SolverCheck: Declarative Testing of Constraints *

Xavier Gillard[1][0000−0002−4493−6041], Pierre Schaus[1][0000−0002−3153−8941], 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 [41], JaCoP [27] and MiniCP [35] revealed the presence of numerous non-trivial bugs, no matter how carefully the test suites of these solvers have been engineered. Beyond the remarkable effectiveness of our technique to assess the correctness and robustness of a solver, our experiments also demonstrated the practical usability of SolverCheck to test actual CP-solvers.

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* [42], *regular* [40], *element* [24]. Nevertheless, the success of the tools developed in our community remains fragile as results of a solver might all 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 results from 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. Else, should the consistency

* 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 [33].

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 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 need 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.

Our contribution with this paper is the following: we propose SolverCheck as a DSL and tool to help improve the quality and robustness of JVM-based CP solvers. Given the very implementation-minded nature of the CP community, we hope that it can benefit the whole community and foster further innovation in the same way as Minizinc [36], XCSP3 [7], CPViz [44], Essence [23], etc... have in the past.

Outline The rest of our paper is organized as follows: Sections 1 and 2 present the background material necessary to understand the purpose and methodology applied in SolverCheck. Then, Section 3 briefly presents other related lines of research and how these relate to our work. After that, Section 4 introduces the various capabilities of SolverCheck through a simple yet illustrative example. Section 5 gives some more details relative to the implementation of our tool. Finally, Section 6 reports on the experiments that were made to validate the effectiveness and practical usability of SolverCheck before conclusions are drawn in Section 7.

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 [46], Choco [41], JaCop [27], Or-tools [38], OscaR [39], and MiniCP [35], 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.

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

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 what 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 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* [45] and *formal specification*. Doing so, PBT changes the role of the test engineer. With PBT he 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 Mixed consistency

In order to solve a CSP, filters are used to reduce the search space. A filter applied on a constraint aims at establishing some consistency property of this constraint by removing some values in the domain of its variables, without removing any solutions. We thus hereby only consider filters for *domain-based* consistencies. That is filters reducing the domain of variables.

In particular, we would like to set the focus on filters where different consistencies are mixed in a constraint. The idea of mixed consistency is to maintain different levels of consistency on the different variables of a constraint. The concept of mixed consistency has been introduced in [17] to handle graph and set variables. It is also used in [29,31].

A *constraint* c over the variables $(x_1, \ldots, x_n)$ (its scope) is a relation over the values of the variables. Like any relation, a constraint c can either be defined *in extension* as the set of all the n-tuples belonging to c. Or it can be defined *in comprehension* using a *checker* $c(v_1, \ldots, v_n)$ stating if $\langle v_1, \ldots, v_n \rangle$ belongs to c.

The *domain* of a variable x is a finite set of discrete values $D(x) \subset \mathbb{Z}$. It inherits the usual properties of proper finite subsets of $\mathbb{Z}$. In particular, it is either empty or it has a minimum (noted $lb(D(x))$) and a maximum (noted $ub(D(x))$). We denote D the set of tuples $D(x_1) \times \ldots \times D(x_n)$. A tuple $\tau = \langle v_1, \ldots, v_n \rangle$ is said to be valid if $\tau \in D$. The element v_i of the tuple is denoted τ_i.

A *partial assignment* is a mapping associating a domain to each variable.

The idea of support is central in the notion of consistency. Intuitively, a support of a value of a variable is a valid tuple, involving this value and satisfying the constraint. The definition of support can also be extended by considering sets larger than the actual variables domains. For instance, one can consider all the integer values between the bounds of the domain. We define $D^{\mathbb{Z}}(x) = \{v \in \mathbb{Z} \mid lb(D(x)) \leqslant v \leqslant ub(D(x)\}$.

Definition 1. *A* support *on c of (x_i, v) in D is a tuple $\tau \in D$ such that $\tau \in c$ and $\tau_i = v$. A* bound(Z) support *on c of (x_i, v) in D is a tuple $\tau \in D^{\mathbb{Z}}$ such that $\tau \in c$ and $\tau_i = v$.*

Different classical levels of consistency can now be defined. Each consistency, however, focuses on a single variable of the constraint. This will allow to later combine them in a mixed consistency.

Definition 2. *(DC) A constraint c is* domain consistent *on x_i with respect to D iff $\forall v \in D(x_i)$, there exists a support on c of (x_i, v) in D.*

Definition 3. *(RC) A constraint c is* range consistent *on x_i with respect to D iff $\forall v \in D(x_i)$, there exists a bound(Z) support on c of (x_i, v) in D.*

Definition 4. *(BC(D)) A constraint c is* bound(D) consistent *on x_i with respect to D iff $(x_i, lb(D(x_i)))$ and $(x_i, ub(D(x_i))$ have a support on c in D.*

Definition 5. *(BC(Z)) A constraint c is* bound(Z) consistent *on x_i with respect to D iff $(x_i, lb(D(x_i)))$ and $(x_i, ub(D(x_i))$ have a bound(Z) support on c in D.*

Definition 6. *(FC) A constraint c is* forward checking consistent *on x_i with respect to D iff when forall $j \neq i$ $D(x_j)$ is a singleton, then c is domain consistent on x_i with respect to D.*

Mixed consistency can now be defined with a consistency level associated to each variable of the constraint.

Definition 7. *Let $\Phi = \langle \phi_1, \ldots, \phi_n \rangle$ with $\phi_i \in$ {DC, RC, BC(D), BC(Z), FC}. The constraint c is Φ* mixed consistent *with respect to D iff c is ϕ_i consistent on x_i with respect to D.*

When Φ is a constant tuple, the above definition reduces to the standard definition of domain consistency or to the other standard levels of consistencies.

Given a consistency ϕ and a constraint c, we associate a filter $\phi_c(x, D)$ yielding a domain D' such that $D' \subseteq D$, $c \cap D = c \cap D'$ (same solutions) and c is ϕ_i consistent on x with respect to D'. A filter $\Phi_c(D)$ is also associated to a tuple of consistency Φ. It yields a domain D' such that $D' \subseteq D$, $c \cap D = c \cap D'$ (same solutions) and c is Φ mixed consistent with respect to D'.

Example 1. Given an array A of integer values, and two variables x, y, the *element(A,x,y)* constraint [24] is satisfied whenever $A[x] = y$. It is not uncommon for CP-solvers to implement a filter achieving the mixed consistency $(RC, BC(D))$ on the two variables (x, y). This kind of filter ensures that all values in the domain of x have a bound(Z) support, and that $lb(y)$ and $ub(y)$ have a support.

Algorithm 1 is a basic implementation of a filter, parameterized with a tuple of filters, achieving mixed consistency. This algorithm repeatedly reduces the domain of each individual variable x_i using its associated filter until a fixed point is reached. Assuming all the filters on the variables are correct, this algorithm yields a domain D' such that $D' \subseteq D$, $c \cap D = c \cap D'$ (same solutions) and c is Φ mixed consistent with respect to D'.

Algorithm 1: Filter achieving mixed consistency

```
1  Filter Φ_c(⟨φ_1, ..., φ_n⟩, D)
2      fixedpoint ← False ;
3      while !fixedpoint do
4          fixedpoint ← True;
5          foreach x_i ∈ scope(c) do
6              D'_{x_i} ← φ_i(x_i, D) ;
7              if D'_{x_i}.isEmpty() then return Fail ;
8              fixedpoint ← fixedpoint ∧ D(x_i) = D'_{x_i} ;
9              D(x_i) ← D'_{x_i};
10     return D ;
```

Generally speaking, a filter of a constraint modifies domains. A filter f_c should be contracting $(f_c(D) \subseteq D)$ and idempotent, that is a repeated application of the filter does not further reduce the domain $(f_c(f_c(D)) = f_c(D))$. In [43], Schulte and Tack have shown that weak monotony is the minimal necessary condition that any filter must fulfill in order to guarantee the soundness and the completeness of constraint propagation. A filter f_c is weakly monotonic iff $\forall D, \forall v \in D : f_c(\{v\}) \subseteq f_c(D)$. A correct filter for some constraint c is thus necessarily weakly monotonic and contracting. The corollary of this property is that a correct filter behaves as the checker applied to a singleton domain (i.e. an assignment).

Two filters f_1, f_2 of a given constraint can be compared thanks to their relative *strength*. A filter f_1 is said to be *stronger* than f_2 (noted $f_1 \sqsubseteq f_2$) iff $\forall D : f_1(D) \subseteq f_2(D)$. Similarly, f_1 is said to be *weaker* than f_2 ($f_1 \sqsupseteq f_2$) iff f_2 is stronger than f_1. Finally, f_1 and f_2 are *equivalent* whenever both $f_1 \sqsubseteq f_2$ and $f_1 \sqsupseteq f_2$ hold.

This paper aims at comparing filters. Therefore, we will say that a filter realizes a given consistency ϕ if it does not remove any further values than the

ones required per the consistency definition. That is, we say that a filter realizes the consistency ϕ iff it is the *weakest filter* (removing as few values as possibly can) complying with the definition of ϕ. Any other filter also realizing ϕ that removes additional (non-solution) values is therefore *stronger than* ϕ.

Example 2. It is clear from definition 2 and 4 that whenever a filter f realizes DC, it also realizes BC(D). However, f possibly removes more values than a hypothetical filter g that enforces BC(D) but not DC. Hence, we have $f \sqsubseteq g$. Thus, f is not the *weakest* filter realizing BC(D). Therefore, in this paper, we would not say that f is equivalent to BC(D) – although it realizes that consistency. Instead, we would say that f is *stronger than* BC(D).

3 Related work

The purpose of our research differs from the line of work started in the late '80s [15,20,14,34,28]. 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 [22,21,26]. Similarly, as of today, formally certified CP solvers [19,12] 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 [47] and only support constraints of arity greater than 3 through a decomposition into equivalent binary constraints (using the hidden variable encoding) [18].

Recently, the SAT/SMT/ASP/QBF communities have undertaken a line of work that closely relates to ours [9,8,3,37]. 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 [46] and Choco [41] 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 ef-

[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.

ficient, is unable to deal with mixed consistencies (eg. that of the *element* [24] 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 implementation of each filter. However, their approach relies on the *table* propagator from the target solver and requires the test-engineer to provide a table with all the solutions of the tested constraint (the authors of [2] provide a python function to help alleviate that burden). SolverCheck uses a different approach: it automatically derives a naive alternate implementation of the propagator which is completely independent from the target solver. Moreover, SolverCheck sets the focus on mixed consistencies, which is not the case of [2]. Additionally, the approach used in SolverCheck makes it easy to test properties that do not depend on a specific consistency level such as "stronger filtering". This kind of comparison is particularly well suited to compare the filtering for NP-hard constraints such as bin-packing [16].

4 What SolverCheck has to offer

We will use the example reproduced in Listing 1.1 as a starting point. The latter is actually the verbatim copy of a property we specified when writing a test suite for JaCoP.

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

```
@Test
public void statelessLexLE() {
 assertThat(
  forAll(listOf("x", jDom())).assertThat(x ->
  forAll(listOf("y", jDom())).assertThat(y ->
    a(statelessJacopLexOrder(false))
    .isEquivalentTo(arcConsistent(lexLE(x.size(), y.size())))
    .forThePartialAssignment(x, y)
  )));
}

// Generate a domain respecting JaCoP's documented limits
public GenDomainBuilder jDom() {
 return domain().withValuesBetween(
   IntDomain.MinInt,
   IntDomain.MaxInt);
}
// Discriminate solutions from non-solutions
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;
 };
}
// Adapter to expose the actual constraint as a SolverCheck Filter
```

```
31 private Filter statelessJacopLexOrder(final boolean lt) {
32   return partialAssignment -> {
33     Store store = new Store();
34     IntVar[][] vars = componentsToVars(store, partialAssignment);
35     store.impose(new LexOrder(vars[0], vars[1], lt));
36     if (!store.consistency()) {
37       return PartialAssignment.error(partialAssignment.size());
38     } else {
39       return vars2Partial(vars);
40     }
41   };
42 }
```

4.1 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 (lines 3-9). 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.

All the other functions declared in the example are actually utility methods: `jDom()` (lines 13-17) provides a means to generate pseudo-random domains [3] having their values in the range of values accepted by JaCoP. The `lexLE()` method (lines 19-29) returns a `Checker` for the Lex constraint. That is, it returns a predicate deciding whether or not an assignment belongs to the constraint relation. The `Checker` API is SolverCheck's mechanism to test constraints that are not built in the framework. The `statelessJacopLexOrder()` method (lines 31-42) adapts the actual constraint from JaCoP (`LexOrder`) and exposes it as a `Filter` that SolverCheck can interact with.

4.2 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 solvers 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

[3] sets of pseudo-random int

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. Additionally, this example illustrates (line 3) how a time limit can be set to check a property.

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 ->
7        a(minicpElement1D(A))
8          .isEquivalentTo(hybrid(element(A), rangeDomain(), bcDDomain()))
9          .forThePartialAssignment(x, y)
10     ))));
11 }
```

4.3 Extensibility

The example from Listing 1.1 also illustrates how SolverCheck's capabilities can be extended to support constraints that were not initially foreseen[4]. 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.

4.4 Dynamic checking

Because there are many cases where existing solvers implement the filtering of their constraints as *incremental propagators*, they do not exactly fit the ideal

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

of *pure* filtering functions having no side effects. Indeed, propagators hold and manipulate some internal state in order to deliver an efficient filtering in practice. But the efficiency gains often come at the expense of an increased risk of error. In order to detect the bugs caused by this internal state handling, SolverCheck proposes two operating modes.

Static Checking. Pseudo random test cases are fed to the filter. Then, the library tests if the outcome of *one application* of the filter delivers the expected result. This corresponds to the way of using SolverCheck which has been presented until now.

Dynamic Checking. The tested solver searches through the state-space, making branch decisions and backtracking, in conditions similar to those of an actual CSP solving.

Algorithm 2 describes how dynamic checks operate. This algorithm accepts five parameters: two stateful filters *trusted* and *tested* matching the interface of Listing 1.3, a property *prop* that must hold of all executions, a natural number N and a pseudo-randomly generated partial assignment *pa*. As opposed to static checking, dynamic checking considers the partial assignment *pa* as the root of a search tree and explores a fraction of that search tree with a series of N dives. That is, it dives N times in the search tree until a leaf (assignment or error) is reached (lines 7–12). At that moment, the library rolls back a few decisions it made when diving (lines 13–15). Then it starts the exploration of a new branch. At each visited node of the search tree, the current states of *tested* and *trusted* are compared to check whether the property is verified (line 12).

Algorithm 2: Dynamic checking, the *dive* algorithm

```
1  Dive (trusted, tested, prop, N, pa)
2      trusted.initialize(pa); tested.initialize(pa);
3      if not prop.holds(trusted, tested) then fail;
4      for N times do
5          CheckBranch(trusted, tested, prop, pa);
6          BackJump(trusted, tested);

7  CheckBranch (trusted, tested, prop, pa)
8      while neither trusted nor tested reached a leaf do
9          trusted.saveState(); tested.saveState();
10         decision ← RandomDecision(pa);
11         trusted.branchOn(decision); tested.branchOn(decision);
12         if not prop.holds(trusted, tested) then fail;

13 BackJump (trusted, tested)
14     while not trusted.isAtRootLevel() and RandomBool() do
15         trusted.restoreState(); tested.restoreState();
```

Writing dynamic checks, in practice As is made clear by the previous paragraphs, using the dynamic checking mode requires slightly more work from the human tester. Indeed, rather than writing a stateless `Filter` adapter similar to the one shown in Listing 1.1 (lines 31–42), the tester must write an adapter matching the `StatefulFilter` interface (Listing 1.3).

The `branchOn()` method stands for the addition of usual branching conditions such as $\neq$, $<$, $>$. The `pushState()` and `popState()` methods adopt the terminology used in trail-based solvers where these methods are at the heart of the backtracking mechanism (see [35] for further information on that matter).

For the rest, thanks to the declarative nature of SolverCheck, the code remains almost identical to what is required in the static case.

Listing 1.3: 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 }
```

5 Implementing SolverCheck

SolverCheck posits that the *implementation* of current CP solvers have become incredibly efficient at the expense of an increased code complexity. Therefore, they no longer fit in Hoare's *"obviously no deficiencies"* category [25]. The idea behind SolverCheck is then fairly simple: the library tests the behavior of an actual (complicated) CP filter by simply comparing it with that of a (generated) implementation that is so outright simple that it is straightforward for anyone to *trust* that second implementation to be correct. In SolverCheck parlance, the filters obtained from a generated implementation are called *trusted filters*.

5.1 Deriving trusted filters

As explained in Section 4.3, SolverCheck trusted filters rely on a `Checker` to decide whether or not an assignment belongs to the tested constraint. On that basis, a naive but easily trusted filter implementation immediately follows from the definitions of Section 2. For that matter, one needs to distinguish *uniform* consistencies (DC, BC(D), BC(Z), RC, FC) from *mixed* consistencies.

Uniform consistencies. Given a checker c, a consistency γ and the partial assignment $pa = (D(x_1), \ldots, D(x_n))$ the trusted filter $\phi_{c,\gamma}$ proceeds as follows. It starts by computing the set $D_{sol} = \{\tau \in D^* \mid c(\tau)\}$ of solutions, where D^* stands for either D (when $\gamma \in \{DC, BC(D)\}$) or $D^{\mathbb{Z}}$ (when $\gamma \in \{RC, BC(Z)\}$). Then it computes a partial assignment pa' associating the domain $D'(x_i) =$

$\bigcup_{\tau \in D_{sol}} \tau_i$ to each variable x_i. Finally, it uses pa' to filter the original domains in pa according to the rules of γ. This is the final output of $\phi_{c,\gamma}$.

Deriving a trusted filter implementation for the uniform FC consistency is trivial: when the domains of all but one variable are singletons, the filter behaves as in the DC case. Otherwise, the original partial assignment is returned.

Mixed consistencies. The trusted filters derived for mixed consistencies are a direct implementation of Algorithm 1. There the filters ϕ_i used to filter the domain of each variable x_i simply consist in the application of a uniform trusted filter which is then projected over x_i.

5.2 Generation of pseudo-random test cases

In order to check that a property holds, SolverCheck generates pseudo-random test cases which are fed to both the trusted and tested filters. Because it is widely accepted among software engineers that extreme values often exhibit extreme behaviors, it was decided that SolverCheck would not use a uniform random distribution to generate its test inputs. Instead, it draws its values from a multi-modal distribution – the modes being the *usually problematic* values (zero, min and max). Doing so, it introduces a bias on the values occurring in the generated test cases.

6 Experimental results

We conducted a series of experiments, all of which are based on three solvers[5]: Choco [41], JaCoP [27] and MiniCP [35]. 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 representative of the kind of constraint typically available in a CP solver.

For each of the selected constraint, we present two distinct experiments. The first one aimed at evaluating the effectiveness of our library when it comes to detecting bugs in an actual solver. In practice, this experiment consisted in a phase of *exploratory testing* during which we went through the documentation of solvers/constraints and wrote specifications matching the documented behavior for each of the tested artifacts. The goal of our second experiment was to assess the usability of our library in practice. That is, we wanted to make sure SolverCheck could actually be used and be useful in a continuous integration setup. To that end, we measured the time it took for our exploratory tests to complete as well as the code coverage they achieved.

[5] Experiments were also realized using AbsCon [30]. 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.

6.1 Exploratory testing

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 (🗲). 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 documented behavior of implemented propagators. And that, even though all these propagators had already been carefully tested by their authors. The biased pseudo-random input generation used to produce "extreme" values naturally led to the identification of several over- and under-flows issues that are often counterintuitive for a human being. Table 1 shows however that it was far from the only type of error identified by our tool.

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

Errors in the state management The exploratory testing outdid our expectations wrt stateless issues detection. As a consequence, the stateful issues detection potential of our library remained unknown. Therefore, we conducted a variation of our exploratory testing experiment and designed it so as to specifically assess that potential. In practice, we manually introduced bugs in the state management of the stateful constraints. To that end, we replaced some *reversible integer* with its primitive counterpart in the source code. 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.

Similarly, we also checked whether SolverCheck would identify bugs after we altered the implementation of the reversible structure to let it discard some modification from the trail. Again, SolverCheck reported a violation trace for all the cases that have been tested.

6.2 Effectiveness in practice

The plots from Figure 1 provide a good illustration of the behavior we observed when SolverCheck is used to test properties with a varying number of variables[6]. In particular, Figure 1a plots the time it takes to test that the DC consistent propagator of each solver actually enforces DC with an increasing number of variables. While the exact duration of these tests is of little interest, the trend it indicates is informative. On the one hand, it clearly indicates an exponential duration blowup becoming significant beyond 12 variables. On the other hand, it also shows that dynamic checking consistently requires a longer amount of time to complete than a corresponding static check. The extreme similarity between the two groups of static and dynamic curves (the same graph with logarithmic y-axis does not show any major difference either) indicates that the test-completion time is dominated by SolverCheck rather than by the tested solver.

These observations had to be expected and directly stem from the algorithms implemented in our library. The need for our trusted filter to explicitly compute the set D_{sol} based on a filtering of D^* is sufficient to explain the exponential blowup in its own right. Similarly, the repeated application of filter operations (as per Algorithm 2) during dynamic checking explains why dynamic tests require longer to complete. Despite being expected and logically understood, both observations clearly highlight a limitation in the capabilities of our library: it does not scale and is not efficient when there are lots of variables to be considered (or when they have large domains).

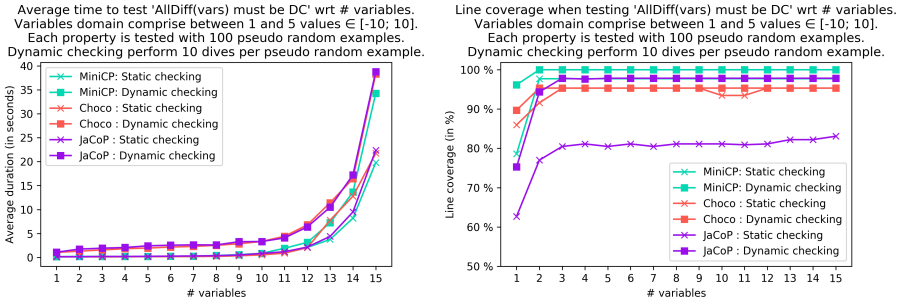
That conclusion should nevertheless be contrasted by the information shown on Figure 1b. It plots the line coverage of the tested propagators as measured during the tests whose runtime are plotted in Figure 1a. From there, we first observe that the coverage stabilizes very quickly. As soon as three variables are considered, the coverage reaches a state where it marginally increases, if at all. We also observe that in most cases, the tests cover about 95% of the lines. This is quite a high score, and is way above the usual 70-80% target from the industry. Finally, we observe that dynamic checking either improves or equates the static checking line coverage for all tests. The gap observed between the static and dynamic line coverage of JaCoP illustrates one of the benefits of dynamic checking. That strategy is able to exercise all parts of the propagators, including the ones related to state restoration of incremental propagators.

7 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

[6] The plots and observations to be made when it is the domain size that varies for a fixed number of variables are substantially the same as the case presented here (varying number of variables, fixed domain size). These are therefore omitted.

Fig. 1: Measuring the efficiency of SolverCheck for testing AllDiff (DC)



(a) (b)

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 and provide counterexamples for each of the witnessed property violations. Besides that, we showed that SolverCheck is successful at its intended purpose. It can easily be integrated in the test suite of any JVM-based solver to produce a high quality set of tests (good coverage) that is easy to maintain. Moreover, given that SolverCheck allows a tester to control every aspect of how tests are generated, we are also confident that SolverCheck can be an integral part of the quality assurance process of any solver. In particular, checking properties with our library can seamlessly be integrated with the continuous integration of any JVM-based solver.

We envision several extensions of this work in the future. In particular, 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 [10,32,48]. Also, it could be extended to target different classes of bugs. So far, SolverCheck is very good at finding value-related bugs like over/under-flows and logical errors in the propagation. 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 [36] 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. Dooms, G., Deville, Y., Dupont, P.: Cp(graph): Introducing a graph computation domain in constraint programming. In: van Beek, P. (ed.) Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3709, pp. 211–225. Springer (2005)
18. 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)
19. 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)
20. Ducassé, M.: Opium$^+$, a meta-debugger for prolog. In: ECAI. pp. 272–277 (1988)
21. 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)
22. 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)
23. 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)
24. 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)
25. Hoare, C.A.R.: The emperor's old clothes. Commun. ACM **24**(2), 75–83 (1981)
26. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: A software analysis perspective. Formal Asp. Comput. **27**(3), 573–609 (2015)
27. 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)
28. Lazaar, N., Gotlieb, A., Lebbah, Y.: A CP framework for testing CP. Constraints **17**(2), 123–147 (2012)
29. Lecoutre, C., Prosser, P.: Maintaining singleton arc consistency. In: Proceedings of the 3rd International Workshop on Constraint Propagation And Implementation (CPAI'2006) held with CP'2006. pp. 47–61. Springer (2006)

30. 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
31. Lesaint, D., Mehta, D., O'Sullivan, B., Quesada, L., Wilson, N.: Solving a telecommunications feature subscription configuration problem. In: Stuckey, P.J. (ed.) Principles and Practice of Constraint Programming, 14th International Conference, CP 2008, Sydney, Australia, September 14-18, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5202, pp. 67–81. Springer (2008)
32. 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)
33. Massart, A., Rombouts, V., Schaus, P.: Testing global constraints. CoRR **abs/1807.03975** (2018), http://arxiv.org/abs/1807.03975
34. 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)
35. Michel, L., Schaus, P., Van Hentenryck, P.: MiniCP: A lightweight solver for constraint programming (2018), available from `www.minicp.org`
36. 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)
37. 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)
38. van Omme, N., Perron, L., Furnon, V.: OR-Tools user's manual. Google Inc. (2014), available from `https://developers.google.com/optimization/`
39. OscaR Team: OscaR: Scala in OR (2012), available from `https://bitbucket.org/oscarlib/oscar`
40. 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)
41. 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
42. 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)
43. Schulte, C., Tack, G.: Weakly monotonic propagators. In: Gent, I.P. (ed.) Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings. Lecture Notes in Computer Science, vol. 5732, pp. 723–730. Springer (2009)

44. 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)
45. Sutton, M.: Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley Professional (jul 2007)
46. Team, G.: Gecode: Generic constraint development environment (2008), https://www.gecode.org/
47. coq development team, T.: The coq proof assistant (2019), https://coq.inria.fr/
48. 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)