Combining finite and continuous solvers

Towards a simpler solver maintenance

Jean-Guillaume Fages and Gilles Chabert and Charles Prud'Homme

TASC - Ecole des Mines de Nantes,
LINA UMR CNRS 6241,
FR-44307 Nantes Cedex 3, France,
`{Jean-Guillaume.Fages,Gilles.Chabert,`
`Charles.Prudhomme}@mines-nantes.fr`

Abstract. Combining efficiency with reliability within CP systems is one of the main concerns of CP developers. This paper presents a simple and efficient way to connect `Choco` and `Ibex`, two CP solvers respectively specialised on finite and continuous domains. This enables to take advantage of the most recent advances of the continuous community within `Choco` while saving development and maintenance resources, hence ensuring a better software quality.

1 Introduction

The Constraint Programming (CP) community is witnessing the emergence of numerous new solvers, most of them coming up with new features. In this competitive context, integrating latest advances and ensuring software quality is challenging. From a more general point of view, spending effort on developing something already well handled by other libraries can be argued to be a waste of resource. `Choco` [9] and `Ibex` [2] are two such solvers, respectively specialised on Finite Domains (FD) and Continuous Domains (CD). While they already have some history, they have recently been completely re-engineered to brand new improved versions.

This paper presents a bridge which has been made so that `Choco` can use `Ibex` as a global constraint. The interval arithmetics provided by `Ibex` greatly enhances modeling possibilities of `Choco`. It enables to express naturally the wide family of *statistical constraints* [6, 7], but also non-linear physics constraints as well as many continuous objective functions. This bridge enables to take advantage of the most recent advances of the CD community within `Choco` for free. It saves development and maintenance resources, and contributes to the software quality. In this way, `Choco` and `Ibex` developers can focus on what they do best, being respectively FD and CD reasonings, and users have access to the whole.

2   Solver overviews

2.1   Choco 3.0

`Choco` is a java library for constraint satisfaction problems and constraint optimisation problems. This solver already has a long history and has been fully re-engineered this year, to a 3.0 version [4, 9]. It roughly contains $60,000$ lines of code.

The `Choco` library contains numerous variables, constraints and search procedures, to provide wide modeling perspectives. Most common variables are integer variables (including binary variables and views [8]) but the distribution also includes set variables, graph variables and real variables. The constraint library provided by `Choco` contains many global constraints, which gives a very expressive modeling language. The search process can also be greatly improved by various built-in search strategies (such as DomWDeg, ABS, IBS, first-fail, etc.) and some optimisation procedures (LNS, fast restart, etc.). Moreover, `Choco` natively supports explained constraints. Last, several useful extra features, such as a FlatZinc (the target language of MiniZinc [5]) parser and some viewing tools, are provided as well.

`Choco` is used by the academy for teaching and research and by the industry to solve real-world problems, such as program verification, data center management, timetabling, scheduling and routing.

2.2   Ibex 2.0

`Ibex` (Interval-Based EXplorer) is also a library for constraint satisfaction and optimization, but written in C++ and dedicated to continuous domains. This solver has been fully re-engineered to a 2.0 version this year [2]. `Ibex` consists of roughly $40,000$ lines of code.

From the perspective of solver cooperation, two features of `Ibex` are of interest: the modeling language and the *contractors*.

Compared to `Choco`, the modeling language is much simpler in the sense that constraints are either numerical equations or inequalities. However, the mathematical expression involved in a constraint can be of arbitrary complexity. The expression is obtained by composition of standard mathematical operators such as $+$, $\times$, $\sqrt{}$, $\sin$, etc. (see §3.2). The modeling language also allows vector and matrix operations; it shares some similarities with Matlab on purpose.

A contractor [3] is the equivalent of a propagator in finite domain except that it is considered as a pure function: it takes a Cartesian product of domains as input and returns a subset of it. `Ibex` contains a variety of built-in contractors for acheiving different level of bound consistency with respect to a set of numerical constraints such as HC4, Shaving, ACID, X-newton, q-intersection, etc.

Finally, `Ibex` also comes with a default black-box solver and global optimizer for immediate usage. It is mainly used so far in academic labs for teaching and research. Its main application field is global optimization and robotics.

3 Embedding Ibex into a Choco constraint

3.1 Motivation

It is worth noticing that combining FD with CD in a CP solver is not new. Since its early beginning, the `Choco` solver has supported real variables, hence it has always been able to solve hybrid discrete continuous problems. However, these older versions included their own interval arithmetics implementation. Another example is the `Gecode` 4.0.0 solver [1], which has recently added floating variables to its distribution, by following the same approach.

Interestingly, it appeared that most of `Choco` users and contributors were concerned by FD problems. Thus, for historical reasons, the `Choco` module over reals has not evolved much within the last years. In the meanwhile, people working on continuous problems have proposed new solvers, such as `Ibex`, able to handle efficiently continuous non-linear equation systems. As a counterpart, such solvers are not competitive on problems involving finite domains, if ever they can handle them.

If no theoretical pitfall stands in the way of implementing state-of-the-art CD techniques in `Choco`, this would require significant resources and ensuring its maintenance over time is presumably even more expensive. Moreover, it would require `Choco` developers to have a high level of expertise on both FD and CD. A symmetric reasoning holds if one would like to implement advanced FD features within `Ibex`. Thus, instead of reimplementing the wheel, it has been decided to make a bridge between `Choco` and `Ibex`. This provides a very good trade-off between solver features and implementation effort.

The choice of using `Ibex` within `Choco`, instead of the opposite, is based on practical reasons. First of all, the functional architecture of contractors in `Ibex` enables to call them from another program very easily. Second, `Choco` has more variable types, hence using an opposite design would require a heavier interface. In particular, `Ibex` would have to implement finite domains. Third, `Choco` offers richer resolution options (black-box search procedures, LNS, explanations...) than `Ibex` so it is better to give the control of the search to `Choco`. Last, calling Java from C++ is more cumbersome since a virtual machine has to be loaded prior to function calls.

3.2 A simple but yet expressive interface

The bridge linking `Choco` and `Ibex` is organised in a master-slave architecture where `Choco` integrates `Ibex` within a global constraint. This constraint, referred to as `RealConstraint`, has no particular semantics but is used as a shell to encapsulate continuous propagators. Each equation system of the model is associated with one generic propagator, `RealPropagator`, in `Choco` and one contractor in `Ibex`. Continuous expressions can embed integer variables by using views. `Choco` drives the propagation algorithm: on domain modifications, targeted propagators are scheduled for a future execution. Any call to the propagation algorithm of a `RealPropagator` is then automatically delegated to

`Ibex` contractors; the resulting domain modifications, if any, are recovered and transmitted back to `Choco`. `Ibex` contractors are called through the Java Native Interface (JNI) which enables a Java program to call functions of a C++ library. Comments apart, this native interface only includes 40 lines of code, whence the easy maintenance. An overview of the `Choco-Ibex` framework is given in Figure 1.
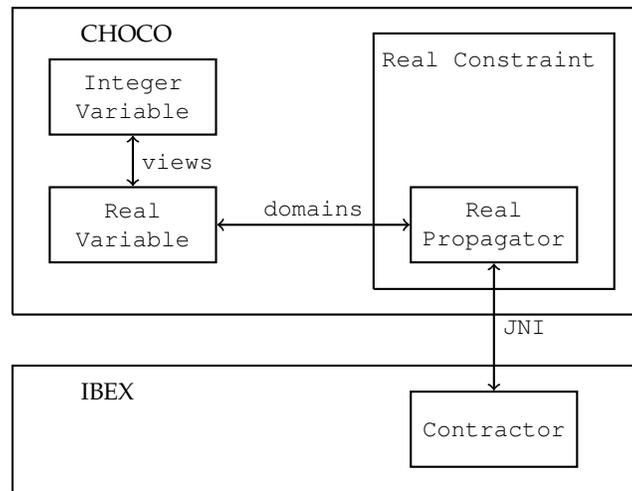

Fig. 1. Scheme of the `Choco-Ibex` bridge.

Listing 1.1 provides the filtering algorithm of `RealPropagator`. First, the propagator copies variable domain bounds in an array (l. $5 - 10$). Second, it calls the `contract` method of the `Ibex` JNI class (see Listing 1.2), with this array and the contractor identifier as input (l. $11 - 12$). This method updates the array of bounds in argument (for a further filtering) and returns an entailment statement. Third, it incorporates these changes into variable domains and, possibly, fails or becomes silent (l. $13 - 28$). As any constraint of `Choco`, a `RealConstraint` can be reified.

Regarding the management of object creations and Java/C++ communication, this architecture does not bring any significant overhead. When the first `RealConstraint` is created, the `Ibex` library is loaded once and for all by the system. Each `Ibex` contractor is created once during the model creation, and its reference is kept in memory. Calling an `Ibex` contractor from a `Choco` propagator has no particular overhead but the translation of the Java primitive double array which represents variable bounds to a native double array. This takes a linear time over the number of variables that are involved, which is presumably less or equal to the contractor time complexity.

Listing 1.1.

The expression of the continuous constraint (equation or inequality) is encoded in a simple `String`. To simplify the interpretation of this `String` by `Ibex`, variables are represented by their indices, surrounded by braces. For instance, the constraint `"({0}+{1}+{2})/3={3}"` means that the fourth variable is the average of the three first ones.

This framework handles any equation system involving the following operators:

```
+, -, *, /, =, <, >, <=, >=,
sign, min, max, abs, sqr, sqrt, exp, log, pow,
cos, sin, tan, acos, asin, atan,
cosh, sinh, tanh, acosh, asinh, atanh, atan2
```

This provides wide modeling perspectives. In particular, the family of *statistical* constraints, such as `Spread` [6] and `Deviation` [7], can be expressed naturally and extended by using neither monolithic ad hoc algorithms nor reformulations. Of course, in continuous domains, equations and inequalities are ubiquitous.

Besides being both simple and expressive, the use of `Strings` enables to get very concise models. As a counterpart, it has no safeguard against user mistakes in the declaration of continuous constraints. Hence building a framework which generates those `Strings` may be a good perspective to make the use of this bridge safer.

4   Practical example: using CD to express balancing

This section introduces the Santa Claus problem as a simple illustration of this framework. Given a set of kids and a set of gifts, the Santa Claus problem consists of giving a gift to each child. The average deviation of gift values must be minimised so that the gift distribution is fair.

The `Choco` model associated with this problem is given in Listing 1.3. It involves integer assignment decision variables as well as real variables related to the average and the average deviation of gift prices. In particular, the objective variable is real, hence the hybrid nature of the problem. On the one hand, the `AllDifferent` constraint is typically not implemented in `Ibex`, as differences have no much meaning over reals. On the other hand, the average and the average deviation constraints are straightforward to formulate as general `Ibex` arithmetic expressions. Thus, we take the best from each solver. The possibility to have real views of integer variables enables to consider integer variables within continuous systems. Hence, even on purely integer problems, this framework makes available a wide family of constraints, for free.

Listing 1.3. Santa Claus `Choco` model

```java
// input data
int n_kids = 3;
int n_gifts = 5;
int[] gift_price = new int[]{11, 24, 5, 23, 17};
int min_price = 5;
int max_price = 24;

// solver
Solver solver = new Solver("Santa Claus");

// FD variables
// VF is the factory for variables' declaration
IntVar[] kid_gift = VF.enumeratedArray("g2k", n_kids, 0, n_gifts, solver);
IntVar[] kid_price = VF.boundedArray("p2k", n_kids, min_price, max_price, solver);
IntVar total_cost = VF.bounded("total cost", min_price*n_kids, max_price * n_kids, solver);

// CD variable
double precision = 1.e-4;
RealVar average = VF.real("average", min_price, max_price, precision, solver);
RealVar average_deviation = VF.real("average_deviation", 0, max_price, precision, solver);

// continuous views of FD variables
RealVar[] realViews = VF.real(kid_price, precision);

// kids must have different gifts
// ICF is the factory for integer constraints' declaration
solver.post(ICF.alldifferent(kid_gift, "AC"));

// compute cost
for (int i = 0; i < n_kids; i++) {
    solver.post(ICF.element(kid_price[i], gift_price, kid_gift[i]));
}
solver.post(ICF.sum(kid_price, total_cost));

// compute the average and average deviation costs
RealVar[] allRV = ArrayUtils.append(realViews, new RealVar[]{average, average_deviation});
RealConstraint ave_cons = new RealConstraint(solver);
ave_cons.addFunction("({0}+{1}+{2})/3={3}", allRV);
ave_cons.addFunction("(abs({0}-{3})+abs({1}-{3})+abs({2}-{3}))/3={4}", allRV);
solver.post(ave_cons);

// set search strategy (selects smallest domains first)
solver.set(IntStrategyFactory.firstFail_InDomainMin(kid_gift));

// find optimal solution (the gift distribution should be fair)
solver.findOptimalSolution(ResolutionPolicy.MINIMIZE, average_deviation);
```

The output stream (Listing 1.4) then provides the following solution:

Listing 1.4. Output

```
******** Optimal solution
Kids #0 has received the gift #4 at a cost of 17 euros
Kids #1 has received the gift #3 at a cost of 23 euros
Kids #2 has received the gift #1 at a cost of 24 euros
Total cost: 64 euros
Average: 21.333333333333332 euros per kid
Average deviation: 2.8888888888888866
```

5 Conclusion

We have proposed a bridge between `Choco` and `Ibex` so that `Choco` can use `Ibex` as a global constraint. We have shown that this framework offers wide modeling possibilities while being simple and generic. This work enables the FD and the CD communities to benefit from the work of each other and focus and their respective field of expertise. It enables to provide a rich and reliable solver while saving development and maintenance resources.

References

1. Gecode 4.0.0. http://www.gecode.org/index.html, 2013.
2. Gilles Chabert. Ibex 2.0. http://www.emn.fr/z-info/ibex/, 2013.
3. Gilles Chabert and Luc Jaulin. Contractor programming. *Artif. Intell.*, 173(11):1079–1100, 2009.
4. Jean-Guillaume Fages, Narendra Jussien, Xavier Lorca, and Charles Prud'homme. Choco3: an open source java constraint programming library. Technical report, 2013. to appear.
5. Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *CP*, pages 529–543, 2007.
6. Gilles Pesant and Jean-Charles Régin. Spread: A balancing constraint based on statistics. In *CP*, volume 3709 of *LNCS*, pages 460–474. Springer, 2005.
7. Pierre Schaus, Yves Deville, Pierre Dupont, and Jean-Charles Régin. The deviation constraint. In *CPAIOR*, volume 4510 of *LNCS*. Springer, 2007.
8. Christian Schulte andirL 11.41(irL 1811i1si59 Td-2811iR]TJ/F5w.-227(the)-227(aR]T)]TJ -R]Twork)-3J -R]T22(on)-223(s