Alt-Ergo-Fuzz: A fuzzer for the Alt-Ergo SMT solver

Hichem Rami Ait El Hara, Guillaume Bury, and Steven de Oliveira

OCamlPro, Paris, France

Abstract

Alt-Ergo is an open source Satisfiability Modulo Theories (SMT) solver programmed in OCaml. It was designed for program verification and it's used as a back end by other software verification tools such as Frama-C, SPARK, Why3, Atelier-B and Caveat, the reliability of which depends on the soundness of Alt-Ergo's answers and the absence of bugs in it.

Fuzzing is an efficient technique to test programs and find bugs. It works by quickly and automatically generating input data with which to test the software. American Fuzzy Lop (AFL) is one of the most well-known and most used fuzzers in both academia and the industry. It has managed to find many bugs in various programs thanks to its grey box fuzzing technique that uses genetic algorithms and program instrumentation to generate test data that maximizes code and execution path coverage in the targeted software.

In this paper we present Alt-Ergo-Fuzz, a fuzzer for Alt-Ergo that we developed with the aim of finding faults and unsoundness bugs to solve and improve its reliability. By using AFL as a back end, the Crowbar OCaml library for test case generation and the CVC5 SMT solver as a reference solver of which the answers will be used to determine whether or not Alt-Ergo's answers are correct, we managed to develop Alt-Ergo-Fuzz, which even as a work in progress and in only twenty days of testing managed to find four never found before bugs in Alt-Ergo.

1 Introduction

Computer systems are entrusted with various tasks, such as transportation, financial operations and industrial production. The roles played by software are more important and critical than ever before which means that the presence of bugs in such software could lead to serious consequences. To guarantee the absence of bugs in programs, various tools using formal methods were developed in the last decades, such as model checkers, static analysers and deductive verification platforms. These tools tend to generate a large amount of logical formulas whose validity needs to be verified. SMT solvers [2], thanks to their expressivity and efficiency, are well-adapted for such tasks. Alt-Ergo [6] is one of these SMT solvers, it was designed for program verification and it is used to determine the validity of logical formulas. Our purpose is to test it efficiently and to detect the eventual bugs it might contain. After seeing how well fuzzers managed to find bugs in other SMT solvers, mainly CVC4 [3] and Z3 [10], we decided to develop Alt-Ergo-Fuzz[1], which is the first fuzzer that was developed for Alt-Ergo.

Related work Other fuzzers were developed to test the CVC4 and Z3 SMT solvers. One of them is OPFuzz [12], it works by mutating existing SMT formulas. The mutations are done by replacing operations or function calls in the SMT formulas by other ones while preserving the well-typedness of the formulas. Another one is YinYang [13], a fuzzer that uses semantic fusion to build test oracles, then verify that the SMT solver decides the right satisfiability for the formula. The semantic fusion in question is done by taking two SMT formulas that have the same satisfiability and fusing them to produce a new formula while preserving the satisfiability,

[1] Alt-Ergo-Fuzz's sources are available at: https://github.com/hra687261/alt-ergo-fuzz

and the tested SMT solver is expected to answer with that satisfiability, if it doesn't then that is likely caused by a bug. When it comes to fuzzing OCaml programs, significant progress has been made thanks to the Crowbar [7] and Monolith [11] libraries, that make it possible to fuzz OCaml programs in a quite straightforward way.

Content of the paper In this paper, we will start by giving some necessary preliminaries about the involved tools in Alt-Ergo-Fuzz. Then we detail how it was implemented. After that we present the results of its experimentation and we talk about its limitations and the envisaged additions to it before we conclude.

2 Preliminaries

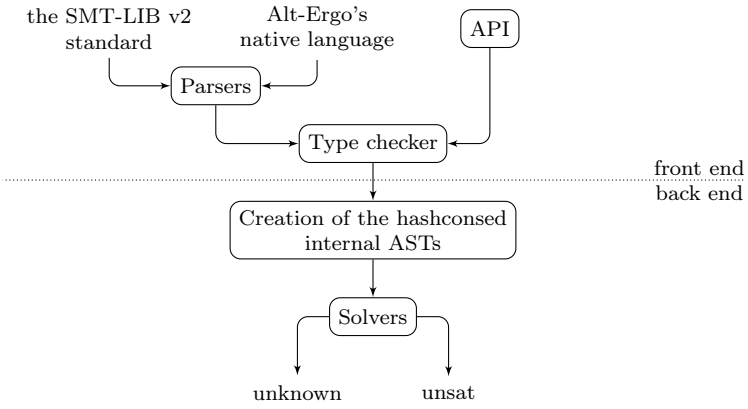

Figure 1: Alt-Ergo's architecture

The Alt-Ergo SMT solver Figure 1 represents the architecture of the Alt-Ergo SMT solver. Alt-Ergo supports both the SMT-LIB v2 standard [1], and its own native language. It can also be used as an OCaml library through its API.

Parsing a file with Alt-Ergo or building formulae using its API produces abstract syntax trees (ASTs) that are type-checked and transformed into typed ASTs. The typed ASTs are then translated into Alt-Ergo's internal language in the form of hashconsed ASTs. Alt-Ergo's core processes the resulting ASTs and uses SAT solvers, decision procedures and quantifier instantiation heuristics to decide on the satisfiability of the input SMT formulas. The result is then printed on the standard output.

Alt-Ergo has two main SAT solvers, the first one uses a Tableaux-like method and the other uses the conflict-driven clause learning (CDCL) algorithm. Each one of these solvers has a variant. In total Alt-Ergo has the following four SAT solvers:

- `Tableaux` – A functional SAT solver with Tableau boolean model simplification.
- `CDCL` – A SAT solver using the CDCL algorithm.
- `Tableaux-CDCL` – The Tableaux solver, but using a CDCL solver for boolean constraints.
- `CDCL-Tableaux` – The CDCL solver, extended with a Tableau boolean model simplification.

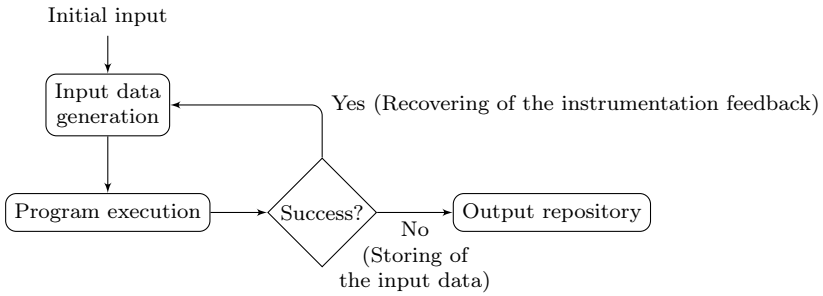

Figure 2: Functioning of AFL

Fuzz testing with AFL Fuzzing or fuzz testing is a very scalable, efficient and widely used software testing technique [8, 9]. It generally functions by continuously generating input data with which to test the software and guiding the data generation in a way that favors bug detection.

AFL [14] is a very popular fuzzer. By using code instrumentation and genetic algorithms, it managed to find many bugs in various kinds of programs. The code instrumentation allows it to get feedback on code and execution path coverage after every execution, while the genetic algorithms are used to exploit that feedback and help produce test cases that expand the code and execution path coverage.

Figure 2 illustrates the functioning of AFL. It starts by taking, as initial input, a folder containing raw data files, which are usually in the form of a random sequence of bytes that it will use as an initial test case, on which to run the targeted software. If the test succeeds, which usually means that the executable didn't crash, then the instrumentation feedback is recovered and exploited to guide future data generations. On the other hand, if the execution fails then the data on which the program was executed is stored in a file located in an output folder and the testing continues.

The Crowbar OCaml library Crowbar [7] is an OCaml library designed to do QuickCheck-like tests [5] and to help with fuzzing OCaml programs. It provides generators for OCaml's basic data types and combiners of generators that can be used to generate more complex data structures. It also allows the definition of properties that the generated data needs to verify for the tests to pass. It can also be combined with a fuzzer like AFL by using the files generated by AFL as a seed for data generation instead of generating data randomly like it's done in the classical QuickCheck testing technique. That, combined with an OCaml compiler switch that supports AFL instrumentation makes it possible to fuzz OCaml programs efficiently.

3 Implementation

In this section we present the structure of our fuzzer and how it works. We start by detailing how the data generation is done, then we describe the body of the fuzzing loop and finally we talk about how we deal with the bugs we detect.

3.1 Data generation

We defined an intermediate AST that we use as a container for the statements we generate. To make our fuzzing as efficient and as fast as possible, we made sure of generating only well-formed formulas. Well-formed means that they have to be well-typed and well-scoped. We are more interested in unsoundness bugs, as they are usually harder to detect and harder to solve, plus they could have worse consequences than other failures. The presence of unsoundness bugs usually means that the solver provides incorrect answers. By generating well-formed formulas, we can translate them directly to Alt-Ergo's internal language and provide them to the solver, without parsing or type checking them since we know that they are well-formed.

The Crowbar library provides a data type for generators that has one parameter which is the type of the data that the generator can produce. In our case, we needed to write a function called `expr_gen` that builds a generator for SMT expressions which can be used to build SMT statements. That function takes as parameters a generation context, the type of the expression we want to build a generator for, and a `fuel` parameter which is an upper bound of the depth that generated expressions can have.

The generation context is used to provide the generator with additional information that can be used in the generation of the expressions. For example, when generating the body of a function. The arguments of the function are provided to the generator so that they can be used during the generation of its body.

When `expr_gen` is called, if the `fuel` is equal to or less than zero, then it selects a generator of a terminal expression of the correct type, so it chooses either a variable generator or a literal or non-literal constant generator. Otherwise, it selects a generator of any expression of the correct type or calls a function that builds that generator. For example, if it chooses to build a generator that generates a call to some function `func`, then a function `func_call_gen` is called with the function `func`'s signature. `func_call_gen` will then call `expr_gen` with the type of each parameter of `func` and a decremented `fuel`. It can then use the generators for the arguments of `func` that it got from calling `expr_gen` to build a generator for a function call to `func`.

Well-scopedness An SMT statement is well-scoped if every identifier in the statement is either a bound variable, a declared constant or a call to a user-defined or uninterpreted function.

Variables can be bound to quantifiers, "let" expressions, patterns from a pattern matching or to a function as its parameters. In the case of quantification, the number of usable universally or existentially quantified variables by type is determined before the generation of the statements. When a statement is generated, it can contain free variables, which are then quantified, by adding a quantifier of each one of the variables. The quantifier will be positioned at the closest common predecessor (of type `bool` to keep the well-typedness) of all the instances of the variable. Which ensures that each quantified formula will be well-scoped.

When it comes to "let" expressions, the variable and its assigned value are first generated, then the body of the expression is generated with a generation context that contains the variable, which makes it possible for it to be used in the generated body. That way, it is guaranteed that each variable bound to a "let" expression will not be out of its scope.

Concerning the variables that are bound as function parameters or to patterns in a pattern matching. They are generated before the rest of the expressions are, therefore adding them as usable variables to the generation context when generating the body of the function or the consequence of the matched pattern is sufficient to guarantee that the variables will be well-scoped in the resulting expression.

The last case is the one of undeclared functions, constants or user-defined data types. The user-defined types are generated and declared before the generation of the statements so that they can be used during it, which makes their scope cover every statement. On the other hand the other identifiers that need to be declared are declared the first time a generated statement uses them. When another statement that comes after uses the same identifiers, they are not redeclared.

3.2 The body of the fuzzing loop

After being able to generate well-formed sequences of statements, it is possible to define the body of the fuzzing loop. The generated sequences of statements need to be initially translated to Alt-Ergo's internal language to make it possible to run Alt-Ergo on them and recover the answers. That makes it possible to test whether or not Alt-Ergo crashes when run on well-formed statements. So it only tests for internal crashes. To check the soundness of Alt-Ergo's answers, CVC5 [4] was chosen as a reference solver, the answers of which will be used to compare Alt-Ergo's answers to.

```
1:  function (stmts: list of SMT statements)
2:      try
3:          ae_stmts ← translate_ae(stmts)
4:          smt2_stmts ← translate_smt2(stmts)
5:
6:          cvc5_ans ← cvc5_solve(smt2_stmts) // CVC5 SMT solver
7:
8:          c_ans ← ae_c_solve(ae_stmts)       // Alt-Ergo with the CDCL solver
9:          ct_ans ← ae_ct_solve(ae_stmts)     // Alt-Ergo with the CDCL-Tableaux solver
10:         t_ans ← ae_t_solve(ae_stmts)       // Alt-Ergo with the Tableaux solver
11:         tc_ans ← ae_tc_solve(ae_stmts)     // Alt-Ergo with the Tableaux-CDCL solver
12:
13:         try
14:             cmp_answers(cvc5_ans, t_ans, tc_ans, c_ans, ct_ans)
15:         catch exn
16:             handle_unsoundness_bug(exn, stmts, cvc5_res, ae_t_res, ae_c_res)
17:         end try
18:     catch exn
19:         handle_failure_bug(exn, stmts)
20:     end try
21: end function
```

Algorithm 1: The testing function

The function represented in Algorithm 1 takes as input a list of statements, in lines 3 and 4 the statements are translated to Alt-Ergo's internal language and the SMT-LIB v2 standard. In line 6, CVC5 is called with the statements in the SMT-LIB v2 standard and its answers are recovered. In lines 8 to 11, Alt-Ergo is called with its four solvers on the statements that are in Alt-Ergo's internal language, the answers are also recovered. If a crash happens in one of the previous steps, an exception is raised, caught at line 18, handled at line 19 and the test fails. Otherwise, the answers of Alt-Ergo's solvers are compared to those of CVC5 in line 14. If there is a contradiction in the answers, then an exception is raised, caught in line 15, handled in line

16 and the test fails, otherwise nothing happens and the test passes.

3.3 Bug management

The management of bugs is the process that is applied when handling the exceptions that are raised when a test fails. In Algorithm 1 it is visible that there are two functions in line 16 and line 19 that handle exceptions. The role of those functions is to make it possible to get information about the bug and reproduce it in a simple manner. To do so, the technique that we chose consists of building a data structure that holds the necessary information about the bug, namely the list of statements that caused the bug and the exception that was raised. In the case in which the exception was raised because of contradictory answers, those answers are also collected in the data structure. That data structure is then stored in a uniquely named file, in a specific folder. The files are split across different folders according to the type of the bug that lead to their creation, whether it is a stack overflow, an unsoundness bug or another kind of bug.

Once that file is stored, it is possible then to read its data and translate the list of statements to rerun the solvers on them in order to reproduce the bug. It's also possible to translate the list of statements to the SMT-LIB v2 standard or Alt-Ergo's native language and store them respectively in a ".smt2" or ".ae" file. The file can be used to modify the statements during the debugging process or to report an issue on Alt-Ergo's issue tracker.[2]

4 Experimentation

The fuzzer was put to practice by running it on Alt-Ergo's 2.4.1 release, by using AFL's parallel mode and launching 4 instances of the fuzzer in parallel on a machine that has four cores for 20 days.

Encountered difficulties Initially the testing showed a significant problem in the efficiency of the fuzzer, its stability would decrease very quickly after it was launched and "Out of memory" exceptions were raised repeatedly. Stability is a measure of the program's determinism when it receives similar raw data. The higher it is, the easier it is for AFL to understand and learn about the behaviour of the program, which in turn improves its capability of generating input data that increases the code and execution path coverage in the tested program. These issues were due to two main reasons, the design of Alt-Ergo and the way in which the testing is done with AFL.

Internally Alt-Ergo relies significantly on side effects with the use of caches that are necessary for its reasoning. These caches come either in the form of hash tables or in the form of references to lists and other data types. That is not a problem when Alt-Ergo is used as it was designed to be used, namely by running its binary on a ".ae" or ".smt2" file, processing the statements of the file in a relational manner and returning answers. What makes it problematic in our case is the second issue that we mentioned earlier, which is how AFL works. When AFL is launched on an executable, it runs one instance of it and provides several generated raw data files to it. As long as there is no crash and no timeout is reached it continues until it reaches its chosen number of iterations. The number of iterations is chosen by AFL internally and it depends on the execution speed of the program and other parameters.

[2] Alt-Ergo's issue tracker: https://github.com/OCamlPro/alt-ergo/issues

Knowing that each raw data file generated by AFL leads to the generation of a list of statements on which Alt-Ergo's four solvers are tested. Doing that repeatedly would quickly lead to too much memory consumption and undesirable behaviour. That is because AFL expects the program to behave in a similar manner when provided with the same or very similar raw data files. That's not the case here, since the way in which a list of statements is processed changes if they are processed in relation to other statements that were processed before or independently.

To solve this issue, we added a resolution context reinitialization functionality that makes it possible to reinitialize the caches that are used by the solvers to the state in which they were initially, before processing any statements. To do so we needed to track these caches, since the functions that have side effects are not always documented as such and because OCaml uses garbage collection, developers don't usually think about freeing memory manually after using it unless they need to. The added functionality made it possible to use one instance of Alt-Ergo on many lists of statements while making sure that they are all processed independently by reinitializing the resolution context after processing each list of statements.

Results During the experimentation, the fuzzer managed to find four unique crashes of Alt-Ergo on well-formed lists of statements that were reported on Alt-Ergo's issue tracker (#474, #475, #481 and #482). There were also false positives of unsoundness bugs which after closer inspection turned out to be due to divisions by zero (#476, #477 and #479), but the fact that Alt-Ergo answered on them in a contradictory manner to CVC5 and without printing a warning or something to say that the answer was influenced by the possible presence of a division by zero is problematic.

5 Limitations

For the fuzzing to be as efficient as possible it needs to perform many tests and be able to generate data that is diverse and complex enough to be able to cover hard to access execution paths and corner cases in the program. When generating statements, the generator requires a certain number of parameters, for example, the maximum depth of a statement, the maximum number of quantified variables by type, the number of statements, etc. Choosing the right parameters to optimize the capability of the executions to find new paths and cover new code while being doable in a reasonable amount of time is not trivial. In the current version of the fuzzer, the values of the various parameters were set in an intuitive manner and chosen after doing extensive testing. It appeared during the testing that with non restrictive parameters, it is very easy to have an explosion of the sizes of the statements and produce statements that take a very long time to be processed by the solvers.

A possible workaround for this issue is to gather some statistics on the structure of SMT files from industrial or research benchmarks, and get from those statistics upper and lower bounds for each one of the parameters. These bounds can then be used to give the fuzzer the possibility of selecting the values of the parameters before generating formulas. Eventually the fuzzer should be able to select the best values for these parameters.

When formulas are too complex, not only do the solvers take too long to process them, but timeouts happen, which is not ideal because the feedback from timed out executions cannot be exploited correctly by AFL. Similarly, when the generated data leads to the detection of bugs that were previously detected. The executions are not likely to be optimal when it comes to the feedback they provide to help with the detection of new bugs. On the other hand, that might be useful because it allows the user to choose the simplest and clearest one of the test

cases that caused the bug to try and resolve it. One way to avoid redetecting bugs is to correct them quickly and relaunch the fuzzer, then again, relaunching the fuzzer means going from scratch and losing the information it gathered on the structure of the code during the time it was running, and regaining that information could require a significant amount of time.

6 Conclusion and future work

We presented in this paper Alt-Ergo-Fuzz, which is the fuzzer we developed for the Alt-Ergo SMT solver. This is the first fuzzing to be done on Alt-Ergo and more extensive testing will hopefully reveal how efficient it can be.

Alt-Ergo-Fuzz still needs a lot of work before becoming a complete software testing tool. One of the things that it needs is a shrinking technique that allows for the diminution of the size of the test cases that cause bugs, which would facilitate finding the bug and solving it. Another one would be a completeness test, for now we only test to see if Alt-Ergo crashes or answers in a contradictory manner to CVC5. Ideally, we should be able as well to test whether Alt-Ergo responds `unknown` for formulas that it is supposed to be able to decide because that might as well be hiding bugs in its reasoning.

References

[1] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. 2021. URL: http://smtlib.cs.uiowa.edu/language.shtml (cit. on p. 2).

[2] Clark Barrett and Cesare Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Cham: Springer International Publishing, 2018, pp. 305–343. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_11 (cit. on p. 1).

[3] Clark Barrett et al. "CVC4". en. In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 171–177. ISBN: 978-3-642-22110-1. DOI: 10.1007/978-3-642-22110-1_14 (cit. on p. 1).

[4] Clark Barrett et al. *The CVC5 automatic theorem prover*. 2021. URL: https://cvc5.github.io (cit. on p. 5).

[5] Koen Claessen and John Hughes. "QuickCheck: A lightweight tool for random testing of Haskell programs". In: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP* 46 (2000). DOI: 10.1145/1988042.1988046 (cit. on p. 3).

[6] Sylvain Conchon et al. *Alt-Ergo*. URL: https://alt-ergo.ocamlpro.com (visited on 2021-10-13) (cit. on p. 1).

[7] Stephen Dolan. *Crowbar*. 2017. URL: https://github.com/stedolan/crowbar (cit. on pp. 2, 3).

[8] Jun Li, Bodong Zhao, and Chao Zhang. "Fuzzing: a survey". In: *Cybersecurity* 1.1 (2018-06), p. 6. ISSN: 2523-3246. DOI: 10.1186/s42400-018-0002-y (cit. on p. 3).

[9] Hongliang Liang et al. "Fuzzing: State of the Art". In: *IEEE Transactions on Reliability* 67.3 (2018-09), pp. 1199–1218. ISSN: 1558-1721. DOI: 10.1109/TR.2018.2834476 (cit. on p. 3).

[10] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". en. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 337–340. ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24 (cit. on p. 1).

[11] François Pottier. "Strong Automated Testing of OCaml Libraries". In: *JFLA 2021 - 32es Journées Francophones des Langages Applicatifs, Feb 2021, Saint Médard d'Excideuil, France*. 2021-02. URL: https://hal.inria.fr/hal-03049511 (cit. on p. 2).

[12] Dominik Winterer, Chengyu Zhang, and Zhendong Su. "On the unusual effectiveness of type-aware operator mutations for testing SMT solvers". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020-11), 193:1–193:25. DOI: 10.1145/3428261 (cit. on p. 1).

[13] Dominik Winterer, Chengyu Zhang, and Zhendong Su. "Validating SMT solvers via semantic fusion". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020-06, pp. 718–730. ISBN: 978-1-4503-7613-6. DOI: 10.1145/3385412.3385985 (cit. on p. 1).

[14] Michał Zalewski. *American Fuzzy Lop*. 2013. URL: https://lcamtuf.coredump.cx/afl (cit. on p. 3).