SpyBug: Automated Bug Detection in the Configuration Space of SAT Solvers

Norbert Manthey[1]* and Marius Lindauer[2]**

[1] Technische Universität Dresden, Dresden, Germany
[2] University of Freiburg, Freiburg, Germany

Abstract. Automated configuration is used to improve the performance of a SAT solver. Increasing the space of possible parameter configurations leverages the power of configuration but also leads to harder maintainable code and to more undiscovered bugs. We present the tool `SpyBug` that finds erroneous minimal parameter configurations of SAT solvers and their parameter specification to help developers to identify and narrow down bugs in their solvers. The importance of `SpyBug` is shown by the bugs we found for four well-known SAT solvers that won prices in international competitions.

1 Introduction

Recently, algorithm configuration [12] has shown to considerably improve the performance of SAT solvers by tuning their parameters, e.g. in the Configurable SAT Solver Challenge (CSSC; [13]). As a consequence, SAT solvers became highly flexible and configurable such that their performance can be optimized for broad range of different SAT formulas. One well-known example is the solver *Lingeling* [4] that exposes 323 parameters to the user and gave rise to 10^{1341} possible parameter configurations in the CSSC'14. In combination with its already strong default performance, *Lingeling* won the industrial track of the CSSC'14.

This flexibility of solvers comes with a price: as it is infeasible to assess the performance of all possible parameter settings, it is also infeasible to verify the correctness of all these settings. The high number of parameters of SAT solvers motivated this work, as we expect to find untested (possibly buggy) configurations when searching long enough. Since the components of a SAT solver highly interact with each other and due to their increasing size and complexity, it is hard to apply traditional methods aiming at formally proving SAT solver to be bug free (see Section 2). Therefore, we propose a new tool, called `SpyBug` (see Section 3) that searches in the space of possible parameter settings and instances to find erroneous solver runs that unexpectedly terminated or returned a wrong proof. Since SAT solvers have many parameters and in most cases only a few parameters actually trigger the bug (often 2 - 10 out of hundreds of parameters), we furthermore use a heuristic to minimize the parameter configuration

* N. Manthey was supported by the DFG grant HO 1294/11-1.
** M. Lindauer was supported by the DFG under Emmy Noether grant HU 1900/2-1.

by removing all parameters that are not necessary for the bug. This automatic procedure helps the developer to narrow down the reason of the bug. In Section 4, we demonstrate the effectiveness of `SpyBug` on the participants of the industrial track of the CSSC 2014 and show that in 4 out of the 6 participants we were able to find bugs which can be often explained by less than 10 parameters. With a few modifications, `SpyBug` could also be applied to other problem domains, e.g. mixed integer programming (MIP), maximum satisfiability (MaxSAT), quantified Boolean formula (QBF) or SAT modulo theory (SMT).

2 Related Work: Bug Detection in Software

Bugs in software are found in several ways. On one hand, *formal methods* can be used to verify the software with respect to a specification, or to formally prove that certain error states cannot be reached. Typically recognizable states are assertions in the code, numerical overflows, or passing memory bounds. Similar bugs are found with *symbolic execution*, or by *fuzz testing*. Finally, we can use *unit tests* to test single components of a software system. A survey on formal methods for software verification can be found in [20].

Depending on the size of the software system under test, certain methods are not feasible due to their computational complexity. Unit tests are possible as soon as there are single components in a system. Here, for example the solvers data structures could be tested. Fuzz testing executes the software binary many times and checks each run for errors. For SAT solvers, there exists the fuzz testing suite [7, 1], whose latter versions also support passing parameters to the used SAT solver and minimizing faulty combinations. However, the required feature to parse parameters from the formulas is not supported by many solvers. The approach of `SpyBug` treats the software under examination as a black box. Instead of the binary, the API of a solver library can be tested [7].

Formal methods have been applied to write verified SAT solvers with clause learning: F. Marić [16] specified a SAT solver in the *Isabelle* interactive theorem prover and then a solver implemented in *Haskell* is created. The SAT solver *versat* [17] is created and verified by using the language *Guru* and translating the code to *C*. In both projects, many techniques of recent systems have not been implemented, and there is a performance gap to modern systems.

For symbolic execution and proving the reachability of erroneous states it is hard to add assertions that check whether a solver state is valid. Therefore, we currently focus on improved fuzz testing, as we want to check the correctness of a highly configurable system without modifying the tool itself. With fuzz testing, we only require that there exists a formal specification of the parameters of the solver, and a standard input format and output format.

因此，我们目前将重点放在改进的模糊测试上，因为我们希望在不修改工具本身的情况下检查高度可配置系统的正确性。在模糊测试中，我们只要求存在求解器参数的形式化规范，以及标准的输入格式和输出格式。

Hutter et al. [10] implemented a similar approach as used in `SpyBug` to identify bugs in two MIP solvers. They modified the algorithm configurator *ParamILS* [12] to record erroneous configurations and to minimize them. The main difference to `SpyBug` is that *ParamILS* will find bugs mostly in well-performing areas of the configuration space since its goal is not to find bugs but to

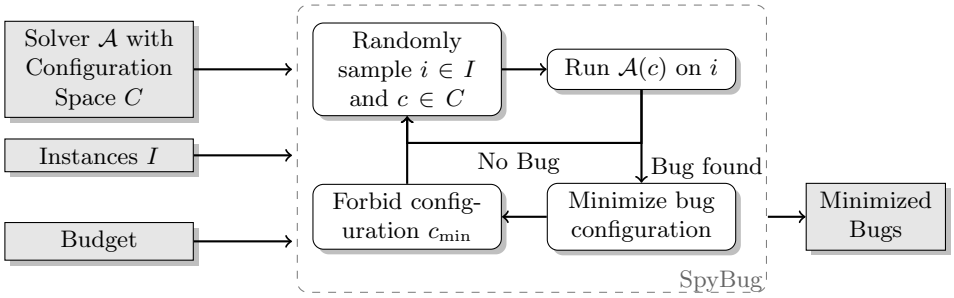

Fig. 1. Workflow of `SpyBug`

optimize the performance of a solver. As *ParamILS* aims at optimization, it is more complicated to set up *ParamILS* compared to setting up `SpyBug`.

3 `SpyBug`'s Framework

The tool `SpyBug` is supposed to help developers by finding bugs in large configuration spaces. We require only a parameter specification, the solver binary and a list of formulas to perform testing.[3] In the following sections, we explain the internals of `SpyBug` and present a use case. The tool is open-source under GPLv2 and available at `http://www.ml4aad.org/spybug/`.

3.1 Workflow

Since the idea of `SpyBug` is to search in the space of parameter configurations, it is built upon a simplified subset of the already well-known interfaces of the state-of-the-art algorithm configurators *ParamILS* [12] and *SMAC* [11] that was also used in the Configurable SAT Solver Challenge [13]. Hence, developers already using algorithm configuration can use `SpyBug` directly out-of-the-box. As illustrated in Figure 1, the input of `SpyBug` is a solver with its configuration space, a set of instances to run the solver on, and a budget how long `SpyBug` runs (i.e., a wall clock time budget or a maximal number of solver runs). In the end, `SpyBug` returns a list of all minimized found erroneous configurations of the given solver.

The main workflow of `SpyBug` consists of a loop:

1. It samples an instance and a valid configuration uniformly at random.
2. It runs the solver with the configuration on the instance with some resource limits (i.e., the used runtime cutoff or memory limit). The solver is wrapped by the so-called *generic wrapper*, which was used in the CSSC and which is responsible to determine whether it was a valid run or not. In a valid

[3] Such a collection could be generated with fuzzing tools [7].

run, the solver either returned `SATISFIABLE` with a corresponding model or `UNSATISFIABLE` – to verify the output, we check the returned model – or it violated the resource limits. Hence, segmentation faults or failed assertions are also considered as bugs.
3. If it was a valid run, we continue with Step 1. Otherwise, we first minimize the found bug and then forbid the minimized configuration (or any super set of this configuration) to be sampled again. If the minimized configuration is empty (i.e., the bug is also triggered by the default configuration), we remove the instance from the instance set because we assume that the solver will always fail to run on this instance.

The *generic wrapper* judges a run based on the output of the solver, and the report of the *runsolver* tool [19]. By replacing the SAT-specific component of this script, `SpyBug` can easily be adapted to other types of solvers, for example MaxSAT solvers, Pseudo-Boolean solvers, QBF solver or SMT solvers.

3.2 Bug Minimization

The minimization of a configuration that triggered a bug is important to narrow down the reason for this bug. In the end, we are interested in the changed error-prone parameters with respect to the default configuration of a solver since the default configuration is often well tested and well studied by the developers. We therefore apply two iterations of a backward elimination, i.e., we iterate over all changed parameters and flip them to the default if the bug remains. The second iteration is done in reversed order of the parameters because a single traversal of the parameters could keep many non-relevant parameters. In preliminary tests, many further non-relevant parameters were removed with the help of a reverse traversal. To not spend too much time on the minimization, we do not use a full minimization procedure.

In this minimization process, we have to consider that the parameter configuration space (pcs) of a solver could be structured. According to the used pcs-format[4], the configuration space can consist of forbidden parameter combinations and conditional parameter clauses. For the first, we simply have to ensure that we do not violate these forbidden combinations. In contrast, conditional parameter clauses define when a parameter is active (e.g., a parameter of a heuristic is only active if this heuristic is indeed used). If we flip a parameter value we check which parameters are inactive and which ones need to be activated. Since a parameter will be set to its default value when it gets activated, the number of parameters to activate does not increase by flipping parameters during minimization.

In order to not find the same faulty configuration twice, the obtained combination of parameters is disallowed for all further solver calls. We furthermore remove all continuous parameters from this parameter configuration, as for such a kind of parameter there might be too many other values that would lead to the same fault in the solver.

[4] `http://www.cs.ubc.ca/labs/beta/Projects/SMAC/v2.08.00/manual.pdf`

Table 1. Overview of participants of the industrial track of the Configurable SAT Solver Challenge 2014. For illustration, the number of configurations refers to a discretized version of the configuration space where each parameter has at most 8 possible values.

Solver	# Parameters	# Configurations	Reference
Minisat-HACK-999ED	10	$8 \cdot 10^5$	[18]
Cryptominisat	36	$3 \cdot 10^{24}$	[21]
Clasp-3.0.4-p8	75	$1 \cdot 10^{49}$	[9]
Riss-4.27	214	$5 \cdot 10^{86}$	[15]
SparrowToRiss	222	$1 \cdot 10^{112}$	[2]
Lingeling	323	$2 \cdot 10^{1341}$	[5]

3.3 An Exemplary Use Case

We present a small use case for the solver *Riss-4.27* [15] (RISS) with its 214 parameters (specified in the file RISS.PCS). We use the solver and parameter specification that were submitted to the CSSC 2014 [13]. As a benchmark set, we use *Circuit Fuzz* formulas [7].

When starting `SpyBug` for the submitted version of *Riss-4.27* with the above setup, the following lines will be printed.

```
found a bug in -agil-r no -probe yes [...] on fuzz_100_20562.cnf
[...]
Minimal bug config: -cp3_strength no [...] -inprocess yes
```

The example highlights the major properties of `SpyBug`: when it finds a bug, it reports the full parameters that have been passed to the solvers wrapper script.[5] Furthermore, the formula that triggered the bug is printed (`SAT_dat.k50.cnf`). Next, the minimized list of parameters is printed. For the given example, the parameter `-agil-r` is not set any longer in the minimized set, as the bug is reproducible with the default value for this parameter. This combination is reported to the user, as shown in the example above.

4 Case Study: Configurable SAT Solver Challenge

To demonstrate the usefulness of `SpyBug`, we evaluated the 6 solvers that have been submitted to industrial track of the CSSC 2014 [13]. We independently analyze the solvers on the three benchmark families, i.e., hardware verification (*IBM* [22]), circuit fuzz (*CF* [7]), and bounded model checking (*BMC* [6]). Similar to automated configuration, we repeat the analysis for the combination of a solver and a benchmark family four times. The execution of a solver on a formula is limited to 300 CPU seconds and 3 GB RAM. We stop the analysis as soon as

[5] When reproducing the buggy configuration with the native binary, the parameters that might be added to the solver call in the wrapper script must be considered.

Table 2. Number of found bugs and their minimized average sizes (± standard deviation) in the configuration space after 4 independent runs of `SpyBug` for at most 2 days or 10.000 randomly sampled configuration-instance pairs.

	# Bugs			∅ Bugs		
	IBM	*CF*	*BMC*	*IBM*	*CF*	*BMC*
Minisat-HACK-999ED	0	0	0	–	–	–
Cryptominisat	2	0	7	2 ± 0	–	3 ± 2
Clasp-3.0.4-p8	0	0	0	–	–	–
Riss-4.27	3	5	2	5 ± 1	13 ± 6	6 ± 4
SparrowToRiss	0	0	1	–	–	7 ± 0
Lingeling	0	2	0	–	8 ± 2	–

we tested 10 000 random configurations, or when reaching 2 days wall clock time. We communicated the found erroneous configurations to the solver developers.

The experiment has been executed on a cluster with 64 GB RAM that is shared among two Intel Xeon E5-2650v2 8-core CPUs with 20 MB L2 cache; running Ubuntu 14.04 LTS 64 bit. Table 1 presents the 6 solvers of the analysis and their number of parameters. While larger configuration spaces might yield better configurations, verifying the absence of bugs becomes harder.

Next, we present a summary of the bugs we found during the analysis in Table 2. For *Minisat-HACK-999ED* and *Clasp-3.0.4-p8*, we did not find a bug during our experiments. *Lingeling* has two misbehaving configurations for the *CF* family and *SparrowToRiss* shows bugs for the *BMC* family. For *Cryptominisat* and *Riss-4.27*, more than 5 bugs could be revealed on different families. The table furthermore presents how many parameters have to be changed from the default configuration to reach the faulty configuration. We note that `SpyBug` minimized the erroneous configurations of *Lingeling* with 323 parameters down to 8 parameters on average which drastically reduces the possible reasons of the responsible bugs. `SpyBug` got the best narrowing of responsible parameters for *Cryptominisat* for which only 2 or 3 parameters on average were necessary.

5 Conclusion

State-of-the-art SAT solvers contain many different techniques such as simplification during search. Combining only a few of these techniques can already result in unsound sequential systems [14]. Implementing competitive SAT solvers is a difficult task, especially if the system should be tuned for different applications. We presented the tool `SpyBug`, which orthogonally to existing tools finds bugs in the configuration space of the SAT solver. One specific use-case of `SpyBug` is to reveal bugs in solvers, before tuning them with many expensive resources. We reported bugs for four SAT solvers that won first prices in international competitions in 2014. For the future, we plan to integrate `SpyBug` into the framework of *SpySMAC* [8], i.e., an automatic framework to tune and analyse of SAT solvers.

References

1. Artho, C., Biere, A., Seidl, M.: Model-based testing for verification back-ends. In: Proc. of TAP'13. pp. 39–55 (2013)
2. Balint, A., Manthey, N.: SparrowToRiss. In: Belov et al. [3], pp. 77–78
3. Belov, A., Diepold, D., Heule, M., Järvisalo, M. (eds.): Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions, Department of Computer Science Series of Publications B, vol. B-2014-2. University of Helsinki (2014)
4. Biere, A.: Lingeling, plingeling and treengeling entering the sat competition 2013. In: Proc. of SAT Competition 2013. pp. 51–52 (2013)
5. Biere, A.: Yet another local search solver and Lingeling and friends entering the SAT competition 2014. In: Belov et al. [3], pp. 39–40
6. Biere, A., Cimatti, A., Claessen, K.L., Jussila, T., McMillan, K., Somenzi, F.: Benchmarks from the 2008 hardware model checking competition (HWMCC'08) (2008), available at `http://fmv.jku.at/hwmcc08/benchmarks.html`
7. Brummayer, R., Lonsing, F., Biere, A.: Automated testing and debugging of SAT and QBF solvers. In: Proc. of SAT'10, pp. 44–57
8. Falkner, S., Lindauer, M., Hutter, F.: SpySMAC: Automated configuration and performance analysis of SAT solvers. pp. 1–8
9. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artificial Intelligence 187-188, 52–89 (2012)
10. Hutter, F., Hoos, H., Leyton-Brown, K.: Automated configuration of mixed integer programming solvers. In: Proc. of CPAIOR'10. pp. 186–202 (2010)
11. Hutter, F., Hoos, H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Proc. of LION'11. pp. 507–523 (2011)
12. Hutter, F., Hoos, H., Leyton-Brown, K., Stützle, T.: ParamILS: An automatic algorithm configuration framework. JAIR 36, 267–306 (2009)
13. Hutter, F., Lindauer, M., Balint, A., Bayless, S., Hoos, H.H., Leyton-Brown, K.: The Configurable SAT Solver Challenge. CoRR (2015), `http://arxiv.org/abs/1505.01221`
14. Järvisalo, M., Heule, M., Biere, A.: Automated Reasoning: 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings, chap. Inprocessing Rules, pp. 355–370. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
15. Manthey, N.: Riss 4.27. In: Belov et al. [3], pp. 65–67
16. Marić, F.: Formal verification of a modern SAT solver by shallow embedding into isabelle/hol. Theoretical Computer Science 411(50), 4333–4356 (2010)
17. Oe, D., Stump, A., Oliver, C., Clancy, K.: versat: A verified modern SAT solver. In: Kuncak, V., Rybalchenko, A. (eds.) Proc. of VMCAI 2012. pp. 363–378. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
18. Oh, C.: MiniSat HACK 999ED, MiniSat HACK 1430ED and SWDiA5BY. In: Belov et al. [3], p. 46
19. Roussel, O.: Controlling a solver execution with the runsolver tool. Journal on Satisfiability, Boolean Modeling and Computation 7(4), 139–144 (2011)
20. Silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 27(7), 1165–1178 (2008)
21. Soos, M.: CryptoMiniSat v4. In: Belov et al. [3], p. 23
22. Zarpas, E.: Benchmarking SAT solvers for bounded model checking. In: Proc. of SAT'05. pp. 340–354 (2005)