Constraints (2014) 19:139–149
DOI 10.1007/s10601-013-9152-4

Toward sustainable development in constraint programming

Nicolas Beldiceanu · Pierre Flener · Jean-Noël Monette · Justin Pearson · Helmut Simonis

Published online: 20 October 2013

Abstract We present a few challenges that we consider important to tackle for the future of constraint programming. The focus is put on simplifying the design and implementation of propagators in solvers.

Keywords Global constraints · Combinatorial structure · Sustainable solver development

1 Introduction

One of the key issues for the design of constraint programming (CP) solvers with global constraints is the dichotomy between on the one hand the wish to have a few general-purpose constraints, and on the other hand the wish to adapt constraints for specific situations in order to achieve concise models and the maximum amount of propagation. There is a clear need to limit the amount of engineering required to

N. Beldiceanu
TASC team (CNRS/INRIA), Mines de Nantes, 44307 Nantes, France
e-mail: Nicolas.Beldiceanu@mines-nantes.fr

P. Flener (✉) · J.-N. Monette · J. Pearson
Department of Information Technology, Uppsala University,
Box 337, 751 05 Uppsala, Sweden
e-mail: Pierre.Flener@it.uu.se

J.-N. Monette
e-mail: Jean-Noel.Monette@it.uu.se

J. Pearson
e-mail: Justin.Pearson@it.uu.se

H. Simonis
4C, University College Cork, Cork, Ireland
e-mail: H.Simonis@4c.ucc.ie

implement a standard set of constraints, and also to limit the number of concepts that have to be learned by the users of such solvers. At the same time, it should be easy to define and implement new constraints for a specific problem, without excessive and repetitive development effort.

Developing CP solvers takes a lot of resources but cannot be avoided since only solvers allow one to put CP into practice. Many pioneering CP solvers, such as Alice [17] or Prolog IV [10], were done by outstanding academicians. But these solvers have disappeared and it took a number of years to reconstruct parts thereof in newer solvers. Some of their features can still not be found in most modern CP solvers, such as computing the lower bound of a sum in the presence of an ALLDIFFERENT constraint [3, 17]. Since not much was published about their implementation, a lot of effort is lost.

A possible origin of the problem is that some people assume that since they could come up with a new algorithm, method, or solver, they will have a competitive advantage over others. In order to preserve this advantage, not all details are revealed, also because standard publications are typically not the appropriate place to do so. If we continue so, then we end up with many groups repeating again and again work already done, while having to guess some obscure parts that were not explained in the original papers.

In order for the CP community to continue to flourish, we consider a number of proposals towards the sustainable development of CP solvers and their associated algorithms: the focus is put on simplifying the design and implementation of propagators in solvers by promoting reuse. However, a key challenge that should be considered by CP solver developers is:

Challenge 1 *How can CP solver developers simplify the migration of the knowledge embedded in their solver to next-generation solvers?*

This issue inevitably pops up for every solver. In fact, it is the part of a solver that one can migrate to a new solver that will be valuable, since the other part will just be lost at some point. Good practice thus includes:

- Promote source code to be associated with submitted and published papers on algorithms; see for instance [27, p. 36].
- Promote open-source solvers, such as Choco, ECLiPSe, Gecode, JaCoP, and Minion.
- Promote algorithms that can be implemented in more than one solver.
- Promote approaches that see an algorithm as the interpretation of a formula or automaton, since formulae and automata can migrate between solvers. In addition, unlike monolithic algorithms, formulae and automata can be verified, composed, and used for different purposes.
- Promote standards for encoding knowledge that can be shared by different solvers (ideally a solver should be limited to an abstract machine that can interpret various types of combinatorial knowledge).
- Promote the creation of nested concepts of increasing abstraction [8] for handling larger classes of structured (conjunctions of) constraints.

We refine some of these ideas in Section 2. Rather than providing general challenges, we focus on a small set of concrete questions that we think are worth investigating.

This does not mean that other issues, such as integrating continuous and discrete constraints, are not important, but we prefer to state focussed challenges.

As a complement, in Section 3, we discuss the way CP interacts with other computer science areas. We list some recurring difficulties that CP is facing, and we indicate possible ways to deal with them.

2 Global constraints and propagators

The main idea is to use more compact ways to describe and implement constraints, by searching for concepts (and their combinations) that occur inside the propagators of many constraints. First, we focus on generic ways to represent constraints and propagators, namely automata (Section 2.1) and indexicals or invariants (Section 2.2). Next, we tackle various challenges recurring in the development of propagators. Section 2.3 deals with the verification and synthesis of propagators. Section 2.4 addresses the topics of visualisation and explanations. The issue of scalability is treated in Section 2.5. We close the section by discussing in Section 2.6 how to reconstruct the Global Constraint Catalogue [5].

2.1 Automata and learning

Automata and multi-valued decision diagrams (MDDs) have become increasingly popular in CP in order to encode some classes of constraints without needing to provide ad-hoc propagators. Automaton constraints (such as REGULAR [25] or AUTOMATON [2]) are available in almost all constraint solvers. However, from a global constraint perspective, automaton constraints have led to the following paradox: If the actual automaton is specific to the instance of the problem, then one needs an ad-hoc procedural algorithm in order to generate the actual automaton for the constraint one wants to post (for example, see the counter-free automata of the PATTERN, INCREASING_GLOBAL_CARDINALITY, and SMOOTH constraints of [5]). This prevents the portability of constraint models between solvers (even if automaton constraints are available in most solvers), as one needs to generate a new automaton for each instance of the problem. However, many automata that are used in practice (say in timetabling) have a very regular structure, which means that, even if they have a huge (say cubic) number of states or transitions, their generator algorithms can be described in very concise ways (for example, see the automata of the INCREASING_NVALUE constraint of [5, pp. 1145–1146]). This leads to the following challenge:

Challenge 2 *Is there a* declarative *way for describing concisely a generator of automata (or MDDs) that have a very regular structure?*

Since unfolding an automaton for a given sequence length can take a lot of memory, we add:

Challenge 3 *Is it possible to perform propagation directly on a description of a declarative generator rather than on the automaton (or MDD)?*

A subarea of machine learning deals with grammatical inference [15], such as learning automata from positive and negative examples [23]. One goal is to learn the smallest automaton (in terms of the number of states and transitions) that covers all the positive examples but none of the negative examples. But, again, this may be counterproductive for the automaton attached to a global constraint, as it may have a huge number of states or transitions but a very concise generator.

Challenge 4 *Are there learning algorithms that learn smallest automaton generators rather than smallest automata?*

A first step in this direction would be defining a language that allows the concise description of automaton generators. One such language could extend automata by replacing states by classes of states. Those classes would be indexed by several indices varying over finite integer sets with arithmetic constraints linking the indices. Transitions would be defined between classes of states, or between subclasses of states defined by additional constraints on the indices.

2.2 Beyond classical indexicals

At the time of their invention in the early 1990s, indexicals [34] have raised great hope about the possibility of avoiding ad-hoc propagators. Indexicals are formulae of the form $x \texttt{ in } \sigma$, meaning that the domain of the variable x must be a subset of the set σ, which can depend on the current domain of other variables. Indexicals allow the encoding of constraint propagation by numerical expressions defining the set of possible values of a variable. However, it turned out that indexicals could not replace or encode the propagators associated to global constraints. After initial work on the efficient compilation of indexicals (for instance [13]), very little work was done in the spirit of indexicals, exceptions being [9, 22]. On the one hand, classical indexicals have a number of known limitations:

- Classical indexicals are low-level in the sense that one has to write a formula for each variable to prune.
- Classical indexicals only address a limited class of numerical constraints[1] but not combinatorial constraints.

On the other hand, indexicals are attractive from the following points of view:

- Indexicals allow one to design rapidly variants of constraints that one encounters in practice.
- Indexicals separate the logical part expressing the pruning from its implementation.
- Indexicals encoding propagation can be reused by all the solvers that handle indexicals.

[1] Apparently, even solvers that handle indexicals do not use them for encoding the propagation of constraints such as DIV and MOD, which are used in the *MiniZinc Challenge*.

Hence the following generic challenge:

Challenge 5 *Can we extend indexicals that manipulate sets of numerical values to indexicals that deal with other combinatorial objects?*

The key strength of propagators is that they take advantage of the problem structure. Our idea is to provide this structure as *formulae involving combinatorial objects that can be interpreted by a dedicated abstract machine.* Note that this quest not only benefits CP but also local search, MIP (mixed integer programming), SAT (Boolean satisfiability), and SMT (SAT modulo theories).

Hereafter we consider two specific cases where the structure could be made apparent. As a first case, many propagators follow this pattern: first, the lower bound of some formula is computed; then the propagator exploits this bound in order to perform pruning by evaluating the regret of each variable-value pair $\langle x, d \rangle$. The lower bound can usually be stated as a formula but the tedious part is the computation of the regret. Unfortunately the information of how to compute the regret is often hidden in the proof of the bound used in the first step, but not available in a machine-usable way. Hence the following challenge:

Challenge 6 *Is it possible to characterise and describe with an appropriate dedicated language an assignment associated with effectively reaching the bound?*

The hope is that if one can explicitly describe such an extremal object, then the regret and the corresponding propagator can be obtained mechanically.

We now turn our attention to the case when the combinatorial structure is a graph. The propagators of many constraints, such as ALLDIFFERENT, GCC, or NVALUE, can be understood as the application of a few graph algorithms on a graph that is derived from the arguments of the constraint. This is also true for propagators attached to graph properties, as shown in [6], where a preliminary effort was done to see a propagator as the interpretation of a graph formula.

While the derived graphs on which we apply standard graph algorithms are often simple (for instance, variable-value bipartite graphs), the problem is that this is done in an ad-hoc way for each constraint. The arising challenge is thus:

Challenge 7 *Is there a* concise *and* compositional *way of denoting how to derive graphs from the arguments of a constraint?*

This would be the basis for a graph-based indexical language where one can directly and fully encode graph formulae representing the desired propagator. Once such a language is clearly defined, one can define dedicated abstract machines to handle or compile such formulae efficiently. Key points in this process are the ability to define compositional graph views (to avoid representing graphs explicitly) and the automatic selection of the most efficient algorithms for the considered graph class.

In addition, *graph invariants* (formulae relating different graph characteristics, such as number of nodes or connected components) can be used to provide

additional filtering [6]. Like for indexicals, those invariants are declarative and do not require the development of new ad hoc propagators. However, so far they have been discovered and proved manually and one-by-one. This leads to the following challenge:

Challenge 8 *Is there a way to discover in a systematic way new graph invariants?*

To the best of our knowledge, this type of question is not addressed by any community (CP, OR, graph theory). Our guess is that most people just focus on solving one problem. However, it makes sense from a theoretical point of view, since it would allow us to put in perspective and understand as a whole some families of propagators.

2.3 Verification and synthesis of propagators

The existence of efficient propagators for many constraints is recognised as a key strength of CP. The core of a solver is responsible for the cooperation and coordination of the propagators. For overall correctness and efficiency, a solver may require, or take advantage of, different properties of propagators, such as their correctness, monotonicity, idempotency, or achievement of a given level of consistency. Verifying those properties for a given propagator is most of the time done manually. Unfortunately, the intricacy of most propagators complicates this verification process greatly. The failure to conform to some property can lead to very subtle errors that show up only in specific situations, and debugging a constraint program at this level is notoriously difficult. It is thus desirable to automate the verification process of propagators, in order to certify that they fulfil the expectations of the solver and user. The arising question is then:

Challenge 9 *Can we adapt the techniques of formal verification in order to certify (semi-)automatically the correctness and other properties of propagators?*

While some properties are very specific to CP (for instance, the achievement of a given level of consistency), the verification of the correctness of a propagator with respect to the description of a constraint corresponds directly to the verification of the correctness of an algorithm with respect to its specification.

Several formal verification methods for this problem have constructive counterparts, which synthesise and verify algorithms at the same time. The synthesis of propagators can save a lot of effort, at least when prototyping new constraints, and has already been applied to specific classes of constraints; see [1, 11, 28, 32] for instance. This leads to the next challenge:

Challenge 10 *Is it possible to use inductive or deductive synthesis to generate (semi) automatically efficient propagators from the declarative description of some global constraints?*

We stress that we do not expect to replace human creativity by a computer. On the contrary we hope for a better collaboration between man and machine, where the computer takes care of the repetitive and tedious parts of the design and implementation, letting the human focus on the higher-level reasoning. Hence, a related challenge can be:

Challenge 11 *Are there higher-level abstractions (such as those of Section 2.2) that recur in the manual synthesis of propagators for some global constraints, so that propagators need not be designed from first principles each time?*

Positive results have already been achieved on some families of propagators (e.g., [21, 26]). More general results are to be expected but will require thorough inspection of existing propagators to discover and factor out recurring abstractions.

2.4 Visualisation and explanation

One of the major challenges for global constraints is providing a conceptual model of their execution for the application programmer, without requiring a detailed understanding of the propagators. Visualisation [12, 14] and explanation [16] can be very helpful in providing this information to the user. Special visualisers for some specific global constraints were already defined in [30], but even the more generic *CP-Viz* tool [31] requires hard-coded extensions for each global constraint. For some uses it may be sufficient to visualise just the state of the decision variables of the constraint. A more detailed view of the internal state of the constraint requires some interface to the propagators. This leads to the following challenge:

Challenge 12 *Is there a mechanism that generates a visualiser of a global constraint based on a declarative description of the constraint? Or, at least, can we define a toolkit to visualise classes of global constraints without tedious and error-prone adaptation for each constraint?*

The same questions can be asked for explanations.

Challenge 13 *Can explanations for global constraints be derived automatically from a declarative description of the constraint or from a high-level description of a propagator? What is required to generate automatically also explanations when synthesising propagators for global constraints? Can explanations be provided in a form that is understandable without describing details of the propagators?*

Those two challenges extend the ones of Section 2.3. Keeping the same spirit, we aim at an automation of the tedious parts of the development of visualisation and explanations, while keeping important design decisions in the hands of the human developer. This can be done for instance by extending the handling of the higher-level abstractions of Section 2.2 to those two use cases.

2.5 Scalability and big data

Scalability was one of the challenges identified in the *2011 Panel on the Future of CP* [24]. Scalability is becoming increasingly important for some industrial applications, such as cloud computing.

- On the one hand, CP provides an opportunity to scale well in terms of memory, because global constraints allow the implicit representation of large sets of similar constraints. This is not the case in SAT/SMT or MIP, where all constraints need to be stated explicitly in order to be actively taken into account by the solver.
- On the other hand, CP suffers from the following problems:
  1. In the context of non determinism, memory management is a weak point of CP solvers, no matter whether copying or trailing or a combination of the two is used for implementing backtracking. The price to pay to be able to backtrack at any point and to maintain incremental data structures is simply too high for large industrial problems.
  2. The convergence to a fixpoint for a conjunction of constraints can be very slow due to a mutual waking-up of high-complexity filtering algorithms.

A popular way to overcome the scalability issue is to use local search [33], which does not have any memory problem with backtracking. Large neighbourhood search [29] was introduced in order to keep the advantage of constraint propagation for small parts of a problem. There are indications that a constraint solver adapted for large neighbourhood search can perform quite well on large problem instances [20] compared to other solution techniques. Also, greedy propagators were introduced for packing [4] and scheduling [18] constraints that handle up to a few million items within a *single* constraint, using a standard CP solver.

This leads to the following challenge:

Challenge 14 *Are there, for some large classes of conjunctions of constraints, (greedy) propagators that address both the memory consumption problem and the slow convergence issue?*

2.6 A systematic reconstruction of the global constraint catalog

The current version of the *Global Constraint Catalogue* [5] contains around 400 individual constraints. Even with software support to find relevant constraints from examples [7], it is nearly impossible to remember all the alternatives and select the best constraints for a new application. Constraints were added as they were introduced in the literature, so there is very little structure beyond the description of each constraint on its own. Variants and extensions of known constraints are always added with their own entries, leading to a confusing multitude of related, but not clearly structured, entries.

A systematic reconstruction of the catalogue may be required to provide a framework for classifying constraints and their variants. This should identify core concepts (for example, ALLDIFFERENT), specialisations (PERMUTATION, ALLDIFFERENT_CONSECUTIVE_VALUES) and generalisations (LEX_ALLDIFFERENT, K_ALLDIFFERENT) of constraints, as well as variants of constraints like open

(OPEN_ALLDIFFERENT) and soft (SOFT_ALLDIFFERENT_VAR, SOFT_ALLDIFFERENT_CTR) versions, and the use of escape values (ALLDIFFERENT_EXCEPT_0).

At the same time, the argument structure and the naming of the constraints should be reconsidered to become more uniform. To have a real impact, this should not just be done for the constraint catalogue, but for as many implementations as possible, creating a more uniform view of global constraints throughout the community.

The aim of this restructuring is also to leverage the big amount of data and meta-data already present in the catalogue to allow a better automated processing. Tools using this data, like the Constraint Seeker [7], will certainly become more common in the future and it is important to make it available in a more structured way.

3 Interface of CP with other computer science areas

Not only real-world application areas and other sciences but also many research areas of computer science (CS) have sub-problems that are NP-hard constraint problems. Such areas include machine learning and data mining (ML/DM), computer-aided verification (CAV), software synthesis, computational linguistics, etc. Even here, with fellow computer scientists, communication problems occur, so the CP community has begun to reach out to some of these areas, witness the Dagstuhl seminar *CP meets ML/DM*,[2] as well as our *CP meets CAV*.[3] CP researchers have been invited to strategic events of other CS areas, witness some of us at the Dagstuhl seminar *Software Synthesis*.[4]

The overwhelming impression from these three recent events is that CP is widely misunderstood, as discussed in the next five paragraphs.

First, the standard reflex in other CS areas is to address the occurring NP-hard constraint problems using SAT/SMT or MIP solvers. If researchers of other CS areas have even heard of CP, then they often either think that the term 'constraint programming' is a synonym of 'constraint solving' (and thus that SAT, SMT, and MIP solvers are also CP solvers and not just constraint solvers, with CP solvers not offering anything distinct), or that the term 'constraint programming' is still synonymous with 'constraint logic programming' (CLP), the latter being thought still synonymous with CLP over rational numbers, CLP(Q), or real numbers, CLP(R).

Second, CP tutorials given at such events have revealed huge conceptual and vocabulary gaps, indicating that the CP community still has trouble communicating what it is doing and what distinguishes it from other approaches to the solving of constraint problems. Especially the beauty of the 'P' (programming) of 'CP' and its difference from the 'P' of 'MIP', say, seem hard to communicate (see e.g., [19]).

Third, deterrents to using CP include the prejudices that there are still no black-box CP solvers (hence that search procedures always have to be customised, possibly through lengthy experiments) and that CP solvers are almost never faster, as well as the valid objection that there is no standard interface to CP solvers. SAT, SMT, and MIP solvers offer these advantages, as well as their own disadvantages, which

[2] http://www.dagstuhl.de/11201

[3] http://www.it.uu.se/research/group/astra/CPmeetsCAV

[4] http://www.dagstuhl.de/12152

however tend to be ignored. Nevertheless, it seems presumptuous to believe that one approach (say SAT, SMT, or MIP) is always good enough. We have heard SAT and SMT users say that if their model is not fast enough then they just wait for a year in order to benefit, without effort of their own, from a solver twice as fast.

Fourth, due to the reflex reliance in other CS areas on SAT, SMT, or MIP solvers, it is also very hard to understand the constraint problems of other CS areas, because the problem descriptions tend to be in terms of the capabilities of the lower-level modelling languages they use.

Fifth, wheels tend to be reinvented if CP is not better known. For instance, the programming language community aims to integrate constraints into programming languages (sometimes upon solving all constraints using a SAT solver), blissfully unaware that this has been the essence of CP since the late 1980s and a distinguishing feature with other constraint solving approaches.

In conclusion, we raise the following challenge:

Challenge 15 *How can we, as a community, promote a better understanding of CP by other computer scientists?*

As part of the answer, we advocate the following measures:

- Organise more out-reach meetings with experts of other CS areas.
- Develop (on-line) material explaining CP to CS experts.
- Maintain (together with at least the SAT/SMT and MIP communities) a showcase of significant benchmarks where CP solvers outperform SAT/SMT and MIP solvers, or where CP practitioners challenge SAT/SMT and MIP practitioners to beat CP solvers (and vice-versa).
- Limit the impact of the absence of a standard interface to CP solvers.

We stress that the mentioned difficulties are not completely specific to CP, but addressing them could enhance the development and dissemination of CP.

References

1. Abdennadher, S., & Rigotti, C. (2005). Automatic generation of CHR constraint solvers. *Theory and Practice of Logic Programming, 5*(4), 403–418.
2. Beldiceanu, N., Carlsson, M., Petit, T. (2004). Deriving filtering algorithms from constraint checkers. In M.G. Wallace (Ed.), *CP 2004. LNCS* (Vol. 3258, pp. 107–122). Springer.
3. Beldiceanu, N., Carlsson, M., Petit, T., Régin, J.C. (2012). An $O(n \log n)$ bound consistency algorithm for the conjunction of an alldifferent and an inequality between a sum of variables and a constant, and its generalization. In L.D. Raedt, C. Bessière, D. Dubois, P. Doherty, P. Frasconi, F. Heintz, P.J.F. Lucas (Eds.), *ECAI 2012* (pp. 145–150). IOS Press.
4. Beldiceanu, N., Carlsson, M., Poder, E., Sadek, R., Truchet, C. (2007). A generic geometrical constraint kernel in space and time for handling polymorphic k-dimensional objects. In C. Bessière (Ed.), *CP 2007. LNCS* (Vol. 4741, pp. 180–194). Springer.
5. Beldiceanu, N., Carlsson, M., Rampon, J.X. (2012). *Global constraint catalog* 2nd edn. (revision a). Tech. Rep. T2012:03, Swedish Institute of Computer Science. Available at http://soda.swedish-ict.se/5195.
6. Beldiceanu, N., Petit, T., Rochart, G. (2006). Bounds of parameters for global constraints. *RAIRO Operations Research, 40*(4), 327–353.
7. Beldiceanu, N., & Simonis, H. (2011). A constraint seeker: Finding and ranking global constraints from examples. In J.H.M. Lee (Ed.), *CP 2011. LNCS* (Vol. 6876, pp. 12–26). Springer.
8. Berger, M. (2010). *Geometry revealed, a Jacob's ladder to modern higher geometry*. Springer.

9. Carlsson, M., Beldiceanu, N., Martin, J. (2008). A geometric constraint over *k*-dimensional objects and shapes subject to business rules. In P.J. Stuckey (Ed.), *CP 2008. LNCS* (Vol. 5202, pp. 220–234). Springer.
10. Colmerauer, A. (1996). *Les bases de Prolog IV*. Internal publication of the Laboratoire d'Informatique de Marseille, France. Available at http://alain.colmerauer.free.fr/.
11. Dao, T.B.H., Lallouet, A., Legtchenko, A., Martin, L. (2002). Indexical-based solver learning. In P. Van Hentenryck (Ed.), *CP 2002. LNCS* (Vol. 2470, pp. 541–555). Springer.
12. Deransart, P., Hermenegildo, M.V., Maluszynski, J. (Eds.) (2000). Analysis and visualization tools for constraint programming, constraint debugging (DiSCiPl project). *LNCS* (Vol. 1870). Springer.
13. Diaz, D., & Codognet, P. (1993). A minimal extension of the WAM for clp(FD). In *ICLP 1993* (pp. 774–790). The MIT Press.
14. Dooms, G., Van Hentenryck, P., Michel, L. (2009). Model-driven visualizations of constraint-based local search. *Constraints, 14*(3), 294–324.
15. de la Higuera, C. (2010). *Grammatical inference: Learning automata and grammars*. Cambridge University Press.
16. Jussien, N. (2003). *The versatility of using explanations within constraint programming.* Habilitation thesis, Université de Nantes, France. Available at http://tel.archives-ouvertes.fr/tel-00293905.
17. Laurière, J.L. (1978). A language and a program for stating and solving combinatorial problems. *Artificial Intelligence, 10*(1), 29–127.
18. Letort, A., Beldiceanu, N., Carlsson, M. (2012). A scalable sweep algorithm for the *cumulative* constraint. In M. Milano (Ed.), *CP 2012. LNCS* (Vol. 7514, pp. 439–454). Springer.
19. Lustig, I.J., & Puget, J.F. (2001). Program does not equal program: constraint programming and its relationship to mathematical programming. *Interfaces, 31*(6), 29–53.
20. Mehta, D., O'Sullivan, B., Simonis, H. (2012). Comparing solution methods for the machine reassignment problem. In M. Milano (Ed.), *CP 2012. LNCS* (Vol. 7514, pp. 782–797).
21. Monette, J.N., Beldiceanu, N., Flener, P., Pearson, J. (2013). A parametric propagator for discretely convex pairs of sum constraints. In C. Schulte (Ed.), *CP 2013. LNCS* (Vol. 8124, pp. 529–544). Springer.
22. Monette, J.N., Flener, P., Pearson, J. (2012). Towards solver-independent propagators. In M. Milano (Ed.), *CP 2012. LNCS* (Vol. 7514, pp. 544–560). Springer.
23. Oncina, J., & Garcia, P. (1992). Identifying regular languages in polynomial time. In *Advances in structural and syntactic pattern recognition* (Vol. 5, pp. 99–108).
24. O'Sullivan, B. (2011). CP panel position – The future of CP. Available at http://www.dmi.unipg.it/cp2011/downloads/slides/panel/osullivan.pdf.
25. Pesant, G. (2004). A regular language membership constraint for finite sequences of variables. In M.G. Wallace (Ed.), *CP 2004. LNCS* (Vol. 3258, pp. 482–495). Springer.
26. Petit, T., Beldiceanu, N., Lorca, X. (2011). A generalized arc-consistency algorithm for a class of counting constraints. In *IJCAI 2011* (pp. 643–648). AAAI Press / IJCAI. Revised edition available at http://arxiv.org/abs/1110.4719.
27. Prosser, P. (2012). *Exact algorithms for maximum clique: a computational study.* Tech. Rep. 2012-333, Department of Computer Science, Glasgow University, Scotland. Available at http://arxiv.org/abs/1207.4616.
28. Raiser, F. (2008). Semi-automatic generation of CHR solvers for global constraints. In P.J. Stuckey (Ed.), *CP 2008. LNCS* (Vol. 5202, pp. 588–592). Springer.
29. Shaw, P. (1998). Using constraint programming and local search methods to solve vehicle routing problems. In M.J. Maher, J.F. Puget (Eds.), *CP 1998. LNCS* (Vol. 1520, pp. 417–431). Springer.
30. Simonis, H., Aggoun, A., Beldiceanu, N., Bourreau, E. (2000). Complex constraint abstraction: Global constraint visualisation. In P. Deransart, M.V. Hermenegildo, J. Maluszynski, (Eds.), *Analysis and visualization tools for constraint programming, constraint debugging (DiSCiPl project). LNCS* (Vol. 1870, pp. 299–317). Springer.
31. Simonis, H., Davern, P., Feldman, J., Mehta, D., Quesada, L., Carlsson, M. (2010). A generic visualization platform for CP. In J.H.M. Lee (Ed.), *CP 2010. LNCS* (Vol. 6308, pp. 460–474). Springer.
32. Tack, G., Schulte, C., Smolka, G. (2006). Generating propagators for finite set constraints. In F. Benhamou (Ed.), *CP 2006. LNCS* (Vol. 4204, pp. 575–589). Springer.
33. Van Hentenryck, P., & Michel, L. (2007). *Constraint-based local search.* The MIT Press.
34. Van Hentenryck, P., Saraswat, V., Deville, Y. *Design, implementation, and evaluation of the constraint language cc(FD).* Tech. Rep. CS-93-02, Brown University, Providence, USA (1993). Based on the unpublished manuscript *Constraint Processing in cc(FD)*, 1991.