IMPLEMENTING GLOBAL CONSTRAINTS AS STRUCTURED NETWORKS OF ELEMENTARY CONSTRAINTS

DÁVID HANÁK

Department of Computer Science and Information Theory
Email: dhanak@cs.bme.hu

Abstract

Global constraints are cardinal concepts of CLP($\mathcal{FD}$), a constraint programming language. They are means to find a set of integers that satisfy certain relations. The fact that defining global constraints often requires the knowledge of a specification language makes sharing constraints between scientists and programmers difficult. Nicolas Beldiceanu presented a theory that could solve this problem, because it depicts global constraints as graphs: an abstraction that everyone understands.
The abstract description language defined by the theory may also be interpreted by a computer program. This paper deals with the problematic issues of putting the theory into practice by implementing such a program. It introduces a concrete syntax of the language and presents three programs understanding that syntax. These case studies represent two different approaches of propagation. One of these offers exhausting pruning with poor efficiency, the other, yet unfinished attempt provides a better alternative at the cost of being a lot more complicated.

I. Introduction

Constraint Logic Programming (CLP, also referred to as Constraint Programming, CP) [1] is a family of logic programming languages, where a problem is defined in terms of correlations between unknown values, and a solution is a set of values which satisfy the correlations. In other words, the correlations *constrain* the set of acceptable values, hence the name. A member of this family is CLP($\mathcal{FD}$), a constraint language which operates on variables of integer values. Like CLP solvers in general, CLP($\mathcal{FD}$) solvers are embedded either into standalone platforms such as the ILOG OPL Studio [2] or host languages, such as C [3], Java [4], Oz [5] or Prolog [6, 7].

CLP($\mathcal{FD}$) systems are particularly useful to model discrete optimization problems. The wide range of industrial applications includes various scheduling and resource allocation tasks, analysis and synthesis of analog circuits, cutting stock problems, graph coloring, etc.

In CLP($\mathcal{FD}$), FD stands for *finite domain,* because each variable is represented by the finite set of integer values which it can take. These variables are connected by the constraints, which *propagate* the change of the domain of one variable to the domains of others. A constraint can be thought of like a "daemon" which wakes up when the domain of one (or more) of its variables has changed, propagates the change and then falls asleep again. This change can be induced either by an other constraint or by the *distribution* or *labeling* process, which enumerates the solutions by successively substituting every possible value into the variables. Constraints can be divided into two groups: simple and global constraints. The former always operate on a fixed number of variables (like $X = Y$), while the latter are more generic and can work with a variable number of arguments (e.g., "$X_1, X_2, \ldots, X_n$ are all different").

Many solvers allow the users to implement user-defined constraints. However, the specification languages vary. In some cases, a specific syntax is defined for this purpose, in others, the host language is used. There are several problems with this. First, CLP($\mathcal{FD}$) programmers using different systems could have serious difficulties sharing such constraints because of the lack of a common description

language. Second, to define constraints, one usually has to know the solver in greater detail than if merely using the predefined ones. Inspired by these problems, Nicolas Beldiceanu suggested a new method for defining and describing global finite domain constraints [8]. After studying his theory, we decided to put it into practice by extending the CLP($\mathcal{FD}$) library of SICStus Prolog [9], a full implementation of the CLP($\mathcal{FD}$) language, with a parser of Beldiceanu's abstract description language. This has been the only attempt in this direction so far.

II. The Theory

In [8], Beldiceanu specifies a description language which enables mathematicians, computer scientists and programmers of different CLP systems to share information on global constraints in a way that all of them understand. It also helps to classify global constraints, and as a most important feature, it enables us to write programs which, given only this abstract description, can automatically generate parsers, type checkers and propagators (pruners) for specific global constraints.

In order to create an inter-paradigm platform, Beldiceanu reached for a device that is abstract enough and capable of depicting relations between members of a set: the *directed graph*. According to the theory, the most important ingredients of a global constraint specification are the following: a regularly structured *initial graph*, an *elementary constraint* (a very simple constraint, such as $X = Y$), and a set of *graph properties* (restrictions on the number of arcs, sources, connected components, etc.).

For each constraint it is specified how the *initial graph* should be built. Its vertices are assigned one or more variables from the constraint, while the arcs connecting the vertices are generated according to a regular pattern. Finally the chosen elementary constraint is assigned to each arc, the variables belonging to the endpoints of the arc being used as the arguments of the elementary constraint. Note that the elementary constraint need not be binary, if it has more arguments, then a hypergraph is built using arcs with the required number of endpoints.

The *final graph* consists of those arcs of the initial graph for which the elementary constraints hold. The global constraint itself succeeds if and only if the specified graph properties hold for this final graph.

The description language also contains terms to express certain preconditions on the arguments of the constraint, i.e., restrictions that must hold in order to be able to apply the constraint on its arguments.

This schema allows us to test whether the relation expressed by a global constraint holds for a given set of concrete arguments. However, it does not deal with the more important case where only the domains of the arguments are specified, but their specific values are unknown. In such a case we need an algorithm to prune the domains of the arguments by deleting those values that would certainly result in a final graph not satisfying the properties. This question is fundamental in practical applications, therefore it is addressed by the second part of the paper, which introduces an implementation for the SICStus Prolog environment.

III. The Implementation

The first step of putting the theory into practice was to define a concrete syntax that corresponds to the abstract syntax and is suitable for parsing. The representation chosen closely resembles the abstract syntax, but follows the syntax of Prolog, too. This has the advantage that it can be effortlessly parsed by a Prolog program. This task also helped to discover that the semantics of an operator expressing a precondition on a set argument of the constraint is unclear in certain contexts, because it is underspecified.

Development was launched with two goals in mind. The first task was to implement a relation checker, a realization of the testing feature offered by the schema, and a dumb propagator built on this checker. By and large, this task is finished. The second task was to implement an active propagator

capable of pruning variable domains based on an analysis of the current state of the graph, with the required properties in view. This task is much bigger, the development is still in an early stage. Both are implemented in SICStus Prolog [10], extending its CLP($\mathcal{FD}$) library by utilizing the interface for defining global constraints. This allows thorough testing of both the program and the theory itself in a trusted environment.

Task 1. The complex relation checker and the generate-and-test propagator. The first stage was to implement the complex relation checker, a program that checks whether the relation defined by the global constraint holds for a given set of values, but does no pruning at all. When called, the relation checker is given a constraint with fully specified arguments (i.e., no variables should appear in them), and reports the result of the restriction checks and whether the graph properties hold for the final graph. It was used to test the formal description of several constraints, whether they really conform to their expected meaning, and some errors in their specification have already been discovered.

The second (and final) stage was to amend the relation checker with a generate-and-test propagator. The idea is that whenever the domain of a variable changes, all possible value combinations of the affected constraint's arguments are tested against the relation checker, and only the values that passed the test are preserved. This classical but extremely inefficient method for finding solutions gives us full and exhaustive pruning.

Task 2. The active propagator. Generate-and-test propagation is naturally out of the question in any practical applications. The active propagator is the first step towards an efficient, applicable pruner. Here the line of thought is reversed: we assume that the constraint holds (or at least the intention of the programmer is that it should hold), and from the required graph properties we try to deduce conclusions on the domains of its variables.

More specifically, when a constraint wakes up, some of the elementary constraints assigned to the arcs of the graph are known to hold (i.e., unless the program backtracks, they will also appear in the final graph), and some are known to fail (i.e., they will be left out of the final graph, too). The state of the rest is yet uncertain. This classification can be completed gradually because the CLP($\mathcal{FD}$) system is monotonic, which means that a value is not removed from a domain unless it is definitely not a solution. Then, by knowing the required graph properties and the graph's current state, the graph can be tightened by forcing some of these uncertain constraints into success or into failure, thus causing the affected variables to be narrowed or instantiated. The global constraint finally becomes entailed when there are no uncertain arcs left.

We chose to represent elementary constraints with simple, *reifiable* CLP($\mathcal{FD}$) constraints. (Reifiable constraints are connected with a Boole variable, and succeed *if and only if* the Boole variable has a value of 1.) This has several advantages. For one, a wide range of predefined constraints is available, already at this early stage of development. For another, the algorithm must be able to determine whether an elementary constraint holds or fails, or force it into success or failure, and the Boole variable linked to the reifiable constraints serves exactly that purpose.

To figure out how to tighten the graph at each step, in other words, to find the rules of pruning, we need to study each graph property separately. There are simpler properties, such as prescribing the number of arcs, for which finding these rules is not very problematic. A few of these are already handled by the propagator. The are more complex properties, like constraining the difference in the vertex number of the biggest and smallest strongly connected components, the pruning rules for these are a lot more complicated.

The current implementation can handle four different graph properties: restrictions on the number of arcs, vertices, sources and sinks. Fortunately, a large number of descriptions relies only on the first two, thus many different constraints can already be propagated. Without going into details, such constraints

are `among`, `disjoint`, `common`, `sliding_sum`, `change`, `smooth`, `inverse`, and variants of these. Current work is concentrated on the perfection of the propagation of these properties, and on the study of the more complicated one that prescribes the number of strongly connected components, upon which many of the existing descriptions are based.

IV. Future Work

We have to work out pruning rules for more of the graph properties. This will be the objective of an international project hopefully starting in Autumn 2003.

Using reifiable constraints as elementary constraints poses a problem: they do not necessarily provide a pruning as strong as expectable. This is so because they naturally do not take other elementary constraints into account, although a more global view would be more effective. This problem requires further study.

Efficiency matters need to be considered more carefully when implementing further propagators. One way to increase efficiency could be to abandon the thought of a common propagator, that is able to parse such descriptions and prune in run time, and implement a *pruner algorithm generator* instead. This generator would take the description and convert it into a piece of code that does the pruning. This would shift the execution of complicated graph algorithms into compile time, where efficiency is a smaller issue. How this can be done must be worked out yet.

Acknowledgements

I would like to thank Nicolas Beldiceanu for his theory and his ideas on the propagation of graph properties. Péter Szeredi, my supervisor always directed my attempts at research and writing with patience yet with great tenacity.

References

[1] J. Jaffar and S. Michaylov, "Methodology and implementation of a CLP system," in *Logic Programming: Proceedings of the 4th International Conference*, J.-L. Lassez, Ed., pp. 196–218, Melbourne, May 1987. MIT Press, Revised version of Monash University technical report number 86/75, November 1986.

[2] P. van Hentenryck, L. Michela, L. Perron, and J.-C. Régin, "Constraint programming in OPL," in *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, G. Nadathur, Ed., vol. 1702 of *Lecture Notes in Computer Science*, pp. 98–116, September 29 - October 1 1999.

[3] ILOG, *ILOG Solver 5.1 User's Manual*, ILOG s.a. `http://www.ilog.com`, 2001.

[4] V. Loia and M. Quaggetto, "Embed finite domain constraint programming into Java and some Web-based applications," *Software— Practice and Experience*, 29(4):311–339, Apr. 1999.

[5] G. Smolka, "Constraints in OZ," *ACM Computing Surveys*, 28(4es):75, Dec. 1996.

[6] P. van Hentenryck, *Constraint Satisfaction in Logic Programming*, MIT Press, Cambridge, Massachusetts, 1989.

[7] D. Diaz and P. Codognet, "A minimal extension of the WAM for clp(FD)," in *Proceedings of the Tenth International Conference on Logic Programming*, D. S. Warren, Ed., pp. 774–790, Budapest, Hungary, 1993. The MIT Press.

[8] N. Beldiceanu, "Global constraints as graph properties on a structured network of elementary constraints of the same type," in *Principles and Practice of Constraint Programming*, pp. 52–66, 2000.

[9] M. Carlsson, G. Ottosson, and B. Carlson, "An open-ended finite domain constraint solver," in *Proceedings of the Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP'97)*, vol. 1292. Springer-Verlag, Berlin, 1997.

[10] Swedish Institute of Computer Science, Uppsala, Sweden, *SICStus Prolog User's Manual*, 2003, http://www.sics.se/isl/sicstuswww/site/documentation.html.