ddSMT 2.0: Better Delta Debugging for the SMT-LIBv2 Language and Friends [⋆]

Gereon Kremer, Aina Niemetz(✉), and Mathias Preiner

Stanford University, Stanford, USA
{gkremer, niemetz, preiner}@cs.stanford.edu

Abstract. Erroneous behavior of verification back ends such as SMT solvers require effective and efficient techniques to identify, locate and fix failures of any kind. Manual analysis of large real-world inputs usually becomes infeasible due to the complex nature of these tools. Delta Debugging has emerged as a valuable technique to automatically reduce failure-inducing inputs while preserving the original erroneous behavior. We present ddSMT 2.0, the successor of the delta debugger ddSMT. ddSMT is the current de-facto standard delta debugger for the SMT-LIBv2 language. Our tool improves and extends core concepts of ddSMT and extends input language support to the entire family of SMT-LIBv2 language dialects. In addition to its *ddmin*-based main minimization strategy, it implements an alternative, orthogonal strategy based on hierarchical input minimization. We combine both strategies into a hybrid strategy and show that ddSMT 2.0 significantly improves over ddSMT and other delta debugging tools for SMT-LIBv2 on real-world examples.

1 Introduction

In recent years, a growing number of formal methods applications (e.g., [6, 8]) rely on Satisfiability Modulo Theories (SMT) solvers as the back end. Current state-of-the-art SMT solvers are typically complex pieces of software, and debugging erroneous behavior requires effective and efficient techniques to analyze failure-inducing input with the purpose of identifying and locating the cause of the failure. Manual analysis of real-world problems that trigger a particular unwanted behavior is very often infeasible for large inputs, mainly due to the complex nature of these tools.

Erroneous behavior is never only triggered by a single unique input, but by a class of inputs that share a common trait. Extracting a *minimal working example*, i.e., an input that is *as small as possible* but still triggers the original faulty behavior, from such a class of inputs usually significantly decreases the time to identify and locate the cause of the failure. While ideally, the notion of size of an input directly correlates to the effort required to determine the failure cause, in practice this is hard to quantify. We instead use metrics such as file size, number of language constructs, and solver runtime until the failure occurs.

[⋆] This work was supported in part by DARPA (award no. FA8650-18-2-7861) and ONR (award no. N68335-17-C-0558).

Finding such minimal working examples, however, is a problem of its own. Manual minimization is typically infeasible in practice, simply due to the large number of possible simplifications that may even depend on each other. Delta debugging techniques, on the other hand, provide automated means to minimize failure-inducing inputs. This typically entails to first read some input, apply a set of rules to simplify the input, and then check that the modified input still triggers the original behavior. Delta debugging in its simplest form [24] extracts a minimal working example by omitting parts of the input that are irrelevant for triggering the original faulty behavior. More input language specific tools perform additional simplifications to further minimize the input. All of these simplifications are typically performed until a fixed point is reached.

For the design of a delta debugger, this process raises a number of questions: How does the debugging tool check for "same behavior" of a tool on some input? Which simplification rules should be employed and how should they be combined? To what (syntactic and semantic) degree should the delta debugger itself understand the input language? In this paper, we address these questions in the context of delta debugging for the SMT-LIBv2 language and its dialects with our delta debugger ddSMT 2.0, the successor of ddSMT [18]. In the following, we will refer to ddSMT 2.0 as ddSMTv2, and to its predecessor as ddSMTv1.

Related Work. Generic delta debugging tools that are agnostic to the input language can be surprisingly efficient for some use cases. For minimizing SMT-LIB input, however, their usefulness is usually rather modest. One such generic tool is linedd [4], which solely performs line-based simplifications. The first delta debugging tool specific to the SMT-LIB language was presented in [7] as deltaSMT and targeted SMT-LIBv1 [22]. Three years later, the SMT community adopted a new input language SMT-LIBv2. In 2013, an updated version of deltaSMT [10] extended the tool syntactically for SMT-LIBv2 compliance, but limited to the feature set of the SMT-LIBv1 language and without full SMT-LIBv2 support. Note that this updated version is not available anymore. In the same year, ddsexpr [5], a generic hierarchical delta debugger for S-expressions (and thus applicable to the SMT-LIB language family), and ddSMTv1 [18], a delta debugger specific to the SMT-LIBv2 language, were presented. The latter implements a variant of Zeller's *ddmin* algorithm [24] and is considered as the current de-facto standard delta debugger in the SMT community. The only other delta debugging tool specific to the SMT-LIBv2 language we aware of is delta [15], a hierarchical delta debugger shipped together with the SMT solver SMT-RAT [9]. A reimplementation of delta in Python is available as pyDelta at [14].

Contributions. In this paper, we present ddSMTv2, a delta debugging tool for the SMT-LIBv2 [2] language and its dialects. It supports the entirety of the SMT-LIBv2 standard as well as non-standardized extensions and derived formats such as the SyGuS input language [21]. Our tool is agnostic to future extensions of the standard in the sense that it does not require any modifications for basic support. It is easy to extend, and extensions will only be required for simplifications that are specific to new language features or a certain dialect of the SMT-LIBv2 language. In this sense it will also immediately support the SMT-LIBv3 [1] language, which is currently under development.

ddSMTv2 is the successor of the delta debugger ddSMTv1 [18] and incorporates, improves and extends its core concepts. It also implements an improved variant of the hierarchical approach of pyDelta as an alternative, orthogonal strategy, and allows to combine these two strategies in a hybrid manner. ddSMTv2 is intended to overcome major weaknesses of ddSMTv1, which is limited to the SMT-LIBv2 language and does not support the full set of standardized background theories or language extensions to the point where it is even unable to parse the input file. ddSMTv2 further extends the set of theory-specific simplifications over both ddSMTv1 and pyDelta, which allows to exploit even more minimization opportunities.

ddSMTv2 is implemented in Python and can be installed via `pip3 install ddsmt`. Its documentation is available at [11], and its source code is available under version 3 of the GNU General Public License (GPLv3) at [13].

2 Detecting Failure-Inducing Inputs

An SMT solver is a fully automated tool to determine the satisfiability of a first order logic formula modulo some background theories and their combinations. For satisfiable inputs, SMT solvers optionally allow to query a model, whereas for unsatisfiable inputs, some optionally generate a proof of unsatisfiability. Additionally, SMT solvers usually provide a plethora of configuration options.

Within the SMT community, the notion of *failure* is generally defined as anything from abnormal termination or crashes (including segmentation faults and assertion failures), to performance regressions (one solver performs significantly worse on an input than a reference solver), unsoundness (answering sat instead of unsat and vice versa), incorrect models or incorrect proofs of unsatisfiability. In the following, we define a *failure-inducing input* to an SMT solver as an SMT-LIB input that triggers a failure. In particular, we do not consider options configured via command line as part of the input.

Strategies to determine if a minimized input still triggers the original faulty behavior typically differ depending on the kind of the failure. For *abnormal termination or crashes*, it is usually sufficient to compare the exit code of the solver call, optionally with additional comparisons of output on the standard output and error channels. For failures that generate error messages that include memory addresses, it is often useful to not compare the full output, but to only match against a specific phrase that occurs in the original error output.

By default, ddSMTv2 does exactly that: it determines if a simplified input has the same erroneous behavior as the original input by comparing the exit code and the output on the standard output and error channels for equality. Standard output and error output can optionally be ignored or matched against user-defined strings via command line options.

Performance regressions are more tricky and typically involve helper scripts that call two solver configurations with some time limit and return a specific exit code in case the performance regression is triggered. The delta debugger will then minimize the input based on this exit code. Inputs that trigger *unsoundness failures* can be dealt with in a similar way. For inputs that reveal performance regressions and unsound answers, ddSMTv2 provides easy-to-use wrapper scripts that can also be adapted to more specific use cases.

Incorrect models and *incorrect proofs* are more involved since they typically require some checking mechanism to determine if a generated model or proof is incorrect. Most SMT solvers implement such mechanisms and will throw an assertion failure in debug mode when such a failure is detected. For cases that are not detected by the solver itself, external checking tools are required. Implementing such checks is considered out of scope for a debugging tool due to their complex nature.

3 Simplification Rules and Staged Simplification

Historically, the set of simplification rules for delta debugging has been in general rather small and mainly limited to removing or reordering parts of the input. Adding *structural and semantic simplifications* on top of these basic transformations has proved successful for the SMT-LIB language, and greatly improves performance over language agnostic minimization techniques. The delta debuggers deltaSMT, delta and ddSMTv1 all support structural and semantic simplifications, albeit to a varying degree. Of these three, ddSMTv1 implements the largest set of language-specific simplifications. The SMT-LIB-agnostic delta debugger ddsexpr, on the other hand, performs structural simplifications only.

Additionally, it is beneficial to devise a strategy for *when* to apply *which kind* of simplification rules to *which part* of the input in order to avoid generating useless test cases. An example for a useless test case is when the declaration of a constant is removed before removing all occurrences of this constant. Such a test case is useless because it is almost guaranteed to fail due to a parse error in the solver instead of triggering the original faulty behavior. It is further beneficial to perform simplifications that promise larger overall reduction (e.g., removal of commands) early on, in order to reduce the burden of more local, theory-specific simplifications (e.g., replacing terms with default values of the same sort).

We require that applying a simplification rule indeed *simplifies* the input and that it is not possible to cycle between applications of simplification rules in order to ensure termination of the minimization procedure. Generally, we define *simplification* in terms of measuring the input size in bytes or in the number of S-expressions. We supplement this with specific syntactic and semantic properties, e.g., the number of variable binders in a quantified formula, or the degree of "sortedness" of children of an S-expression. Intuitively, we say that given an input $\mathcal{A}$, a simplification rule yields a simpler input $\mathcal{B}$ if the constructs in $\mathcal{B}$ are simpler according to some metric specific to the rule, or if $\mathcal{B}$ is smaller than $\mathcal{A}$ in terms of size. As an example for such a metric, consider a simplification rule that replaces a value with another value. Such a transformation is only interpreted as simpler if the value to be replaced does not already fall into the class of simpler values, e.g., for integer values we define the set of simpler values as $\{0, 1\}$. Thus, replacing value 1234 with 0 is a simplification, but replacing 1 with 0 is not.

In ddSMTv2, possible input simplifications are generated by so-called *mutators*, which implement simplification rules. They either perform small local changes to a given S-expression, or introduce global modifications on the input based on that S-expression. Each mutator implements a *filter* method, which checks if the mutator is applicable to the given S-expression. If this is the case, the mutator can be queried to suggest (a list of) possible local and global simplifications. Mutators are not required to be equivalence or satisfiability preserving. They may extract semantic information from the input when needed, e.g., to infer the sort of a term, to query the set of declared or defined symbols, to extract indices of indexed operators, and more. ddSMTv2 applies a considerably larger set of simplifications than ddSMTv1 and currently implements 48 mutators, which range from generic simplifications on S-expressions that require no understanding of SMT-LIB, to more theory-specific mutators that make full use of SMT-LIB semantics. Each of these mutators is enabled by default and can optionally be disabled. Extending ddSMTv2 with a new simplification boils down to implementing a filter method and methods to query local and/or global mutations in a new mutator class, and registering this class as an active mutator.

4 Parsing and Input Representation

While the question about the syntactic and semantic degree of understanding of the input language may seem silly at first glance, it is indeed warranted and actually crucial for the overall design of the delta debugger. The two extreme cases are aiming at *full understanding* of the language, and *no understanding*, i.e., treating the input as a sequence of bytes. The trade-off at hand is mainly between the ability to easily devise *language compliant* simplifications, and the burden of infrastructure required for *parsing* and *representing* the input, which is an additional burden on *maintenance* in case the input language changes.

Both deltaSMT and ddSMTv1 aim at full understanding, while most of the others try for some intermediate level of abstraction, i.e., a level that does not require full understanding of the input language but allows for smarter simplifications than just manipulating bytes. The line-based delta debugger linedd minimizes input by removing lines, whereas ddsexpr is syntax-aware in the sense that it understands S-expressions, but without any SMT-LIBv2 specific semantics. Both delta and pyDelta extend understanding of S-expressions with some semantic properties, however, in the case of delta only to a very basic degree (it is, e.g., not even aware of sorts). Outside of the context of the SMT-LIBv2 language, applying an intermediate abstraction approach was successful for the original *ddmin* algorithm [24], which considers change sets (e.g., commits or individual hunks of a commit), and in [23], where the authors use local semantics of certain C++ constructs. Another example is presented in [16], which exploits the hierarchical structure of an input, independent of the concrete semantics.

Our main target language is SMT-LIBv2, which is a hierarchically structured language where, to cite the SMT-LIBv2 standard [2], "every expression [...] is a legal S-expression of Common Lisp". In contrast to ddSMTv1, in ddSMTv2 we aim for an intermediate level of abstraction to ease the burden on infrastructure and maintenance and choose to use S-expressions as the main representation of the input, just like ddsexpr does. However, additionally, we extract a comprehensive set of semantic properties to allow for SMT-LIBv2 specific and compliant simplifications. Language compliant transformations are a requirement for the specific use case of minimizing SMT-LIBv2 input to debug erroneous behavior of SMT solvers. This is mainly to avoid generating nonsensical test cases, i.e., test cases that an SMT solver will refuse to parse. Even when such test cases are refused immediately, if the overwhelming majority of generated test cases is nonsensical it can significantly impact the efficiency of our debugging tool. Note that we explicitly do not disallow delta debugging non-compliant input.

ddSMTv2 features a simple S-expression parser and represents S-expressions as a lightweight wrapper around built-in Python tuples and strings. Semantic information is recovered in an ad-hoc manner after parsing. This allows for minimal infrastructure and maintenance overhead for input parsing and representation. The parser component of ddSMTv2 has less than 100 LOC, and the ad-hoc semantic analysis accounts for less than 400 LOC. Adding support for new versions, dialects or non-standardized extensions of the SMT-LIB language does not require any changes to the parser.

This is in stark contrast to deltaSMT and ddSMTv1, which both aim to get a full understanding of the input, with all its negative consequences: deltaSMT dedicates about 50% (more than 2000 LOC) of its Java code base and ddSMTv1 even over 80% (3000 LOC) of its Python code base to parsing and input representation. Note that the former targets SMT-LIBv1, whereas the latter provides full SMT-LIBv2 support for most of the standardized theories. In both tools, parsing is a disproportionate part of the code base and extending the tools to support new theories or language constructs usually requires extensive modifications to their input parsers. These modifications have significantly complicated or even inhibited the development of these tools in the past: adding support for the theory of floating-point arithmetic in ddSMTv1 required touching more than 1000 LOC; deltaSMT, on the other hand, has never seen full support of SMT-LIBv2 and fails to parse almost all inputs from our test set.

5 Delta Debugging Strategies

Our delta debugger ddSMTv2 implements two minimization strategies which we call ddmin and hierarchical. These two can be combined into a third strategy called hybrid, which aims to utilize the best of both worlds. All three strategies use the same input representation and have access to the same pool of available mutators. However, they differ in *how* they apply mutators to simplify the input.

Strategy ddmin. Our ddmin strategy implements a variant of the minimization strategy of ddSMTv1 and tries to perform simplifications on multiple S-

Algorithm 1: Main loop of ddmin strategy.

```
   Input: S-Expression $input$
1  do                                                        // run to fixed point
2      $simplified := False$
3      for $M \in mutators$ do
4          $sexprs := \{e \mid e \in input \wedge \mathsf{filter}_M(e)\}$, $size := |sexprs|$
5          while $size > 0$ do
6              for $subset \in \mathsf{partition}(sexprs, size)$ do
7                  $candidate := \mathsf{apply}_M(input, subset)$
8                  if $\mathsf{check\_result}(candidate)$ then
9                      $input := candidate$, $simplified := True$
10             $sexprs := \{e \mid e \in input \wedge \mathsf{filter}_M(e)\}$, $size := size/2$
11 while $simplified$
12 return $input$
```

Algorithm 2: Core simplification loop of hierarchical strategy

```
   Input: S-Expression $input$
1  do                                                        // run to fixed point
2      $simplified := False$
3      for $sexpr \in input$ do                              // BFS traversal
4          for $M \in mutators$ do
5              if $\neg\mathsf{filter}_M(sexpr)$ then continue
6              for $candidate \in \mathsf{apply}_M(input, sexpr)$ do
7                  if $\mathsf{check\_result}(candidate)$ then
8                      $input := candidate$, $simplified := True$
9                      continue with simplified $sexpr$ in Line 3
10 while $simplified$
11 return $input$
```

expressions in the input in parallel. Algorithm 1 shows the main loop of this strategy. For each active mutator M, the algorithm first collects all S-expressions in the input that can be simplified by M (Line 4). Simplifications are applied and checked in a fashion similar to Zeller's original *ddmin* algorithm [24]: the set of S-expressions *sexprs* is partitioned into subsets of size *size*; each S-expression $e \in subset$ is substituted in *input* (Line 7) with a simplification suggested by M; the resulting simplified input *candidate* is then checked if it still triggers the original behavior (Line 8). Once all subsets of a given size are checked, *sexprs* is updated based on the current input and partitioned into smaller subsets. As soon as all subsets of size 1 were checked, the algorithm repeats these steps with the next available mutator. The main loop of strategy ddmin is run until a fixed point is reached, i.e., the input cannot be further simplified. Strategy ddmin applies mutators in two stages. The first stage targets top-level S-expressions (e.g., specific kinds of SMT-LIB commands) until a fixed point to aggressively simplify the input before applying more expensive mutators in the second stage.

Strategy hierarchical. The main loop of the hierarchical strategy performs a simple breadth-first traversal of the S-expressions in the input, and applies all enabled mutators to every S-expression, as shown in Algorithm 2. Once a simplification is found (Line 7), all pending checks for the current S-expression are aborted and the breadth-first traversal continues with the simplified S-expression *sexpr* (Line 9). This process is repeated until a fixed point is reached, i.e., until no further simplifications are found for any S-expression. The main simplification loop (Line 3) is applied multiple times, with varying sets of mutators. In the initial stages, strategy hierarchical aims for aggressive minimization using only a small set of selected mutators, in the next-to-last stage it employs all but a few mutators that usually only have cosmetic impact, and in the last stage it includes all mutators. We observed that breadth-first traversal yields significantly better results than a depth-first traversal, most probably since it tends to favor simplifications on larger subtrees of the input.

Strategy hybrid. This strategy combines strategies ddmin and hierarchical in a sequential portfolio manner. It first applies ddmin until a fixed point is reached, and then calls strategy hierarchical on the simplified input. We chose this order of strategies after observing in our experiments that ddmin is usually faster in simplifying input, while hierarchical often yields smaller inputs.

6 Experimental Evaluation

We compare the different strategies implemented in ddSMTv2 against the existing delta debuggers ddsexpr, ddSMTv1, delta, linedd, and pyDelta. For this purpose, we compiled a set of SMT-LIB and SyGuS test cases from different sources. Every test case consists of an input file, a solver binary and command line configuration options for that binary. Our set of test cases includes those used in [18] and instances reported in bug reports of the SMT solvers Bitwuzla [19], CVC4 [3], Yices [12], and Z3 [17]. The test cases from [18] include issues encountered with development versions of the SMT solvers Boolector [20] and CVC4. Note that we excluded 9 test cases from this set because they did not trigger any faulty behavior on our experimental setup. In total, we collected 244 test cases consisting of inputs that trigger assertion failures, unexpected behavior or wrong solver answers. We performed all experiments on a cluster with Intel Xeon E5-2620v4 CPUs with 2.1GHz and 128GB memory and used a 1 hour wall-clock time limit and 8GB of memory for each delta debugger/test case pair. Table 1 summarizes the results on all 244 test cases.

A first immediate observation is the value of a simpler and more generic parser: ddSMTv1 fails to parse more than 20% of the inputs, mostly due to the lack of support for newer standard and non-standard SMT-LIBv2 constructs. Examples include the `check-sat-assuming` command, algebraic datatypes, some operators of the theory of strings, the SyGuS language extension, and the non-standardized extension to encode problems of separation logic. We also observe that each strategy of ddSMTv2 simplifies significantly more inputs than any

	ddsexpr	ddSMTv1	delta	linedd	pyDelta	ddmin	hier.	hybrid
Parse Errors	0	54	1	0	0	0	0	0
Incorrect Output	0	0	1	1	1	0	0	0
Timeouts	155	81	128	3	126	6	122	6
Any Simplification	219	175	114	209	119	242	242	242
Smallest Output	2	10	0	3	58	89	59	168
Avg. Reduction (%)	-40	-63	+288	-26	-4	-75	+571	-77
Avg. Reduction w/o ERR (%)	-40	-80	+291	-26	-4	-75	+571	-77
Avg. Reduction w/o TO/ERR (%)	-32	-73	+617	-26	-57	-76	-59	-79

Table 1: Results summarized over all 244 test cases.

other tool. The only inputs that could not be simplified by ddSMTv2 were already very small (83 and 98 bytes). Strategy hybrid achieves the smallest output on 168 test cases (more than two thirds) and an average reduction in file size by 77% (79% not counting timeouts), while only timing out on 6 test cases.

Some debuggers increase the input size (in bytes), indicated by positive reductions. Eliminating let binders or inlining function definitions frequently increase the size of the input. A positive reduction occurs if the debugger times out while performing such simplifications, or if it is unable to find viable simplifications after the input size increased. In rare individual cases, incorrect outputs were produced that did not trigger the issue under investigation. This happened because of the unchecked removal of unused variables (delta), incorrect handling of timeouts (linedd) and defective handling of quoted symbols (pyDelta).

The hybrid strategy performs significantly better than ddSMTv1, even on the set of instances that both can reduce without any timeout or error. On these commonly reduced instances (107), the results from hybrid are smaller in most cases (99), and on average smaller by about a third.

On inputs that both ddmin and hybrid reduce without timeout or error (238), the hybrid strategy produces smaller outputs on 125 cases and never generates larger results. On average, over all 238 inputs the outputs are about 5% smaller. This may seem marginal, but can make a big difference for users in practice.

Figures 1–2 show the direct comparison of ddmin, hierarchical, hybrid and ddSMTv1 in terms of output size and overall runtime as scatter plots, where a dot represents a test case and dots on the "T" lines correspond to timeouts. While strategy hierarchical tends to produce smaller output files, it is considerably slower than ddmin and runs into the time limit on 116 more test cases. As a result of this observation, we combined both strategies into the hybrid strategy, which first uses ddmin to quickly reduce the input before applying hierarchical to achieve maximum reduction. Comparing hybrid to the best of strategies ddmin and hierarchical, we see that hybrid usually achieves the smallest output and is only slower on test cases that are comparably fast to minimize. If the runtime of ddSMTv2 exceeds a few minutes, there is no discernible performance penalty.

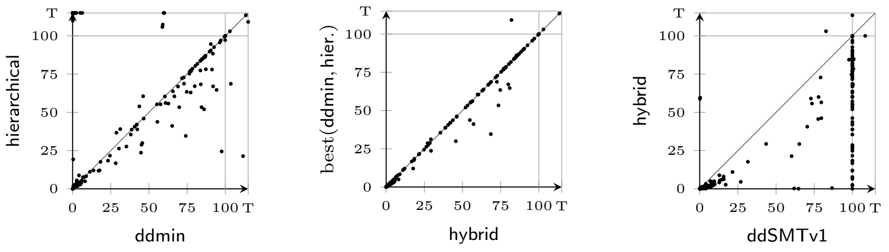

Fig. 1: Output size (in % of original size).

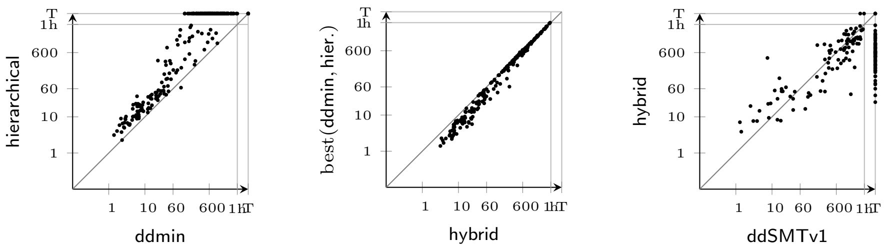

Fig. 2: Overall runtime (in seconds).

In comparison to ddSMTv1, strategy hybrid obtains significantly smaller output files on almost all inputs while having a similar runtime on inputs where ddSMTv1 terminates within the given time limit.

All strategies allow to use multiple worker processes to perform checks asynchronously. Though there is potential for significant runtime improvements, the current impact is rather limited. With 8 worker processes, hierarchical achieves on average a 2x speedup, and up to 6x speedup on a few instances. Both ddmin and hybrid, on the other hand, slow down on average (by 25% and 9%, respectively).

The artifact is available at https://zenodo.org/record/4721925.

7 Conclusion

We have presented ddSMTv2, a delta debugger for the SMT-LIBv2 language and its dialects. Our tool improves substantially over its predecessor ddSMTv1, which is the current de-facto standard in the SMT community for delta debugging SMT-LIB input. We have shown how a more generic parser approach not only lowers the maintenance overhead of the tool itself, but also makes the delta debugger more robust and easier to extend for future SMT-LIB extensions. Our experimental evaluation has shown that ddSMTv2 significantly outperforms existing delta debugging tools on a variety of real-world test cases from different SMT solvers. Further, our experiments suggest that combining different minimization strategies is beneficial in practice to quickly obtain small output files.

References

1. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Version 3.0 - Preliminary Proposal (2021), http://smtlib.cs.uiowa.edu/version3.shtml
2. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK) (2010)
3. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_14
4. Bayless, S.: linedd (2015), https://github.com/sambayless/linedd
5. Biere, A.: ddsexpr (2013), http://fmv.jku.at/ddsexpr
6. Bjørner, N.: SMT in verification, modeling, and testing at microsoft. In: Biere, A., Nahir, A., Vos, T.E.J. (eds.) Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers. Lecture Notes in Computer Science, vol. 7857, p. 3. Springer (2012). https://doi.org/10.1007/978-3-642-39611-3_3
7. Brummayer, R., Biere, A.: Fuzzing and delta-debugging SMT solvers. In: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories. pp. 1–5 (2009)
8. Cook, B.: Formal reasoning about the security of amazon web services. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10981, pp. 38–47. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_3
9. Corzilius, F., Kremer, G., Junges, S., Schupp, S., Ábrahám, E.: SMT-RAT: an open source C++ toolbox for strategic and parallel SMT solving. In: Heule, M., Weaver, S.A. (eds.) Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9340, pp. 360–368. Springer (2015). https://doi.org/10.1007/978-3-319-24318-4_26
10. Dobal, F.: DeltaSMT for SMT-LIBv2 (2013), updated version of [7], unavailable
11. Aina Niemetz and Mathias Preiner and Gereon Kremer: ddSMTv2 documentation, https://ddsmt.readthedocs.io
12. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 737–744. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_49
13. Aina Niemetz and Mathias Preiner and Gereon Kremer: ddSMTv2, https://github.com/ddsmt/ddsmt
14. Kremer, G.: pyDelta (2021), https://github.com/nafur/pydelta
15. Kremer, G., Nalbach, J.: delta, https://github.com/smtrat/smtrat/tree/master/src/delta, SMT-RAT's delta debugger
16. Misherghi, G., Su, Z.: HDD: hierarchical delta debugging. In: Osterweil, L.J., Rombach, H.D., Soffa, M.L. (eds.) 28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006. pp. 142–151. ACM (2006). https://doi.org/10.1145/1134285.1134307

17. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
18. Niemetz, A., Biere, A.: ddSMT: a delta debugger for the SMT-LIB v2 format. In: Proceedings of the 11th International Workshop on Satisfiability Modulo Theories, SMT. pp. 8–9 (2013)
19. Niemetz, A., Preiner, M.: Bitwuzla at the SMT-COMP 2020. CoRR **abs/2006.01621** (2020), https://arxiv.org/abs/2006.01621
20. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. J. Satisf. Boolean Model. Comput. **9**(1), 53–58 (2014). https://doi.org/10.3233/sat190101
21. Raghothaman, M., Reynolds, A., Udupa, A.: The SyGuS Language Standard Version 2.0. Tech. rep. (2019), https://sygus.org/assets/pdf/SyGuS-IF_2.0.pdf
22. Ranise, S., Tinelli, C.: The SMT-LIB Standard: Version 1.2. Tech. rep., Department of Computer Science, The University of Iowa (2006), https://smtlib.cs.uiowa.edu/papers/format-v1.2-r06.08.30.pdf
23. Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., Yang, X.: Test-case reduction for C compiler bugs. In: Vitek, J., Lin, H., Tip, F. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012. pp. 335–346. ACM (2012). https://doi.org/10.1145/2254064.2254104
24. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Trans. Software Eng. **28**(2), 183–200 (2002). https://doi.org/10.1109/32.988498