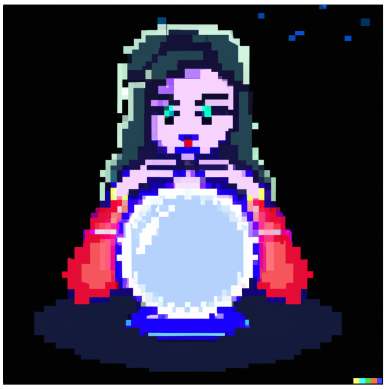
AST - Final Report

PASCAL STREBEL and ROBIN SCHMIDIGER

This research project examines the robustness of Z3, a state-of-the-art Satisfiability Modulo Theories (SMT) solver, focusing specifically on its string solvers, Z3Seq and Z3Str3. While Z3 has proven to be effective in handling theories like arithmetic and bit-vectors, the robustness of its string-related problem-solving capabilities remains a subject of exploration. String manipulation, with its inherent complexity and undecidability, poses significant challenges in SMT solving. Edit distance, a fundamental metric for quantifying the similarity between two strings, is a crucial operation in numerous real-world applications. We designed an approach that leverages the concepts of edit distance to rewrite (un)satisfiable SMT formulas that can be used to test SMT solvers. We then implemented this approach in our automated tool MadamASTra to comprehensively assess the performance and reliability of Z3's string solvers. The analysis encompasses various aspects, including handling strings of different lengths, considering multiple edit operations, and evaluating the solvers' scalability with increasingly complex inputs. The findings from this work provide valuable insights into the application of Z3 for real-world applications that heavily rely on string operations and contribute to a better understanding of Z3's limitations and strengths in the domain of string manipulations. Additionally, the work provides a basis and offers recommendations for potential further research to assess the robustness of SMT string solvers.

Additional Key Words and Phrases: Automated Software Testing, SMT Solvers, String Solvers, Levenshtein Distance



1 INTRODUCTION

Software development has evolved significantly over the years, with complex applications becoming increasingly prevalent in various domains. As software systems become more intricate, the need for reliable and efficient methods to verify their correctness and robustness becomes crucial. Automated software testing plays a key role in this regard, enabling developers to detect and fix bugs, ensure quality, and improve the overall reliability of their software.

One area of software testing that has gained significant attention is the evaluation of Satisfiability Modulo Theories (SMT) solvers. SMT solvers are powerful tools used for automated reasoning and decision-making in various domains such as formal verification, program synthesis, and software analysis. They use logical theories to determine the satisfiability of formulas expressed in a particular logical language such as arithmetic, bit vectors, strings, and uninterpreted functions [Monniaux 2016].

Authors' address: Pascal Strebel, pstrebel@ethz.ch; Robin Schmidiger, schmrobi@ethz.ch.

The robustness of an SMT solver is of major importance, as it directly affects the reliability and accuracy of its results. Robustness refers to the ability of a solver to efficiently process a wide range of input scenarios, avoid errors, and consistently deliver reliable solutions. Ensuring the robustness of an SMT solver is essential, as even a single incorrect result can have significant consequences in critical systems. But as SMT solvers are complex and highly optimised software systems, it is difficult to ensure their required correctness.

While most popular SMT solvers, including Z3 [de Moura and Bjørner 2008], exhibit high robustness in some theories such as arithmetic and bit-vectors, they do not necessarily do so for strings. This is primarily due to the error-prone nature of string manipulations and because any comprehensive theory of strings is undecidable [Bjørner et al. 2009].

Therefore, our objective in this work is to evaluate the reliability of the widely used SMT solver Z3, specifically its two string solvers: Z3Seq [Berzish et al. 2021] and Z3Str3 [Berzish et al. 2017].

To accomplish this, we generate random words and use rewrite rules to encode their edit distance [Levenshtein 1965] in an SMT formula, whose satisfiability we determine using Z3. The edit distance between two words measures the minimum number of operations (insertions, removals, or replacements) required to convert one of them into the other. Since edit distance has been studied extensively and its theoretical limits are well-defined, we have prior knowledge regarding the feasibility of transforming one word into another within a specified number of steps, which allows us to generate and rewrite formulas that are (un)satisfiable by design.

Based on these insights, we aim to evaluate the robustness of Z3 by examining its ability to consistently and accurately solve the generated SMT formulas within the expected time frame. To the best of our knowledge, no previous work has explored the application of real-world problems, such as transforming one word into another with minimum number of operations, for the purpose of generating and rewriting formulas to evaluate SMT solvers.

Despite extensive employment of our tool MadamASTra, which incorporates this approach, our experiments have not been able to trigger any erroneous behavior in Z3. This observation suggests that Z3's string solvers exhibit a high degree of robustness in handling the generated SMT formulas related to the edit distance problem.

Nevertheless, the results of this work will not only shed light on the robustness of Z3, but can also provide developers and researchers with insight into the capabilities of SMT solvers in addressing real-world problems related to string manipulation and constraints. By delving into the realm of automated software testing and robustness assessment of SMT solvers, this work aims to contribute to the broader field of automated software testing, promoting the development of more dependable software systems.

The remainder of this report is structured as follows:

- Section 2 provides a detailed overview of our technique, delving into subtleties and highlighting key aspects.
- In Section 3, we introduce our tool MadamASTra, outline the experimental setup we utilized, and present our findings from the investigation of Z3's robustness.
- Section 4 offers a discussion on related work in the field of SMT solver testing, with particular emphasis on strings. Thus, we postion our work within the existing body of knowledge.
- Following this, in Section 5, we conclude by analyzing the implications of our findings and providing insights into potential future research directions.

2 APPROACH

The edit distance x between strings s_1 and s_2 denotes the minimum number of insertions, removals and replacements of a single character to transform s_1 into s_2. That is, it is possible to find a sequence of x `insert`, `remove`, and `replace`

operations to get from s_1 to s_2, and it is impossible to find a sequence of less than x such operations that achieves the same. Those thoughts form the basis for our generation of satisfiable and unsatisfiable SMT formulas, respectively.

In terms of SMT solving, we can model the `insert`, `remove`, and `replace` operations as either interpreted or uninterpreted functions, with the difference of whether we want to specify their behavior explicitly or implicitly. In either case, they have the same signatures, shown in Figure 1.

```
(declare-fun insert (String Int String) String)     // e.g. insert "h" 0 "ello" -> "hello"
(declare-fun remove (Int String) String)            // e.g. remove 4 "hello" -> "hell"
(declare-fun replace (String Int String) String)    // e.g. replace "a" 1 "hello" -> "hallo"
```

Fig. 1. Signatures for the `insert`, `remove`, and `replace` operations as SMT declarations with intuitive examples for each of them.

Without enforcing further constraints, however, the behavior suggested by the intuitive examples in Figure 1 is not adopted by Z3. Instead of general simple definitions, Z3 interprets the functions with different if-then-else cases that do not reflect the actual specifications of the operations, as demonstrated in Figure 2.

```
...
(assert (= (insert "e" 3 (replace "y" 1 "bit")) "byte"))
...
```

```
(define-fun insert ((x!0 String) (x!1 Int) (x!2 String)) String
    (ite (and (= x!0 "e") (= x!1 3) (= x!2 "e")) "byte"
    ...
```

Fig. 2. Given usages of the `insert`, `remove`, and `replace` functions (above), Z3 infers a model where they are defined using multiple if-then-else cases (below), which is unsuitable for our purpose because it does not reflect the actual behavior of the operations.

In order to specify the functions `insert`, `remove`, and `replace` more precisely after all, but not to get into a different theory of quantifiers (e.g., by saying that for all strings, if they are used with the function `insert`, they become longer by exactly 1 character), which leads to timeouts more quickly and is not the content of our project, we define the functions explicitly, i.e., as interpreted functions. For instance, the SMT encoding of the `insert` operation that we ultimately employed is shown in Figure 3. The operations `remove` and `replace` are defined analogously.

```
(define-fun insert ((to_insert String) (index Int) (source String)) String
  (ite
      (and (>= index 0) (<= index (str.len source)))
      (str.++ (str.substr source 0 index) to_insert (str.substr source index (str.len source)))
      source
  )
)
```

Fig. 3. SMT encoding of the `insert` operation. In particular, the character `to_insert` is inserted into the string `source` at position `index` if the `index` exists, i.e., is greater than or equal to 0 and less than or equal to the length of `source`. Otherwise, nothing happens, i.e., `source` is returned as it was before. Note that this definition does not enforce `to_insert` to be a single character, so we do this with additional constraints on each application of the function.

We then automatically generate formulas including application series of these operations using a classical dynamic programming algorithm for edit distance between two strings with backtracking. To rewrite a satisfiable SMT formula, we encode the steps derived by the algorithm, i.e., a minimum number of operations to be applied to transform the first string into the second string, directly into an SMT formula, which is thus satisfiable. To rewrite an unsatisfiable SMT formula, we use a random sequence of `insert`, `remove`, and `replace` operations, but less of them than the actual edit distance between the two strings is, thus the formula must be unsatisfiable. Our formulas may contain more or fewer unknowns, as shown in Figures 4 and 5, respectively, and can be used to test the string solvers of Z3.

```
φ1: (assert (= (remove 4 (insert "p" 2 (replace "w" 0 "host"))) "wops"))
φ2: (assert (= (remove int_const_2 (insert str_const_1 int_const_1 (replace str_const_0 int_const_0 "host"))) "wops"))
```

Fig. 4. Satisfiable SMT formulas generated by our technique. Formula φ_1 contains not only the `insert`/`remove`/`replace` operations to be applied, but also the corresponding input values to get from "*host*" to "*wops*". φ_1 is not a challenge for Z3, but an implicit model for the correctness of our technique. In formula φ_2, these values are replaced by uninterpreted constants, i.e., unknowns whose value should be found. The more unknowns there are, the more difficult it becomes for Z3 to determine whether the formula is satisfiable. Note that we must additionally enforce all string constants to have length 1 (via `(assert (= (str.len str_const_x) 1))`).

```
φ3: (assert (= (replace str_const_1 int_const_1 (remove int_const_0 "host")) "wops"))
```

Fig. 5. Unsatisfiable SMT formula generated by our technique. Since we know the edit distance between "*host*" and "*wops*" to be 3, it is impossible to get from "*host*" to "*wops*" with only 2 randomly chosen operations. Therefore, φ_3 is unsatisfiable by construction.

Thus, we have created a unique method to encode the real-world problem of converting one word into another with minimal number of steps in pre-labeled SMT formulas.

3 IMPLEMENTATION AND RESULTS

We created MadamASTra, a tool incorporating the approach described in Section 2 as a form of metamorphic testing. In the following subsections, we delve into implementation details, describe our experimental setup, and present results.

3.1 Overview

Creating MadamASTra involved implementing three distinct behaviors: Search, Try, and Log. Each behavior contributes to the comprehensive assessment of Z3's performance and robustness in handling SMT formulas related to the edit distance problem. In summary, the behaviors accomplish the following:

- Search aims to explore Z3's robustness by generating random strings, configuring parameters such as `mode` (`unsat` or `sat`) and `solver` (`seq` or `z3str3`), and rewriting an SMT formula based on the chosen configuration. MadamASTra then runs Z3 on that SMT formula and compares its result to the expected result. It repeats this process for a fixed number of iterations.
- Try focuses on testing specific fixed word pairs, while parameters such as `mode` and `solver` are varied, to uncover potential vulnerabilities or limitations in Z3's handling of the edit distance problem. In the case of unsatisfiable formulas, the non-deterministic behavior of our approach leads to a variety of cases arising even for fixed words. The process terminates as soon as Z3 makes a mistake.
- Log serves the purpose of generating SMT formulas based on a given configuration but does not run Z3. Instead, the formulas are persisted for later analysis. It is useful to ensure that any bugs can be reproduced.

Algorithm 1 provides a high-level overview of MadamASTra's Search behavior. We start each iteration of the main procedure by generating two random strings. The longer the strings, the higher the difficulty, as their edit distance tends to be higher. Next, we compute the edit distance between them and encode it in an SMT formula that is (un)satisfiable by construction, using the approach described in Section 2. We then spawn a shell and run Z3 with the SMT formula. If Z3 makes an error, i.e., erroneously labels the formula as (un)satisfiable, we persist it. This procedure is repeated for a fixed number of iterations. Note that we do not consider the results "*unknown*" or "*timeout*" to be errors by Z3, as they are not subject to this work. This behavior allows us to test Z3's ability to handle various configurations and detect any potential inconsistencies or incorrect results produced by the SMT solver.

Algorithm 1 MadamASTra Search

```
Input : string_length
configs ← {sat, unsat} × {seq, z3str3}
bugs ← {}
for i ← 1..n do
    w1, w2 ← stringgenerator.generate(string_length)
    mode, solver ← configs.next()
    if mode == sat then
        φ ← get_sat_z3_formula(w1, w2)
    else
        φ ← get_unsat_z3_formula(w1, w2)
    end if
    input = wrap_formula(φ, solver)
    res = z3_driver.run(input)
    if res! = mode then
        bugs ← bugs ∪ {φ}                          ▷ Z3 made a mistake!
    end if
end for
```

Algorithm 2 is structured similarly to Algorithm 1, but uses two fixed strings instead of randomly generated ones. The loop is terminated once Z3 makes an error. This behavior allows us to identify scenarios where Z3 might fail to provide accurate solutions, and thus gain insight into specific cases that challenge the SMT solver's robustness.

Algorithm 2 MadamASTra Try

```
Input : w1, w2
configs ← {sat, unsat} × {seq, z3str3}
bugs ← {}
while bugs == {} do
    mode, solver ← configs.next()
    if mode == sat then
        φ ← get_sat_z3_formula(w1, w2)
    else
        φ ← get_unsat_z3_formula(w1, w2)
    end if
    input = wrap_formula(φ, solver)
    res = z3_driver.run(input)
    if res! = mode then
        bugs ← bugs ∪ {φ}                          ▷ Z3 made a mistake!
    end if
end while
```

3.2 Experimental Setup and Technicalities

MadamASTra is implemented in Python (3.10.6) and uses Z3 as a binary (4.8.15). Although a Python API exists for Z3, we deliberately take the detour with generating an SMT formula so that our technique is applicable to any SMT solver.

Recall from Section 2 that our approach for rewriting unsatisfiable formulas is non-deterministic, thus calling `get_unsat_z3_formula` in Algorithms 1 and 2 twice with the same inputs will likely result in two different SMT-formulas. Therefore, we provide the unsat formula generator with a seed to ensure that our results are reproducible.

Each iteration of the Search behavior executes a separate instance of Z3, which can result in a significant runtime. To mitigate this, we have implemented a timeout mechanism to limit the runtime of Z3 for each SMT formula to a few seconds. Our tool MadamASTra currently uses a default timeout of 30 seconds, which can be changed via the command line interface (CLI). Although this approach places an upper limit on the total runtime of MadamASTra, the total execution time for about 120 iterations of the Search loop can still be about an hour. It is important to note, however, that each iteration of the Search behavior is independent of all previous iterations. This allows for parallelization so that multiple iterations can be executed simultaneously on different processor cores. Taking advantage of the parallel processing capabilities of modern machines can significantly reduce the overall average runtime of MadamASTra. By distributing the workload across multiple cores, we can achieve more efficient execution and speed up the testing process, ultimately reducing the overall robustness evaluation time.

3.3 Results

MadamASTra enabled us to comprehensively evaluate Z3's robustness in handling SMT formulas related to the edit distance problem. Through iterations of random string generation, configurable parameter selection, and systematic testing of specific string pairs, we aimed to expose any inconsistencies in Z3's performance. The logged SMT formulas also provided valuable insights for further analysis and potential improvements in the solver's robustness. In the following, we discuss the results obtained from executing MadamASTra, analyze the observed behavior of Z3, and draw conclusions regarding its robustness in handling the edit distance problem.

After numerous iterations of the Search behavior, we are pleased to report that we did not encounter any errors in Z3's response. Over the course of 150′000 iterations, Z3 consistently provided accurate and reliable results within the expected time frame. Moreover, we also did not observe any bugs in Z3's behavior during the execution of the Try behavior. Overall, our results indicate that Z3 showcases a high level of robustness when dealing with SMT formulas related to the edit distance problem. This outcome provides reassurance about the reliability and accuracy of Z3 as an SMT solver for handling real-world problems related to string manipulation and constraints.

While our testing did not reveal any bugs or inconsistencies, it is important to note that further evaluation and analysis may uncover corner cases or scenarios that challenge Z3's robustness. It is advisable to continue exploring different inputs, configurations, and problem domains to comprehensively assess the solver's capabilities and identify potential areas for improvement.

4 RELATED WORK

While strings are among the most researched theories in the SMT community in the last decade, the methods presented are often complicated and support a rich vocabulary of operations. An example work for this is [Bugariu and Müller 2020], which, similar to what we do, proposes a synthesis approach to generating formulas on strings that are (un)satisfiable by construction, with which they detected a total of 5 bugs in Z3. A more general approach is taken by Semantic Fusion

[Winterer et al. 2020]. The rewriting rules presented there are very intuitive, but have nevertheless discovered 37 bugs in Z3. With both of these approaches, however, the generated formulas become increasingly complex, which can lead to timeouts and which is why generated formulas must first be reduced before reporting a bug. Moreover, they are completely detached from real-world problems and thus may focus on testing aspects of an SMT string solver that are not relevant in practice. While we also employ simple rewriting rules, our approach, on the other hand, incorporates the generation of short formulas with a strong focus on the real-world problem of editing one word into another.

The work of [Zhang et al. 2023] focuses less on soundness and more on performance regressions happening across multiple versions of a string solver. Although this is not our main focus, they demonstrate performance aspects of Z3's string solvers Z3Seq and Z3Str3, which we noticed as well when running MadamASTra.

5 CONCLUSION

In this final section, we summarize the key aspects of our work to evaluate the robustness of Z3 when dealing with SMT formulas related to the edit distance problem. We reflect on the contributions of our work and propose directions for future research.

We have designed an interesting approach for rewriting (un)satisfiable SMT formulas based on the edit distance problem, exploring the application of a real-world problem for the purpose of evaluating SMT solvers. Based on this, we implemented and used our tool MadamASTra to thoroughly test the robustness of Z3's string solvers.

The results of our tests demonstrated Z3's high reliability and accuracy. No errors occurred while executing MadamASTra, indicating the robustness of Z3 in solving SMT problems related to edit distances.

There are, however, several areas where our work can build a basis to potentially uncover bugs or inconsistencies in Z3's behaviors:

- Although we have tested various random string pairs and configurations, it would be beneficial to expand the range of input cases. Examining extreme or borderline cases, as well as complex or extensive formulas, could reveal previously unknown scenarios in which Z3's robustness might be challenged. In addition, the inclusion of real datasets or domain-specific examples in the testing process could provide insights into the performance of the SMT solver in practical applications.
- By increasing the complexity and diversity of the generated SMT formulas, Z3 could be subjected to a further stress test. By including a wider range of constraints, branching conditions, or nonlinear expressions (i.e., additional theories) in the formulas, we may uncover potential areas where Z3's performance or accuracy are affected. Exploring advanced properties and functionalities of Z3, such as quantifiers or theory-specific inference, could also provide valuable insights into the robustness of Z3.
- Using MadamASTra with other SMT solvers for comparative analysis would be a possibility for future work. We have created a proper basis for this by having the formula generation of MadamASTra provide SMT formulas that can be given as input to many other SMT solvers. A comparison of Z3's performance with other leading solvers could provide a comprehensive understanding of its strengths and weaknesses compared to alternative solutions. This comparative analysis could reveal subtle differences in behavior or highlight specific areas where Z3 excels or needs further improvement.

In summary, our work successfully evaluated the robustness of the SMT solver Z3 for SMT formulas based on the edit distance problem. The absence of bugs or errors in Z3's responses during the execution of MadamASTra demonstrates its reliability and accuracy. However, there is still room for improvement, such as exploring more diverse input cases,

increasing the complexity of the generated formulas, and performing comparative analysis with other solvers. By addressing these potential areas of improvement and further investigating the behavior of Z3, we can improve our understanding of the robustness of SMT solvers and contribute to the advancement of automated software testing methods. This work serves as a foundation for future research and opens up avenues for refining and expanding the capabilities of SMT solvers in tackling complex software verification tasks.

REFERENCES

Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. 2017. Z3str3: A String Solver with Theory-aware Heuristics. In *2017 Formal Methods in Computer Aided Design (FMCAD)*. 55–59. https://doi.org/10.23919/FMCAD.2017.8102241

Murphy Berzish, Mitja Kulczynski, Federico Mora, Florin Manea, Joel D. Day, Dirk Nowotka, and Vijay Ganesh. 2021. An SMT Solver for Regular Expressions and Linear Arithmetic over String Length. arXiv:2010.07253 [cs.LO]

Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. 2009. Path Feasibility Analysis for String-Manipulating Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, Stefan Kowalewski and Anna Philippou (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 307–321.

Alexandra Bugariu and Peter Müller. 2020. Automatically Testing String Solvers. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1459–1470. https://doi.org/10.1145/3377811.3380398

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

V. I. Levenshtein. 1965. Binary codes with correction of deletions, insertions and symbol substitutions. , 845-848 pages.

David Monniaux. 2016. A Survey of Satisfiability Modulo Theory. arXiv:1606.04786 [cs.LO]

Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT Solvers via Semantic Fusion. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 718–730. https://doi.org/10.1145/3385412.3385985

Yao Zhang, Xiaofei Xie, Yi Li, Yun Lin, Sen Chen, Yang Liu, and Xiaohong Li. 2023. Demystifying Performance Regressions in String Solvers. *IEEE Transactions on Software Engineering* 49, 3 (2023), 947–961. https://doi.org/10.1109/TSE.2022.3168373