Optimizing the Automatic Test Generation by SAT and SMT Solving for Boolean Expressions

Paolo Arcaini*, Angelo Gargantini†, and Elvinia Riccobene*
*Dip. di Tecnologie dell'Informazione - Università degli Studi di Milano, Italy
{paolo.arcaini,elvinia.riccobene}@unimi.it
†Dip. di Ing. dell'Informazione e Metodi Matematici - Università di Bergamo, Italy
angelo.gargantini@unibg.it

***Abstract*—Recent advances in propositional satisfiability (SAT) and Satisfiability Modulo Theories (SMT) solvers are increasingly rendering SAT and SMT-based automatic test generation an attractive alternative to traditional algorithmic test generation methods. The use of SAT/SMT solvers is particularly appealing when testing Boolean expressions: These tools are able to deal with constraints over the models, generate compact test suites, and they support fault-based test generation methods. However, these solvers normally require more time and greater amount of memory than classical test generation algorithms, limiting their applicability. In this paper we propose several ways to optimize the process of test generation and we compare several SAT/SMT solvers and propositional transformation rules. These optimizations promise to make SAT/SMT-based techniques as efficient as standard methods for testing purposes, especially when dealing with Boolean expressions, as proved by our experiments.**

I. Introduction

In the context of model-based test generation, propositional satisfiability (SAT) and Satisfiability Modulo Theories (SMT) solvers are increasingly considered an attractive alternative [9], [2] to traditional algorithmic test generation methods. Besides their capability of dealing with complex constraints, these techniques can achieve the goal of compact test suites without compromising their fault detection capability [8].

Although SAT and SMT solvers are already successfully employed in several projects of software testing and verification, in some areas, like testing of Boolean specifications, they are rarely used. For Boolean expressions, classical testing criteria (like MCDC [3] or MUMCUT [15]) are widely used together with simple yet fast algorithms for test generation. These algorithms, however, do not explicitly consider the expression fault classes and they often also require expressions to be in a particular normal (usually disjunctive) form.

Recent results [9] show how it is possible to reduce the problem of finding fault detecting test cases for Boolean expressions to a logical satisfiability problem, which can be solved by a SAT/SMT-based algorithm. This approach does not require the specifications under test to be expressed in a particular normal form, so avoiding possible overhead due to the formula transformation, generates test cases directly targeting specific fault classes and uses several reduction policies to minimize the size of resulting test suites.

The process of automatic test generation by SAT/SMT techniques requires, however, more time and memory than standard generation algorithms and this fact limits its use in practice. The contribution of this paper is to improve this process by proposing a number of optimizations that promise to make SAT/SMT techniques as efficient as standard methods for test generation purposes with the mentioned benefits. Some optimizations regard the actual use of the tools (e.g. avoiding exchanging files and using native libraries instead). Other optimizations improve the SAT/SMT-based process of automatic test generation, independently from a specific input specification and selected testing criterion. Others are specific to the process instantiated for testing Boolean expressions.

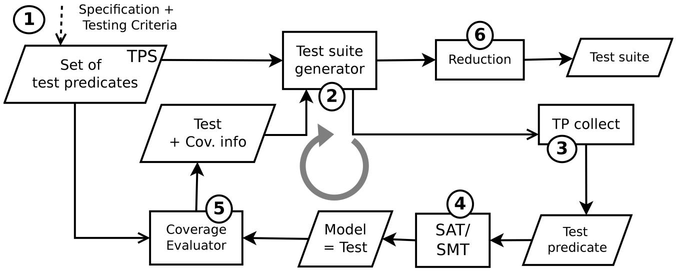

Figure 1. Test generation process

We also propose a comparison of different SAT and SMT solvers usable in the test generation process and that are able to support (not necessarily all) the proposed optimizations. On the base of the best (among those used) tool, we show evidence that the proposed optimizations are effective. And, on the base of our experiments, we conclude that SAT/SMT solvers can be successfully applied to Boolean testing and that SMT-solvers may have better performance than SAT solvers.

II. Test Generation Process

The overall test generation process by SAT/SMT solvers optimized in this paper is presented in [9] and is depicted in Fig. 1. A set TPS of test predicates is generated from a specification (① in Fig. 1) depending on the testing criteria.

The generation of the complete test suite (② in Fig. 1) can be performed by taking a test predicate tp and trying to generate a test that covers it by using a SAT/SMT solver which is able to find a model for tp. A core optimization employed in this paper consists in finding tests that cover as many test predicates as possible instead of single test predicates. We call *collection* the set of test predicates sharing the same model, and *collecting* the process of grouping test predicates. The collecting phase is shown in Fig. 1 as ③ and again uses a SAT/SMT solver. Although the collecting process is able to produce very compact test suites [9], it is very expensive in terms of solver calls, since it requires a call of the solver to check if a test predicate $\mathtt{tp}$ is compatible with Tpc, i.e. if there exits a common model for $\mathtt{tp}$ and Tpc. If compatible, $\mathtt{tp}$ is added to the collection Tpc. If $\mathtt{tp}$ is not compatible, it must be checked if it is satisfiable; otherwise it is removed from TPS in order to avoid trying to collect a test predicate which is actually infeasible. Note that infeasible test predicates consume computing resources without producing usable tests.

Once a collection Tpc is built, the SAT/SMT solver is invoked (④ in Fig. 1) to find a model for Tpc and, therefore, for all the test predicates collected in it. The *coverage evaluation* (⑤ in Fig. 1) gathers information useful to skip the generation for test predicates already covered (also called *monitoring*), while *post reduction* (⑥ in Fig. 1) is the last step of the process, removes unnecessary test predicates (if any), and it can be performed in a negligible amount of time.

We here assume that the specifications under test are Boolean expressions in general form (GF) and we perform the experiments using fault-based testing criteria that explicitly target faults in Boolean expressions (like those in [11]). The test predicates have form $\mathtt{tp}_i = \varphi \oplus \varphi'_i$ (called *detection conditions*), where $\oplus$ denotes the exclusive or (xor), φ is the Boolean specification, and φ'_i are all the possible faulty implementations.

III. Optimizations

O.1 Simplification of the test predicate: Since test predicates have form $\varphi \oplus \varphi'$ and that φ and φ' often have a common subexpression, we can use several equivalences (like $(a \wedge b) \oplus (a \wedge c) \equiv a \wedge (b \oplus c)$) that allow to factor a part of the formula and to push the $\oplus$ operator near the literals.

O.2 Optimizing the transformation to CNF: Almost all SAT solvers require CNF input formulas, while Boolean expressions we consider and their test predicates have *general* form. Efficient transformation to CNF is still a research topic [13], and we have experimented several translations to CNF.

O.3 Avoiding the transformation to clausal form: Converting a non-clausal formula to CNF requires a great effort (it can grow exponentially in length) and it may destroy the initial structure of the formula, which could be used for efficient satisfiability checking. SAT/SMT solvers taking Boolean expressions in general form (GF) may perform better.

O.4 Using the API and avoiding the exchange of files: A simple optimization regards the way the solvers are invoked: Instead of using input files and calling the solvers in a command shell, we can embed the solver in the test generation process itself.

A. Collecting Optimizations

Since collecting is the most powerful technique to generate small test suites, but it is also the most expensive [9], a great effort should be spent to improve this part of the generation process.

O.5 Marking feasible test predicates: A first optimization consists in marking if a test predicate is *feasible*. Feasible test predicates do not need to be checked for feasibility when they are not compatible with the collection.

O.6 Collection with witness: The collecting algorithm returns the collected test predicates and the SAT/SMT solver is called after the collecting process to find a model for the conjunction of the collected test predicates (④ in Fig 1). Since the solver has been already invoked to check if the last test predicate added to the collection is compatible with the other test predicates previously collected, a further call to the SMT solver is useless.

O.7 Checking if the witness is a model: When trying to add the current test predicate $\mathtt{tp}$ to the collection Tpc, one could check if the witness for the collection Tpc is already a model for $\mathtt{tp}$. In this case $\mathtt{tp}$ can be added to Tpc without any further call of the SMT solver.

O.8 Collecting incrementally: Most modern SAT/SMT solvers maintain the logical context of a given problem and allow incremental satisfiability checking. This feature can be exploited when collecting test predicates.

O.9 Collecting incrementally with backtracking: Most SMT solvers, like Yices [6] and Z3 [5], have the further feature of removing an added formula from the current logical context. We can, therefore, incrementally collect all the test predicates having a common model: Before adding a single test predicate $\mathtt{tp}$ to the collection, $\mathtt{tp}$ is added into the context. If the context has still a model, then $\mathtt{tp}$ is added to the collection, otherwise $\mathtt{tp}$ is removed from the context.

O.10 Double incremental collecting: In case all the test predicates have the form $\varphi \oplus \varphi'_i$ (if the $\oplus$ is not pushed), one can use the following *Xor elimination* logical equivalence:

$$\bigwedge_{i=1}^{n} (\varphi \oplus \varphi'_i) \equiv (\varphi \wedge \bigwedge_{i=1}^{n} \neg\varphi'_i) \vee (\neg\varphi \wedge \bigwedge_{i=1}^{n} \varphi'_i)$$

to simplify the collecting process. One starts with two contexts: $\mathtt{c}_\top$ initially containing only φ, and $\mathtt{c}_\bot$ containing $\neg\varphi$. When a test predicate $\mathtt{tp}_i = \varphi \oplus \varphi'_i$ must be checked for compatibility with all the test predicates already collected, $\neg\varphi'_i$ is added to $\mathtt{c}_\top$ (if still valid), while φ'_i is added to $\mathtt{c}_\bot$(if still valid). (1) If both contexts are still satisfiable, then $\mathtt{tp}_i$ is accepted; (2) if only one context is satisfiable, then $\mathtt{tp}_i$ is still accepted but the context without model is invalidated and no longer considered; (3) if no valid context is satisfiable, then $\mathtt{tp}_i$ is refused and the valid contexts are restored.

B. Limiting Collecting

To make the collecting process of test predicates faster, another approach consists in limiting the test predicates that can be possibly collected. In this case, instead of reducing the time necessary to collect every possible test predicate, one could try to limit the number of test predicates that are collected. The collection would not contain all the uncovered test predicates which could be possibly collected together (we can say that it is a *partial* collection), and this may reduce the effectiveness of the collecting process itself. We devise the following policies.

O.11 Quit after N: Quit collecting when a maximum *number* of test predicates are added to the collection. The collecting process is very fast with small N, but it produces bigger test suites. With increasing N, it behaves similarly to the unlimited collection but also the time may increase.

O.12 Collecting until useful: Quit collecting as soon as it becomes useless, i.e. when the collection admits an unique model. We can discover if a model is unique by using the SAT/SMT solver and the following proposition.

Prop. 1: Let ψ be a feasible predicate, m be a model of ψ, and $asExpr(m)$ be the function that returns the conjunction of the variables having value *true* in m and the negation of the variables having value *false* in m. m is the unique model of ψ if $\psi \wedge \neg asExpr(m)$ is not satisfiable.

Note that O.12 collects all the useful test predicates and, therefore, it does not impact over the test suite size.

O.13 Checking uniqueness after N: Add the first N predicates (if possible) in a classic way. After that, check if the model of the collection is unique: If it is, quit collecting.

IV. Experimental Results

For experimentation, we initially consider the same set of Boolean specifications for TCAS introduced by [14]. As SAT solver we select SAT4J [12], MiniSAT [7], PicoSAT [1] and NFLSAT [10]. As SMT solver, we use Yices [6], which includes a very efficient SAT solver; it claims to be "*competitive as an ordinary SAT and MaxSAT solver*" [6]. To implement optimization O.4 we use the solvers (if possible) together with the Java Native Access (JNA) libraries, which simply require native shared libraries. NFLSAT does not require inputs as CNF, but it cannot work with JNA since it comes as executable binary. Yices has a very rich API, accepts GF Boolean expressions and supports a very efficient backtracking technique.

Initial experiments revealed that O.1 always speeds up the generation process and that (O.2) the test generation made with the Tseitin algorithm is faster (by around 75%) than the generation made with an equivalence preserving CNF translation algorithm even if it increases the number of literals. For the rest of reported experimentations, we assume the use of these two optimizations.

A comparison among the solvers using only optimizations O.3 and O.4 whenever possible is reported in Fig. 2. NFLSAT is faster than the others SAT solvers used at command line (CLI): Optimization O.3 has a positive effect: NFLSAT and Yices, which take GF predicates, perform better than the other solvers that accept only CNF. However, NFLSAT is much slower than the other solvers when applying O.4 which is not supported by NFLSAT. All the solvers with JNA perform considerably better than the CLI counterparts: O.4 drastically reduces the time necessary to generate the tests.

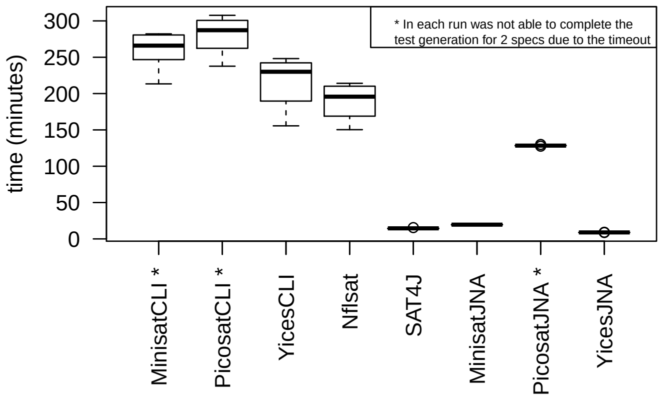

Figure 2. Comparing the solvers using TCAS specs

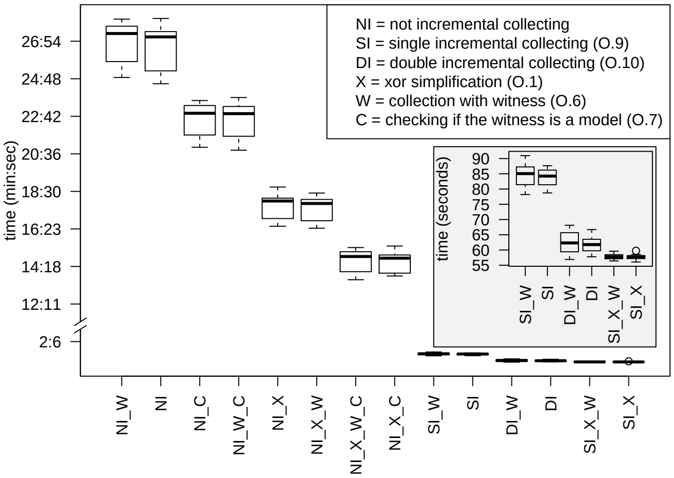

Figure 3. Optimization evaluation with Yices

In order to thoroughly compare the best solvers (SAT4J, Yices and MiniSAT with JNA), we select 12200 random specifications with different number of variables and complexity. We found that Yices performs better than the other two SAT solvers, therefore, SMT solving is competitive even for test generation for Boolean expressions. From here on, we use Yices with JNA.

A. Optimization Evaluation

Fig. 3 reports the time required to complete the test generation for 40 specifications (the 20 TCAS specifications and 20 most complex random specifications) for 50 runs, depending on the optimizations O.1, O.6, O.7, and incremental collecting in three variants: not applied, single incremental collecting (O.9), and double incremental collecting (O.10). The grey box enlarges the cases in which incremental collecting is applied. It is evident that the incremental collecting significantly improves the performances.

We found that the only ineffective optimization is O.6: It seems that an extra call of the SMT solver after the collecting phase does not impact over the final time.

All the other optimizations are effective. The incremental collecting (O.9 and O.10) boosts the performances by significantly reducing the time. The best performances are obtained when the single collecting (O.9) is applied together with O.1.

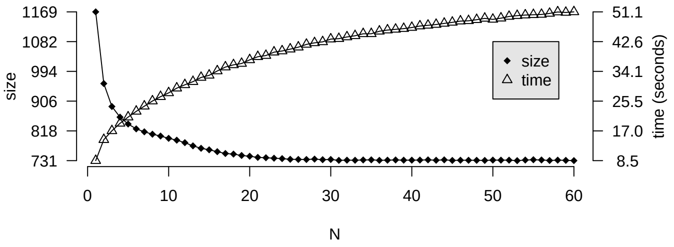

Figure 4. Quit collecting after N (O.11)

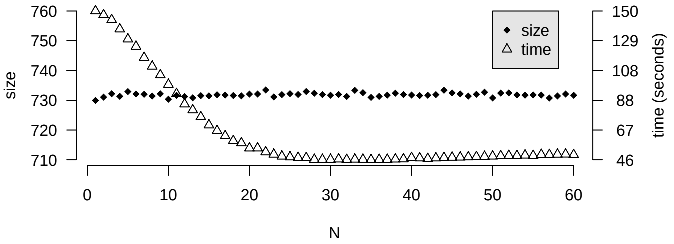

Figure 5. Checking uniqueness after N (O.13)

In this case, the generation for all the test suites for all the 40 specifications requires only 57.6 secs, with an average of 1.44 secs for specification. This proofs that a well engineered and optimized SMT-based test generation process can be used in practice for Boolean specifications instead of the classical algorithms.

B. Limiting Collecting

In this experiment, we test how all the limiting policies introduced in Sect. III-B may affect the test generation process (used together with O.9). Fig. 4 depicts the effects of O.11. As the figure shows and as expected, the size of the test suite decreases with increasing N, but the time required increases as well. For small N the size rapidly decreases, but after a threshold (around 15) the test suite size is reduced only marginally. This option gives the user more control over the collecting process: A suitable value of N can be chosen to balance between test suite compactness and test generation time. Note that the test suite becomes of comparable size w.r.t. the case without limiting when N approaches 60, but the generation time still remains smaller.

O.12 and O.13 can find test suites as small as those found without limiting. However, O.12 requires almost a tripled time with respect to the best combination without limiting (O.9 and O.1), and, therefore, O.12 is not effective. Fig. 5 depicts the effects of O.13 depending on N; for values of N lower than around 20 the time is greater than the best combination without limiting, for values of N greater than 20 the time is lower: O.13 is effective for N greater than 20.

Overall, we can state that limiting the collecting is effective in reducing the time for test generation with possible no negative effects over the test suite size.

V. Conclusion and Future Work

We have presented a set of optimizations to improve a process of automatic test generation by SAT/SMT techniques. Although we propose and apply the optimizations only for a selected number of SAT and SMT solvers and for fault-based testing of Boolean expressions, most techniques are general enough and can be applied to other approaches, as well, to speed up the model-based test generation even not for Boolean expressions. Some optimizations exploit specific features of a SAT or an SMT solver, others requires specific form of the input formulas or are applicable only to fault-based test predicates. However, most of the proposed optimizations modify the test generation process and can be applied regardless the notation and the tool used for test generation.

Experimenting these optimizations through a set of benchmark case studies, we make apparent that a well engineered and optimized SAT/(but better an)SMT-based test generation process can be used in practice for Boolean specifications instead of the classical algorithms like MUMCUT and MCDC.

In this paper, we assume completeness of the input space. As future work, we plan to review the proposed process in the presence of constraints on input values, and to adapt optimizations accordingly. We plan also to study the application of the optimizations to the generation of combinatorial tests by SMT solvers [2], [4] and to experiment some heuristics regarding test predicate ordering.

References

[1] A. Biere. PicoSAT essentials. *JSAT*, 4(2-4):75–97, 2008.

[2] A. Calvagna and A. Gargantini. A formal logic approach to constrained combinatorial testing. *J. of Automated Reasoning*, 45(4):331–358, 2010.

[3] J. Chilenski and S. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5), 1994.

[4] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 129–139, New York, NY, USA, 2007. ACM.

[5] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] B. Dutertre and L. de Moura. The Yices SMT solver. Technical report, SRI Available at http://yices.csl.sri.com/tool-paper.pdf, 2006.

[7] N. Een and N. Sörensson. Minisat v2. 0 (beta). *Solver description, SAT race*, 2006.

[8] A. Gargantini. Dealing with constraints in boolean expression testing. In *CSTVA 2011- 3rd Workshop on Constraints in Software Testing, Verification, and Analysis, Berlin, March 25, 2011*, 2011.

[9] A. Gargantini and G. Fraser. Generating minimal fault detecting test suites for general boolean specifications. *Information and Software Technology, Elsevier*, 53:1263–1273, 2011.

[10] H. Jain and E. M. Clarke. Efficient SAT solving for non-clausal formulas using DPLL, graphs, and watched cuts. In *Proc. 46th ACM/IEEE Design Automation Conf. DAC '09*, pages 563–568, 2009.

[11] K. Kapoor and J. P. Bowen. Test conditions for fault classes in Boolean specifications. *ACM Transactions on Software Engineering and Methodology*, 16(3):10, 2007.

[12] D. Le Berre and A. Parrain. The SAT4J library, release 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation(JSAT)*, 7:59–64, 2010.

[13] S. Prestwich. *Handbook of satisfiability*, chapter 2 - CNF Encodings, pages 75–98. IOS Press, 2009.

[14] E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a Boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, May 1994.

[15] Y. T. Yu, M. F. Lau, and T. Y. Chen. Automatic generation of test cases from boolean specifications using the MUMCUT strategy. *Journal of Systems and Software*, 79(6):820–840, 2006.