Breaking Constraint Modelling Languages with Metamorphic Testing (extended abstract)

Wout Vanroose
DTAI, KU Leuven, Belgium

Ignace Bleukx
DTAI, KU Leuven, Belgium

Jo Devriendt
DTAI, KU Leuven, Belgium

Dimos Tsouros
DTAI, KU Leuven, Belgium

Hélène Verhaeghe
DTAI, KU Leuven, Belgium

Tias Guns
DTAI - Data Analytics Lab, KU Leuven - VUB, Belgium

Abstract

Constraint-solving techniques are used in a range of high-stakes applications ranging from scheduling production lines [9] to automated verification of computer programs [10]. These applications require constraint solvers to provide correct and reliable solutions to the constraint specifications. However, modern solvers and modelling languages are complex pieces of software, which are inevitably prone to bugs. Bugs in a solver or modelling language can cause a range of undesired behaviour: from crashes of the program to returning an invalid solution to the constraints, with a potentially major impact on the application at hand.

To mitigate the number of bugs in computer programs, it is good practice to use some kind of *automated testing* during software development. Unit testing is such a technique to test isolated parts of the code. A unit test consists of a small use case of the code and is typically written by the developers of the software. While unit testing is very useful to verify the intended behaviour of a program, it is time-consuming for developers [3]. Moreover, tricky edge cases may be overlooked by developers when designing the test suite.

Fuzz testing is an aggregate of several techniques which test computer programs on random inputs. These techniques can either be generation-based or mutation-based. The former generates input for the program from scratch, while the latter uses given inputs and applies mutations on them to generate a new input. In the field of Constraint Programming (CP), generation-based fuzz-testing has already been adopted as an automatic testing technique for solvers. For example, the propagation algorithms of the MINION solver have been automatically tested throughout the solvers' development [2]. The input used for testing such propagation algorithms is a randomly generated CSP instance within the grammar supported by the solver. The output of the solver is verified using other equivalent algorithms.

Mutation-based fuzz testing has been applied to test SAT Modulo Theory (SMT) solvers in the form of *metamorphic testing* [11, 15]. Given a satisfiable set of expressions, the goal is to combine these using a *solution preserving mutation*. The combined expressions are then used as input to the solver. Depending on the type of expressions and mutations, this technique can result in deeply nested expressions. SMT solvers accept such nested structures by default, and can hence be thoroughly tested using such a technique.

While CP-solvers do not expect nested expressions as input, constraint modelling languages do. These languages automatically flatten, rewrite and decompose nested expressions into individual constraints accepted by the solver as input. Therefore, constraint modelling languages are an essential tool to make effective use of constraint solvers. The CP community has developed many such languages. Popular examples are the text-based MiniZinc [12] language, the XML-based language XCSP [14], the Essence system [1] and the Python-based language CPMpy [6]. To the

best of our knowledge, the metamorphic testing framework has not been employed to test the logic of any of these modelling languages.

Hence, we propose a method to fuzz-test constraint modelling languages using mutational testing. We take inspiration from the work on SMT testing, as well as taking the specifics of constraint solvers into account: namely the use of global constraints, optional objective functions and integer decision variables/expressions.

The very simple and general approach to using metamorphic testing for modelling languages can be summarized by the following algorithm:

Algorithm 1 TestMetamorphic

Input: Input constraints $\mathcal{C}$, set of solution preserving mutations $\mathcal{M}$
while $sat(\mathcal{C})$ **do**
  $M \leftarrow$ pick a mutation from $\mathcal{M}$
  $E \leftarrow$ pick n (sub)expressions from $\mathcal{C}$
  $\mathcal{C} \leftarrow \mathcal{C} \cup (M(E))$

To investigate the use of metamorphic testing, we apply our method to the CPMpy constraint modelling language. The system is able to translate high-level user constraints to solvers belonging to several constraint-solving paradigms. These include CP (OR-Tools [13]), MIP (Gurobi [7]), SMT (Z3 [5]), SAT (PySAT [8]), knowledge compilers (PySDD [4]) and even to MiniZinc [12]. Thereby supporting any solver supported by MiniZinc. CPMpy allows a user to arbitrarily nest Boolean and numerical expressions, making it perfect to use in a metamorphic testing framework. To transform the high-level user specification to solver-specific constraints, CPMpy uses a set of *mutation functions*. This is one of the uses of mutations we wish to test.

In this talk, we discuss several types of metamorphic mutations that are applicable to constraint models, including logical mutations, semantic fusion, and solution-preserving mutations that are inherent to modelling language rewrite systems.

While the work is still ongoing, we will present some initial results, and discuss follow-up issues such as how to not overwhelm the developer with instances causing a bug.

▶ **Example 1.** Finishing this abstract, we present one interesting bug found during initial experimentation. It involves a mutation which uses the "XOR" ($\oplus$) operator to combine two constraints c_1 and c_2 in the following way:

1. $\neg(c_1 \oplus c_2)$
2. $\neg c_1 \oplus c_2$
3. $c_1 \oplus \neg c_2$
4. $\neg(\neg c_1 \oplus \neg c_2)$

By construction, adding any of these constraints to the constraint set $\mathcal{C}$ will not change the set of solutions as they are implied by $c_1 \wedge c_2$. To discover the bug, our method sampled a starting model with one constraint: $\mathcal{C} = \{Circuit(x)\}$ with x a list of variables, and applied these logical mutators:

1. Negation morph: $\mathcal{C} = \{Circuit(x), \neg\neg Circuit(x)\}$
2. Xor morph: $\mathcal{C} = \{Circuit(x), \neg\neg Circuit(x), \neg Circuit(x) \oplus \neg\neg Circuit(x)), \dots\}$

After solving the transformed model, no solution was returned. Because the mutations should be solution-preserving, we know there is some underlying bug. After debugging, we noticed reified *Circuit* constraints were incorrectly handled, due to new variables being introduced in the decomposition.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases Constraint Solving, Automated testing, Modelling Languages

Funding This research received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (Grant No. 101002802, CHAT-Opt).

References

1 Özgür Akgün, Alan M. Frisch, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Conjure: Automatic generation of constraint models from problem specifications. *Artif. Intell.*, 310:103751, 2022. doi:10.1016/j.artint.2022.103751.

2 Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Metamorphic testing of constraint solvers. In John N. Hooker, editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 727–736. Springer, 2018. doi:10.1007/978-3-319-98334-9_46.

3 Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014*, pages 201–211. IEEE Computer Society, 2014. doi:10.1109/ISSRE.2014.11.

4 Adnan Darwiche, Pierre Marquis, Dan Suciu, and Stefan Szeider. Wannes meert, pysdd. In *Recent trends in knowledge compilation (dagstuhl seminar 17381)*, volume 7. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

5 Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

6 Tias Guns. Increasing modeling language convenience with a universal n-dimensional array, cppy as python-embedded example. In *Proceedings of the 18th workshop on Constraint Modelling and Reformulation at CP (Modref 2019)*, volume 19, 2019.

7 Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL: https://www.gurobi.com.

8 Alexey Ignatiev, António Morgado, and João Marques-Silva. Pysat: A python toolkit for prototyping with SAT oracles. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10929 of *Lecture Notes in Computer Science*, pages 428–437. Springer, 2018. doi:10.1007/978-3-319-94144-8_26.

9 Ahmet B. Keha, Ketan Khowala, and John W. Fowler. Mixed integer programming formulations for single machine scheduling problems. *Comput. Ind. Eng.*, 56(1):357–367, 2009. doi:10.1016/j.cie.2008.06.008.

10 Shuvendu K. Lahiri and Shaz Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 171–182. ACM, 2008. doi:10.1145/1328438.1328461.

11 Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 701–712. ACM, 2020. doi:10.1145/3368089.3409763.

12 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume

4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007. doi:10.1007/978-3-540-74970-7_38.

13 Laurent Perron and Vincent Furnon. Or-tools. URL: https://developers.google.com/optimization/.

14 Olivier Roussel and Christophe Lecoutre. XML representation of constraint networks: Format XCSP 2.1. *CoRR*, abs/0902.2362, 2009. URL: http://arxiv.org/abs/0902.2362, arXiv:0902.2362.

15 Dominik Winterer, Chengyu Zhang, and Zhendong Su. Validating SMT solvers via semantic fusion. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 718–730. ACM, 2020. doi:10.1145/3385412.3385985.