Proceedings Paper:
Akgun, Ozgur, Gent, Ian Philip, Jefferson, Christopher Anthony et al. (2 more authors) (2018) Metamorphic testing of constraint solvers. In: Hooker, John, (ed.) Principles and Practice of Constraint Programming:24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings. Lecture Notes in Computer Science . Springer , Netherlands , pp. 727-736.

https://doi.org/10.1007/978-3-319-98334-9_46

Metamorphic Testing of Constraint Solvers

Özgür Akgün, Ian P. Gent, Christopher Jefferson,
Ian Miguel, Peter Nightingale

School of Computer Science, University of St Andrews, St Andrews, UK
{ozgur.akgun, ian.gent, caj21, ijm, pwn1}@st-andrews.ac.uk

Abstract. Constraint solvers are complex pieces of software and are notoriously difficult to debug. In large part this is due to the difficulty of pinpointing the source of an error in the vast searches these solvers perform, since the effect of an error may only come to light long after the error is made. In addition, an error does not necessarily lead to the wrong result, further complicating the debugging process. A major source of errors in a constraint solver is the complex constraint propagation algorithms that provide the inference that controls and directs the search. In this paper we show that metamorphic testing is a principled way to test constraint solvers by comparing two different implementations of the same constraint. Specifically, specialised propagators for the constraint are tested against the general purpose table constraint propagator. We report on metamorphic testing of the constraint solver Minion. We demonstrate that the metamorphic testing method is very effective for finding artificial bugs introduced by random code mutation.

1 Introduction

Metamorphic testing [24] involves generating new test cases from existing ones, where the expected result of a new test case can be generated from the result of an existing test via a *metamorphic relation*. By comparing the results of the original test with the new one we can identify cases where the metamorphic relations are broken, indicating the presence of errors in the implementation. As an illustrative example in the context of a constraint satisfaction problem (CSP), given any unsolvable CSP, adding a constraint or removing domain values from a variable should result in another unsolvable CSP.[1]

Metamorphic testing cannot be used completely in isolation — a solver that immediately returns that a problem is unsolvable would pass all of the metamorphic tests given in this paper. However, the big advantage of metamorphic testing of a constraint solver is that it transforms the relatively hard problem of validating the behaviour of a constraint solver against an independent oracle into comparing a solver against its own behaviour on a different problem.

One major area of constraints research is the creation of new propagation algorithms for constraints. Here metamorphic testing shines — we can check

[1] The authors have experienced both of these conditions being violated in both their own, and other, solvers.

the correctness and propagation level of a new propagation algorithm for a constraint by comparing it with a previously existing algorithm. Assuming the two algorithms do not contain exactly the same bug (which is unlikely, if the two algorithms were independently designed and implemented), comparing one against the other provides us with a high level of confidence that both are correct.

The central argument of this paper is that metamorphic testing is a good fit for constraint solving. While it can be difficult to solve a constraint problem, it is easy to generate one randomly, and there are simple transformations that can take one constraint problem and produce another with the same set of solutions.

2 The Constraint Solver Minion

Throughout this paper, we use the Minion constraint solver to illustrate metamorphic testing in constraint solvers. We begin with an overview of the most important features of Minion's input language, for a more extensive discussion of Minion's features see [10].

Variable Types: Minion provides four implementations of integer variables. `BOOL` (Initial domain must be $\{0, 1\}$), `DISCRETE` and `BOUND` (Initial domain must be a range $\{x..y\}$) and `SPARSEBOUND`. `BOUND` and `SPARSEBOUND` only support changing the bounds of a variable during search.

Constraints: The current release of Minion features 72 constraints, including arithmetic and logical constraints, element [8], alldiff [11] and gcc [19]. Constraints can also be combined, either disjunctively or conjunctively [14].

Reification : Every constraint in Minion can be reified or reifyimplied [14] (also called half reification [6]): these combine a constraint C and a Boolean b into the new constraint $b \iff C$ and $b \implies C$ respectively.

Search Orders: Minion provides ten variable orderings, including static orderings, smallest domain first, conflict ordering [18] and weighted degree [2].

Global Propagation Level: Minion can perform higher levels of consistency, including Singleton Arc Consistency [5].

Some of Minion's features make testing particularly challenging. Several constraints have different implementations for the different types of variables. Also, Minion allows propagators to dynamically change the set of variables whose changes they are informed of. Further, these changes can either automatically revert when search backtracks, or remain in their new location [8]. This means testing must verify how a propagator behaves over an entire search.

3 Instance-based Testing in Minion

The original method of testing Minion involved a set of small test instances, for which the following information was derived by hand:

`SOLCOUNT n` The number of solutions to the problem is `n`.

`NODECOUNT n` The number of search nodes until the first solution under MINION's default search strategy is `n`.

`CHECKONESOL sol` The first solution with MINION's default search strategy is `sol` (given as a list).

`SOLCOUNT` tests are used to test variable and value ordering heuristics, while `NODECOUNT` is used to check that a constraint propagator achieves an expected level of consistency. At the time of writing, this test suite includes 312 instances. New tests are added whenever a bug is discovered in MINION.

For illustration, the bugs found in the inequality constraint ($\sum c_i \times x_i \leq y$ for constants c_i and variables x_i and y) before metamorphic testing was introduced include: Summing a list of Boolean variables to ≥ -1; summing variables to greater than 65,536 (which overflowed the `unsigned short` type in C); crashing if any $c_i = 0$; and sums where all variables were assigned at the root node.

This test-based approach is effective at preventing the reintroduction of previously identified bugs. However, we still found users discovering a large number of errors. This motivates the adoption of the more proactive metamorphic approach. Instance-based testing remains in use to complement the metamorphic tester described in the following section, and remains the primary means of testing variable and value ordering heuristics.

4 Metamorphic Testing in MINION

In this section we explain how MINION uses metamorphic testing for propagators. Given a constraint propagator to test, testing begins by generating a problem instance consisting of a single occurrence of that constraint with a random domain for each variable in its scope. As an example, consider `difference(x,y,z)`, which implements the constraint $|x - y| = z$. The leftmost instance in Fig. 1 shows a corresponding instance. The tester adds several random additional constraints, as presented in the middle instance of Fig. 1. These extra constraints are necessary because many bugs only occur when multiple propagators interact. Optionally, the tester may at this stage create an optimisation problem by `MAXIMISING` or `MINIMISING` a randomly chosen variable. Finally, the tester transforms this instance into another instance with the same set of solutions by replacing the constraint being tested with an equivalent `table` constraint, as presented in the rightmost instance of Fig. 1. The tester then compares the searches of the middle and rightmost instances.

We test the reified version of each constraint propagator separately. The production of metamorphic instances proceeds similarly, but differs in the construction of the table constraint. The scope of the table constraint includes that of the reified constraint, and the reification variable.

The rationale for using a table constraint is that it can represent any other constraint and many propagators for the table constraint achieve **GAC** [15]. Any errors in MINION's implementation of the table constraint propagator would likely be found while testing it against all other constraints. Also, MINION includes several implementations of table constraints [15,20], which are compared.

```
MINION 3
**VARIABLES**
DISCRETE x1 {-3..0}
DISCRETE x2 {0..4}
DISCRETE x3 {0..1}
**CONSTRAINTS**
difference(x1,x2,x3)
**EOF**
```

```
MINION 3
**VARIABLES**
DISCRETE x1 {-3..0}
DISCRETE x2 {0..4}
DISCRETE x3 {0..1}
**CONSTRAINTS**
difference(x1,x2,x3)
diseq(x1,x3)
eq(x2,x3)
**EOF**
```

```
MINION 3
**VARIABLES**
DISCRETE x1 {-3..1}
DISCRETE x2 {-0..4}
DISCRETE x3 {0..1}
**TUPLELIST**
tab 3 3
-1 0 1 0 0 0 0 1 1
**CONSTRAINTS**
table([x1,x2,x3],tab)
diseq(x1,x3)
eq(x2,x3)
**EOF**
```

Fig. 1: Stages of production of metamorphic instances.

The testing process is automated in Python. For each propagator, there is a Python function that, given a list of domains for each variable, returns the list of allowed tuples. MINION is run on the original and transformed instances with the default static variable ordering. The search trees of both instances are compared according to the metamorphic relations described in the following section.

5 Tree Comparison

The tester compares the search trees explored by MINION in solving two different instances, which differ only in the expression of the constraint being tested. If the tested propagator achieves GAC, MINION will explore identical search trees for both instances. If the propagator achieves less than GAC then the search tree may include extra nodes and values but will still find the same set of solutions.

We begin with a formal definition of a two-way branching search tree, which is commonly employed in modern constraint solvers, including MINION:

Definition 1. *Consider a CSP with a list of variables V and a function D which gives the initial domain of every variable. A search node contains a function N which maps each $v \in V$ to a subset of $D(v)$ and one of the following labels:*

Fail *: A **fail** node has no children and represents no solutions. In this case $\forall v \in V.|N(v)| = 0$ (some solvers do not empty all domains at a failure node. We require this to make node comparison easier)*

Solution *: In a **solution** node, $|N(v)| = 1$ for all $v \in V$. N then represents a single assignment to every variable, which is a solution to the CSP.*

Branch *: In a **branch** node, there is a branching literal $\langle v, x \rangle$, where $v \in V$ and $x \in D(v)$ and two child nodes, where in the left v is assigned x, and in the right x is removed from the domain of v. $|D(v)|$ must be > 1.*

This definition is purposely loose — we could easily place (and check) extra requirements on search trees, in particular between a search node and its children in the case of branching nodes but this has not appeared necessary thus far.

When the tester compares two propagators which achieves GAC, comparing search trees is easy – check the search trees are identical, including equality of the variable domains at each node. For weaker propagators we need a more complex method of comparing trees, which is given in Definition 2.

Definition 2. *Consider a set of variables V with initial domains D and search nodes N_1, N_2 from two search trees of CSPs with variables V with initial domains D (but possibly different constraints). The tree rooted on N_1 is a subtree of the tree rooted on N_2 if $\forall v \in V.N_1(v) \subseteq N_2(v)$ and the following are true:*

- *If N_2 is a **solution** node, so is N_1, similarly if N_2 is a **fail** node, so is N_1.*
- *If N_2 is a **branch** node with branch literal $l = \langle v, x \rangle$, then one of the following cases is true:*
  - *N_1 is also a **branch** node, N_1 also branches on l, and N_1's left child is a subtree of N_2's left child (and similarly for right child).*
  - *$x \notin N_1(v)$, there are no solutions in the subtree under N_2's left child and N_2's right child is a subtree of N_1.*

The most complex part of Definition 2 is the final branching case. In this case, we branch on a literal in N_2 that does not occur in N_1. All we know about the left branch of this node is that it will contain no solutions. We will show later this is equivalent to being a supertree of a fail node.

We also need to define propagators, and how they change search states. The property the tester uses to compare search trees is given in Lemma 1 below. Intuitively, Lemma 1 shows that if we run propagators on a search state, then either adding more literals to the search state, or replacing one or more propagators with weaker propagators, never leads propagation removing more literals. This lemma uses the fact that a GAC propagator removes every literal which can correctly be removed and that propagators are inflationary (they never add literals to the search state). This lemma applies to a wide range of propagators, including non-monotonic propagators or propagators which use randomised algorithms[23]. This paper does not contain a full discussion of propagators: for a general overview see [1,22].

Lemma 1. *Consider two search states N and M on the same set of variables V, where $\forall v \in V.N(v) \subseteq M(v)$ (from here denoted by $N \subseteq_V M$), and two lists of inflationary propagators $P = \langle p_1, \ldots, p_n \rangle$ and $Q = \langle q_1, \ldots, q_n \rangle$ where for all i, p_i and q_i are both propagators for a constraint c_i and the p_i achieves GAC. If N_P is a result of applying elements of P to N until a fixed point is reached, and similarly for M_Q, then $N_P \subseteq_V M_Q$.*

Proof. We proceed by induction. Consider a search state S where $N_P \subseteq_V S \subseteq_V M$, and a propagator $q_i \in Q$. Then the result S_Q of applying q_i to S must be contained in M as q_i is inflationary, and must satisfy $N_P \subseteq_V S_Q$, because $N_P \subseteq S$, and N_P is a fixed point for P and therefore all of the p_i.

Given Lemma 1, we can now prove our main result, which is that the definition of subtrees given in Definition 2 is correct, and the tester uses this in metamorphic testing to check the correctness of propagators.

Lemma 2. *Consider a set of variables V with initial domains D, and two lists of propagators $P = \langle p_1, \ldots, p_n \rangle$ and $Q = \langle q_1, \ldots, q_n \rangle$, where for all i p_i and q_i are both propagators for a constraint c_i, and the p_i achieves GAC. Given a static variable and value ordering, a search tree generated by P is a subtree of a search tree generated by Q.*

Proof. We will proceed inductively. To begin, apply P and Q to D to get the root nodes of the search states. By Lemma 1 we know that $\forall v \in V.P(D) \subseteq Q(D)$, and as correct propagators never remove solutions, the set of assignments to $P(D)$ and $Q(D)$ which are solutions will be the same. Now consider any pair of search states N_P and N_Q where $\forall v \in V.N_P(v) \subseteq N_Q(v)$ and the assignments to N_P and N_Q which are solutions are the same. We then continue our induction by considering the possible types of N_Q.

If N_Q is a **branch** node with branching literal $l = \langle v, x \rangle$, there are two possibilities: either $x \in N_P(v)$ or $x \notin N_P(v)$. If $x \notin N_P(v)$, then the left child of N_Q is created from N_Q by reducing the domain of v to $\{x\}$ and then applying Q. None of the assignments to this search state will be solutions, as $x \notin N_P(v)$ and the set of assignments to N_P and N_Q which are solutions are the same. Therefore this child and all of its children are not **solution** nodes.

If $x \in N_P(v)$, then the branching literal we pick for N_P must also be l, as the value and variable order is static. The left child of N_P is created by reducing the domain of v to $\{x\}$ and then running P to a fixed point. The left child of N_Q is created similarly, and by Lemma 1 these children will satisfy our inductive hypothesis. Similarly, the right children also satisfy the inductive hypothesis.

If N_Q is a **fail** node, then as $\forall v \in V.N_P(v) \subseteq N_Q(v)$, N_P is also a **fail** node. Similarly, if N_Q is a **solution** node, then as the assignments to N_Q and N_P which are solutions are the same, then N_P must also be a **solution** node.

6 Practical Experience

Metamorphic testing was introduced into Minion in 2007. It is impossible to measure reliably the number of bugs it has discovered over time, as it has been used during the development of every constraint [13,14,20,15,19,11,12,9] added to Minion since 2007 and many bugs will be found at this stage. We can say no bug has been reported in any propagator which was published after development in Minion, except for one which was not added to the metamorphic tester.

6.1 Mutation Testing

In order to test the robustness of metamorphic testing, we performed mutation testing [4,16]. We tested three constraints: strict (`lexless`) and non-strict (`lexleq`) lexicographic ordering, `difference`, and the reified and reifyimplied variants of these three constraints. `difference` is an example of a propagator with complex arithmetic, while `lexless` and `lexleq` are global constraints. These constraints were still simple enough to check which mutations actually introduced bugs. We ran the tester for the `difference` constraints for 1000 tests,

and the `lex` constraints for 100 tests (the metamorphic tester tests global constraints less as tests take much longer to run). We ran the tester 10 times on each mutation. After filtering out non-compiling mutations, for each constraint we have 20 mutations generated by changing a random Boolean operator (an ROR mutation [17]) and 10 mutations generated by removing a random line of code (an SDL mutation [17]).

The tester found each bug in at least one out of the ten test runs. The non-buggy mutations fall into three groups. Five mutations introduced inefficiencies without changing propagation. Three mutations changed the propagation level of a non-GAC propagator, which is not currently be detected. Finally, one mutation improved performance by removing unnecessary code!

Three mutations were only detected in some runs of the tester. Two *lex* bugs passed more than half of the time, each passing six out of ten test runs. These bugs only affected the first invocation of the constraint, and required several variables already to be assigned by other constraints. This demonstrates the importance of introducing extra constraints to the models. One *difference* instance passed eight out of ten times. This bug introduced a problem in reifying the constraint, by acting as if the domain of the first variable had no holes in it. This problem only very occasionally resulted in an incorrect solution.

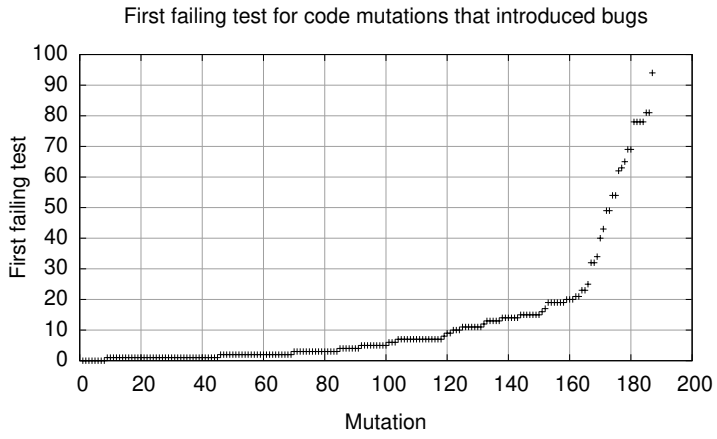

Fig. 2: Number of tests before detecting a bug introduced by a code mutation

Figure 2 shows the median number of tests that were needed before detecting a bug that was introduced by a source code mutation, sorted by first failing test. In many cases bugs are detected with less than 25 test instances.

6.2 Limitations of Metamorphic Testing

There have been three wrong-answer bugs found in released versions of Minion since the introduction of metamorphic testing. One involved a constraint which

was not added to the metamorphic tester. The other two involved large domains – the first involved domain values larger than 2^{30}, the second searches with more than 2^{31} search nodes. Another source of occasional bugs has been bad behaviour on inputs which were supposed to be rejected – the tester only tests valid constraint problems.

6.3 Related Work

Many large A.I. systems have developed similar test frameworks, where many problems, sometimes randomly generated, are tested for correctness. We discuss a few of the most relevant here.

Testing of the Gecode solver [7] has evolved similarly to MINION's. Gecode has an extensive list of fixed tests and a tester which uses random problems. Gecode's random tests creating a random search state which contains a known solution. Literals not in the known solution are removed one by one and the propagator is run, checking no literals from the known solution are ever removed. Other properties of propagators (such as if reaching a fixed point) are also checked. This approach has some limitations compared to MINION, such as not testing multi-solution problems, optimisation problems, or if the propagator works when backtracking. Also, search-tree comparison metamorphic testing should be easier to add to other solvers, as it only requires solvers to output their search tree.

Brummayer and Biere [3] generate random inputs for SMT solvers. Tthey compare the results of multiple solvers looking for inconsistencies. This was very successful, but not useful for solvers like Minion with a unique input format. Also, their technique could not be used to detect the level of propagation achieved.

Reger, Suda and Voronkov discuss the testing of the Vampire theorem prover [21]. They use a fixed set of benchmarks, testing the solver's many configuration options against each other, and also validating the generated proof.

7 Conclusions

Overall, metamorphic testing has been a great success for MINION. It is impossible to measure the number of bugs it has discovered, as it is used by developers when creating new propagators, when large numbers of bugs are found and fixed during development.

While metamorphic testing has been very useful in MINION, it is not immediately applicable to all solvers. One major limitation is learning solvers, where new constraints are added during search. Significant metamorphic tests could still be performed in such solvers by checking that the solver produces the same set of solutions, and optimisation problems achieve the same optimal solution. It would be interesting to investigate if more subtle forms of metamorphic testing would be beneficial in such cases.

Acknowledgements We thank EPSRC for funding this work via the grants EP/P015638/1 and EP/P026842/1. Dr Jefferson holds a Royal Society University Research Fellowship.

References

1. Apt, K.: Principles of Constraint Programming. Cambridge University Press, New York, NY, USA (2003)
2. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: ECAI 04. pp. 482–486 (2004)
3. Brummayer, R., Biere, A.: Fuzzing and delta-debugging SMT solvers. In: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories. pp. 1–5. SMT '09, ACM, New York, NY, USA (2009)
4. Budd, T.A.: Mutation Analysis of Program Test Data. Ph.D. thesis, New Haven, CT, USA (1980)
5. Debruyne, R., Bessière, C.: Some practicable filtering techniques for the constraint satisfaction problem. In: Proc. 15th International Joint Conference on Artificial Intelligence (IJCAI 97). pp. 412–417 (1997)
6. Feydy, T., Somogyi, Z., Stuckey, P.J.: Proceedings of the 17th International Conference Principles and Practice of Constraint Programming, chap. Half Reification and Flattening, pp. 286–301. Springer, Berlin, Heidelberg (2011)
7. Gecode Team: Gecode: Generic constraint development environment (2006), available from `http://www.gecode.org`
8. Gent, I., Jefferson, C., Miguel, I.: Watched literals for constraint propagation in minion. In: Proceedings of the Twelfth International Conference on Principles and Practice of Constraint Programming (CP 2006). vol. 4204, pp. 182–197. Springer Berlin / Heidelberg (2006)
9. Gent, I.P., Jefferson, C., Linton, S., Miguel, I., Nightingale, P.: Generating custom propagators for arbitrary constraints. Artificial Intelligence 211, 1 – 33 (2014), `http://www.sciencedirect.com/science/article/pii/S000437021400023X`
10. Gent, I.P., Jefferson, C., Miguel, I.: Minion: A fast scalable constraint solver. In: ECAI. vol. 141, pp. 98–102 (2006)
11. Gent, I.P., Miguel, I., Nightingale, P.: Generalised arc consistency for the alldifferent constraint: An empirical survey. Artificial Intelligence 172(18), 1973–2000 (2008)
12. Gent, I., Jefferson, C., Miguel, I., Nightingale, P.: Data structures for generalised arc consistency for extensional constraints. AAAI (CPPOD-19-2006-A), 191–197 (2007), `http://www.aaai.org/Papers/AAAI/2007/AAAI07-029.pdf`
13. Jefferson, C., Kadioglu, S., Petrie, K.E., Sellmann, M., Živný, S.: Same-relation constraints. In: Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP'09). No. 5732 in LNCS (2009), `http://zivny.cz/publications/jkpsz09cp-preprint.pdf`
14. Jefferson, C., Moore, N.C.A., Nightingale, P., Petrie, K.E.: Implementing logical connectives in constraint programming. Artificial Intelligence 174(16-17), 1407–1429 (11 2010)
15. Jefferson, C., Nightingale, P.: Extending simple tabular reduction with short supports. In: IJCAI (2013)
16. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering 37(5), 649–678 (Sept 2011)
17. King, K.N., Offutt, A.J.: A fortran language system for mutation-based software testing. Software: Practice and Experience 21(7), 685–718, `https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380210704`
18. Lecoutre, C., Saïs, L., Tabary, S., Vidal, V.: Reasoning from last conflict(s) in constraint programming. Artificial Intelligence 173(18), 1592 – 1614 (2009), `http://www.sciencedirect.com/science/article/pii/S0004370209001040`

19. Nightingale, P.: The extended global cardinality constraint: An empirical survey. Artificial Intelligence 175(2), 586–614 (2011)
20. Nightingale, P., Gent, I.P., Jefferson, C., Miguel, I.: Short and long supports for constraint propagation. J. Artif. Int. Res. 46(1), 1–45 (Jan 2013), http://dl.acm.org/citation.cfm?id=2512538.2512539
21. Reger, G., Suda, M., Voronkov, A.: Testing a saturation-based theorem prover: Experiences and challenges. In: Gabmeyer, S., Johnsen, E.B. (eds.) Tests and Proofs. pp. 152–161. Springer International Publishing, Cham (2017)
22. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming. Elsevier (2006)
23. Schulte, C., Tack, G.: Weakly monotonic propagators. In: Gent, I. (ed.) Fifteenth International Conference on Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, vol. 5732, pp. 723–730. Springer-Verlag (2009), http://www.gecode.org/paper.html?id=SchulteTack:CP:2009
24. T. Y. Chen, S. C. Cheung, S.M.Y.: Metamorphic testing: a new approach for generating next test cases. Tech. Rep. Rep. HKUST-CS98-01 (1998)