Declarative pearl: I put a SAT solver in your SAT solver so you can find backdoors slowly

Martin Mariusz Lester @ ORCID
Department of Computer Science, University of Reading, United Kingdom

Abstract

The classic algorithm for SAT solving, DPLL, runs in exponential time. The algorithm is entirely deterministic, except for choices about which variable to branch on and whether to try true or false first. This choice can be made deterministically using heuristics, but often randomness is employed. For the algorithm to find a solution in reasonable time, it must make only a small number of choices, although (thanks to backtracking) some of these may be wrong. By modelling the choices as nondeterminism, we obtain a declarative C SAT solver, executable using a SAT solver.

The declarative C SAT solver works using a bounded model checker to encode the problem of solving a SAT instance as a larger SAT instance, then using an existing SAT solver to resolve the nondeterminism. This is much less efficient than applying the existing SAT solver directly to the SAT instance of interest. However, by reframing SAT solving as an optimisation problem of minimising the amount of nondeterminism required, we can get an empirical measure of the hardness of an easy SAT instance, called a backdoor, which is more informative than syntactic measures such as number of variables or clauses.

2012 ACM Subject Classification Software and its engineering → Software verification; Software and its engineering → Constraint and logic languages

Keywords and phrases SAT solving, declarative C, bounded model-checking

1 Pure Nondeterminism

NP problems, such as SAT, are characterised by the existence of polynomial-time verifiable certificates for their solutions. In the case of SAT, the certificate is an assignment of true/false to Boolean variables in the SAT instance. *Bounded model-checking* encodes a run of a potentially nondeterministic imperative program into a SAT instance, where the certificate gives a path through the program to a point of interest (such as assertion violation). The *declarative C* paradigm [7, 8] exploits this equivalence to repurpose C as a declarative language for constraint programming.

It is straightforward to encode the problem of solving a SAT instance as a declarative C program. For example, we can encode the CNF $(x_1 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee \neg x_1)$ as follows:

Listing 1 A SAT instance in declarative C.

```
int main() {
    uint8_t x1, x2, x3;
    __VERIFIER_assume(x1 <= 1 && x2 <= 1 && x3 <= 1);
    __VERIFIER_assume(x1 || !x3);
    __VERIFIER_assume(x2 || x3 || !x1);
    __VERIFIER_error();
}
```

If we pass the program to a bounded model-checker such as *CBMC* [3], it will encode the problem of finding an execution that leads to the error as a larger SAT instance, then solve it with a SAT solver. It will translate the solution back into a counterexample trace, yielding a solution to the original SAT instance.

While it is amusing to solve a small SAT instance by turning it into a large SAT instance and applying an existing SAT solver, we gain no real insight by doing so. The "circuit"

encoded by the large SAT instance includes the same inputs as the original SAT instance, but bloated with auxiliary circuitry to encode the run of the C program. Nonetheless, this is exactly how the XCSP3 solver *Exchequer* [6] behaves when presented with a SAT instance encoded as an XCSP3 problem. While the performance is bad compared with applying the SAT solver directly to the original small SAT instance, it may still beat other constraint-solving approaches [1].

2 Nondeterministic DPLL

The classic SAT-solving algorithm is a variant of *DPLL* [2], which combines pure literal elimination, unit propagation and backtracking search. The backtracking search is invoked only when pure literal elimination and unit propagation can make no further progress. It picks a variable and guesses whether it should be true or false; if it is wrong, it tries the other option on backtracking. With good heuristics for picking a variable and guessing its value, DPLL can solve many SAT instances quickly.

What happens if we interpret a DPLL implementation as a declarative C program? Rather than picking a variable using heuristics or randomness, we can make the choice nondeterministic. There is no longer any need for backtracking in the C program, as the SAT solver will choose the correct value if one exists. However, to make the loop unrolling performed by the bounded model-checker tractable, we do need to bound the number of iterations of the algorithm.

As in the purely nondeterministic case, the resulting SAT instance is larger, and solving it with a SAT solver is less efficient than applying a solver directly to the original SAT instance. However, now the true nondeterminism in the larger instance is limited to just variable/value choice in the (no longer backtracking) search; the rest of the instance is quickly forced to consistency with the deterministic parts of the C program.

Many SAT instances become “easy”, in the sense that they can be solved quickly using just pure literal elimination and unit propagation, once a small number of variable assignments have been guessed correctly. These assignments constitute a *backdoor* [10, 9, 4]. The size of a SAT instance’s backdoor is a better metric of its hardness than purely syntactic measures, such as number of variables or clauses. Unfortunately, computing it is at least as hard as solving the instance. However, by treating solution of our declarative implementation of DPLL as an optimisation problem, where we aim to minimise the number of nondeterministic choices, we obtain a method of finding SAT backdoors using a SAT solver, albeit an inefficient one.

3 The UNSAT Case

Solutions to NP problems, such as SAT, are characterised by the existence of polynomial-time verifiable certificates. However, for co-NP problems, such as UNSAT, this is not the case (unless NP = co-NP). Yet SAT solvers can also produce certificates for unsatisfiability. What happens if we treat an UNSAT certificate verifier as a declarative program? We obtain a method for finding UNSAT certificates (a co-NP problem) using a SAT solver (which solves NP problems). But under the assumption that NP $\neq$ co-NP, and observing that UNSAT certificates can be extremely large [5], this method will be neither complete nor efficient. Nonetheless, this provides another example of the succinctness and flexibility of the declarative C paradigm.

References

1 Gilles Audemard, Christophe Lecoutre, and Emmanuel Lonca. Proceedings of the 2022 XCSP3 competition. *CoRR*, abs/2209.00917, 2022. URL: https://doi.org/10.48550/arXiv.2209.00917, arXiv:2209.00917, doi:10.48550/ARXIV.2209.00917.

2 Sam Buss and Jakob Nordström. Proof complexity and SAT solving. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 233–350. IOS Press, 2021. URL: https://doi.org/10.3233/FAIA200990, doi:10.3233/FAIA200990.

3 Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004. URL: https://doi.org/10.1007/978-3-540-24730-2_15, doi:10.1007/978-3-540-24730-2_15.

4 Jan Dreier, Sebastian Ordyniak, and Stefan Szeider. SAT backdoors: Depth beats size. *J. Comput. Syst. Sci.*, 142:103520, 2024. URL: https://doi.org/10.1016/j.jcss.2024.103520, doi:10.1016/J.JCSS.2024.103520.

5 Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2008, Fort Lauderdale, Florida, USA, January 2-4, 2008*, 2008. URL: http://isaim2008.unl.edu/PAPERS/TechnicalProgram/ISAIM2008_0008_60a1f9b2fd607a61ec9e0feac3f438f8.pdf.

6 Martin Mariusz Lester. Solving xcsp3 constraint problems using tools from software verification. In *ModRef 2022: The 21st workshop on Constraint Modelling and Reformulation*, 2022. URL: https://modref.github.io/papers/ModRef2022_SolvingXCSP3ConstraintProblemsUsingToolsFromSoftwareVerification.pdf.

7 Martin Mariusz Lester. Coptic: Constraint programming translated into C. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II*, volume 13994 of *Lecture Notes in Computer Science*, pages 173–191. Springer, 2023. URL: https://doi.org/10.1007/978-3-031-30820-8_13, doi:10.1007/978-3-031-30820-8_13.

8 Martin Mariusz Lester. Cutting the cake into crumbs: Verifying envy-free cake-cutting protocols using bounded integer arithmetic. In Martin Gebser and Ilya Sergey, editors, *Practical Aspects of Declarative Languages - 26th International Symposium, PADL 2024, London, UK, January 15-16, 2024, Proceedings*, volume 14512 of *Lecture Notes in Computer Science*, pages 100–115. Springer, 2024. URL: https://doi.org/10.1007/978-3-031-52038-9_7, doi:10.1007/978-3-031-52038-9_7.

9 Stefan Szeider. Backdoor sets for DLL subsolvers. *J. Autom. Reason.*, 35(1-3):73–88, 2005. URL: https://doi.org/10.1007/s10817-005-9007-9, doi:10.1007/S10817-005-9007-9.

10 Ryan Williams, Carla P. Gomes, and Bart Selman. Backdoors to typical case complexity. In Georg Gottlob and Toby Walsh, editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, pages 1173–1178. Morgan Kaufmann, 2003. URL: http://ijcai.org/Proceedings/03/Papers/168.pdf.