Model-Based API Testing for SMT Solvers*

Aina Niemetz, Mathias Preiner, and Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria

Abstract

Verification back ends such as SMT solvers are typically highly complex pieces of software with performance, correctness and robustness as key requirements. Full verification of SMT solvers, however, is difficult due to their complex nature and still an open question. Grammar-based black-box input fuzzing proved to be effective to uncover bugs in SMT solvers but is entirely input-based and restricted to a certain input language. State-of-the-art SMT solvers, however, usually provide a rich API, which often introduces additional functionality not supported by the input language. Previous work showed that applying model-based API fuzzing to SAT solvers is more effective than input fuzzing. In this paper, we introduce a model-based API testing framework for our SMT solver Boolector. Our experimental results show that model-based API fuzzing in combination with delta debugging techniques is effective for testing SMT solvers.

1 Introduction

State-of-the-art Satisfiability Modulo Theories (SMT) solvers are typically highly complex pieces of software, and since they usually serve as back-end to some application, the level of trust in this application strongly depends on the level of trust in the underlying solver. Full verification of SMT solvers, however, is difficult due to their complex nature and still an open question. Hence, solver developers usually rely on traditional testing techniques such as unit and regression tests. At the SMT workshop in 2009, in [10] Brummayer et al. presented FuzzSMT, a grammar-based black-box fuzz testing tool that has been shown to be effective to uncover bugs in SMT solvers, in particular in combination with delta debugging tools for minimizing failure inducing input. In [14], we presented such a delta debugging procedure for the SMT-LIB v2 language [3]. Generational input fuzzers such as FuzzSMT typically generate random but valid input in a certain language. If this input triggers faulty behavior of the system under test, minimizing this input as much as possible while preserving its failure-inducing characteristics with delta debugging procedures enables the localization of bugs in a time efficient manner.

State-of-the-art SMT solvers usually provide a rich application programming interface (API) as the direct connection between an application and its solver back-end. This API often introduces additional functionality not supported by the input language (e.g., the SMT-LIB language). By merely generating randomized valid input sequences it may therefore not be possible to test the full feature set an SMT solver actually provides. In [1], Artho et al. proposed to apply a model-based testing framework to SAT solvers. This framework randomly generates valid sequences of API calls rather than randomized valid input, and further allows to test all possible valid system configurations by randomly setting and combining configuration options of the solver. In case of an error, an API trace is generated and replayed by a dedicated trace interpreter to reproduce the undesired behavior. Delta debugging techniques reduce an API error trace while preserving its failure-inducing characteristics and enable to locate the cause

*Supported by Austrian Science Fund (FWF) under NFN Grant S11408-N23 (RiSE).

of the error in a time efficient manner. Applying this approach to the SAT solver Lingeling [8] yields convincing results, and is in particular promising for other solver back-ends.

In this paper, based on the results in [1] we introduce a model-based API testing framework for our SMT solver Boolector [16], consisting of the model-based API tester BtorMBT, the trace execution tool BtorUntrace, and the delta debugger ddMBT. It is an integral part of the testing workflow in the development process of Boolector, complemented by basic unit testing, a regression test suite and parser testing tools (e.g., FuzzSMT [9] in combination with ddSMT [14] to test valid input, and the ddsexpr tool set [6] to test for robustness and correct error handling). Since version 1.6, our model-based tester BtorMBT and our trace execution tool BtorUntrace are shipped together with Boolector, and in particular BtorMBT can be considered as continued work in progress while Boolector is under active development. Our model-based testing workflow improved considerably since version 1.6 and our practical experience is extremely positive. Our experimental results confirm the claim in [1] that model-based API testing in particular in combination with delta debugging is effective for testing verification back-ends.

2 Workflow

The core test case generation engine in our model-based testing framework is our model-based API testing tool BtorMBT, which implements a model of Boolector's API and generates valid sequences of API calls. In case that one of these sequences causes an error, Boolector generates an API error trace, which allows to replay and reproduce faulty behaviour with our trace execution tool BtorUntrace. Our delta debugging tool ddMBT then minimizes such an API error trace while preserving its fault-inducing characteristics when replayed with BtorUntrace. Figure 1 describes the general workflow of our framework and its components as follows.

Data Model Boolector, our system under test, is an SMT solver for the quantifier-free theory of fixed-size bit-vectors, arrays, and uninterpreted functions as defined in the SMT-LIB v2 [2], and natively supports the use of non-recursive first-order lambda terms.

Option Model Boolector provides multiple solver engines, which are configurable via more than 70 options in total. All options and their default values and value ranges can be queried via its API. BtorMBT identifies valid option values based on these queries and defines invalid option combinations. Else, any random combination of options is allowed.

API Model Boolector provides a rich public API with full access to the complete feature set of the solver. It is available in C and Python, but since both BtorMBT and BtorUntrace, which tightly integrate Boolector via its API, are written in C, we will in the following focus on

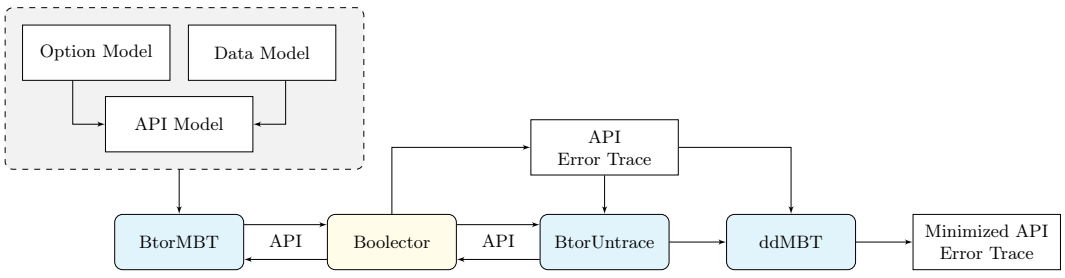

Figure 1: General workflow of model-based API testing for Boolector.

its C API, which consists of more than 150 API functions. Figure 2 illustrates the API model of Boolector as implemented in BtorMBT. It incorporates Boolector's option model and the data model as above, and will be described in more detail in Section 3.

BtorMBT Our model-based API testing tool BtorMBT generates test cases as valid sequences of calls to Boolector's API and implements the API model illustrated in Fig. 2. It aims to exploit the full feature set of Boolector as available via its API. In case of an error, Boolector generates an API error trace, which can then be replayed with our trace execution tool BtorUntrace. Note that delegating API tracing to the solver assumes that it traces API sequences correctly. Within our model-based API testing workflow it would therefore be more reliable to let BtorMBT do the tracing, which we leave to future work. However, API tracing as provided by the solver itself is still a valuable feature since it allows to reproduce erroneous behavior outside of our test framework without the need for the original setup of the tool chain that triggered the error. We will describe BtorMBT in more detail in Section 3.

API Error Trace Boolector provides the possibility to trace all API calls with their arguments into a dedicated trace file. Given such an API trace, the trace execution tool BtorUntrace then replays the sequence of API calls listed in the trace file and reproduces undesired behavior in case of an error. An example of an API trace generated by Boolector is given in Fig. 3. We will describe API tracing and the trace in Fig. 3 in Section 4 in more detail.

BtorUntrace The trace execution tool BtorUntrace allows to replay a sequence of API calls given an API trace file as above. We will describe BtorUntrace in Section 4 in more detail.

ddMBT Given an API Error trace file which logs a sequence of API calls leading to undesired behavior of Boolector, the delta debugger ddMBT minimizes the API trace while preserving this behavior reproduced via BtorUntrace. We will describe ddMBT in more detail in Section 5.

3 Test Case Generation with BtorMBT

Our model-based API tester BtorMBT is a dedicated tool for testing random configurations of Boolector. It is explicitly tailored to Boolector and supports the quantifier-free theories of fixed-size bit-vectors, arrays and uninterpreted functions, extended with non-recursive first-order lambda terms. BtorMBT serves as the test case generation engine in our model-based testing workflow and fully supports all functionality provided by Boolector via its API. In contrast to input fuzzers such as FuzzSMT [9, 10], which generate a random but valid input file to be handed to the system under test, BtorMBT tightly integrates Boolector via its C API and generates test cases in the form of valid sequences of API calls. In case that an API sequence triggers an error, Boolector produces an API error trace, which can then be replayed with BtorUntrace for debugging purposes. BtorMBT further allows to test Boolector's cloning feature [16], which generates a disjunct copy of a Boolector instance, in a test setting we refer to as *shadow clone testing*. We will describe shadow clone testing in more detail in Section 3.2.

Note that Boolector makes heavy use of runtime assertions and provides means to internally check key features of the solver. This includes model validation for satisfiable instances, checking the inconsistency of the set of inconsistent assumptions (also called failed assumptions [12]) for unsatisfiable instances when incremental solving is enabled, and checks for Boolector's cloning feature. Errors triggered by BtorMBT therefore include failed internal checks, assertion failures, segmentation faults, and any other kind of abort.

3.1 Architecture

The general architecture of BtorMBT is defined by a state machine implementing the data, options and API Model of Boolector as illustrated in Fig. 2. Test case generation with BtorMBT is performed in rounds, each with a different configuration of Boolector. One round corresponds to a sequence of states from state New to state Delete and the states are defined as follows.

New In each round, a fresh instance of Boolector is generated. Further, all parameters that influence formula size and structure such as probability distributions and maximum numbers for generating and releasing expressions are (re)initialized with random values (within certain ranges).

Set Options Boolector provides multiple solver engines, with some of them relying on an underlying SAT solver. As back-end SAT solver, Boolector supports the solvers Lingeling [8], PicoSAT [5] and MiniSat [12]. BtorMBT randomly chooses a solver engine and, if required, a SAT solver to use. The solver engine is then configured by randomly choosing and setting configuration options and their values within their predefined ranges. Note that option combinations identified as invalid according to the option model of Boolector are explicitly excluded. Further note that some options, e.g., incremental solving, are chosen with higher probability than others, depending on their relevance.

Generate Initial Expressions After a new Boolector configuration and all parameters that influence the formula size and structure have been set up, an initial set of inputs and non-input expressions is generated. The set of inputs is divided into randomly sized shares of uninterpreted functions, array variables, and Boolean and bit-vector constants and variables. Non-input expressions are randomly generated by combining inputs and already existing non-input expressions until the maximum number of non-input expressions is reached. Note that in case that the chosen solver engine only supports the quantifier-free theory of fixed-size bit-vectors, only bit-vector expressions are generated.

Main After generating an initial set of expressions, in state Main a random number of operations that influence the structure of the input formula is performed in random order: (1) new expressions are generated, (2) existing expressions are released, (3) and existing Boolean expressions are added to the input formula as assertions and, in the incremental case, assumptions. Note that when selecting expressions to generate new non-input expressions, in order

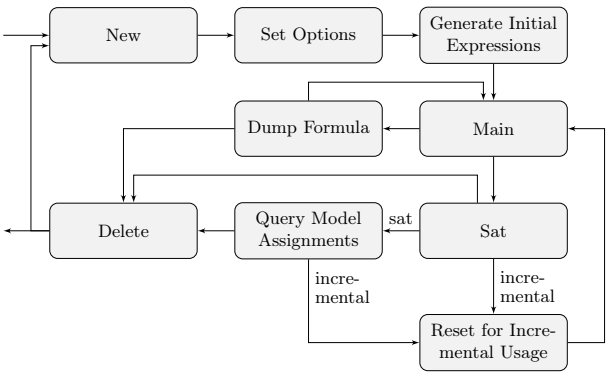

Figure 2: The API model of Boolector as implemented in BtorMBT.

to increase expression depth, expressions from the initial set are chosen with lower probability. After finalizing the current input formula, BtorMBT randomly performs various operations that operate on the current state of the input formula and possibly manipulate the current state of the Boolector instance e.g., simplifying the input formula by means of rewriting and other techniques, or generating a clone of the current Boolector instance. Next, BtorMBT randomly picks between dumping the input formula (state Dump Formula) or determining its satisfiability (state Sat). Note that the latter is chosen with higher probability.

Dump Formula Boolector allows to dump the current state of the input formula (without assumptions) anytime during the solving process and supports BTOR [11], SMT-LIB v2 [3] and AIGER [4] as output format. Depending on the structure of the formula, either of these formats is chosen randomly. Note that AIGER is a bit-blasted and-inverter-graph (AIG) representation of the input formula and can therefore only be produced if it is a bit-vector formula without uninterpreted functions, arrays and lambda terms. If the output format is BTOR or SMT-LIB v2, the formula is dumped to a temporary file. This file is then parsed into a temporary Boolector instance in order to check the dump for errors. If the output format is AIGER, the formula is dumped to stdout without checking for correctness since Boolector does not provide support for parsing input files in AIGER format. Checking the correctness of AIGER dumps is left to future work. Next, BtorMBT randomly picks between concluding the current round (state Delete) and continuing (state Main) with equal probability.

Sat After setting up the current input, a call to determine its satisfiability is issued. Boolector supports incremental solving under assumptions and, in case of unsatisfiability, allows to determine the set of failed assumptions [12], i.e., those assumptions that are inconsistent with the input formula. If failed assumptions checking is enabled, the set of failed assumptions is internally checked for inconsistency with the current input. In case of satisfiability, Boolector provides a model of the input formula, and if model checking is enabled, this model is internally checked for validity. Next, if the input formula is satisfiable and model generation is enabled, BtorMBT continues with printing and querying model assignments (state Query Model Assignments). If incremental solving is enabled, BtorMBT may randomly choose to continue with an incremental step (state Reset for Incremental Usage). Else, it proceeds to conclude the current round (state Delete).

Query Model Assignments If the input formula is satisfiable and model generation is enabled, calls to query the model assignments of all generated expressions are issued. Further, if model printing is enabled, the model of the input formula is printed to stdout. Boolector supports model output formats based on BTOR [11] and SMT-LIB v2 [3], and BtorMBT may pick either of them randomly. Note that as of version 2.5, model output in SMT-LIB format is not yet fully standardized.

Reset for Incremental Usage Prior to performing an incremental step, parameters such as maximum numbers and probability distributions are reinitialized with random values (within certain ranges). Note that the value ranges for these parameters may differ from the ranges employed in state New.

Delete State Delete concludes one round (one test case) with releasing all generated expressions and deleting the current Boolector instance.

```
1  new
2  return b1
3  set_opt b1 1 incremental 1
4  set_opt b1 14 rewrite-level 0
5  bitvec_sort b1
6  return s1@b1
7  array_sort b1 s1@b1 s1@b1
8  return s3
9  array b1 s3@b1 array1
10 return e2@b1
11 var b1 s1@b1 index1
12 return e3@b1
13 var b1 s1@b1 index2
14 return e4@b1
15 read b1 e2@b1 e3@b1
16 return e6@b1
17 read b1 e2@b1 e4@b1
18 return e8@b1
19 eq b1 e3@b1 e4@b1
20 return e9@b1
21 ne b1 e6@b1 e8@b1
22 return e-10@b1
23 assert b1 e9@b1
24 assume b1 e-10@b1
25 sat b1
26 return 20
27 failed b1 e-10@b1
28 return true
29 sat b1
30 return 10
31 release b1 e2@b1
32 release b1 e3@b1
33 release b1 e4@b1
34 release b1 e6@b1
35 release b1 e8@b1
36 release b1 e9@b1
37 release b1 e-10@b1
38 release_sort b1 s1@b1
39 release_sort b1 s3@b1
40 delete b1
```

Figure 3: An example API trace as generated by Boolector.

3.2 Shadow Clone Testing

As of version 2.0, Boolector provides a cloning feature which allows to generate a disjunct copy of a Boolector instance [16]. A clone captures the current state of the solver and can be either a deep copy (full clone) or a term layer copy (term layer clone) of the original instance. As a deep copy, a *full clone* includes the underlying SAT solver and the (bit-blasted) AIG layer (if present) and requires corresponding cloning support of the SAT solver back-end (e.g., Lingeling [7,8]). A full clone is required to behave exactly the same as the instance it has been cloned from. Generating full clones for producing independent subproblems is, e.g., one of the key requirements for the work splitting approach implemented in PBoolector [18], a parallel prototype implementation of Boolector. A *term layer clone*, on the other hand, only copies the term layer of the original instance, which does not guarantee exact same behavior but is sufficient for many applications (e.g., generating a dual solver instance for the dual-propagation-based optimization of the lemmas on demand approach described in [15]).

In order to test and guarantee that a full clone behaves *exactly* the same as the instance it has been cloned from, BtorMBT provides a dedicated *shadow clone* test setting similar to shadow clone testing as implemented for Lingeling. Shadow clone testing is randomly enabled and when enabled, BtorMBT initially generates a full clone (the shadow clone) of the current Boolector instance, which then mirrors every API call to the original instance and cross-checks return values for equivalence. Additionally, Boolector implements extensive checks for freshly generated clones and internally checks the state of the shadow clone after each API call. A shadow clone may be initialized anytime prior to the first SAT call and is usually randomly released and regenerated multiple times after being initialized, at different stages during one test round, to prevent that clones are only generated and checked prior to (incremental) API calls.

4 API Trace Execution with BtorUntrace

Our SMT solver Boolector allows to record all API calls with their arguments to a trace file, which then serves as input for our trace execution tool BtorUntrace. An example of an API trace generated by Boolector is given in Fig. 3, with each line of the trace either listing an API call or the return value of an API call in chronological order. A line representing an API

call consists of an identifier, the Boolector instance to issue the call to, and the arguments to the call. A line representing the return value of an API call must immediately follow the line of the call and consists of the keyword `return` and the return value, which can either be an identifier or a numerical value. As an example, consider the API call in line 7 and its return value in line 8. Identifier `array_sort` in line 7 refers to the API call to create an array sort with bit-vector sort s1 as its first (index sort) and second (element sort) argument, issued to Boolector instance b1. Line 8 identifies the return value of this call as array sort s3.

BtorUntrace is a dedicated tool for replaying traces generated by Boolector and tightly integrates Boolector via its C API. In our model-based API testing workflow, BtorUntrace is used in combination with BtorMBT to reproduce faulty behavior when a test case generated by BtorMBT fails. However, BtorUntrace is also useful outside of our testing workflow when debugging undesired behavior triggered by any (real world) application of Boolector. Since BtorUntrace only requires the API trace to replay a faulty run of Boolector, it is, e.g., not necessary to have the original (possibly complex) setup of the tool chain available for debugging purposes. Further, some errors triggered via the API may not be triggered with a dump of the corresponding input formula since some (sequences of) Boolector API calls can not be expressed in the input file formats it supports.

5 API Error Trace Minimization with ddMBT

Our delta debugger ddMBT minimizes a given API error trace while preserving its failure-inducing characteristics based on the exit code and error message produced by Boolector when replaying the (minimized) trace with BtorUntrace. Trace minimization with ddMBT works in rounds until fixpoint, with each round divided into three phases. In the first phase, lines of the trace file are eliminated in a divide-and-conquer manner similar to the original delta debugging algorithm proposed in [13]. In the second and third phase, children of terms are substituted with fresh variables and already existing expressions of the same sort. These three substitution strategies are in practice usually sufficient to obtain a trace file small enough to allow efficient debugging. In some cases, however, modifying numeric parameters such as bit-widths, shift widths, or indices for slicing might be beneficial. We leave these enhancements to future work.

6 Experimental Evaluation

In the following, similarly as in [1], we evaluate the effectiveness of our model-based API tester BtorMBT in terms of code coverage, throughput and the success rate when inserting defects into the code of Boolector. We further compare the performance of BtorMBT to grammar-based input fuzz testing, in particular to FuzzSMT [9, 10], the only currently available input fuzzer for SMT. Note that since FuzzSMT originally is an input fuzzer for the SMT-LIB v1 [17] language, we applied an available patch [19] for SMT-LIB v2 [2] support. However, this patch only provides SMT-LIB v2 compliant output of SMT-LIB v1 test cases. As a consequence, extensions of the language introduced in SMT-LIB v2, e.g., support for incremental solving, are not included, which may have a considerable impact on the performance of the tool. Measuring this impact without extending the tool to support the full feature set of the SMT-LIB v2 language, however, is difficult. We still include a comparison with FuzzSMT since up until now it was the de facto state-of-the-art for generating random test cases in SMT, and leave the extension of the tool to fully support SMT-LIB v2 to future work.

6.1 Configuration

Since we aim to evaluate BtorMBT and FuzzSMT on as even terms as possible, we provide a script for FuzzSMT that simulates option fuzzing and the round-based behavior as implemented in BtorMBT. In the following, we refer to this script as FuzzSMT, and compare against the version of BtorMBT released together with the current version 2.4 of Boolector.

We optionally switch off option fuzzing (while still randomly choosing solver engines and SAT solvers) and refer to BtorMBT and FuzzSMT without option fuzzing as BtorMBTno and FuzzSMTno. Note that we compiled Boolector with all three supported SAT solvers Lingeling [8], PicoSAT [5] and MiniSat [12].

We evaluated BtorMBT and FuzzSMT on runs of 100k rounds and chose a time limit of 2 seconds per round. Note that since increasing this time limit did not increase code coverage for 100k rounds for configuration BtorMBT, we chose this limit as a good compromise between throughput and test coverage.

In order to be able to determine if either of the tools is able to identify faulty mutations of Boolector (see Section 6.4), we thoroughly tested our base version 2.4 of Boolector prior to our experimental evaluation and run 10M rounds with each BtorMBT and FuzzSMT, all of which did not result in a single error.

We performed all our experiments on a cluster with 30 nodes of 2.83 GHz Intel Core 2 Quad machines running Ubuntu 14.04.5 LTS.

6.2 Code Coverage

We used the tool gcov of the GNU Compiler Collection (GCC) suite to determine the code coverage over 100k non-faulty rounds of BtorMBT and FuzzMBT with and without option fuzzing. The evolution of line coverage for configurations BtorMBT, BtorMBTno, FuzzSMT and FuzzSMTno as measured by gcov over 100k rounds is illustrated in Fig. 4.

After 10k runs, BtorMBT already achieves 75% line coverage without option fuzzing, and 87% when option fuzzing is enabled. FuzzSMT, on the other hand, reaches a coverage of 64% and 72%. After 100k runs, BtorMBT improves coverage up to 78% (+3%) and 90% (+12%), while FuzzSMT achieves a coverage of 65% (+1%) and 73% (+1%).

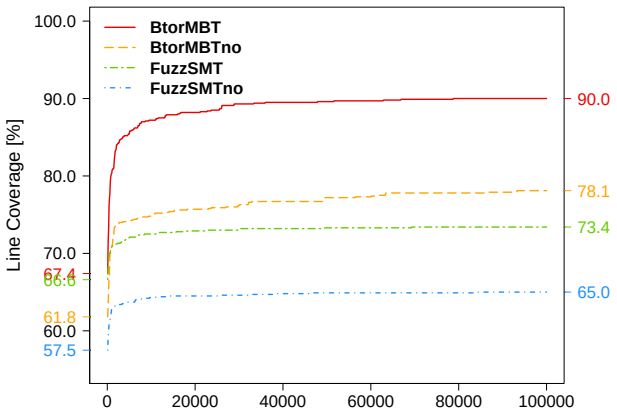

Figure 4: Code coverage evolution over 100k rounds.

Unsurprisingly, BtorMBT covers more than 98% of Boolector's API, with error handling code as the uncovered rest that is not triggered due to the fact that all test cases were error free. FuzzSMT, on the other hand, only achieves a coverage of 52%, which is likely to be improved by introducing full SMT-LIB v2 support, however, not up to the coverage rate BtorMBT achieved. Note that for both tools, a coverage rate of 100% for error free test cases is in general impossible due to the fact that error handling code is not triggered. Further, since FuzzSMT only generates input in SMT-LIB v2 format, all code related to parsing BTOR format (3%) remains unused.

6.3 Throughput

When fuzz testing an SMT solver, no matter if it is input fuzzing or API fuzzing, the number of tests completed within a certain time frame (the throughput) is an important measure of efficiency and effectiveness of the test method. A high number may indicate that the generated test cases are too trivial and therefore less likely to trigger errors. A low number, on the other hand, may be caused by too difficult and therefore too time consuming test cases, which may considerably slow down progress when testing. We aim to perform as many good test cases, i.e., test cases with a high code coverage rate, in as little time as possible, which is a balancing act between the two extremes above.

For 100k rounds, BtorMBT achieves a throughput of on average 45 rounds per second, which increases by 20% when shadow clone testing is disabled. FuzzSMT, on the other hand, achieves a far lower throughput of 7 test cases per second since it first generates an input file that is then handed to the SMT solver. Further, FuzzSMT is written in Java, and (re)starting the Java VM in each round introduces additional overhead which further decreases throughput.

Note that in 100k rounds with BtorMBT, 20% of all calls to determine satisfiability are incremental. Further, one in four solved instances is satisfiable, which corresponds to a rather unbalanced ratio of 1:3 of satisfiable to unsatisfiable instances. We leave improving this ratio to future work.

6.4 Defect Injection

In our final experiment, we evaluated the success rate of BtorMBT and FuzzSMT in identifying faulty configurations of Boolector. We compiled a set of test configurations TC (4626 in total), which consists of two subsets TC_A and TC_D and contains configurations where we introduced artificial defects into the source code of Boolector. Set TC_A contains 2305 configurations with a randomly inserted abort statement, and set TC_D consists of 2321 configurations where we deleted a random statement from the code. All 4626 configurations in set TC are faulty configurations. However, some defects, e.g., modifications of heuristics due to a missing statement, may result in performance bugs rather than producing incorrect results or any other erroneous behavior and are therefore impossible to detect with either test method.

For each faulty test configuration, we set a limit of 100k rounds for BtorMBT. However, since the low throughput of FuzzSMT (7 rounds per second) would require too much runtime for our experiment with 100k rounds even on a cluster with 30 nodes (26 days in the worst case), we limited the number of rounds for FuzzSMT to 10k and compare its results to the number of faulty configurations identified by BtorMBT within 10k rounds.

Table 1 shows the number of faulty configurations identified by BtorMBT and FuzzSMT with and without option fuzzing within 100k and 10k rounds. Overall, within 10k rounds BtorMBT has a 11% higher success rate than FuzzSMT, which is increased by 14% to 80.4% when the limit is extended to 100k rounds. Disabling option fuzzing, on the other hand, decreases the number of configurations identified as faulty for both tools by 12%.

Not surprisingly, for both tools the success rates for configuration set TC_A correspond to their code coverage as determined in Section 6.2. The number of successfully identified faulty configurations in set TC_D, on the other hand, is significantly lower due to the fact that set TC_D contains test cases with defects that concern error handling code or decrease performance rather than introducing erroneous behavior. Further, in case of BtorMBT, since dumps in AIGER format are not tested for correctness it is not possible to detect configurations that produce incorrect AIGER output. The same applies in case of FuzzSMT for configurations that produce incorrect dump output in any format since it is not checked for correctness.

		BtorMBT		**BtorMBTno**		**FuzzSMT**		**FuzzSMTno**	
		Found	[%]	Found	[%]	Found	[%]	Found	[%]
100k	TC_A (2305)	2088	90.6	1789	77.6				
	TC_D (2321)	1629	70.2	1366	58.9				
	TC (4626)	**3717**	**80.4**	**3155**	**68.2**				
10k	TC_A (2305)	2028	88.0	1719	74.6	1735	75.3	1523	66.1
	TC_D (2321)	1510	65.1	1277	55.0	1304	56.2	1153	49.7
	TC (4626)	**3538**	**76.5**	**2996**	**64.8**	**3039**	**65.7**	**2676**	**57.8**

Table 1: Number of faulty configurations of Boolector identified by BtorMBT, BtorMBTno, FuzzSMT and FuzzSMTno within 100k and 10k rounds.

7 Conclusion

In this paper, we presented a model-based API testing tool set for our SMT solver Boolector. It consists of several dedicated tools, the model-based API tester BtorMBT, the API trace exection tool BtorUntrace and the API trace minimizing tool ddMBT.

Our API fuzzer BtorMBT generates random but valid sequences of calls to Boolector's API and allows to test random configurations of Boolector on random input formulas. With a success rate of 80% on our artificial set of faulty configurations of Boolector and a line coverage of 90% over 100k rounds, our experiments suggest that BtorMBT is an effective method for testing Boolector. Our extremely positive practical experience confirms this claim. Based on our results, we believe that applying our techniques to other SMT solvers is effective in general. An interesting direction for future work would be introducing symbolic execution techniques into BtorMBT in order to be able to direct trace generation towards maximal code coverage.

Our model-based API testing framework is the core component of the testing workflow in the development process of Boolector. However, it still needs to be complemented by several other tools to cover cases that can not be tested with BtorMBT alone. The solver front end, for example, can only be tested by using the solver as standalone tool with files in BTOR or SMT-LIB format as input. For that purpose, we use a suite of regression tests and FuzzSMT, even though its support for the SMT-LIB v2 format is incomplete. Another example is parser testing, which is incomplete with BtorMBT since Boolector is currently not able to dump incremental input and in general does not use the full feature set of the SMT-LIB language when dumping. Further, dumping with Boolector only produces valid input files. However, a parser must be tested for correct parse error handling on invalid input, too. We test Boolector's parsers by means of the tool fzsexpr of the ddsexpr tool set [6], which generates (mostly) invalid input by mutating existing files based on lines, S-expressions and characters.

Currently, BtorMBT produces a rather unbalanced ratio of 1:3 of satisfiable to unsatisfiable instances. Further, AIGER dump output is not checked for correctness since Boolector does not allow to parse AIGER input files. We leave improving the ratio of satisfiable to unsatisfiable instances and checking the correctness of AIGER dumps to future work.

References

[1] Cyrille Artho, Armin Biere, and Martina Seidl. Model-based testing for verification back-ends. In Margus Veanes and Luca Viganò, editors, *Tests and Proofs - 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings*, volume 7942 of *Lecture Notes in Computer Science*, pages 39–55. Springer, 2013.

[2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at `www.SMT-LIB.org`.

[3] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at `www.SMT-LIB.org`.

[4] Armin Biere. AIGER Format and Toolbox. `http://fmv.jku.at/aiger`.

[5] Armin Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008.

[6] Armin Biere. ddsexpr. `http://fmv.jku.at/ddsexpr`, 2013.

[7] Armin Biere. Yet another Local Search Solver and Lingeling and Friends Entering the SAT Competition 2014. In Adrian Balingt, Anton Belov, Marijn Heule, and Matti Järvisalo, editors, *SAT Competition 2014*, volume B-2014-2 of *Department of Computer Science Series of Publications B*, pages 39–40. University of Helsinki, 2014.

[8] Armin Biere. Splatz, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2016. In Tomáš Balyo, Marijn Heule, and Matti Järvisalo, editors, *SAT Competition 2016 – Solver and Benchmark Descriptions*, volume B-2016-1 of *Department of Computer Science Series of Publications B*, pages 44–45. University of Helsinki, 2016.

[9] Robert Brummayer. FuzzSMT. `http://fmv.jku.at/fuzzsmt`, 2009.

[10] Robert Brummayer and Armin Biere. Fuzzing and Delta-Debugging SMT Solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories (SMT'09)*, page 5. ACM, 2009.

[11] Robert Brummayer, Armin Biere, and Florian Lonsing. BTOR: Bit-Precise Modelling of Word-Level Problems for Model Checking. In *Proceedings of the 1st International Workshop on Bit-Precise Reasoning, BPR 2008, affiliated with the 20th International Conference on Computer Aided Verification, CAV 2008, Princeton, NJ, USA, July 14, 2008*, 2008.

[12] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[13] Ralf Hildebrandt and Andreas Zeller. Simplifying failure-inducing input. In *ISSTA*, pages 135–145, 2000.

[14] Aina Niemetz and Armin Biere. ddSMT: A Delta Debugger for the SMT-LIB v2 Format. In *Proceedings of the 11th International Workshop on Satisfiability Modulo Theories (SMT'13), affiliated to SAT'13, Helsinki, Finland*, pages 36–45, 2013.

[15] Aina Niemetz, Mathias Preiner, and Armin Biere. Turbo-charging lemmas on demand with don't care reasoning. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 179–186. IEEE, 2014.

[16] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation JSAT*, 9:53–58, 2015.

[17] Silvio Ranise and Cesare Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006. Available at `www.SMT-LIB.org`.

[18] Christian Reisenberger. PBoolector: A Parallel SMT Solver for QF_BV by Combining Bit-Blasting with Look-Ahead. Master's thesis, Johannes Kepler University Linz, 2014.

[19] Christoph Wintersteiger. Patch for FuzzSMT to produce SMT-LIB v2 output. `http://fmv.jku.at/fuzzsmt/fuzzsmt-smt2.patch.gz`, 2012.