SolverCheck: Declarative Testing of Constraints *

Xavier Gillard[1], Pierre Schaus[1], and Yves Deville[1]

Université Catholique de Louvain, BE
{xavier.gillard, pierre.schaus, yves.deville}@uclouvain.be

Abstract. This paper introduces SolverCheck, a property-based testing (PBT) library specifically designed to test CP solvers. In particular, SolverCheck provides a declarative language to express a propagator's expected behavior and test it automatically. That language is easily extended with new constraints and flexible enough to precisely describe a propagator's consistency. Experiments carried out using Choco [38], JaCoP [26] and MiniCP [32] revealed the presence of numerous non-trivial bugs, no matter how carefully the test suites of these solvers have been engineered.

Introduction

Constraint Programming (CP) owes much of its success to the declarative aspect of its models and the expressiveness of its constraints. Obviously, CP wouldn't have been the achievement we all know if it weren't for the efficiency of the propagators that have been devised over the years to enforce some degree of consistency for the constraints enlisted in the catalog [5]. E.g. *alldiff* [39], *regular* [37], *element* [23]. Nevertheless, the success of the tools developed in our community remains fragile as results of a solver might be invalidated by a bogus implementation of one single propagator. As it turns out, the algorithms and data structures involved in those propagators are quite advanced and sometimes rely on state-restoration mechanisms. This is why, ensuring the correctness and robustness of their implementation is crucial to the success of CP as a whole. However, checking the correctness of a propagator by focusing solely on the absence of solution removal is far from enough. Indeed, in order to be able to tackle real world problems, it is essential that a solver be both correct and efficient. In practice, the efficiency of a propagator is the result of a balance between the strength of the enforced consistency and the complexity of the algorithm used to implement it. Hence, being able to test the consistency level imposed by a propagator becomes a necessity. Indeed, in the event where the consistency should be weaker than announced, some problem instances might become intractable and that intractability could hardly be analyzed or reasoned about.

In that context, we propose SolverCheck: an open-source property-based testing (PBT) library inspired by QuickCheck [13] for Haskell. It has been specifically

* Massart and Rombouts have worked on a preliminary version of this work for their MSc. thesis which we supervised. They presented it at the CP-2018 Doctoral Programme [30].

designed and engineered to improve the quality of the tests used to validate CP solvers. In practice, SolverCheck makes it easy to both test the *correctness* of the propagators and to test the level of consistency enforced by the latter. Moreover, SolverCheck aims at being an extensible framework. Therefore, it comes with simple interfaces through which a user can easily describe the relation imposed by a new constraint. Concretely, this relation is described using a *Checker*, a predicate deciding whether or not a tuple belongs to the constraint relation. Similarly, the consistency level that can be tested needs not necessarily be one of the classical consistency level (DC, BC(D), BC(Z), RC, FC)[6] as SolverCheck permits the definition of custom mixed consistencies matching the exact expected behavior of some given propagator. Additionally, SolverCheck is able to perform *dynamic checking* and hence to explicitly test the correctness of the state-restoration mechanisms involved in the targeted propagators.

The rest of our paper is organized as follows: Section 1 presents the background material necessary to understand the purpose and methodology applied in SolverCheck. Then, Section 2 briefly presents other related lines of research and how these relate to our work. After that, Section 3 introduces the various capabilities of SolverCheck through a simple yet illustrative example. Finally, Section 4 reports on the experiments that were made to validate the effectiveness of SolverCheck before conclusions are drawn in Section 5.

1 Property-based testing

SolverCheck adopts the so-called *property-based testing* paradigm which tackles the weaknesses of the classical *example-based testing* methodology. All the open-source solvers that we are aware of, in particular Gecode [42], Choco [38], JaCop [26], Or-tools [35], OscaR [36], and MiniCP [32], maintain a test suite to test the solver at the granularity of the constraints. The test suites of most of the solvers[1] follow the classical example-based approach.

As the name suggests, example-based testing relies on a tester to describe concrete situations (example, with actual variables and domain instantiation) supposedly representative of a class of errors. By combining many such examples, the tester creates a broad test suite covering a large number of potential problems. However, we point out two weaknesses of this approach. First, example-based unit tests are expensive to write and to maintain. Manually finding interesting instances to test is no easy task. It requires some expertise and intuition. Also, test code is often treated as a second class citizen: the quality standards applied to that fraction of the code are less stringent than for the rest of the code base. Therefore, it results that the code composing the test suites is often crippled with duplicate fragments. Moreover, the hard-coded instances fail to clearly communicate the intent regarding which important property is being tested with a given example. For instance, the objective of testing a global constraint's consistency level does not shine from any given test example. Add

[1] Gecode, and likewise Choco for some of its propagators, are a notable exception which is covered in the related work section.

to that picture the fact that example-based tests often opt for an all imperative coding style, and the original goal of the test becomes difficult to grasp. Meanwhile, example-based testing does not offer any means to improve on that floor or to test that kind of property in a generic way.

Property-based testing (PBT) addresses those weaknesses by a combination of *fuzzing* [41] and *formal specification*. Doing so, PBT changes the role of the test engineer. With PBT the test engineer must express the general *properties* that must hold for all executions of a given software rather than manually crafting lots of *test cases* (example-based testing). These properties are expressed in a high-level declarative language which abstracts away the details of actual test cases. As the name suggests, this method is *test-based*. Hence, it is inherently incomplete. Nevertheless, moving the burden of actual test case generation from the human tester to an automated tool makes PBT a remarkably effective approach to identifying bugs in practice.

2 Related work

The purpose of our research differs from the line of work started in the late '80s [15,19,14,31,27]. Indeed, that rich body of investigations aimed at verifying whether the *CP program* (today, one would rather talk about CP *model* instead) was correct. SolverCheck, on the other hand, aims at testing the implementation of a CP solver, which is a different concern by large. It also differs from the research embodied in FocalTest [11] which uses CP to define *smart generators* for PBT. Instead, SolverCheck provides a PBT library to assess the correctness and robustness of CP solvers.

Even though the properties to be tested are formally specified, SolverCheck is a *testing library*, not a formal verification tool. That distinction typically makes it simpler to use. Indeed, despite the many advances in the domain, proof-checkers for general purpose languages either require some human guidance, do not support all language constructs [1,4], or are currently unable to deal with programs as large and complex as modern CP-solvers [21,20,25]. Similarly, as of today, formally certified CP solvers [18,12] are nowhere close to the state of refinement and efficiency of state-of-the-art solvers. For instance, these rely on (efficient but suboptimal) OCaml code extracted from Coq [43] and only support constraints of arity greater than 3 through a decomposition into equivalent binary constraints (using the hidden variable encoding) [17].

Recently, the SAT/SMT/ASP/QBF communities have undertaken a line of work that closely relates to ours [9,8,3,34]. Just like SolverCheck, these techniques also apply fuzzing in order to ensure the quality of the tools they develop. However, that body of work ignores the specifics of a CP solver. In particular, they disregard consistency related issues (mixed or not). Meanwhile, as explained earlier, this is one of the essential aspects of the reasoning and development of a CP solver.

As it has already been mentioned, Gecode [42] and Choco [38] adopt an original test strategy which allows them to test the consistency (DC, BC(D))

imposed by some of their propagators[2]. Their approach, albeit elegant and efficient, is unable to deal with mixed consistencies (eg. that of the *element* [23] constraint).

Last year, Akgün et al. proposed at the CP conference an interesting approach based on metamorphic testing [2] to test the implementation of a solver. Their goal, as well as their initial intuition is the same as those behind SolverCheck. Both target the testing of propagators implemented in actual CP solvers, and both rely on having two distinct implementations of each filter. However, their approach relies on the *table* propagator from the target solver while SolverCheck automatically derives a naive alternative implementation of the propagator which is completely independent from the target solver. This allows SolverCheck to set the focus on a rich set of consistencies (among which, mixed consistencies), when [2] supports only GAC.

3 What SolverCheck has to offer

We will use the example reproduced in Listing 1.1 as a starting point. The latter is actually an excerpt of the test suite we wrote when testing JaCoP.

Listing 1.1: Example: JaCoP LexOrder($\leqslant$) must enforce GAC.

```
1  @Test
2  public void statelessLexLE() {
3   assertThat(
4    forAll(listOf("x", jDom())).assertThat(x ->
5    forAll(listOf("y", jDom())).assertThat(y ->
6      a(statelessJacopLexOrder(false))
7      .isEquivalentTo(arcConsistent(lexLE(x.size(), y.size())))
8      .forThePartialAssignment(x, y)
9    )));
10 }
```

3.1 Declarative testing

The declarative aspect of the test code reproduced in Listing 1.1 is obvious. No mention is ever made in the code about any concrete test case. Instead, that code snippet uses a declarative style close to that of a domain-specific language to express a *property*, a *specification* of what the code should do. The details of the actual tests that are used to validate the implementation are left to the system. Assuming a basic knowledge of Java, it is clear from Listing 1.1 that any reader – familiar with SolverCheck or not – will grasp the expressed property. In our example, it states that for any two given lists x and y of variables, the filtering of the domains imposed by the actual `LexOrder` constraint from JaCoP should strictly enforce domain consistency.

[2] Actually, both solvers adopt a slightly different approach, but this is not relevant for our matter as they are based on the same idea. For the full details, see http://bit.ly/cst-gecode and http://bit.ly/cst-choco.

Naturally, such a short code snippet is not fully self-contained, and the test engineer is expected to write a little bit of glue code – which we call an adapter. The sole purpose of that adapter is to ensure the interoperability of SolverCheck and the targeted solver. Concretely, implementing such an adapter amounts to writing a few functions performing the conversion between the types of SolverCheck and those of the target solver. These functions are typically very simple, and the effort to write them is only required once per solver. The adapter is written once and reused for all constraints and properties.

3.2 Consistency

Despite its apparent simplicity, the example from Listing 1.1 is a good illustration of the flexibility provided by SolverCheck. It shows how to parameterize the consistency level used to test a given propagator. It would only take a change of line 8 in the example to modify the property expressed in Listing 1.1 and let it state that the propagator should enforce BC(Z) rather than DC. For that purpose, the only change required would be to replace `arcConsistent(lexLE(x.size(), y.size()))` by `boundZConsistent(lexLE(x.size(), y.size()))`.

Because solver developers tend to be pragmatic people who favor general case efficiency over the compliance to pure mathematical consistency definitions, it is often the case that discrepancies exist between the implemented artifacts and the theoretical framework. To cope with that reality, SolverCheck offers facilities to express that a filtering should be stronger than (`isStrongerThan(·)`), weaker than (`isWeakerThan(·)`) or equivalent to (`isEquivalentTo(·)`) a given consistency level. This is illustrated by line 7 in our illustrating example. However, a relative positioning wrt a "standard" consistency level might be deemed too weak. This is why SolverCheck also supports the definition of custom mixed consistencies. The example of Listing 1.2 illustrates how the exact mixed consistency of a propagator is specified with SolverCheck (line 8). That example shows that for any array A of integer and pair of variables x and y, MiniCP's $element(A, x, y) \equiv A[x] = y$ constraint does not comply with any of the standard consistencies. Instead, the property states that each value in the domain of x should have a support in y whereas only the upper and lower bounds of $D(y)$ should have a support in x.

Note that in addition to illustrating the expression of mixed consistency filters, our example also shows how to impose a time limit on the checking of properties (Line-3). This feature means that one can force a test to stop (with an inconclusive result) when a given duration has elapsed. This is useful in CI systems where builds are expected to complete (relatively) swiftly.

Listing 1.2: A[x] = y has a mixed consistency

```
1 @Test
2 public void elementIsHybridConsistent() {
3   given(TIMELIMIT, TimeUnit.SECONDS).assertThat(
4     forAll(listOf("A", integer())).assertThat(A ->
5     forAll(domain("x")).assertThat(x ->
6     forAll(domain("y")).assertThat(y ->
```

```
      a(minicpElement1D(A))
        .isEquivalentTo(hybrid(element(A), rangeDomain(), bcDDomain()))
        .forThePartialAssignment(x, y)
    ));
}
```

3.3 Extensibility

The example from Listing 1.3 illustrates how SolverCheck's capabilities can be extended to support constraints that were not initially foreseen[3]. To that end, it suffices to implement a new `Checker` for the desired constraint. That is a predicate on assignment which is true iff the assignment belongs to the constraint relation.

On top of the assertions meant to test the strength of a propagator, SolverCheck provides several extension points making it possible to check virtually any property of the tested filter. For instance, in the snippet `a(tested).is(property)`, the method `is(·)` will accept any predicate on partial assignments for its property argument. In particular, this is how the checks `isContracting()`, `isIdempotent()` and `isWeaklyMonotonic()` have been implemented in the library.

Listing 1.3: Extending SolverCheck to support new constraints

```
public Checker lexLE(int x_sz, int y_sz) {
 return assignment -> {
   var xs = assignment.subList(0, x_sz);
   var ys = assignment.subList(x_sz, x_sz+y_sz);
   for (int i=0; i < min(x_sz, y_sz); i++) {
    if (xs.get(i) < ys.get(i)) return true;
    if (xs.get(i) > ys.get(i)) return false;
   }
   return x_sz <= y_sz;
 };
}
```

3.4 Dynamic checking

Because there are many cases where existing solvers implement the filtering of their constraints as *incremental propagators*, it is necessary to ensure that their behavior is correct at any time during a search. In particular, one needs to make sure that the stated properties remain satisfied even after many state restorations. This is why SolverCheck offers a *dynamic checking* mode in addition to the *static checks* that have been presented up to now. The definition of such dynamic checks for properties is SolverCheck's way to testing that both the propagators and the state restoration mechanisms of the solver are correct.

As in the static case, whenever SolverCheck performs a dynamic check for some given property, it starts by generating a pseudo-random input. However, in the dynamic case, that random input is seen as the root of search tree and

[3] SolverCheck comes with built-in checkers for the usual constraints `alldiff`, `element`, `gcc`, etc.

a series of dives is performed in order to explore a fraction of it. At each node of the search tree, the system verifies whether the checked property still holds. Should it not be the case, then SolverCheck would report a trace leading to a state where the property is violated. When SolverCheck encounters a leaf, it decides to roll back the search until an arbitrary node (chosen on the latest visited branch) before starting a new dive.

In practice, using the dynamic checking mode requires slightly more work from the human tester. Indeed, rather than writing a plain stateless adapter as discussed in Section 3.1, the test engineer must write an adapter matching the `StatefulFilter` interface (Listing 1.4) to let SolverCheck know how the search is to be driven. Fortunately, this typically amounts to very little effort and needs only be written once for all properties. Once that is done, it is sufficient to use the *stateful* keyword to square the expected filtering declaration in order to let SolverCheck validate the property with dynamic checks. Concretely, in our example, it would suffice to adapt line 8 of Listing 1.1 to make it look like `stateful(arConsistent(...))`.

Listing 1.4: Interface of a Stateful Filter in SolverCheck

```
1 public interface StatefulFilter {
2    void setup(PartialAssignment initialDomains);
3    void pushState();
4    void popState();
5    void branchOn(int variable, Operator op, int value);
6    PartialAssignment currentState();
7 }
```

4 Evaluation

We conducted a series of experiments, all of which are based on three solvers[4]: Choco [38], JaCoP [26] and MiniCP [32]. These solvers have been chosen because, on the one hand, they run on the JVM which is our target platform; and on the other hand, because they have been carefully developed by domain experts. Among the large panel of possible constraints, we picked seven that were deemed representatives of the kind of constraints typically available in a CP solver.

We observed five different kind of outcomes during this experiment and summarize our findings in Table 1. The first possibility occurs when SolverCheck wasn't able to detect any mismatch between the tested propagators and their documented behavior (✔in Table 1). An other possible result is observed when a propagator prunes more values than announced but never removes any solution (🡅). The defective cases are split in three categories: the cases where a propagator was weaker than announced (🡇), the cases where it provided an incorrect answer (✖) and those when an undesired behavior happened at runtime

[4] Experiments were also realized using AbsCon [28]. However, even though we highlighted some defects in this solver, we chose not to report on the outcome of these experiments because we are still discussing some of our findings with the maintainer of that solver.

(↯). Among others, this covers program crashes (cast errors, memory exhaustion, ...) and infinite loops. All of our findings have been reported to, and accepted by the solvers maintainers. As of today, the vast majority of the findings we reported have been fixed.

As shown per Table 1, SolverCheck was remarkably efficient at identifying discrepancies between the actual and the documented behavior of implemented propagators. And that, even though all these propagators had already been carefully tested by their authors. In that sense, this experiment outdid our expectations wrt static issues detection. As a consequence, the dynamic checking potential of our library remained unknown. Therefore, we conducted a variation of our experiment where we manually introduced bugs in the state management of the stateful constraints. Then we used dynamic checking to test the properties of the targeted constraints. For all the seeded bugs, SolverCheck correctly reported a trace where the bug expressed itself.

Table 1: Findings of the exploratory testing phase

Solver	Alldiff	Element	Table	Sum	GCC	Lex	Regular
Choco	↓✗	↓	↯↓	↓✗	✓	↯	✓
JaCoP	✓	↑	✗	✓	✓	↓	↯
MiniCP	↯	✓	↓✗	↯✗	N/A	N/A	N/A

It is interesting to note that all the test cases that revealed "problems" were small and intelligible to a human being. Not all of them were trivial though, as some highlighted subtle corner cases that were not initially foreseen by the programmers.

5 Conclusions and future work

In this paper we introduced SolverCheck, an open-source property-based testing library to effectively check the correctness of the propagators of any JVM-based solver. We showed how the library can be used to declaratively specify the properties which must hold for a constraint, and presented the two modes in which the tests can be operated.

Furthermore, we demonstrated the practical effectiveness of SolverCheck through an experimental study based on Choco, JaCoP and MiniCP. These results are promising as they show that our library has been able to identify bugs in the aforementioned solvers despite their being heavily tested. This shows that SolverCheck is successful at its intended purpose. On that basis, we are confident that SolverCheck can and should be an integral part of the quality assurance process of any JVM-based solver.

We envision several extensions of this work in the future. We believe that our library can be adapted and extended to cope with the specifics of scheduling constraints. For instance, it could be extended to generate trusted filters matching

the filtering of an edge-finding propagator [10,29,44]. Also, it could be extended to target different classes of bugs. For instance, we think that it would be interesting to leverage the features of SolverCheck to target aliasing issues which are also a common source of bugs in solvers supporting views. Beyond that, our library could benefit from the use of checkers that operate directly on partial assignments. With these, a trusted filter would not necessarily need to always test all assignments. Other possible extensions include microbenchmarking and the ability to test solvers outside of the JVM world through language-agnostic tests using MiniZinc [33] or XCSP3 [7].

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer (2016)
2. Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Metamorphic testing of constraint solvers. In: Hooker, J.N. (ed.) Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11008, pp. 727–736. Springer (2018)
3. Artho, C., Biere, A., Seidl, M.: Model-based testing for verification back-ends. In: Veanes, M., Viganò, L. (eds.) Tests and Proofs - 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7942, pp. 39–55. Springer (2013)
4. Barnes, J.: SPARK: The Proven Approach to High Integrity Software. Altran Praxis, http://www.altran.co.uk, UK (2012)
5. Beldiceanu, N., Carlsson, M., Rampon, J.X.: Global constraint catalog, 2nd edition. Tech. Rep. 2010:07, Swedish Institute of Computer Science (2010)
6. Bessiere, C.: Constraint propagation. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, Foundations of Artificial Intelligence, vol. 2, pp. 29–83. Elsevier (2006)
7. Boussemart, F., Lecoutre, C., Piette, C.: Xcsp3: An integrated format for benchmarking combinatorial constrained problems. arXiv preprint arXiv:1611.03398 (2016)
8. Brummayer, R., Järvisalo, M.: Testing and debugging techniques for answer set solver development. TPLP **10**(4-6), 741–758 (2010)
9. Brummayer, R., Lonsing, F., Biere, A.: Automated testing and debugging of SAT and QBF solvers. In: Strichman, O., Szeider, S. (eds.) Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6175, pp. 44–57. Springer (2010)
10. Carlier, J., Pinson, E.: Adjustment of heads and tails for the job-shop problem. European Journal of Operational Research **78**(2), 146 – 161 (1994), project Management and Scheduling
11. Carlier, M., Dubois, C., Gotlieb, A.: Focaltest: A constraint programming approach for property-based testing. In: Cordeiro, J., Virvou, M., Shishkov, B. (eds.) Software and Data Technologies - 5th International Conference, ICSOFT 2010, Athens, Greece, July 22-24, 2010. Revised Selected Papers. Communications in Computer and Information Science, vol. 170, pp. 140–155. Springer (2010)
12. Carlier, M., Dubois, C., Gotlieb, A.: A certified constraint solver over finite domains. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7436, pp. 116–131. Springer (2012)
13. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of haskell programs. In: Odersky, M., Wadler, P. (eds.) Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000. pp. 268–279. ACM (2000)
14. Dahmen, M.: A debugger for constraints in prolog. Tech. Rep. ECRC-91-11, ECRC (1991)

15. Debruyune, R., Fekete, J.D., Jussien, N., Ghoniem, M.: Proposition de format concret pour des traces générées par des solveurs de contraintes réalisation rntl oadymppac 2.2. 2.1 (2001), http://pauillac.inria.fr/~contraintes/OADymPPaC/Public/d2.2.2.1.pdf
16. Derval, G., Régin, J., Schaus, P.: Improved filtering for the bin-packing with cardinality constraint. Constraints **23**(3), 251–271 (2018)
17. Dubois, C.: Formally Verified Decomposition of Non-binary Constraints into Equivalent Binary Constraints. In: Magaud, N., Dargaye, Z. (eds.) Journées Francophones des Langages Applicatifs 2019. JFLA2019, Les Rousses, France (Jan 2019)
18. Dubois, C., Gotlieb, A.: Solveurs cp (fd) vérifiés formellement. In: Journées Francophones de Programmation par Contraintes (JFPC'13). pp. 115–118. aix-en-provence, France (Jun 2013)
19. Ducassé, M.: Opium$^+$, a meta-debugger for prolog. In: ECAI. pp. 272–277 (1988)
20. Filliâtre, J., Marché, C.: The why/krakatoa/caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4590, pp. 173–177. Springer (2007)
21. Filliâtre, J., Paskevich, A.: Why3 - where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (2013)
22. Frisch, A.M., Grum, M., Jefferson, C., Hernández, B.M., Miguel, I.: The design of ESSENCE: A constraint language for specifying combinatorial problems. In: Veloso, M.M. (ed.) IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007. pp. 80–87 (2007)
23. Hentenryck, P.V., Carillon, J.: Generality versus specificity: An experience with AI and OR techniques. In: Shrobe, H.E., Mitchell, T.M., Smith, R.G. (eds.) Proceedings of the 7th National Conference on Artificial Intelligence, St. Paul, MN, USA, August 21-26, 1988. pp. 660–664. AAAI Press / The MIT Press (1988)
24. Hoare, C.A.R.: The emperor's old clothes. Commun. ACM **24**(2), 75–83 (1981)
25. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: A software analysis perspective. Formal Asp. Comput. **27**(3), 573–609 (2015)
26. Kuchcinski, K., Szymanek, R.: Jacop-java constraint programming solver. In: CP Solvers: Modeling, Applications, Integration, and Standardization, co-located with the 19th International Conference on Principles and Practice of Constraint Programming (2013)
27. Lazaar, N., Gotlieb, A., Lebbah, Y.: A CP framework for testing CP. Constraints **17**(2), 123–147 (2012)
28. Lecoutre, C., Tabary, S.: Abscon 112: towards more robustness. In: 3rd International Constraint Solver Competition (CSC'08). pp. 41–48. Sydney, Australia (2008), https://hal.archives-ouvertes.fr/hal-00870841
29. Martin, P., Shmoys, D.B.: A new approach to computing optimal schedules for the job-shop scheduling problem. In: Cunningham, W.H., McCormick, S.T., Queyranne, M. (eds.) Integer Programming and Combinatorial Optimization, 5th International IPCO Conference, Vancouver, British Columbia, Canada, June 3-5, 1996, Proceedings. Lecture Notes in Computer Science, vol. 1084, pp. 389–403. Springer (1996)

30. Massart, A., Rombouts, V., Schaus, P.: Testing global constraints. CoRR **abs/1807.03975** (2018), http://arxiv.org/abs/1807.03975
31. Meier, M.: Debugging constraint programs. In: Montanari, U., Rossi, F. (eds.) Principles and Practice of Constraint Programming - CP'95, First International Conference, CP'95, Cassis, France, September 19-22, 1995, Proceedings. Lecture Notes in Computer Science, vol. 976, pp. 204–221. Springer (1995)
32. Michel, L., Schaus, P., Van Hentenryck, P.: MiniCP: A lightweight solver for constraint programming (2018), available from `www.minicp.org`
33. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: Minizinc: Towards a standard CP modelling language. In: Bessiere, C. (ed.) Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4741, pp. 529–543. Springer (2007)
34. Niemetz, A., Preiner, M., Biere, A.: Model-based api testing for smt solvers. In: Proceedings of the 15th International Workshop on Satisfiability Modulo Theories, SMT. p. 10 (2017)
35. van Omme, N., Perron, L., Furnon, V.: OR-Tools user's manual. Google Inc. (2014), available from `https://developers.google.com/optimization/`
36. OscaR Team: OscaR: Scala in OR (2012), available from `https://bitbucket.org/oscarlib/oscar`
37. Pesant, G.: A regular language membership constraint for finite sequences of variables. In: Wallace, M. (ed.) Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3258, pp. 482–495. Springer (2004)
38. Prud'homme, C., Fages, J.G., Lorca, X.: Choco Documentation. TASC - LS2N CNRS UMR 6241, COSLING S.A.S. (2017), http://www.choco-solver.org
39. Régin, J.: A filtering algorithm for constraints of difference in csps. In: Hayes-Roth, B., Korf, R.E. (eds.) Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1. pp. 362–367. AAAI Press / The MIT Press (1994)
40. Simonis, H., Davern, P., Feldman, J., Mehta, D., Quesada, L., Carlsson, M.: A generic visualization platform for CP. In: Cohen, D. (ed.) Principles and Practice of Constraint Programming - CP 2010 - 16th International Conference, CP 2010, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6308, pp. 460–474. Springer (2010)
41. Sutton, M.: Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley Professional (jul 2007)
42. Team, G.: Gecode: Generic constraint development environment (2008), https://www.gecode.org/
43. coq development team, T.: The coq proof assistant (2019), https://coq.inria.fr/
44. Vilím, P.: Timetable edge finding filtering algorithm for discrete cumulative resources. In: Achterberg, T., Beck, J.C. (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems - 8th International Conference, CPAIOR 2011, Berlin, Germany, May 23-27, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6697, pp. 230–245. Springer (2011)