above.

Of course, numerical techniques can also handle several equations simultaneously, i.e., a system of equations. Many of them are derived from a multivariate interval Newton, one famous variant being the Hansen-Sengupta algorithm [17]. In case of linear equalities, many dedicated algorithms also exist (see [35] or [37]). One can be easily convinced that all these algorithms act as contractors.

4.2 Propagation

The propagation operator allows the implementation of interval variants of the classical `AC3` algorithm such as *hull consistency* (see, e.g., [3, 11]) but also many more algorithms. This operator illustrates by itself the potential of our framework, in terms of contractor design. The key idea is to propagate contractors instead of constraints.

Given a list of contractors, the principle is to obtain the fixpoint of their composition, i.e., $(C_1 \circ \cdots \circ C_m)^\infty$ at a lower computation price. We shall denote by `propag` our operator. We have:

$$\text{set}(\texttt{propag}(C_1, \ldots, C_m)) = \text{set}((C_1 \circ \cdots \circ C_m)^\infty)$$

which means that `propag` is a *pure* efficiency contractor (it has no impact on the output).

An implementation of this operator will be proposed in §6.3.

Let us now recall that the *hull consistency* enforces bound consistency for each equation. In practice, only a relaxation of bound-consistency is calculated via the `HC4Revise` algorithm. The resulting propagation algorithm is called `HC4`. In our framework, given a list of constraints $c_1, \ldots, c_m$, the `HC4` algorithm is simply reimplemented as follows

$$\texttt{HC4}(c_1, \ldots, c_m) \leftarrow \texttt{propag}(\texttt{HC4Revise}(c_1), \ldots, \texttt{HC4Revise}(c_m))$$

A more interesting direct application is a multi-level propagation. Sometimes, we would like to group constraints we know to have strong "dependencies". For instance, one may find relevant to perform an intermediate fixpoint with constraints that strictly share the same variables. Consider three sets of constraints $\{c_1, c_2, c_3\}, \{c_4\}$ and $\{c_5, c_6\}$. The following contractor will ensure that a constraint in a subset (say, c_4) will not be wakened between constraints of other subsets (say, c_1 and c_2).

$$\texttt{propag(propag(}c_1, c_2, c_3\texttt{)},}c_4\texttt{,propag(}c_5, c_6\texttt{))}$$

4.3 Shaving

The purpose of this section is to show that other operators exist besides `propag`.

Shaving is an operation that allows implementing refutation techniques, similar to SAC [5] with discrete domains. With continuous domains, refutation is used to shrink endpoints only (instead of *any* value inside the domains) so that the structure of interval is always maintained (whence the fancy name *shaving*).

Detailing the algorithm here would take too much room. We shall only give a rough description.

A shaving operator `shave` takes a contractor C. Given a box $[x]$, the resulting contractor $\texttt{shave}(C)$ contracts "slices" with C, i.e., subboxes obtained from $[x]$ by restricting the domain $[x]_k$ of one component to a small subinterval $[x_k^-, x_k^- + \epsilon]$ or $[x_k^+ - \epsilon, x_k^+]$. When the result of the subcontraction is an empty set, the slice is removed. Otherwise, contraction is tried on a smaller slice. This recursion leads to consistent endpoints. More formally, the resulting box $[y] := (\texttt{shave(C)})(\texttt{[x]})$ satisfies $\forall k = 1, \ldots, n$

$$C([y]_1 \times \cdots \times [y]_{k-1} \times \{y_k^-\} \times [y]_{k+1} \times \cdots \times [y]_n) \neq \emptyset$$

and

$$C([y]_1 \times \cdots \times [y]_{k-1} \times \{y_k^+\} \times [y]_{k+1} \times \cdots \times [y]_n) \neq \emptyset.$$

The shaving operator can also be given many variants mainly because of the slicing which can either be *optimistic* ("try large slices first") or *pessimistic* ("check consistency of endpoints first").

Given m constraints $c_1, \ldots, c_m$, a box consistency operator (see [4, 45, 47]) can be reimplemented in our framework as follows:

$$\texttt{propag}(\texttt{shave}(\texttt{univNewton}(c_1)),\ldots,\texttt{shave}(\texttt{UnivNewton}(c_m)))$$

where `UnivNewton` is an univariate interval Newton contractor (see §4.1). The 3B consistency defined in [31] can be reimplemented as follows:

$$\texttt{shave}(\texttt{HC4}(c_1, \ldots, c_m)).$$

5 Revisiting examples

We now revisit all the examples in our framework. By convention, the sub-paving of the precision contractor will always be represented in white in the pictures. As a preliminary example, consider the system of equations in Figure 1 page 8 and remember that the goal is to enclose the solution set into a sub-paving. By a direct application of our formalism, we then need two contractors:

1. A contractor C_f, to remove unfeasible points.
2. The precision contractor C_ε.

The desired output is given in Figure 9. This time, the result of the outer contraction appears as a sub-paving with only unfeasible points. The roots are all in the sub-paving of the precision contractor.

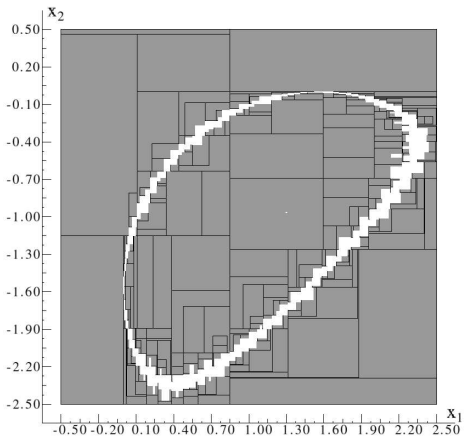

Fig. 9. System of equations, with an outer and precision contractor. The sub-paving of the outer contractor is in light gray.

5.1 A decomposable problem

Assume that each subsystem (called *block*) has to be solved by a combination of propagation and interval Newton. Of course, a multivariate interval Newton (denoted by `newton` henceforth) requires a square system. When one variable is fixed (e.g., x_C and y_C when the second triangle is to be solved), it must not be considered as a variable by `newton` but rather as an interval constant. Such information typically enters as a "static" parameter (see §6.1). For that reason, the `newton` contractor takes two arguments: the set of constraints and the set of variables. Similarly, let us refine the behaviour of the precision contractor C_ε (see §4). This contractor will also take here a subset $\mathcal{S}$ of variables in parameter. A box $[x]$ is emptied by $C_\varepsilon(\mathcal{S})$ if and only if the domains of the components in $\mathcal{S}$ all have a diameter lower than ε.

In the sequel, d_{PQ} denotes the distance constraint between two points P and Q.

For each block (here: a triangle), the solving strategy is implemented thanks to the following operator `block_contractor`:

$$\texttt{block\_contractor}(\{c_1,\ldots,c_k\},\{x_1,\ldots,x_k\}) \leftarrow \\ \texttt{HC4}(c_1,\ldots,c_k) \cap \texttt{newton}(\{c_1,\ldots,c_k\},\{x_1,\ldots,x_k\})$$

For instance, a paver run with $\texttt{block\_contractor}(\{d_{BD}, d_{CD}\}, \{x_D, y_D\})$ (and C_ε) with x_C and y_C fixed (i.e., their domains have a diameter less than ε) will solve the second triangle in the required manner, i.e., using propagation and Newton.

The step-by-step solving method proposed in §2.1 can then be directly implemented with the following contractor:

$$\begin{array}{l} \texttt{block\_contractor}(\{d_{AC}, d_{BC}\},\{x_C, y_C\}) \\ \cap\ (\ C_\varepsilon(\{x_C, y_C\}) \\ \quad \cup\ (\ \texttt{block\_contractor}(\{d_{BD}, d_{CD}\}, \{x_D, y_D\}) \\ \quad\quad \cap\ (\ C_\varepsilon(\{x_C, y_C, x_D, y_D\}) \\ \quad\quad\quad \cup\ \texttt{block\_contractor}(\{d_{DE}, d_{CE}\}, \{x_E, y_E\})))) \end{array}$$

provided that variables are bisected block after block. In more details, the variables of the first block $\{x_C, y_C\}$ must be bisected (say, in a round-robin fashion) until precision is achieved for both of them. The bisection procedure must then proceed to the second block, and so on. Branching procedures are not covered in this paper but this mechanism is orthogonal to contractors and one should be easily convinced that a block-wise bisection is easy to set up.

5.2 Enhanced description of the ring

In the set inversion problem, we need to characterize $\{x \mid f(x) \in [y]\}$. This means that two sets have to be described by contractors: $\{x \mid f(x) \in [y]\}$ (the ring)

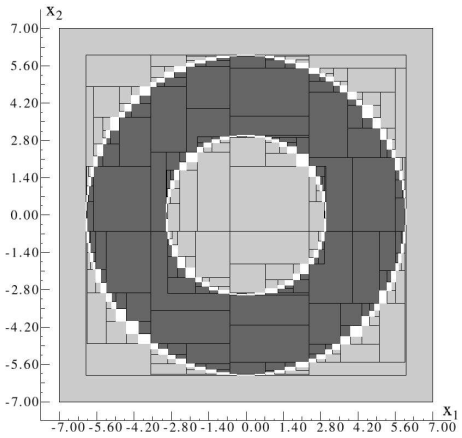

Fig. 10. A description of the ring involving outer and inner contractors.

and $\{x \mid f(x) \notin int([y])\}$ (the complementary of the ring). Put $[y] = [y^-, y^+]$ and $g(x) := f(x) - y$.

To deal with unfeasible points, we need an outer contractor. This contractor, denoted by $C_{f\in[y]}$, can be simply defined as follows:

$$C_{f\in[y]} := [x] \mapsto \pi_x(C_g([x], [y])).$$

The points inside the ring have to be classified with an *inner contractor*. As explained in §2.2, the latter comes naturally with the negation of the problem. We have

$$\begin{array}{rcl} c(x) & \iff & f(x) \in [y] \iff y^- \leq f(x) \text{ and } f(x) \leq y^+ \\ & \iff & f(x) \in [y^-, +\infty) \text{ and } f(x) \in (-\infty, y^+]. \end{array}$$

Hence,

$$\begin{array}{rcl} \bar{c}(x) & \iff & f(x) \in (-\infty, y^-) \text{ or } f(x) \in (y^+, +\infty) \\ & \implies & f(x) \in (-\infty, y^-] \text{ or } f(x) \in [y^+, +\infty) \end{array}$$

Therefore, the contractor

$$C_{f\notin[y]}([x]) := [x] \mapsto \pi_x(C_g([x], (-\infty, y^-])) \cup \pi_x(C_g([x], [y^+, +\infty)))$$

only removes unfeasible points for $\bar{c}$, i.e., feasible points for c.

Now, $\text{set}(C_{f\in[y]}) \cap \text{set}(C_{f\notin[y]})$ is the boundary of the ring, which is non empty. Hence, we have to add a precision contractor to ensure the termination of the algorithm. The result is depicted in Figure 10. Since $C_{f\in[y]}$ and $C_{f\notin[y]}$ are independent, the ring represented on the figure can be interpreted as an approximation of the set to be described (see proposition 2).

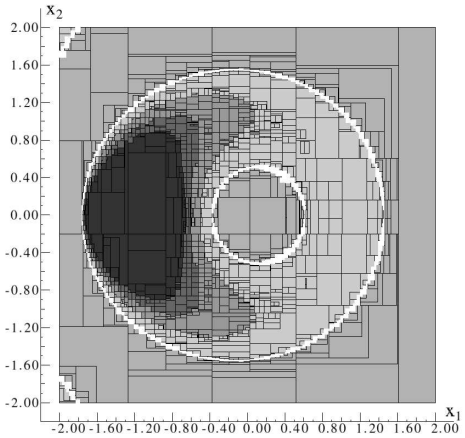

Fig. 11. An appropriate paving, with smooth boundaries between surface levels.

5.3 Level surfaces

First of all, the contractor $C_{f\in[y_i,+\infty)}$ (as defined in the previous paragraph) could be associated to the i^{th} level surface $\{f \geq y_i\}$. Of course, in this way, the contractor of the lowest surface level would preempt most of the boxes, leading to a bad result (independence does not hold).

Introducing a priority between the different contractors (in addition to independence) does not require any extra concept. We just need to adapt the contractors to their actual semantics: a (sufficiently small) box $[x]$ must be classified with the contractor of the i^{th} surface level only if

$[x]$ is inside the i^{th} surface level
and $[x]$ does not intersect the $(i-1)^{th}$ surface level
  or $[x]$ has a small width (lower than ε)

The desired contractor C_i is then obtained by simply rewriting these conditions in terms of sub-contractors:

$$C_i := C_{f\notin[y_i,+\infty)} \cup \Big(C_{f\in[y_{i-1},+\infty)} \cap C_\varepsilon\Big)$$

or, using a *sub-distributivity* rule:

$$C_i := C_{f\notin[y_{i-1},+\infty]} \cap \Big(C_{f\in[y_i,+\infty)} \cup C_\varepsilon\Big).$$

Finally, we need as before an outer contractor for all the surface levels, namely, $C_{n+1} := C_{f\in(-\infty,y_n]}$. Moreover, the intersection of $C_1 \cap \ldots \cap C_{n+1}$ is $\{x \mid f(x) = y_n\}$ and, as usual, this set can be treated by a precision contractor C_ε. Figure 11 shows the desired output.

5.4 Set inclusion

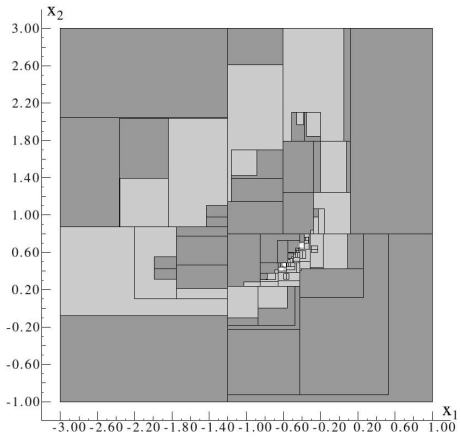

Fig. 12. The set inclusion problem solved by a more adapted algorithm. Boxes in dark gray does not belong to set(c_1). Boxes in light gray are inside set(c_2). We resort to splitting only in regions where boundaries of set(c_1) and set(c_2) are close.

The set inclusion problem can be solved efficiently in our formalism by using the following simple reasoning.

A box $[x]$ can be discarded from the search either if it does not belong to set(c_1) or if it is included in set(c_2). Indeed, in both cases, $[x]$ cannot compromise the assertion to be proven since $\forall x \in [x]$ we have $c_1(x) \Longrightarrow c_2(x)$.

Hence, an inner contractor C_1 for c_1 and an outer contractor C_2 for c_2 can be used jointly. Only the box that both (possibly) contain points inside set(c_1) and outside set(c_2) are bisected. The result is depicted in Figure 12.

No precision need to be introduced if $int(\text{set}(c_1)) \subseteq set(c_2)$ (where *int* stands for interior) since the algorithm will end in this case (we have $\text{set}(C_1) \cap \text{set}(C_2) = \emptyset$). But we still need a precision contractor C_ε in case of non inclusion.

5.5 Bounded-Error Parameter estimation

As we evoked above, modeling this problem causes no difficulty. The overall inner contractor is the union of the inner contractors related to each measure, say, $C_{g_i \notin [y_i]}$ with $g_i(p) := f(p, t_i)$ and $t_i \in [t_i]$. Here, the interval extension of g_i is a *thick* function since an interval constant $[t_i]$ substitutes for the variable t_i.

Likewise, the intersection of the outer contractors $C_{g_i \in [y_i]}$ defines an outer contractor for the whole problem.

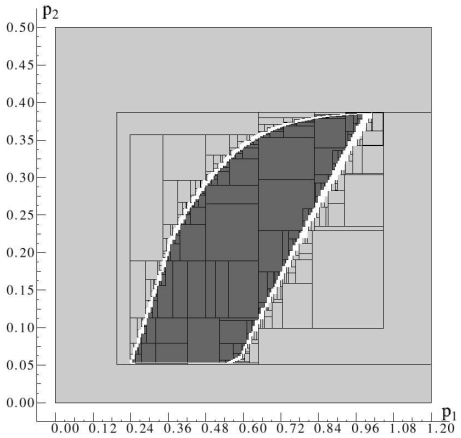

Fig. 13. Parameter estimation solved with inner and outer contractors based on proj-union and proj-intersection. The precision of the paver is still 0.01.

For efficiency, we have to split t_i inside the i^{th} outer and inner contractor. Consider first outer contraction: (p_1, p_2) is feasible if at least one value in $[t_i]$ satisfies $f(p, t_i) \in [y_i]$. Hence, if a point $(p, t_i) \in \mathbb{R}^3$ is insensitive to $C_{f \in [y_i]}$ then p must be insensitive to the outer contractor. The *proj-union* defined in §4 exactly fits this requirement. We define:

$$C_{outer} := C^{\cup [t_1]}_{f(p,t_1) \in [y_1]} \cap \ldots \cap C^{\cup [t_{10}]}_{f(p,t_{10}) \in [y_{10}]}$$

Similarly, the *proj-intersection* fits the requirement for the inner contraction. Indeed, a point $(p, t_i) \in \mathbb{R}^3$ sensitive to $C_{f \notin [y_i]}$ must be discarded by the inner contractor. We define:

$$C_{inner} := C^{\cap [t_1]}_{f(p,t_1) \notin [y_1]} \cup \ldots \cup C^{\cap [t_{10}]}_{f(p,t_{10})) \notin [y_{10}]}$$

A paving resulting from the combination of these contractors is shown in Figure 13.

6 The `Quimper` System

Besides theoretical investigations, contractor programming has given rise to a real system named `Quimper` (**QU**ick **I**nterval **M**odeling and **P**rogramming in a bounded-**ER**ror context). This system includes today three different programs: `qPave` (a graphical tool for paving sets in 2D), `qSolve` (a tool for listing numerical results, typically in high dimension) and `qTraj` (a graphical tool tailored to constraint problems derived from differential equations).

These are light-weight programs that only manage input/ouput, i.e., the language for writing contractors (the "Quimper language") and the interface for

configuring, running the paver and handling generated pavings. They are linked to a C++ library called `Ibex` (Interval-Based Explorer) that implements a contractor programming framework.

`Ibex` is based itself on the `Profil/Bias` library [28] for the low-level interval arithmetics. However, part of this library has been wrapped into functions that manage all the borderline cases (infinite bounds, values out of definition domains, empty sets, etc.) so that arithmetic operations are always exception-free. This explains why a modified release of `Profil/Bias` is included in the `Ibex` package.

All these software components are under GPL licence and can be downloaded online [7]. A user guide for `Quimper` (including the grammar of the `Quimper` language), as well as a complete documentation of `Ibex` classes are available on the same site. An archive containing all the examples of this paper in the `Quimper` syntax is also provided.

`Ibex/Quimper` has been developed by the first author but this software should be considered as a prototype since a real collaborative open-source project continuing this work is about to be launched. This new project will take advantage of the existing code.

The purpose of this section is to give some insight into the `Ibex` system. For convenience, we shall adopt some object-oriented coding notations. A contractor C is a class with the main function being `contract`, i.e., for a box $[x]$

$$C.\texttt{contract}([x])$$

contracts $[x]$ with C.

Generalizing *constraint* propagation to *contractor* propagation required letting the interface of contractors inherit from constraints. First, we can ask a contractor whether the domain of a given variable can impact the result of the contraction or not. This notion simply generalizes the incidence graph of a constraint network.

A contractor C, as a class, therefore implements a Boolean function `involves` that takes in argument the index j of a variable, i.e.,

```
C.involves(j)
```

returns `false` or `true`. Note that, by default, a contractor always returns `true`. In the parlance of object-oriented programming, this function needs not necessarily be *overridden*.

Second, we also added a few parameters to the `contract` function above, besides the box to be contracted, as explained in the next section.

6.1 Indicators

Every contractor has its own set of *specific* parameters. For instance, the precision contractor C_ε (`maxdiamGT` in the `Quimper` syntax) takes a parameter ε. The interval Newton also needs a parameter for controlling the termination of the iteration: when a step does not reduce any interval by more than a given ratio, the procedure stops. This ratio can be set externally. These parameters are related to the semantics specific to contractors (e.g., the precision contractor has by definition a threshold) or their implementations. They are usually set once for all (as *constructor* parameters). We shall not consider this type of parameters any further.

There is another type of parameters required by constraint programming algorithms for efficiency reasons, called *indicators*. The purpose of these parameters is to notify contractors during the search about the context in which they are called. For instance, we may inform a contractor that only a contraction on a given variable is actually expected (any other contraction being superfluous). If the time complexity of the contractor depends on the number of variables, some work is spared. We may also inform the contractor that only the domain of a given variable has been modified since the last call. Again, if the contractor works incrementally, this will speed up contraction.

Two indicators have been integrated into our system, corresponding precisely to the examples just given. The first is named *scope* and contains the subset of variables to be treated. The second is named *impacted* and contains the subset of variables whose domain has been impacted.

The semantics of indicators is constrained by one single fundamental property: they can be ignored by a called contractor without spoiling soundness (i.e., losing solutions), whence their name. If one develops a contractor that ignores the *scope* indicator, this contractor can still be passed as argument to an operator, should the latter be based on the communicability of this information (e.g., `propagate` below). The consequence is only a loss of efficiency.

Hence, only two indicators are proposed today but the nice point with indicators is that new ones can be invented at any time and integrated progressively in the subsequent implementations of contractors. In other words, backward compatibility is complete. We just had to put all the indicators in a dictionary structure. The real signature of the `contract` function is:

`contract(box` $[x]$`, dictionary` indicators).

As an example, one can ask a contractor C to focus on the two first variables only in the following way:

C.`contract`$([x], \{scope = \{0, 1\}\})$.

6.2 Numerical Contractors

There is nothing particular in the implementation of numerical contractors in `Quimper`, except that *indicators* have to be managed.

- The binary test $0 \in f([x])$ is actually not proposed in `Quimper`, but such a contractor would necessarily ignore all the indicators.
- `HC4Revise` is implemented. The complexity of this contractor is linear in the length of the constraint expression. Hence, contracting for a single variable amounts to contracting for all the variables (up to a small constant factor). In `Quimper`, `HC4Revise` ignores the *scope* indicator.
- The univariate interval Newton is implemented. As contrary to `HC4Revise`, this contractor deals with one particular variable at a time. Setting the *scope* indicator to a single variable divides the contraction time by the number of variables.
- Linear relaxations are not yet implemented. Now, since a linear programming solver is called iteratively to reduce the bounds of each initial variable, the *scope* indicator would have to be taken into account as with univariate Newton.

The multivariate interval Newton is also implemented and ignores the *scope* and *impacted* indicators.

6.3 Propagation

Here is how the propagation operator is implemented in `Quimper`.

Our convention for indices is to use i and j for contractors and variables respectively. When a couple (i, j) is *revised*, the i^{th} contractor has to work on the j^{th} variable. If the revision succeeds (a significant part of the i^{th} domain is removed) then the agenda is updated with the following classical procedure:

`update_agenda`(integer i, integer j, subset scope)
| `forall` $i' \neq i$ such that $C_{i'}$.`involves`(j)
| | `forall` j' such that $C_{i'}$.`involves`(j')
| | | `if` $j' \in$ scope$\backslash\{j\}$ `then` add (i', j') in the agenda

Now, if a revision (i, j) fails, there still may be a residual contraction. Furthermore, some subcontractors may not take into account the *scope* indicator so that unsolicited contractions can appear as the propagation loop goes along. At some point, the accumulation of these small contractions on a variable j can add up to a significant contraction. The agenda must be updated in consequence. In this case, the j^{th} variable is the source but there is not a particular contractor. The agenda must then be updated with all the couples (i, j), i describing the set of contractors. This is what the next procedure does.

```
update_agenda(integer j, subset scope)
 | forall i such that C_i.involves(j)
 | | forall j' such that C_i.involves(j')
 | | | if j' ∈ scope then add (i, j') in the agenda
```

Now, let us give the main procedure. Two local boxes are defined, the first is used to measure the result of each *revision* (the contraction of C_i over x_j) while the latter measure the contractions collected since the last time x_j triggered an update.

```
contract(box [x], subset scope, subset impacted)
 box [x^rev] ← [x]          // keeps track of domains before last revision
 box [x^upd] ← [x]          // keeps track of domains before last update
 forall j' ∈ impacted
 | update_agenda(j',scope)
 integer last_i ← −1         // keeps track of the last contractor called
 while the agenda is not empty
 | pop (i, j) from the agenda
 | | subset scope' ← {j};          // we want to revise the jth variable only
 | | subset impacted' ← {};        // no domain has been touched ...
 | | if (i ≠ last_i)               // ... unless we popped a different contractor
 | | | impacted' ← { all variables };    // so that all may have been impacted
 | | | forall j' such that C_i.involves(j')  // and all the constraint variables
 | | | | [x^rev]_j' ← [x]_j'                  // are now potentially impactable.
 | | | last_i ← i;                          // i becomes the last contractor called
 | | C_i.contract([x], scope', impacted')
 | | if [x^upd]_j\[x]_j is sufficiently small   // check accumulated contraction
 | | | if [x^rev]_j\[x]_j is sufficiently small // check last revision
 | | | | update_agenda(i,j,scope)        // fine propagation
 | | | else
 | | | | update_agenda(j,scope)          // coarse propagation
 | | | | [x]_j^upd ← [x]_j
```

6.4 Some performance results

All the examples in this paper have been executed almost instantaneously (less than 0.1 second) when the precision was not high (this corresponds to the figures with apparent boxes).

Figure 4.(a) was obtained by setting the precision of C_ϵ to $\epsilon := 0.01$. The paving contains 17768 boxes and was generated in $\sim 1.2s$.

Figure 5.(a) was obtained with two natural contractors corresponding to the constraints $c_1(x)$ and $(\bar{c_1}(x) \wedge c_2)$, with a precision set to $\epsilon := 0.005$. The paving contains 40844 boxes and was generated in $\sim 2.2s$.

Figure 6.(a) was obtained with $\epsilon := 0.001$ and by splitting 10 times the parameter domain inside the contractors $C^\cup$ and $C^\cap$. Computation time was 26 seconds

in this case, mainly because of the parameter splitting process which occurs systematically. The paving includes 18249 boxes.

7 Conclusion

We have presented a new framework for interval programming. The benefits of this framework are twofold. First, a large class of constraint-based problems can now be addressed with a unique simple algorithm, called *paver*. Second, a full control on the solving process is at hand, including the core of the propagation loop.

This framework is entirely built on the concept of *contractor*: inputs are contractors instead of constraints. Similarly, outputs are related to contractor sets instead of constraints sets. As just said, this –apparently small– change in modeling can lead to significant improvements in the design of new solvers but also in a declarative way. In a sense, the *imperative* aspect of solvers is now subsumed in the *declarative* one: the end user write constraints while the constraint programmer write contractors.

This approach is supported by a real system called `Quimper` which solves quickly all the different problems mentioned in this paper.

There are plenty of possible extensions for this work. The first one is perhaps to deal with global optimization. This extension would probably resort to *dynamic* contractors, i.e., contractors parameterized by a value that can be updated during the solving process. Orthogonally to *contractors* that potentially remove all unfeasible points, *local finders* that find peculiar feasible points seem to be a key feature as well.

References

1. H. Batnini and M. Rueher. Décomposition Sémantique pour la Résolution de Systèmes d'Equations de Distance. *JEDAI, Journal Electronique d'Intelligence Artificielle*, 2(1), 2004.
2. F. Benhamou and F. Goualard. Universally Quantified Interval Constraints. In *CP'00: 6th International Conference on Principles and Practice of Constraint Programming*, pages 67–82, 2000.
3. F. Benhamou, F. Goualard, L. Granvilliers, and J-F. Puget. Revising Hull and Box Consistency. In *ICLP*, pages 230–244, 1999.
4. F. Benhamou, D. McAllester, and P. Van Hentenryck. CLP(intervals) revisited. In *International Symposium on Logic programming*, pages 124–138. MIT Press, 1994.
5. C. Bessière and Debruyne R. Optimal and Suboptimal Singleton Arc Consistency Algorithms. In *IJCAI, 19th International Joint Conference on Artificial Intelligence*, pages 54–59, 2005.
6. C. Bliek, B. Neveu, and G. Trombettoni. Using Graph Decomposition for Solving Continuous CSPs. In *CP'98: 4th International Conference on Principles and Practice of Constraint Programming*, pages 102–116. Springer, 1998.

7. G. Chabert. *IBEX, an Interval-Based EXplorer*. http://www.ibex-lib.org.
8. H. Collavizza, F. Delobel, and M. Rueher. Extending Consistent Domains of Numeric CSP. In *IJCAI, Sixteenth International Joint Conference on Artificial Intelligence*, pages 406–413, 1999.
9. E. Davis. Constraint Propagation with Interval Labels. *Artificial Intelligence*, 32(3):281–331, 1987.
10. R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.
11. F. Delobel, H. Collavizza, and M. Rueher. Comparing Partial Consistencies. *Reliable Computing*, 5(3):213–228, 1999.
12. A. Goldsztejn. A branch and prune algorithm for the approximation of non-linear AE-solution sets. In *SAC'06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1650–1654, 2006.
13. A. Goldsztejn and L. Jaulin. Inner and Outer Approximations of Existentially Quantified Equality Constraints. In *CP'06: 12th International Conference on Principles and Practice of Constraint Programming*, pages 198–212. Springer, 2006.
14. C. Grandón, G. Chabert, and B. Neveu. Generalized Interval Projection: A New Technique for Consistent Domain Extension. In *IJCAI, 20th International Joint Conference on Artificial Intelligence*, pages 94–99, 2007.
15. L. Granvilliers and F. Benhamou. Algorithm 852: RealPaver: An Interval Solver using Constraint Satisfaction Techniques. *ACM Transactions on Mathematical Software*, 32(1), 2006.
16. E. Hansen. *Global Optimization using Interval Analysis (second edition)*. Dekker, 2003.
17. E.R. Hansen and S. Sengupta. Bounding Solutions of Systems of Equations Using Interval Analysis. *BIT Numerical Mathematics*, 21(2):203–211, 1980.
18. P. Herrero, M.A. Sainz, J. Vehí, and L. Jaulin. Quantified Set Inversion Algorithm with Applications to Control. *Reliable Computing*, 11(5):369–382, 2005.
19. E. Hyvönen. Constraint Reasoning Based on Interval Arithmetic: The Tolerance Propagation Approach. *Artificial Intelligence*, 58(1-3):71–112, 1992.
20. ILOG. *ILOG Solver*. http://www.ilog.com/products/cp/.
21. L. Jaulin. Localization of an Underwater Robot using Interval Constraint Propagation. In *CP'06: 12th International Conference on Principles and Practice of Constraint Programming*, 2006.
22. L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied Interval Analysis*. Springer, 2001.
23. L. Jaulin and E. Walter. Guaranteed Bounded-Error Parameter Estimation for Nonlinear Models with Uncertain Experimental Factors. *Automatica*, 35(5):849–856, 1999.
24. C. Jermann, G. Trombettoni, B. Neveu, and P. Mathis. Decomposition of Geometric Constraint Systems: a Survey. *IJCGA, International Journal of Computational Geometry and Applications*, 16(5–6):479–511, 2006.
25. N. Jussien, G. Rochart, and X. Lorca. The CHOCO Constraint Programming Solver. In *CPAIOR'08 workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, 2008.
26. R. B. Kearfott. *GlobSol*. http://interval.louisiana.edu/GlobSol.
27. R.B. Kearfott. *Rigorous Global Search: Continuous Problems*. Springer, 1996.
28. O. Knüppel. *Profil/Bias*. http://www.ti3.tu-harburg.de/Software/PROFILEnglisch.html.

29. M. Z. Lagerkvist and C. Schulte. Advisors for Incremental Propagation. In *CP'07: 13th International Conference on Principles and Practice of Constraint Programming*. Springer, 2007.
30. Y. Lebbah, C. Michel, and M. Rueher. Efficient Pruning Technique Based on Linear Relaxations. In *COCOS*, volume 3478 of *Lecture Notes in Computer Science*, pages 1–14, 2003.
31. O. Lhomme. Consistency Techniques for Numeric CSPs. In *IJCAI, 13th International Joint Conference on Artificial Intelligence*, pages 232–238, 1993.
32. J-P. Merlet. *Alias*. http://www-sop.inria.fr/coprin/logiciels/ALIAS.
33. J-P. Merlet. Solving the Forward Kinematics of a Gough-type Parallel Manipulator with Interval Analysis. *International Journal of Robotics Research*, 23(3):221–236, 2004.
34. R. Moore. *Interval Analysis*. Prentice-Hall, 1966.
35. A. Neumaier. *Interval Methods for Systems of Equations*. Cambridge University Press, 1990.
36. B. Neveu, G. Chabert, and G. Trombettoni. When Interval Analysis Helps Inter-Block Backtracking. In *CP'06: 12th International Conference on Principles and Practice of Constraint Programming*. Springer, 2006.
37. S. Ning and R.B. Kearfott. A Comparison of Some Methods for Solving Linear Interval Equations. *SIAM Journal of Numerical Analysis*, 34(1):1289–1305, 1997.
38. S. Ratschan. *RSolver*. http://rsolver.sourceforge.net.
39. S. Ratschan. Quantified constraints under perturbation. *Journal of Symbolic Computation*, 33(4), 2002.
40. F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., 2006.
41. N. Sahinidis. *BARON, Branch-And-Reduce Optimization Navigator*. http://www.andrew.cmu.edu/user/ns1b/baron/baron.html.
42. D. Sam-Haroud and B. Faltings. Consistency Techniques for Continuous Constraints. *Constraints*, 1:85–118, 1996.
43. C. Schulte, M. Lagerkvist, and G. Tack. *Gecode*. http://www.gecode.org/.
44. P. Van Hentenryck. *The OPL optimization programming language*. MIT Press, Cambridge, 1999.
45. P. Van Hentenryck, D. McAllester, and D. Kapur. Solving Polynomial Systems Using a Branch and Prune Approach. *SIAM Journal of Numerical Analysis*, 34(2):797–827, 1997.
46. P. Van Hentenryck and L. Michel. *Constraint-Based Local Search*. The MIT Press, 2005.
47. P. Van Hentenryck, L. Michel, and Y. Deville. *Numerica: A Modeling Language for Global Optimization*. MIT Press, Cambridge, 1997.
48. X-H. Vu. *Rigorous Solution Techniques for Numerical Constraint Satisfaction Problems*. PhD Thesis, Swiss Federal Institute Of Technology In Lausanne, 2005.