Testing Global Constraints

Aurélie Massart, Valentin Rombouts, and Pierre Schaus

UCLouvain/ICTEAM Belgium

Abstract. Every Constraint Programming (CP) solver exposes a library of constraints for solving combinatorial problems. In order to be useful, CP solvers need to be bug-free. Therefore the testing of the solver is crucial to make developers and users confident. We present a Java library allowing any JVM based solver to test that the implementations of the individual constraints are correct. The library can be used in a test suite executed in a continuous integration tool or it can also be used to discover minimalist instances violating some properties (arc-consistency, etc) in order to help the developer to identify the origin of the problem using standard debuggers.

Keywords: Constraint Programming · Testing · Filtering

1 Introduction

The filtering algorithms inside constraint programming solvers ([1,2,3,4] etc.) are mainly tested using test suites implemented manually. Creating such unit tests is a significant workload for the developers and is also error prone.

The most elementary yet important test to achieve for a constraint is that no feasible solution is removed. One can always implement a checker verifying the feasibility of the constraint when all the variables are bound. By comparing the number of solutions generated with both the checker and the tested filtering algorithm, one can be confident that no solution is removed. This procedure can be repeated for many (small) instances (possibly randomly generated). Alternatively, one can compare with a decomposition of the constraint into (more) elementary ones. This latter approach can improve the coverage of the test suite.

Those unit tests verifying the non removal of feasible solutions do not verify other properties of constraints generally more difficult to test. For instance, the domain-consistency property is rarely tested outside some hard-coded small test examples.

We introduce CPChecker as a tool to ease the solver developer's life by automating the testing of properties of filtering algorithms. For instance, *algorithm A should filter more than algorithm B* or *Algorithm A should achieve arc or bound-consistency*, etc. The tool does not ensure that the tested filtering does not contain any bug - as it is impossible to test all possible input domains - but it can reveal the presence of one, if a test fails. The large variety of input domains pseudo-randomly generated should make the user confident that the tool would allow to detect most of the bugs.

Many constraint implementations are stateful and maintain some reversible data structures. Indeed, global constraints' filtering algorithms often maintain an internal state in order to be more efficient than their decomposition. This reversible state is also a frequent source of bugs. CPChecker includes the trail-based operations when testing constraints such that any bug due to the state management of the constraint also has a high chance to be detected. CPChecker is generic and can be interfaced with any JVM trailed based solvers. CPChecker is able to generate detailed explanations by generating minimal domain examples on which the user's filtering has failed, if any.

Related work In [5,6,7], the authors introduce tools to debug models. Some researches have also been done to help programmers while debugging codes for constraint programming [8]. To the best of our knowledge, these tools, unlike CPChecker, do not focus on the filtering properties of individual constraints.

In the next sections, we first detail how to test static filtering algorithms before explaining the testing of stateful filtering algorithms for trailed based solvers. Finally we introduce how CPChecker can be integrated into a test suite.

2 Testing Static Filtering Algorithms

CPChecker is able to test any static filtering algorithm acting over integer domains. Therefore, the user needs to implement a function taking some domains (array of set of ints) as input and returning the filtered domains[1]:

```
abstract class Filter {
 def filter(variables: Array[Set[Int]]): Array[Set[Int]]
}
```

CPChecker also needs a trusted filtering algorithm serving as reference with the same signature. The least effort for a user is to implement a checker for the constraint under the form of a predicate that specifies the semantic of the constraint. For instance a checker for the constraint $\sum_i x_i = 15$ can be defined as

```
 def sumChecker(x: Array[Int]): Boolean = x.sum == 15
```

One can create with CPChecker an Arc/Bound-Z/Bound-D/Range Consistent filtering algorithm by providing in argument to the corresponding constructor the implementation of the checker. For instance

```
class ArcFiltering(checker: Array[Int] => Boolean) extends
    Filter
val trustedArcSumFiltering = new ArcFiltering(sumChecker)
```

[1] Most of the code fragments presented are in Scala for the sake of conciseness but the library is compatible with any JVM-based language.

This class implements the `filter` function as a trusted filtering algorithm reaching the arc consistency by 1) computing the Cartesian product of the domains, 2) filtering with the checker the non solutions and 3) creating the filtered domains as the the union of the values. Similar filtering algorithms' (Bound-Z, Bound-D and Range) have been implemented from a checker.

Finally the `check` and `stronger` functions permit to respectively check that two compared filtering algorithms are the same or that the tested filtering is stronger than the trusted one.

```
def check/stronger(trustedFiltering: Filter, testedFiltering:
    Filter) : Boolean
```

The testing involves the following steps:

1. Random Domains generation [2].
2. Execution of the tested and trusted filtering algorithms (from CPChecker's filterings or another trusted one) to these random domains.
3. Comparison of the domains returned by the two filtering algorithms.

This process is repeated by default 100 times although all the parameters can be overridden for the creation of random domains, number of tests, etc.

2.1 Generation of Random Test Instances

In order to test a filtering implementation, CPChecker relies on a *property based testing* library called *ScalaCheck*[9][3]. This library includes support for the creation of random generators and for launching multiple test cases given those. CPChecker also relies on the ability of *ScalaCheck* of reducing the instance to discover a smaller test instance over which the error occurs.

2.2 Example

Here is an example for testing with CPChecker the arc-consistent *AllDifferent* constraint's in ***OscaR*** [2] solver :

```
object ACAllDiffTest extends App {
  def allDiffChecker(x: Array[Int]): Boolean = x.toSet.size ==
    x.length
  val trustedACAllDiff: Filter = new ArcFiltering(
   allDiffChecker)
  val oscarACAllDiff: Filter = new Filter {
    override def filter(variables: Array[Set[Int]]): Array[Set
   [Int]] = {
      val cp: CPSolver = CPSolver()
      val vars = variables.map(x => CPIntVar(x)(cp))
      val constraint = new AllDiffAC(vars)
      try {
```

[2] A seed can be set to reproduce the same tests.

[3] Similar libraries exist for most programming languages, all inspired by QuickCheck for Haskell.

```
10          cp.post(constraint)
11        } catch {
12          case _: Inconsistency => throw new NoSolutionException
13        }
14        vars.map(x => x.toArray.toSet)
15      }
16    }
17    check(trustedACAllDiff, oscarACAllDiff)
18  }
```

The trusted filtering algorithm is created thanks to the `ArcFiltering` class at line 3. The checker for AllDifferent simply verifies that the union of the values in the array has a cardinality equal to the size of the array, as defined at line 2. The tested filtering implements the `filter` function using ***OscaR***'s filtering. It first transforms the variables into ***OscaR***'s variables (line 7) then creates the constraint over them (line 8). It is then posted to the solver which filters the domains until fix-point before returning them.

3 Testing stateful constraints

Incremental Filtering Algorithms usually maintain some form of state in the constraints. It can for instance be reversible data-structures for trailed-based solvers. CPChecker allows to test a stateful filtering algorithm by testing it during a search while checking the state restoration. In terms of implementation, the incremental `check` and `stronger` functions compare `FilterWithState` objects that must implement two functions. The *setup* function reaches the fix-point while setting up the solver used for the search. The *branchAndFilter* function applies a branching operation on the current state of the solver and reaches a new fix-point for the constraint. The branching operations represent standard branching constraints such as $=, \neq, <, >$ and the `push/pop` operations on the trail allowing to implement the backtracking mechanism (see [10] for further details on this mechanism).

```
1  abstract class FilterWithState {
2    def setup(variables: Array[Set[Int]]): Array[Set[Int]]
3
4    def branchAndFilter(branching: BranchOp): Array[Set[Int]]
5  }
```

The process of testing an incremental/stateful filtering algorithm is divided into four consecutive steps :

1. Domains generation
2. Application of the *setup* function of the tested and trusted filtering algorithms.
3. Comparing the filtered domains returned at step 2.
4. Execution of a number of fixed number dives as explained next based on the application of *branchAndFilter* function.

3.1 Dives

A dive is performed by successively interleaving a push of the state and a domain restriction operation. When a leaf is reached (no or one solution remaining) the dive is finished and a random number of states are popped to start a new dive as detailed in the algorithm 1.

Algorithm 1: Algorithm performing dives

```
Dives (root, trail, nbDives)
    dives ← 0
    currentDomains ← root
    while dives < nbDives do
        while !currentDomains.isLeaf do
            trail.push(currentDomains)
            restriction ← new RandomRestrictDomain(currentDomains)
            currentDomains ← branchAndFilter(currentDomains, restriction)
        dives ← dives + 1
        for i ← 1 to Random(1,trail.size-1) do
            trail.pop()
```

3.2 Illustration over an Example

The next example illustrates CPChecker to test the ***OscaR***[2]'s filtering for the constraint $\sum_i x_i = 15$. It should reach Bound-Z consistency.

```
1  object SumBCIncrTest extends App {
2
3    def sumChecker(x: Array[Int]): Boolean = x.sum == 15
4    val trusted = new IncrementalFiltering(new BoundZFiltering(
       sumChecker))
5    val tested = new FilterWithState {
6      val cp: CPSolver = CPSolver()
7      var currentVars: Array[CPIntVar] = _
8
9      override def branchAndFilter(branching: BranchOp): Array[
       Set[Int]] ={
10       branching match {
11         case _: Push => cp.pushState()
12         case _: Pop => cp.pop()
13         case r: RestrictDomain => try {
14             r.op match {
15               case "=" => cp.post(currentVars(r.index) == r.
       constant)
16               ...}
17           } catch {
18             case _: Exception => throw new NoSolutionException
19           }
20       }
21       currentVars.map(x => x.toArray.toSet)
22     }
23
24     override def setup(variables: Array[Set[Int]]): Array[Set[
       Int]] = {
25       currentVars = variables.map(x => CPIntVar(x))
26       try {
27         solver.post(sum(currentVars) == 15)
28       } catch {
29         case _: Exception => throw new NoSolutionException
30       }
31       currentVars.map(x => x.toArray.toSet)
32     }
33   }
34   check(trusted, tested)
35 }
```

In this example, two `FilterWithState` are compared with the `check` function.

In CPChecker, the `IncrementalFiltering` class implements the `FilterWithState` abstract class for any `Filter` object. Therefore, the `IncrementalFiltering` created with a `BoundZFiltering` object is used as the trusted filtering (line 4) which it-self relies on the very simple `sumChecker` function provided by the user and assumed to be bug-free.

4 Custom Assertions

To ease the integration into a JUnit like test suite, CPChecker has custom assertions extending the *AssertJ*[11] library. The classes `FilterAssert` and `FilterWithStateAssert` follow the conventions of the library with the `filterAs` and `weakerThan` functions to respectively test a filtering algorithm, as in the `check` and `stronger` functions. An example of assertion is:

```
1 assertThat(tested).filterAs(trusted1).weakerThan(trusted2)
```

5 Code Source

CPChecker's code source is publicly available in the *Github* repository[4]. This repository also contains several examples of usage of CPChecker with both *Scala* solver and *Java* solvers, namely ***OscaR***[2], ***Choco***[1] and ***Jacop***[3]. From those examples, *CPChecker* detected that the arc consistent filtering of the *Global Cardinality* constraint of ***OscaR*** was not arc consistent for all the variables (the cardinality variables). This shows the genericity of *CPChecker* and that it can be useful to test and debug filtering algorithms with only a small workload for the user. Further details on the architecture and implementation of CPChecker can be found in the Master Thesis document available at the github repository[4].

6 Conclusion and Future Work

This article presented CPChecker, a tool to test filtering algorithms implemented in any JVM-based programming language based on the JVM. Filtering algorithms are tested over domains randomly generated which is efficient to find unexpected bugs. Principally written in *Scala*, CPChecker can be used to test simple and stateful filtering algorithms. It also contains its own assertions system to be directly integrated into test suites. As future work, we would like to integrate into CPChecker properties of scheduling filtering algorithms [12] such as edge-finder, not-first not-last, time-table consistency, energy filtering, etc. for testing the most recent implementation of scheduling algorithms [13,14,15,16,17,18].

[4] https://github.com/vrombouts/Generic-checker-for-CP-Solver-s-constraints

References

1. Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017.
2. OscaR Team. OscaR: Scala in OR, 2012. Available from `https://bitbucket.org/oscarlib/oscar`.
3. Krzysztof Kuchcinski, Radoslaw Szymanek and contributors. Jacop solver. Available from `https://osolpro.atlassian.net/wiki/spaces/JACOP/`.
4. Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`.
5. Micha Meier. Debugging constraint programs. In *International Conference on Principles and Practice of Constraint Programming*, pages 204–221. Springer, 1995.
6. Peter J. Stuckey Carleton Coffrin, Siqi Liu and Guido Tack. Solution checking with minizinc. In *ModeRef2017, The Sixteenth International Workshop on Constraint Modelling and Reformulation*, 2017.
7. Nadjib Lazaar, Arnaud Gotlieb, and Yahia Lebbah. A cp framework for testing cp. *Constraints*, 17(2):123–147, 2012.
8. Frédéric Goualard and Frédéric Benhamou. *Debugging Constraint Programs by Store Inspection*, pages 273–297. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.
9. ScalaCheck. `http://scalacheck.org`.
10. Laurent Michel, Pierre Schaus, and Pascal Van Hentenryck. Minicp: A minimalist open-source solver to teach constraint programming. *Technical Report*, 2018.
11. AssertJ Library. `http://joel-costigliola.github.io/assertj/index.html`.
12. Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-based scheduling: applying constraint programming to scheduling problems*, volume 39. Springer Science, 201.
13. Steven Gay, Renaud Hartert, and Pierre Schaus. Time-table disjunctive reasoning for the cumulative constraint. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 157–172. Springer, 2015.
14. Cyrille Dejemeppe, Sascha Van Cauwelaert, and Pierre Schaus. The unary resource with transition times. In *International conference on principles and practice of constraint programming*, pages 89–104. Springer, 2015.
15. Hamed Fahimi and Claude-Guy Quimper. Linear-time filtering algorithms for the disjunctive constraint. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 2637–2643. AAAI Press, 2014.
16. Petr Vilím. Global constraints in scheduling. 2007.
17. Petr Vilím. Timetable edge finding filtering algorithm for discrete cumulative resources. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 230–245. Springer, 2011.
18. Alexander Tesch. A nearly exact propagation algorithm for energetic reasoning in o (n2logn). In *International Conference on Principles and Practice of Constraint Programming*, pages 493–519. Springer, 2016.