Model-Based Testing for Verification Back-Ends

Conference Paper · June 2013

DOI: 10.1007/978-3-642-38916-0_3

CITATIONS
11

READS
37

3 authors, including:

Cyrille Artho
KTH Royal Institute of Technology
117 PUBLICATIONS **1,463** CITATIONS

SEE PROFILE

Armin Biere
Johannes Kepler University Linz
218 PUBLICATIONS **10,495** CITATIONS

SEE PROFILE

Model-Based Testing for Verification Back-ends*

Cyrille Artho[1], Armin Biere[2], and Martina Seidl[2,3]

[1] National Institute of Advanced Industrial Science and Technology (AIST),
Research Institute for Secure Systems (RISEC), AIST Amagasaki
c.artho@aist.go.jp
[2] Institute for Formal Models and Verification,
Johannes Kepler University, Linz, Austria.
{martina.seidl, armin.biere}@jku.at
[3] Business Informatics Group
Vienna University of Technology, Vienna, Austria

Abstract. Many verification tools used in practice rely on sophisticated SAT and SMT solvers. These reasoning engines are assumed and expected to be correct, but, in general, too complex to be fully verified. Therefore, effective testing techniques have to be employed. In this paper, we show how to employ model-based testing (MBT) to test sequences of application programming interface (API) calls and different system configurations. We applied this approach to our SAT solver Lingeling and compared it to existing testing approaches, revealing the effectiveness of MBT for the development of reliable SAT solvers.

1 Introduction

Rigorous formal techniques provide the tools for verifying crucial stability and correctness properties of hardware and software systems in order to increase their reliability as well as the trust of their users. Examples of successful verification techniques include model checking and automated theorem proving (cf. [1] for a survey). For applying these techniques, dedicated software is required which provides (semi-)automatic support during the verification process. Solving verification problems is not a trivial task and therefore, many sophisticated approaches have been developed. Many of these approaches break down the original problem to the problem of deciding the satisfiability of propositional logic (SAT) and extensions (SMT) [2]. For SAT, the prototypical NP-complete problem, not only a myriad of results are available giving a profound understanding of its theoretical properties, but also very efficient tools called SAT solvers [3] have been made available over the last ten years.

When a SAT solver serves as back-end in a verification tool, its correctness and stability is of particular importance, as the trust put in the system to be verified strongly depends on the trust in the verification system, and hence in the

* This work was partially funded by the Vienna Science and Technology Fund (WWTF) under grant ICT10-018 and by the Austrian Science Fund (FWF) under NFN Grant S11408-N23 (RiSE).

SAT solver. Efforts have been made to verify SAT solvers, but since the implementation of modern SAT solvers relies on sophisticated low-level optimizations, complete verification is hardly possible. To ensure robustness of a SAT solver, one has to rely on traditional testing techniques. In particular, grammar-based black-box testing and delta debugging have shown to be of great value [4].

In this paper, we present a *model-based testing framework* for verification back-ends like SAT solvers. This framework allows testing different system configurations and sequences of calls to the application programming interface (API) of the verification back-end. Whereas in previous approaches only the input data has been randomly generated, we suggest to randomly produce valid sequences of API calls. Possible sequences are described by the means of a state machine.

Additionally, we randomly vary the different configurations of the verification back-end. Often, a verification tool implements a huge number of options which enable/disable/configure different pruning techniques and heuristics. The optimal settings for the options is strongly dependent on the problem to be solved, so there is no general optimal setting. We use a model to describe the different configurations of a verification back-end. Guided by this model, we instantiate the verification back-end randomly. If a defect in the verification back-end triggers a failure, we show how to reduce a failure producing trace by *delta debugging*.

The main contribution of this paper is therefore to introduce model-based API testing and model-based option testing, their combination with delta debugging, and an empirical evaluation showing the effectiveness of our framework.

We realize the proposed testing framework for the SAT solver Lingeling [5], which is an advanced industrial-strength SAT solver, with top rankings in recent SAT competitions.[4] It is used in many verification applications both in industry and academia. To evaluate the presented approach, we set up three experiments where we randomly seed some faults in Lingeling and compare the new approach to other well-established testing techniques.

This paper is structured as follows. First, we introduce basic notions of SAT solving and testing techniques in Section 2. Then we introduce a general architecture for model-based testing verification back-ends in Section 3, which is instantiated for the SAT solver Lingeling as described in Section 4. Experiments that underpin the effectiveness of our framework are shown in Section 5. Section 6 discusses related work, and Section 7 concludes with a discussion of related approaches and an outlook to future work.

2 Fuzzing and Delta Debugging for SAT Revisited

Since a detailed discussion of SAT solving is not within the scope of this work, we shortly revisit only the concepts and terminology important for the rest of the paper. A comprehensive introduction to the state-of-the-art in SAT solving is given in [2]. Additionally, we recapitulate the general idea of *grammar-based black-box testing* (vulgo *fuzz testing* or simply *fuzzing*) and *delta debugging*.

[4] http://baldur.iti.kit.edu/SAT-Challenge-2012/

2.1 Background

In general, SAT solvers implement conceptually simple algorithms to decide the (un)satisfiability of a propositional formula. A propositional formula is a conjunction of clauses. A clause is a disjunction of literals, with a literal being a variable or a negated variable. The task of a SAT solver is to find an assignment to each variable such that the overall formula evaluates to true in case of satisfiability or to show that there is no such assignment in case of unsatisfiability. A variable may be assigned the value true or false. A negated variable $\neg x$ is true (resp. false) if it is assigned false (resp. true). A clause is true if at least one of its literals is true. A formula is true if all of its clauses are true. Propositional formulas of the described structure are said to be in *conjunctive normal form* (CNF), which is the default representation for state-of-the-art SAT solvers.

For solving a propositional formula, most state-of-the-art SAT solvers implement a variant of the algorithm by Davis, Logeman, and Loveland (DLL) [6] which traverses the search space in a depth-first manner until either all clauses are satisfied or until at least one clause is falsified. In the latter case the SAT solver backtracks if not all assignments have been considered. For the application of SAT solvers on reasoning problems of practical relevance, a naive implementation of this algorithm is insufficient. Very sophisticated pruning techniques like learning and effective heuristics and data structures have to be realized within a SAT solver, such that the source code of a SAT solver has thousands lines of code usually written in the programming language C; e. g., the SAT solver Lingeling [5] consists of more than 20,000 lines of code. For making a SAT solver efficient on a certain set of formulas, mostly the right configuration, i. e., a specific combination of the options and parameter settings of the solver, has to be found. Due to very sophisticated pruning techniques and well-thought-out implementation tricks, a SAT solver can be tuned in such a manner that it solves most problems occurring in applications in a reasonable amount of time, although the worst case runtime of course remains exponential.

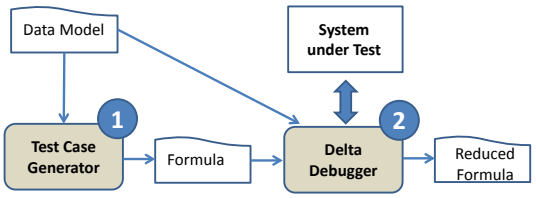

Fig. 1. Fuzzing + Delta Debugging.

Brummayer et al. [4] showed that fuzz testing and delta debugging is effective in testing and debugging large SAT solver implementations, with a high degree of automation. The basic workflow is shown in Fig. 1. It consists of the *test case generator* for generating random formulas according to a grammar provided by a data model and the *delta debugger* for reducing the size of the formula such that the failure still occurs. In the following, we shortly review the two components as we will extend this approach in the rest of this paper. Please note that this approach is not restricted to propositional logic, but may be also used for other languages with more complex concepts (cf. [7,8,9]).

2.2 Test Case Generation

In grammar-based black-box testing no knowledge about the internal structure of the system under test (SUT) is available. The SUT is fed with randomly generated input data for automatically testing stability and correctness of the system. In order to ensure that the input data can be parsed, i. e., not only scenarios with malformed input data are tested, a model is provided which describes the set of syntactically valid inputs. This model may be specified by the means of a textual grammar, hard-coded in fuzzing tools like CNFuzz and FuzzSAT [4] or it may follow the approach of [10], where the structure of the formulas to be generated is specified in a domain-specific language.

For propositional formulas, several models have been proposed whose practical hardness may be configured by a few parameters, like the ratio of variables and clauses [11,12]. Since the language of propositional logic is not very complex, in general only few syntactical restrictions have to be considered. As argued in [4], besides the high degree of automation, the main success factor of fuzzing SAT solvers is based on the fact that a high throughput of test cases is achieved. Therefore, a balance between hard and trivial formulas has to be found.

2.3 Delta Debugging

Given an input which observably triggers a failure of the system under test, the delta debugger has the goal to simplify the input while preserving the failure. On this simplified input the analysis of reasons for the failure, i. e., the debugging, becomes easier. In the context of SAT, input formulas often consist of tens of thousands of clauses. For a human developer it is hardly feasible to manually step through the code of the SAT solver when such a huge input is processed. In order to reduce the input to a new syntactically correct test case, and also for the delta debugging process itself, knowledge on the structure of the input data is useful. For SAT formulas in CNF, delta debuggers remove either clauses or some literals of a clause. Delta debuggers for non-CNF formulas like qprodd[5] or for SMT like deltaSMT[6] and ddSMT[7] need more sophisticated reduction techniques, since the underlying data structures are trees instead of lists of lists (see also [13]).

In contrast to the test case generator, the delta debugger has to call the system under test. In order to simplify given input data, the delta debugger goes through the following process: First, the SUT is run on the given input data. Then the delta debugger tries to reduce the size of the input based on some heuristics. The SUT is run again, now with the reduced test case. If the failure is still observed, the delta debugger tries to perform more simplifications. Otherwise, it undoes the changes and applies different reductions. The latter steps are repeated until either a time limit is reached or the obtained test case fulfills some predefined quality criteria.

[5] http://fmv.jku.at/qprodd/
[6] http://fmv.jku.at/deltasmt/
[7] http://fmv.jku.at/ddsmt/

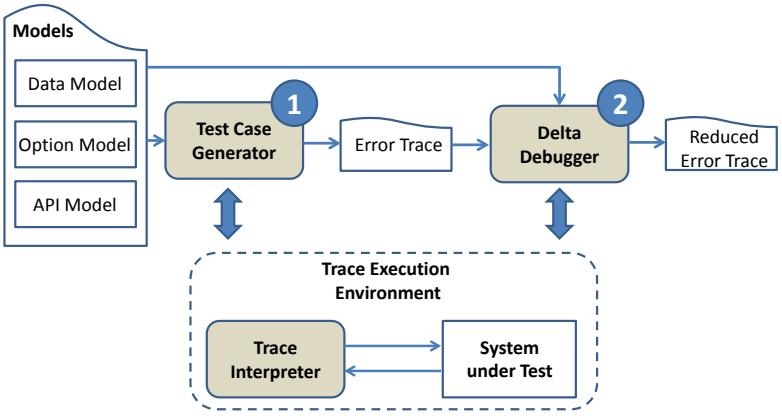

Fig. 2. Extended Workflow for Fuzzing and Delta Debugging.

3 General Architecture

The testing approach discussed in the previous section is agnostic of the SUT. Communication with the SUT is done via a file, which is generated by the test case generator and reduced by the delta debugger in the case a failure has been found. For solving propositional formulas, all SAT solvers which are able to process the standard DIMACS format, can be plugged into the testing workflow shown in Fig. 1. In that workflow, the test case generator and delta debugger produce and reduce propositional formulas in CNF. However, little control over the execution behavior of the SUT is possible with the consequence that not all features of the system are covered by the generated test cases.

In particular, incremental SAT solving as implemented in most modern SAT solvers, cannot be tested. Incremental SAT solving is used for many applications, e. g., for enumerating all solutions of a formula. After solving a satisfiable formula, the solver neither terminates nor is reset; instead, additional constraints are provided. The SAT solver checks if the formula is still satisfiable with the new constraints. Furthermore, the SUT is run with certain options set, but there might be defects which only show up under a certain combination of options.

To circumvent these problems, we propose to use a model-based testing approach for verification back-ends like SAT solvers. In particular, we suggest to fuzz not only the input data, but to generate sequences of API calls, to cover more features of a solver. The sequences of valid API calls are described by a state machine. To test different combination of options, setting options is also fuzzed. The range of possible options is also defined by a model. The adopted workflow is shown in Fig. 2. The goal is still to produce syntactically valid input data, which uncovers defects of the SUT. The individual components of the proposed approach are described in the following.

3.1 Test Case Generation

The test case generator takes three different kinds of models as input: (1) the data model, which is basically the same as the data model used in the approach described in the previous section. Additionally, (2) the option model and (3) the API model have to be provided.

The *option model* describes valid options and valid combinations of options of the SUT. For example, if an option requires an integer of a certain range, this constraint is documented in the option model. Further, the model might contain probabilities stating the chance that a certain option is selected to be set. This feature is necessary to avoid that options which are not so relevant are tested too extensively at the expense of other, more important options. The selection of the probabilities is based on the experience of the modeler.

The *API model* describes valid traces of API calls. It documents how the API has to be used. To generate a trace, the results of fuzzing options and input data have to be included. Hence the test case generator has to combine the three models. For example, the input data may be either read from a file or it may be programmatically handed over by dedicated API calls. As the call of certain functions might be optional, also the API model may be equipped with probabilities for the selection of functions which are not mandatory to be called.

Both the API model and the option model contain information specific to the SUT. Since verification back-ends like SAT solvers often have similar functionality and APIs, reuse is achievable by specifying a generic model and appending to each state the API calls to be performed. If another system with similar functionality has to be tested, only the names of API calls have to be changed.

In principle, the test case generator could be used without communicating to the trace execution environment which is described in Section 3.3. If the test case generator has a direct exchange with the SUT while generating the test data, the results of API calls can be directly considered during the search for a trace which triggers a failure.

3.2 Delta Debugging

For the API testing approach, the delta debugger not only has to reduce the input data, i. e., the formula, but the trace itself such that the failure still occurs. As a trace is a linear sequence of API calls, no complicated rewriting is needed when a call is removed as it would be necessary if the internal node of a tree is removed. However, the delta debugger has to obey the description of the API model in order to maintain a valid trace. For example, there might be calls which may not be removed, like the initialization and release routines or the function which starts the actual solving process.

As for the test case generator, the delta debugger communicates with the system under test in order to incorporate the result of a selected action into the reduction process. Since the mere removal of an API call may not be enough to obtain the expected reduction, it might be necessary also to vary the arguments of a call.

3.3 Trace Execution

In our approach, both the test case generator and the delta debugger communicate with the SUT to achieve better results. For the test case generator this means to get a high coverage rate for uncovering defects. For the delta debugger this means to reduce the failure-triggering traces as much as possible. Communication can be achieved in two ways: either the test case generator and delta debugger directly call API functions. Although this is the more direct implementation, it reduces the reusability of the framework. Alternatively, calls could be attached to the transitions of the API model. If a transition is taken, its attached function is invoked as in Modbat [14]. A potential issue of that approach is that such a testing framework may only support certain programming languages. Furthermore, the testing framework has to interpret the output and return values of the SUT; this again makes the testing framework tailored towards a given system.

Alternatively, the SUT may be wrapped into an execution environment, where a trace interpreter interacts with the SUT. The trace interpreter has to be developed for each SUT individually and is able to call the functions of the SUT directly. The output of the SUT may then be translated into a format which can be processed by the testing framework. The trace interpreter allows to replay the trace reduced by the delta debugger which can then undergo manual debugging in order to find and eliminate defects.

Using a trace interpreter to replay a trace has another practical advantage. If a solver is used as a verification back-end in a larger verification system, it might happen that the solver triggers a failure when a certain sequence of API calls is performed. It might be difficult to reproduce the failure by simply dumping the formula and passing it as command line argument, because internally the solver follows another sequence of API calls. In order to report the defect to the solver developer without giving away the whole verification system, the trace of the failing run of the solver might be produced (under the assumption that the solver is equipped with a logging functionality). The solver developer can replay the trace, analyze the undesired behavior, and fix the bug.

3.4 Discussion

With three different kinds of input models, better control on the test case generation is achieved, assuming the models reflect the behavior of the SUT in an accurate manner. If a model is too restrictive, code coverage is decreased for the test case generator and also less reduction can be achieved by the delta debugger. If the model is too lax, i. e., not precise enough, (external) contracts of API functions might be violated and invalid traces are generated.

In order to obtain good code coverage and increase flexibility for delta debugging, but also for reducing modeling effort, the testing framework is able to deal with *under-approximative models* by relying on a callback feature to give certain feedback from the SUT back to the testing framework. Then the testing framework can react immediately when contract violating traces occur. To

use this feature, the SUT also has to be equipped with *API contract assertions* similar to assertions used in specification-based testing (cf. for example [15]).

4 Case Study: Model-Based Testing for Lingeling

In this section, we show how the presented testing framework is used for testing the SAT solver Lingeling. Therefore, the three different models have to be specified as well as the different components presented in the previous section. The framework is available at `http://fmv.jku.at/lglmbt`. Before we discuss the details of the testing framework, we shortly review the features of Lingeling.

4.1 Lingeling at a Glance

Lingeling [5] is a SAT solver that interleaves searching with very powerful preprocessing techniques. Preprocessing techniques are effective, but computationally expensive techniques which are traditionally applied only at at beginning of the solving process. In Lingeling, such techniques are integrated within the search process by binding their applications to a given extent. These bindings can be controlled both by command line and programmatically, which cause Lingeling to have more than 140 options with many of them requiring an integer value to be selected. For this purpose, fuzzing the options is extremely valuable for testing combinations of different features. In the implementation of Lingeling, much emphasis is spent on a compact representation of clauses for processing very large input formulas as they occur in practice. The implementation is done in C and consists of more than 20,000 lines of code. Without dedicated tool support for testing as proposed in this paper, finding and eliminating defects would hardly be possible.

Actually, the work presented in this paper can be seen as a crucial technique for enabling the integration of an incremental SAT solving API into Lingeling, which in turn made it possible to use a state-of-the-art SAT solver back-end in our SMT solver Boolector. A substantial part of the success of Boolector in the SMT 2012 competition is attributed to this fact.

Besides explicitly running Lingeling from the command line, the solver can also be included as a library. The API includes more than 80 functions. Additionally to the sequential version of Lingeling, there is also a multi-threaded variant which builds on top of the sequential version. The testing framework discussed in the following has only been used for testing the core library and the sequential solver. The API functions used by the multi-threaded front-end are hard to test with, since they mainly define call-backs. We leave it to future work to extend the framework to the multi-threaded case.

4.2 Test Case Generation

The goal of the test case generator is to produce traces which are valid sequences of calls to Lingeling's API. For our prototype we encoded the models necessary to

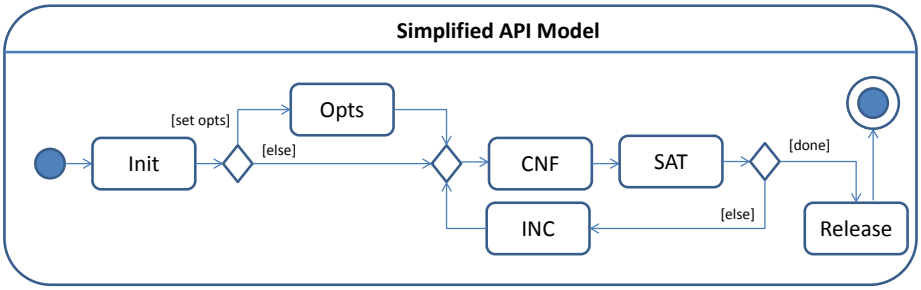

Fig. 3. Simplified API Model of Lingeling.

describe valid traces, input data, and options directly in C, which allows direct communication with the solver. For test case generation, no intermediate layer is necessary. By sacrificing generality, the prototypical implementation is tailored towards testing Lingeling and allows to gain a first understanding of the power of the suggested approach for testing a state-of-the-art SAT solver. A sample trace is shown in Fig. 4.

API Model. The API model (see Fig. 3) documents some contracts which have to be fulfilled when using the API. The omitted features deal with additional optimization techniques which have to be called at certain positions within the model. After initialization (state Init), options (state Opts) may be set. The path to be taken is decided by random. By empirical evaluation it turned out that setting options with a probability of 0.5 is a good choice. If the path to Opts is taken, then options are set according to the option model. In the next step, the formula to be solved has to be generated. Here, knowledge of the data model is necessary.

```
init
option actstdmax 80
option bias 2
option ccereleff 3
option cgrmineff 200000
add -58
add 1
add 2
add 0
add -1
add -2
add 0
assume 1
setphase -2
sat
release
```

Fig. 4. Example of a Trace.

After having created the formula, optimizations are performed with a certain probability. The formula is then handed to the solver. After completing the solving process, the incremental feature of the solver may be tested by changing to the state Inc (this is only possible if the formula is SAT), to extend the formula with additional constraints, and to start the solving again. Alternatively, the solving process could be stopped. If this is done according to the API contract, some functions to free memory have to be called.

Option Model. The description of the options to be generated uses an introspective API function of Lingeling which allows to query the solver for its available options and how they shall be initialized. A list of options to be excluded from testing is also provided, including options related to logging. An option is set to a new value with a probability of 50 %. The choice of the new value depends on the range of valid option values.

Data Model. Lingeling processes propositional formulas in CNF. In our framework, this formula is randomly generated. Unlike in previous work, the formula is not written to a file, but it is fed programmatically to the solver. API calls are used to add literals, represented as positive and negative integers. The generated formulas should not be trivial, but they should also not be too hard, to avoid that the solver does not terminate. It also has to be ensured that no tautological clauses are generated, i. e., clauses which contain a literal in both polarities.

Experiences showed that formulas with between 10 and 200 variables give the best results. If n is the number of variables, the number of clauses is given by $(n * x)/100$ where x is a number between 390 and 450. Again, the values are based on many years of solver development experience, but related to the phase transition threshold of SAT solving.

The length of individual clauses is decided as follows. Clauses of length one, two or three are special and are handled differently than other clauses. For example, in unary clauses (clauses of length one), the truth value of its literal can be decided immediately and therefore be propagated to all other occurrences of the respective variable. The generation of these three kinds of clauses is fostered by giving them a higher probability to be generated than other clauses. The length of a clause is naturally constrained by the number of variables occurring in a formula. A variable in a clause is negated with the probability of 50 %.

For testing incremental SAT solving, additional clauses have to be generated which are added to the current formulas between calls to the solving routine. These clauses are generated in the same way as just described, over already existing and a certain small number of new variables.

4.3 Delta Debugging and Trace Execution

We developed a delta debugger which reduces a given trace as follows. First, the file containing the trace is parsed and a list of all commands is built. At the moment, about 30 different commands are supported, having either one or no argument. Then the original trace is replayed in order to obtain the *golden* exit status of the execution, which should also be returned by the execution of the reduced trace. Then the rewriting of the trace is initiated.

In principle, only sub-traces are extracted, but it has to be ensured that the API model discussed in the previous subsection is not violated, i. e., certain parts like the initialization and the release commands may not be removed. Also the values of the solver options are changed during the trace reduction process, with the hope that another configuration of the solver triggers the failure earlier.

For replaying the traces, a simple interpreter is provided. This interpreter executes not only the traces produced by the fuzzer and delta debugger, but also traces produced during all runs of the solver. The solver is equipped with a logging functionality implemented by the means of a macro calling a certain API function of Lingeling, which outputs every API call in the required format. Logging can be enabled through an API call or by setting an environment variable (`LGLAPITRACE`) to point to the trace file.

For dealing with inadequate API models and for realizing the previously described call back functionality, Lingeling internally executes a state machine. If an invalid state transition would be caused by an API call not possible in the current state, a special assertion fails. This gives the feedback to the caller of the API function that the invocation was incorrect. With this information the caller could adopt its behavior accordingly, i. e., in the case of the delta debugger a different kind of reduction is performed.

5 Experimental Evaluation

Our experience in using the presented framework when developing Lingeling is extremely positive. This section describes experiments to corroborate this, measuring efficiency in terms of throughput, code coverage and detect defection capability.

5.1 Experiment 1: Code Coverage

We measured the code coverage with the tool `gcov` of the GNU compiler collection. The evolution of the code coverage for 10,000 runs is shown in Fig. 5. CNF Fuzzing achieves code coverage of about 75 % after 10,000 runs which could be improved by 5 % by MBT without option fuzzing, and by additional 5 % by MBT with option fuzzing. The difference between CNF Fuzzing and the MBT approaches might be explained by the fact that CNF Fuzzing does not test the incremental feature of the solver. Coverage of 100 % is not possible due to the fact that only correct formulas and traces are generated, so the error handling code is never called. We observed that even for more runs, the values do not change anymore. Creating corrupted input for testing error handling is not within the scope of this work, but might be interesting in the future.

5.2 Experiment 2: Throughput

The effectiveness of random resp. fuzz testing depends not only on the quality of the generated tests, but also on the number of test cases executed per second, which we define as *throughput*. For our MBT approach we achieved a throughput in the order of 251 test cases per second: 919,058 test cases were executed during one hour of running the model-based tester on an Intel Xeon E5645 2.40 GHz CPU.

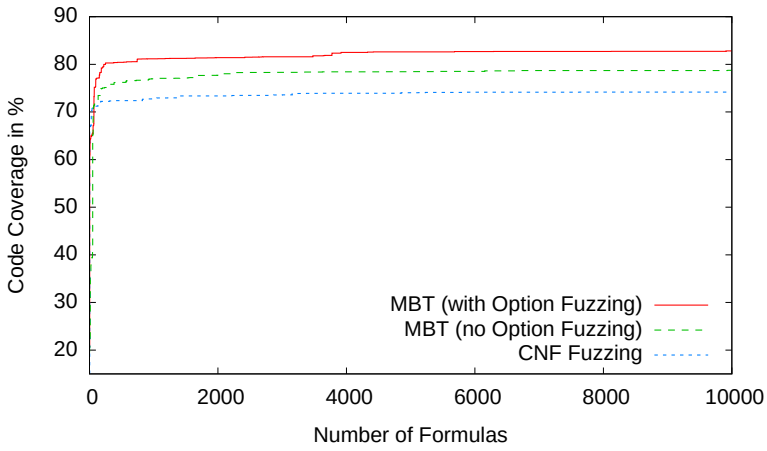

Fig. 5. Comparison of Code Coverage.

Note, that roughly 10 % of these test cases are actually terminated early due to contract violations. This occurs because the model is not precise enough to entirely exclude invalid API call sequences. Those are executed at the point where the contract violation is detected (by API contract assertions in the library) and thus can still be considered valid test cases. This feature of our approach allows a trade-off between the effort needed to capture API contracts precisely in the model and the effectiveness of testing.

To measure the throughput of file-based fuzzing we piped the output of the fuzzer to Lingeling to avoid disk I/O. Still the throughput did not reach more than 25 runs per second, measured for batches of 100 runs, on the same machine. This is an order of magnitude slower than for MBT. In both cases we used a binary compiled with `-O3` but assertion checking enabled (so no `-DNDEBUG`). By this huge difference in the throughput, the benefits of accessing the solver via the API become directly visible. No time is wasted with I/O and parsing.

Note that fuzz testing is "embarrassingly parallel" and we have successfully used the combination of fuzz testing and delta debugging on a cluster with 128 cores, with the goal to produce smaller failure traces than an existing but large and impossible to delta debug trace obtained from an external user or from failing runs on huge benchmarks.

5.3 Experiment 3: Fault Seeding

In a third experiment we randomly inserted defects by either adding an abort statement or deleting a line in the code of Lingeling. These "mutations" were restricted to the core component of Lingeling in the file `lglib.c` consisting of 19,141 lines of C code (computed by the Unix `wc` utility). A mutation is consid-

name	variant	number of successful runs (out of 404)		average time in seconds	average trace size	average #lines covered	relative trace size	relative #lines covered
model-based, no option fuzzing	actual	398	98.51 %	1.67	2172.94	3796.49	100.00 %	100.00 %
	Δ dbg	398	98.51 %	81.43	291.66	2873.58	13.42 %	75.69 %
model-based, with option fuzzing	actual	397	98.27 %	1.69	2228.53	3951.01	100.00 %	100.00 %
	Δ dbg 1	396	98.02 %	72.30	290.54	2916.24	13.05 %	73.99 %
	Δ dbg 2	390	96.53 %	158.25	51.23	1318.45	2.26 %	32.79 %
file-based, no option fuzzing	actual	357	88.37 %	1.64	2908.97	4356.41	100.00 %	100.00 %
	Δ dbg	347	85.89 %	154.25	1141.55	4084.62	39.53 %	95.24 %
file-based, with option fuzzing	actual	324	80.20 %	1.21	2666.28	3573.27	100.00 %	100.00 %
	Δ dbg	314	77.72 %	179.78	31.19	1325.90	1.18 %	38.05 %
regression, no option Δ dbg	actual	354	87.62 %	0.85	1056.34	4267.77	100.00 %	100.00 %
	Δ dbg	346	85.64 %	37.32	360.15	4207.34	49.11 %	100.00 %
regression, with option Δ dbg	actual	354	87.62 %	0.87	1056.33	4271.84	100.00 %	100.00 %
	Δ dbg	349	86.39 %	160.96	82.83	1641.41	9.51 %	38.83 %

Table 1. Mutation Experiments: 963 mutations, 681 compilable, 404 defective.

ered as invalid (counted as "not compilable") if the line it affects contains the use of a macro for initializing an option or tracing the API.

This experiment was run on two identical computers (Intel Xeon CPU E5645 2.40 GHz, the same hardware as for Exp. 2) for roughly one week and produced the results presented in Tab. 1. The table is split vertically into three parts: *model-based* fuzzing / delta debugging (Δ dbg) as presented in this paper, *file-based* fuzzing / delta debugging as in [4], and *regression* runs over 93 collected and hand-crafted CNF files used for many years in the development of various SAT solvers. The regression testing is of course deterministic and the base-line regression suite (before introducing mutations) takes 10 seconds to complete.

We include runs with and without fuzzing options resp. with and without delta debugging of options. In the experiments for model-based testing with option fuzzing, we distinguish two variants of delta debugging. The first variant "Δ dbg 1" only reduces options explicitly set in the failing test, while default options are not changed. The second variant "Δ dbg 2" considers all options for delta debugging. Note, running regressions only allows to delta-debug options but does not really allow to fuzz them.

The 3rd and 4th columns show the success rate of fuzzing resp. delta debugging with respect to all 404 mutations, for which at least one method was able to produce a failure: an assertion violation, segmentation fault, etc. As in Exp. 2 the executable is optimized (`-O3`) but does include assertion checking code (no `-DNDEBUG`). Mutation and compilation time are not taken into account.

Both model-based approaches (with and without option fuzzing) have the highest success rate. Actually, for 31 mutations only these two were successful in producing a failure within a time limit of 100 seconds. The file-based fuzzers did not produce any failure that was not found by model-based testing as well. The

regression suite was slightly more successful and detected three failures that no other method could detect. The 5th column contains the average time needed to produce a failure (not including time-outs).

For each compilable mutation, testing resp. fuzzing continued until the first failure or the time limit of 100 seconds was reached. Each failing test case was then subjected to delta debugging with a time limit of one hour. The algorithm for delta debugging depends on the type of testing: trace shrinking for MBT, and CNF reduction for file-based fuzzing and regression testing. Even with a time limit of one hour per test case, some delta debugging runs timed out and thus the success rate dropped slightly (except for model-based testing without option fuzzing, the first row below the header in Table 1).

In order to be able to compare the effectiveness of trace based and CNF based techniques, we show in the remaining columns the size of failing test cases as well as the number of lines executed. The *size* of a failing test is measured in terms of the size of the API trace produced either directly by the model-based tester or obtained implicitly after tracing the API calls when reading and solving the CNF file. Commands to set options are not counted. This size metric allows to compare sizes of test cases across different testers (with and without fuzzing options).

We consider the number of lines executed during one test case as an important metric for the *quality* of the test case. To obtain the number of executed resp. covered lines, the binary was recompiled with debugging support (`-g`). The compiler was also instructed to include code for producing coverage information. After running the test case the number of executed lines was determined with the help of `gcov`. For each tester resp. delta debugger the average numbers are calculated over all successful runs, while the relative numbers give the same information, but are normalized w. r. t. the tester.

The experiments showed that our MBT approach is substantially more effective in finding defects than previously used techniques. Taking the time-outs into account, it is also faster and even without option fuzzing produces much smaller test cases. Fuzzing and in particular delta debugging of options is particularly effective in reducing the size of traces. We see a reduction of almost two orders of magnitude by delta debugging options, while delta debugging without touching options gives a reduction of slightly less than one order of magnitude.

6 Related Work

Only few publications about testing and debugging verification back-ends exist. Grammar-based black-box fuzzing and delta debugging for SAT and its extension QSAT have been presented in [4] where the authors showed state-of-art solvers contain defects, not revealed by running the standard benchmark sets as used in competitions. Several works deal with the generation of random formulas (e. g., [10,12,11]), but these focus on theoretical properties of formulas and not on their suitability for supporting the solver development process. Similar approaches are available for SMT [7] and ASP [8].

Model-based testing for verification back-ends as proposed in this work has—to the best of our knowledge—never been applied specifically to verification back-ends, but only to arbitrary software systems. Since the literature on model-based testing is too vast to be discussed in detail, we refer to [16] for a survey. Fuzzing options has been realized in the ConFu approach [17], which randomly tests different configurations of a tool during runtime. To this end, the tester has to annotate the parameters of the function to be tested with constraints. For *model-based option testing*, research on model-based testing software product lines are probably the most related (see for example [18]). A software product line is a family of software systems derived from shared assets. By the means of variability models, the possible configurations (the options) are described which are applicable. However, the variability found in software product lines is more complex than the configuration facilities found in SAT solvers in terms of combination constraints, therefore, for a SAT solver a more focused realization of option fuzzing is possible.

Model-based API testing is for example realized in the tool Modbat [14], which is a Scala-based tool providing an embedded domain-specific language (DSL) for specifying the model. Modbat supports only the testing of Java bytecode, but provides a more sophisticated event handling than necessary for our purposes. In the .NET framework, the Abstract State Machine Language (AsmL) can be used for the automatic generation of method parameters and the automatic derivation of test sequences [19]. In this context, also work has to be mentioned which uses contracts as provided by the API for the generation of test data [20].

Delta debugging for SAT solvers has been described in [7], where the size of a formula is reduced. There, the input data (the formula) is reduced such that a failure still occurs. Shrinking techniques for reducing the size of execution traces are for example described in [21]. Delta debugging traces in the context of SAT solving has—to the best of our knowledge—never been presented before.

7 Conclusion

We propose to apply model-based testing for verification back-ends, like SAT solvers. In this approach, not only the input data is randomly generated, but also sequences of valid API calls. This makes it possible to test, for example, the incremental features of SAT solvers. These incremental features play an important role in verification applications. Besides that, we additionally included option fuzzing in our testing framework, which randomly selects different configurations of the SUT.

We combined the presented model-based testing approach with delta debugging, to reduce failure triggering traces. This combination of model-based testing and delta debugging is a powerful tool for testing verification back-ends. As proof of concept we implemented the proposed testing framework for the SAT solver Lingeling and performed an extensive empirical evaluation. Different kinds of experiments confirmed the effectiveness of model-based testing in combination with delta debugging. Based on these experiments and on our long-time expe-

riences in solver development, we believe that the techniques described in this paper are effective in general, and are particularly useful when applied to other formal reasoning engines like SMT solvers, theorem provers, or model checkers.

In future work, we plan to compare our dedicated mutation tool to more general approaches like Milu [22] and extend the presented testing framework to multi-threaded and reentrant engines. Testing our SMT solver Boolector through its API is another target. Furthermore, we plan to investigate how the design of the input models is correlated with the quality of the generated test cases. Today many developers of SMT and SAT solvers rely on fuzzing and delta debugging from our previous work. Our new approach described in this paper is much more effective and efficient, and is hoped to have a similar impact.

8 Acknowledgements

We would like to thank Jürgen Holzleitner for sharing his idea of using option fuzzing to increase coverage, as described in his Master Thesis [23]. It increased the efficiency and productivity of our development process considerably, and particularly allowed us to produce much more complex code. Later it was confirmed by John Hughes, through private communication, that techniques for fuzzing parameters of implementations have been used extensively in applications of QuickCheck [15] too.

References

1. D'Silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. TCAD **27**(7) (2008) 1165–1178
2. Biere, A., Heule, M., van Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. IOS Press (2009)
3. Prasad, M.R., Biere, A., Gupta, A.: A survey of recent advances in SAT-based formal verification. STTT **7**(2) (2005) 156–173
4. Brummayer, R., Lonsing, F., Biere, A.: Automated Testing and Debugging of SAT and QBF Solvers. In: Proc. of SAT. Volume 6175 of LNCS., Springer (2010) 44–57
5. Biere, A.: Lingeling and Friends at the SAT Competition 2011. FMV Report Series Technical Report **11**(1) (2011)
6. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM **5**(7) (1962) 394–397
7. Brummayer, R., Biere, A.: Fuzzing and delta-debugging SMT solvers. In: Proc. of the Workshop on Satisfiability Modulo Theories, ACM (2009) 1–5
8. Brummayer, R., Järvisalo, M.: Testing and debugging techniques for answer set solver development. TPLP **10**(4-6) (2010) 741–758
9. Cuoq, P., Monate, B., Pacalet, A., Prevosto, V., Regehr, J., Yakobowski, B., Yang, X.: Testing static analyzers with randomly generated programs. In: NASA Formal Methods. Volume 7226 of LNCS. Springer (2012) 120–125
10. Creignou, N., Egly, U., Seidl, M.: A Framework for the Specification of Random SAT and QSAT Formulas. In: Proc. of TAP. Volume 7305 of LNCS., Springer (2012) 163–168

11. Nudelman, E., Leyton-Brown, K., Hoos, H., Devkar, A., Shoham, Y.: Understanding Random SAT: Beyond the Clauses-to-Variables Ratio. In: Proc. of CP. Volume 3258 of LNCS. Springer (2004) 438–452
12. Pérez, J.A.N., Voronkov, A.: Generation of Hard Non-Clausal Random Satisfiability Problems. In: Proc. of AAAI/IAAA, AAAI / The MIT Press (2005) 436–442
13. Misherghi, G., Su, Z.: HDD: hierarchical Delta Debugging. In: Proc. of ICSE, ACM (2006) 142–151
14. Artho, C., Biere, A., Hagiya, M., Potter, R., Ramler, R., Tanabe, Y., Yamamoto, F.: Modbat: A model-based API tester for event-driven systems. In: Dependable Systems Workshop. (2012)
15. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of haskell programs. ACM Sigplan Notices **35**(9) (2000) 268–279
16. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Softw. Test., Verif. Reliab. **22**(5) (2012) 297–312
17. Dai, H., Murphy, C., Kaiser, G.E.: Confu: Configuration fuzzing testing framework for software vulnerability detection. IJSSE **1**(3) (2010) 41–55
18. Cichos, H., Oster, S., Lochau, M., Schürr, A.: Model-Based Coverage-Driven Test Suite Generation for Software Product Lines. In: Proc. of MoDELS. Volume 6981 of LNCS., Springer (2011) 425–439
19. Barnett, M., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., Veanes, M.: Towards a Tool Environment for Model-Based Testing with AsmL. In: FATES. Volume 2931 of LNCS., Springer (2003) 252–266
20. Liu, L.L., Meyer, B., Schoeller, B.: Using Contracts and Boolean Queries to Improve the Quality of Automatic Test Generation. In: Proc. of TAP. Volume 4454 of LNCS., Springer (2007) 114–130
21. Jalbert, N., Sen, K.: A trace simplification technique for effective debugging of concurrent programs. In: Proc. of FSE, ACM (2010) 57–66
22. Jia, Y.: Milu. Available at http://www0.cs.ucl.ac.uk/staff/Y.Jia/Milu/ (2012)
23. Holzleitner, J.: Using feedback to improve black box fuzz testing of SAT solvers. Master's thesis, Johannes Kepler University Linz (2009)