ddSMT: A Delta Debugger for the SMT-LIB v2 Format*

Aina Niemetz and Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria
http://fmv.jku.at/

Abstract

Delta debugging tools automatically minimize failure-inducing input and enable efficient localization of erroneous code. In particular when debugging complex verification backends such as SMT solvers, delta debuggers provide an effective debugging approach where other debugging techniques are infeasible due to the input formula size. In this paper, we present ddSMT, a delta debugger for the SMT-LIB v2 format, which supports all SMT-LIB v2 logics and in particular handles macros and scopes defined by the commands *push* and *pop*. We introduce its architecture and describe its workflow in detail.

1 Introduction

Delta debugging algorithms [1, 5, 6, 7, 9] based on the algorithms introduced in [8, 10] typically minimize failure-inducing input by omitting parts irrelevant to the original erroneous behaviour. The resulting simplified failure-inducing input represents a minimal configuration in the sense that all of its possible subsets are necessary to cause the test to fail. Sat Modulo Theories (SMT) solvers serve as a backend for various applications in the field of e.g. deductive software verification, model checking and automated test generation. These applications heavily rely on the correctness of the underlying SMT solver – a highly complex tool, where debugging faulty behaviour becomes increasingly difficult with respect to the input formula size and structure. Rather than manually tracing error paths in order to find the actual error location, delta debugging provides means to automatically minimize input for failing SMT solvers and enables solver developers to localize failure-inducing code in a time efficient manner. Further, as shown in [5], delta debugging in combination with fuzz testing is a particularly effective approach to uncover bugs in SMT solvers.

In 2009, deltaSMT, a delta debugger for quantifier-free logics of the previous SMT-LIB [3] version[1] developed by our group has been presented in [5]. It is tailored to the SMT-LIB v1 language, hence incompatible with SMT-LIB v2 [4], which is a major upgrade of its predecessor. Further, deltaSMT does not employ the original delta debugging algorithm proposed in [8], but exploits the hierarchical structure of the input formula similar to the hierarchical delta debugging approach described in [9]. Representing the input formula as a directed acyclic graph (DAG), deltaSMT tries to simplify nodes in a breadth first search (BFS) manner. Nodes are substituted one-by-one, depending on their sort, with either constant 0, constant 1, or one of their children. Unfortunately, this substitution approach is also one of the limitations of deltaSMT, as in the worst case, too many node-by-node substitution attempts (no matter if successful or unsuccessful) have a negative impact on the overall runtime. Further, we encountered various cases, where deltaSMT was struggling or even unable to simplify certain input files.

*This work was partially funded by the Austrian Science Fund (FWF) under NFN Grant S11408-N23 (RiSE).

[1] http://smtlib.cs.uiowa.edu/papers/format-v1.2-r06.08.30.pdf

```
1  (set-logic UFNIA)
2  (declare-sort sort1 0)
3  (declare-fun x () sort1)
4  (declare-fun y () sort1)
5  (assert (= x y ))
6  (push 1)
7    (define-sort sort2 () Bool)
8    (declare-fun x () sort2)
9    (declare-fun y () sort2)
10   (assert (and (as x Bool) (as y Bool)))
11   (assert (! (not (as x Bool)) :named z))
12   (assert z)
13 (pop 1)
14 (assert (forall ((z Int)) (exists ((zz Int)) (= z zz))))
15 (check-sat)
16 (get-value ((let ((x 1) (y 1)) (= x y))))
17 (exit)
```

Figure 1: A simple example in SMT-LIB v2 format.

More recently and independently, an update of deltaSMT for SMT-LIB v2 by Pablo Federico Dobal and Pascal Fontaine has been released[2]. This version does not provide full SMT-LIB v2 support but syntactically extends the original tool for SMT-LIB v2 compliance, but without support for important new SMT-LIB v2 features such as quantifiers or *push* and *pop* commands. Note that in the following, we will refer to this update of deltaSMT as deltaSMT2.

In this paper we present ddSMT, a delta debugger for the SMT-LIB v2 format. It supports all SMT-LIB v2 logics. It is not based on deltaSMT, but tries to overcome its limitations with a different algorithmic approach, which we will introduce in detail in the following.

2 The Delta Debugger ddSMT

The delta debugger ddSMT is a tool for minimizing failure-inducing input in SMT-LIB v2 format based on the exit code of a given command (typically a call to an SMT solver) when executed on that input. It is implemented in Python 3 and not only supports all SMT-LIB v2 logics, but in particular handles macros (command *define-fun*), named annotations (attribute *:named*), and scopes defined by the commands *push* and *pop*. The tool is intended to be easy to maintain and extend and further provides a dedicated, modular and standalone SMT-LIB v2 parser, which particularly should be useful for prototyping other (Python) tools working on the SMT-LIB v2 language.

2.1 Architecture

One of the challenges introduced in v2 of the SMT-LIB language is the addition of the commands *push* and *pop*, which enables scoping of assertions, and sort and function declarations. Hence, SMT-LIB v2 distinguishes between local scoping of sorted variables and variable bindings (as defined by *forall*, *exists* and *let* terms) and global scoping as defined by the commands *push* and *pop*. Note that in the following, if distinction is needed, we refer to locally defined scopes as *term-level scopes*, and globally defined scopes as *command-level* scopes. Further note that ddSMT does not distinguish between actual functions and variables (or uninterpreted constants

[2] `http://www.verit-solver.org/veriT-toolsDownload.php`

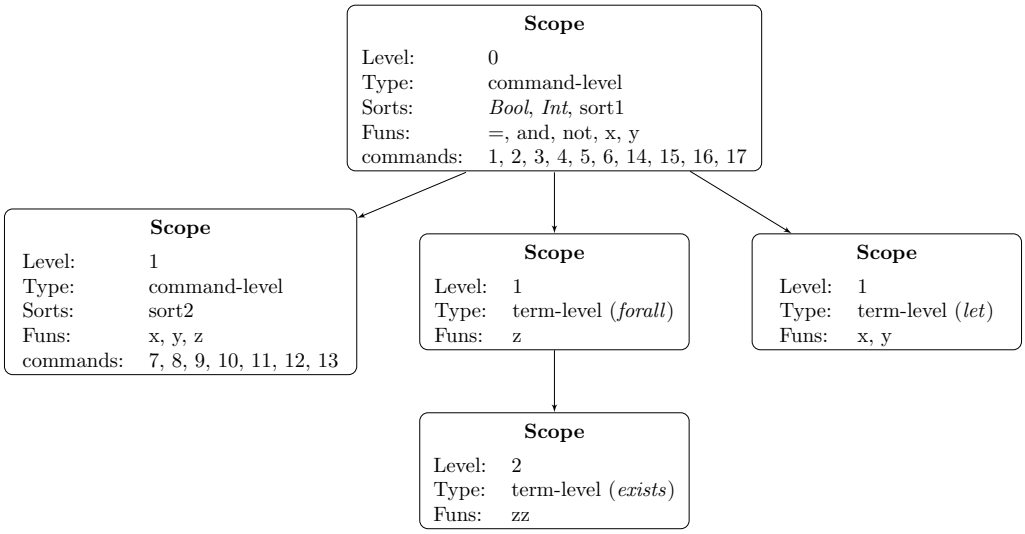

Figure 2: The basic internal structure of ddSMT given the Example in Figure 1.

in first order terminology) explicitly. Hence, in the following, if we do not make an explicit distinction, *function* may refer to either of them.

Internally the tool represents the given SMT-LIB v2 input as a tree of scopes. Each scope maintains a nesting level, a set of nested scopes, and a set of functions. Command-level scopes additionally maintain a set of commands and a set of sorts. Note that this structure enables a visibility handling of sorts and functions similar to related techniques in compiler construction, where a sorts (resp. functions) cache provides access to currently visible sorts (resp. functions) in constant time.

Example 1. To illustrate the basic internal structure of ddSMT as described above, consider the input file given in Fig. 1. As shown in Fig. 2, it defines two command-level scopes (the root scope at level 0 and the scope defined by given *push* and *pop* commands at level 1), and three term-level scopes defined by given *forall*, *exists* and *let* terms, respectively.

All sorts and functions defined at theory level are treated as being defined at level 0. Further, named annotations (attribute *:named*) are internally handled as if additionally a corresponding function definition had been given (in this particular case: `(define-fun z () Bool (not x))`). Commands are maintained by the scope they appear in, with a *push* command as the last command before a new scope is opened, and a *pop* command as the last command before the current scope is closed. Note that for better readability, we refer to the resp. commands in Fig. 2 by the line number they appear in Fig. 1.

In our example, the root scope maintains the predefined sorts *Bool* and *Int*, as well as the user-defined sort *sort1*. It further declares the predefined functions =, *and* and *not*, and the user-defined functions x and y (both of sort *sort1*). The command-level scope at level 1 maintains the user-defined sort *sort2*, and further declares functions x and y (both of sort *sort2*) and the named annotation z (of sort *Bool*). The term-level scope defined by *forall* at level 1 declares variable z of sort *Int*, whereas its nested term-level scope defined by *exists* at level 2 declares variable zz of sort *Int*. Finally, the term-level scope defined by *let* at level 1 declares variables x and y of sort *Int*.

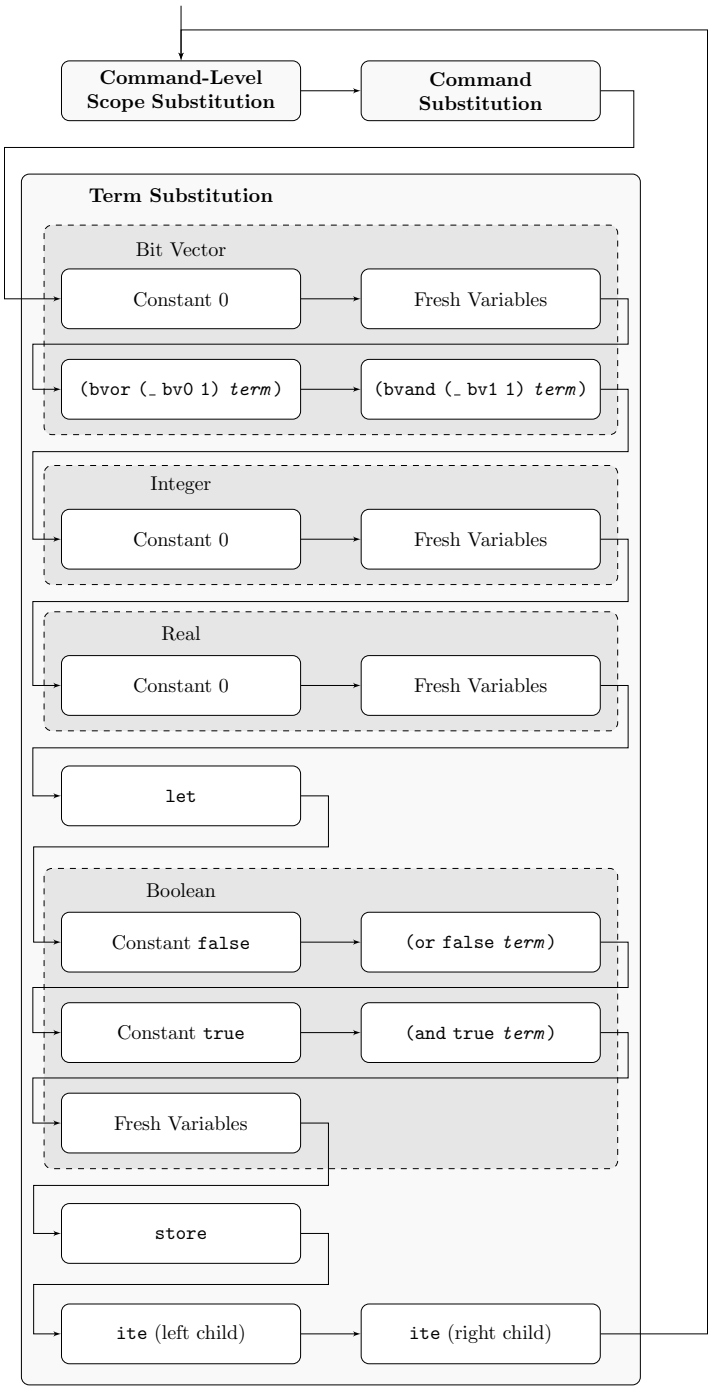

Figure 3: The general workflow and delta debugging phases of ddSMT .

2.2 General Workflow

The SMT-LIB v2 input is simplified by eliminating command-level scopes and commands, and substituting terms with simplified expressions. Note that *eliminating* scopes resp. commands refers to *substituting* nodes by *None* (the Python null object). In contrast to deltaSMT, ddSMT does not employ a hierarchical delta debugging approach on a BFS and node-by-node substitution base, but tries to exploit the strength of the original delta debugging algorithm (a divide-and-conquer strategy) as follows.

As illustrated in Fig. 3, ddSMT works in rounds, where each round is divided into several substitution phases. In each phase, nodes are first filtered and collected by a specific characteristic (e.g. nodes with a bit vector sort), and then substituted using a modified version of the original delta debugging algorithm as described in Fig. 8. The individual substitution phases are described as follows.

Command-level scope substitution Starting with the nested scopes of the root scope, command-level scopes are eliminated level-wise, in BFS manner, until a fixpoint is reached.

Command substitution After the command-level scope substitution phase, any command in any of the remaining command-level scopes irrelevant to the original failure-induced behaviour except the *set-logic* and *exit* commands, which are mandatory for starting and terminating SMT-LIB v2 scripts, is eliminated (while preserving the order of remaining commands) until a fixpoint is reached. Note that in the initial round, in order to prevent lots of likely unsuccessful test runs when eliminating e.g. *declare-fun* commands previous to term substitution, ddSMT considers *assert* commands only.

Further note that ddSMT does not ensure that the resulting simplified output is legal in the sense that e.g. variables must be declared previous to being used – the elimination of commands is solely tied to the exit code of the given command. This usually does not pose a problem though, as this kind of syntactically invalid input should be treated accordingly by a tool working on the SMT-LIB v2 language. In case the above behaviour poses a problem, e.g. when debugging parser related faulty behaviour, this can be easily handled by appropriate wrapper scripts (e.g. to check on specific solver output).

Term substitution Internally, ddSMT represents SMT-LIB v2 terms as DAGs with exactly one root. The SMT-LIB v2 format defines three commands with terms as arguments: *assert*, *define-fun* and *get-value*. Commands of each of these kinds are handled separately, with *define-fun* commands being processed prior to *assert* and *get-value* commands in order to prevent redundant substitution work due to the fact that functions defined via *define-fun* are usually referenced in *assert* and *get-value* commands multiple times. For each of these sets of commands, term substitution replaces terms w.r.t. their resp. sort (and other characteristics) in several steps as indicated in Fig. 3 until a fixpoint is reached. Note that individual steps (e.g. substitution of bit vector terms with constant 0) are defined by the characteristics of both the terms to be substituted and the substitution itself. Further note that steps depending on the SMT-LIB v2 logic in use are skipped if inapplicable (e.g. the substitution of *Real* terms with constant 0 or fresh variables if given logic is not a *Reals* logic). The individual steps are described as follows.

Initially, and depending on the SMT-LIB v2 logic in use, first bit vector, then *Int*, then *Real* terms are substituted with constant 0 and fresh variables, respectively. Additionally, if given formula is defined over the theory of *Fixed_Size_Bit_Vectors*, terms of the form `(bvor (_`

`bv0 1)` *term*`)` and `(bvand (_ bv1 1)` *term*`)` are replaced by their resp. child *term*. Next, *let* terms are replaced by their child term. After that, Boolean terms are substituted by constant *false*, constant *true*, and fresh variables, respectively. Subsequently, terms of the form `(or false` *term*`)` and `(and true` *term*`)` are replaced by their resp. child *term*. If the logic in use is an array logic, *store* terms are replaced by their child array terms. Finally, *ite* terms are substituted with their left and right child, respectively.

Note that in those steps of the term substitution phase, where terms are replaced with a simpler expression rather than one of their child terms (e.g. substituting Boolean terms with constant *false*), constant terms are skipped. Further note that each step of the term substitution phase (e.g. substituting bit vector terms with constant 0) is performed until a fixpoint is reached.

If any of the above substitution phases succeeded, i.e. if in any of the above phases, scopes, commands or terms have been eliminated or replaced successfully, ddSMT tries to iteratively simplify the current configuration even further until a fixpoint is reached.

Example 2. Continuing Ex. 1, consider the input file given in Fig. 1 and an executable failing on this input by not providing support for *get-value* commands as simulated by the Shell script given in Fig. 4. The input file is simplified by ddSMT in two rounds as follows.

In round one, first all redundant command-level scopes are eliminated. In this case, the scope defined by the *push* and *pop* commands in line 6 and 13 is redundant. The resulting simplified input is depicted in Fig. 5a. Next, all commands irrelevant to the failure-induced behaviour are successfully eliminated. As mentioned earlier, in the first round command substitution only considers *assert* commands. Hence, commands 5 and 6 (but not command 7, which is a *check-sat* command) are eliminated. The resulting simplified input is depicted in Fig. 5b. After command substitution, ddSMT subsequently performs term substitution on argument terms of *define-fun*, *assert* and *get-value* commands, in the order specified. As the current simplified input (as depicted in Fig. 5b) only contains a single *get-value* command, term substitution for *define-fun* and *assert* commands is skipped and the argument term of the *get-value* command at line 6 is the only one to be processed as follows. The original input in Fig. 1 is defined over the theory of *Ints* (but not over the theory of *Fixed_Size_Bit_Vectors* or *Reals*), hence all bit vector and *Reals* related steps are skipped. The *let* expression in line 6 contains two non-constant *Int* terms, `x` and `y`, which are first (and successfully) replaced by constant 0. The resulting simplified input is depicted in Fig. 6a. As no more non-constant *Int* terms remain, subsequent substitution with fresh variables is skipped. Next, the *let* term is successfully replaced by its child term (due to the fact that all occurrences of its variable bindings have been substituted by constant 0, previously). The resulting simplified input is depicted in Fig. 6b. Finally, the remaining non-constant Boolean term `(= 0 0)` is successfully replaced by constant *false*. As depicted in Fig. 6c, in the current simplified input the only remaining term (in the argument term of the *get-value* command at line 6) is a Boolean constant. Hence, all further term substitution steps operating on Boolean and *ite* terms are skipped and the first round concludes with the simplified input depicted in Fig. 6c.

In round two, the only successful substitution phase is command substitution, where commands 2, 3, and 4 are eliminated. The final result is depicted in Fig. 7.

2.3 `substitute`: The Delta Debugging Core Algorithm

The core of the actual delta debugging in ddSMT is the substitution algorithm described in Fig. 8. Command-level scopes and commands are substituted with *None*, whereas terms, depending

```
1  #!/bin/sh
2  if [ `grep -c "\<get-value\>" $1` -ne 0 ]; then exit 1 fi
3  exit 0
```

Figure 4: A simple Shell script simulating an executable failing on the input given in Fig. 1.

```
1  (set-logic UFNIA)
2  (declare-sort sort1 0)
3  (declare-fun x () sort1)
4  (declare-fun y () sort1)
5  (assert (= x y))
6  (assert (forall ((z Int)) (exists ((zz Int)) (= z zz))))
7  (check-sat)
8  (get-value ((let ((x 1) (y 1)) (= x y))))
9  (exit)
```

(a) The simplified input after command-level scope substitution.

```
1  (set-logic UFNIA)
2  (declare-sort sort1 0)
3  (declare-fun x () sort1)
4  (declare-fun y () sort1)
5  (check-sat)
6  (get-value ((let ((x 1) (y 1)) (= x y))))
7  (exit)
```

(b) The simplified input after subsequent command substitution.

Figure 5: The input of Fig. 1 during the first substitution round in Ex. 2.

```
1  (set-logic UFNIA)
2  (declare-sort sort1 0)
3  (declare-fun x () sort1)
4  (declare-fun y () sort1)
5  (check-sat)
6  (get-value ((let ((x 1) (y 1)) (= 0 0))))
7  (exit)
```

(a) The result of substituting non-constant *Int* terms with constant 0.

```
1  (set-logic UFNIA)
2  (declare-sort sort1 0)
3  (declare-fun x () sort1)
4  (declare-fun y () sort1)
5  (check-sat)
6  (get-value ((= 0 0)))
7  (exit)
```

(b) The result of substituting the *let* term with its child term.

```
1  (set-logic UFNIA)
2  (declare-sort sort1 0)
3  (declare-fun x () sort1)
4  (declare-fun y () sort1)
5  (check-sat)
6  (get-value (false))
7  (exit)
```

(c) The result of substituting the remaining Boolean term with constant *false*.

Figure 6: Continuing from Fig. 5, all three simplified inputs are the result of individual steps of term substitution in the first round.

```
1  (set-logic UFNIA)
2  (check-sat)
3  (get-value (false))
4  (exit)
```

Figure 7: The final result of simplifying the input of Fig. 1 in Ex. 2 after the second substitution round. In round two, commands 2, 3, and 4 are eliminated during command substitution.

```
1  def substitute (subst_fun, superset):
2      granularity = len(superset)
3      while granularity > 0:
4          nsubsets = len(superset) / granularity
5          subsets = split(superset, nsubsets)
6          for subset in subsets:
7              nsubsts = 0
8              for item in subset:
9                  if not item.is_substituted():
10                     item.substitute_with(subst_fun(item))
11                     nsubsts += 1
12             if nsubsts == 0:
13                 continue
14
15             dump (tmpfile)
16
17             if test():
18                 dump(outfile)
19                 subsets.delete(subset)
20             else:
21                 # reset substitutions of current subset
22                 restore_previous_state()
23         superset = subsets.flatten()
24         granularity = granularity / 2
```

Figure 8: The core substitution algorithm in ddSMT in Python-style pseudo code.

on their sort, are replaced by constant 0, *false*, *true*, fresh variables, or one of their children, respectively. Each substitution phase utilizes `substitute` as follows. Given a substitution function `subst_fun` and a set of nodes filtered by some specific filter criteria (e.g. nodes with a bit vector sort) as `superset`, this set is gradually split into `nsubsts` subsets, where the `granularity`, i.e. the number of items, of each subset initially starts at `len(superset)`. Note that this basically means that in a first attempt, all nodes of `superset` will be substituted. For each `subset` of these subsets, all items are substituted by the application of the substitution function `subst_fun` to the resp. item before issuing the original command (usually a call to an SMT solver) on the current configuration. If this (so called) test run succeeds, i.e. if the exit code of the current run matches the exit code of the original configuration, the current simplified input is stored for immediate reuse in `outfile`. Otherwise, all substitutions of the current `subset` are reset and we continue with the next subset.

Note that previously substituted nodes will be skipped. This is due to the fact that `superset` initially contains either the original node (if it is yet to be substituted) or its most current substitution.

	TS	Files	Red. [%]			Time [s]			Runs			Mem. [MB]		
			avg	min	max	avg	min	max	avg	min	max	avg	min	max
deltaSMT	1	2	0	0	0	257	14	500	4051	655	7446	113	108	117
	2	95	94.0	0	99.9	49	0.1	1738	599	5	7296	111	33	153
	3	5	66.6	0	93.8	12	3	34	608	262	1297	107	76	126
	4	53	99.6	98.8	99.9	8	0.6	20	463	4	852	128	52	142
	5	-	-	-	-	-	-	-	-	-	-	-	-	-
ddSMT	1	2	90.0	83.9	96.0	44	9	79	1412	782	2041	13	10	16
	2	95	94.7	68.2	99.9	92	0.1	1594	1499	2	3790	15	10	24
	3	5	80.4	66.8	87.2	23	14	35	1533	1171	1764	11	10	12
	4	53	99.8	99.3	99.9	57	1	246	431	13	1240	28	15	42
	5	5	97.4	95.7	98.3	12	5	16	247	215	371	39	10	59

Table 1: Comparison between deltaSMT (for SMT-LIB v1) and ddSMT on test sets (*TS*) 1 to 5. Test set 1 to 4 are randomly generated bit vector formulas originally given in SMT-LIB v1, test set 5 contains non-quantifier-free test cases for CVC4. *Red.* denotes the overall reduction in percent of the original file size, *Time* denotes the overall runtime in seconds, and *Mem.* denotes the maximum resident set size in MB.

3 Experimental Evaluation

Our delta debugger ddSMT has recently been released[3] under version 3 of the General Public License (GPLv3)[4] and is currently still a work in progress. Its parser is tested on the complete SMT-LIB v2 benchmark set (available at [3]) and the delta debugger itself has been tested on a wide range of crafted instances and SMT-LIB v2 benchmarks using simple shell scripts in place of actual solver calls in order to achieve a wider distribution over the SMT-LIB v2 logics. Additionally, ddSMT has further been applied to actual failure inducing test cases encountered during the development of our solver Boolector[5], as well as the open source SMT solver CVC4[6], a joint effort between the NYU and the University of Iowa.

As of May 23rd 2013, deltaSMT2, which we understand to be still work in progress, does not produce legal intermediate output for bit vector logics and is thus not able to simplify any of the test cases available for Boolector. Further, deltaSMT2 does not support non-quantifier-free logics such as AUFLIA or AUFLIRA and is hence not applicable to any of the test cases available for CVC4. Unfortunately, it was therefore not possible to evaluate the overall performance of ddSMT in comparison to deltaSMT2 . Instead, we translated quantifier-free SMT-LIB v1 input to SMT-LIB v2 and run deltaSMT-0.2 and ddSMT-0.96-beta on various sets of test cases (*TS*) as indicated in Table 1. Test sets 1 to 4 are randomly generated bit vector formulas originally given in SMT-LIB v1 and serve as test cases for Boolector, whereas test set 5 contains non-quantifier-free SMT-LIB v2 test cases for CVC4. Input reduction (*Red.*) is given in percent of the file size of the original input file, *Time* denotes the wall clock runtime in seconds, and *Mem.* indicates the maximum resident set size per run in MB. All experiments were performed on a 3.4 GHz Intel Core i7-2600 machine with 16GB RAM, running a 64 Bit Arch Linux OS.

Overall and even though the bit vector test cases where originally given in SMT-LIB v1 (i.e. they do not employ SMT-LIB v2 features such as e.g. *push* and *pop* commands, which could be fully exploited by ddSMT), our first results look promising. Even for test cases, where deltaSMT failed to simplify given input at all, ddSMT successfully achieved reductions by at least 81.1% of the original input file size. Note that except for the test cases denoted in Table 1,

[3] `http://fmv.jku.at/ddsmt`
[4] http://www.gnu.org/licenses
[5] `http://fmv.jku.at/boolector`
[6] http://cvc4.cs.nyu.edu

we currently still miss real test cases in SMT-LIB v2 logics other than QF_BV, QF_AX and QF_AUFBV. We therefore would like to encourage the SMT community to actually use ddSMT and thus further its development, and appreciate any comments, suggestions or bug reports.

4 Conclusion

In this paper, we introduced our delta debugger ddSMT, a tool for minimizing failure-inducing input in SMT-LIB v2 format. It supports all SMT-LIB v2 logics and in particular handles macros, named annotations, and scopes defined by the commands *push* and *pop*. Especially in combination with fuzz testing, ddSMT provides an effective approach to find and localize bugs in tools working on the SMT-LIB v2 language.

Recently, model-based delta-debugging (and fuzzing) in the context of testing and debugging verification backends was reported to be more effective than file based delta-debugging [2], in particular in combination with option resp. configuration fuzzing. Even though the delta-debugger ddSMT presented in this paper does not work on the API level of an SMT solver directly, we believe that the "programmatic nature" of the SMT-LIB v2 format using commands allows ddSMT to be equally effective.

In future work we will compare the effectiveness of API level delta-debugging with the approach presented in this paper. We further plan to evaluate ddSMT in combination with fuzzing SMT-LIB v2 input with command-level scopes.

We want to thank Morgan Deters for providing actual test cases for the SMT solver CVC4.

References

[1] Cyrille Artho. Iterative Delta Debugging. In Hana Chockler and Alan J. Hu, editors, *Haifa Verification Conference*, volume 5394 of *Lecture Notes in Computer Science*, pages 99–113. Springer, 2008.

[2] Cyrille Artho, Armin Biere, and Martina Seidl. Model-based testing for verification backends. In *Proc. 7th Intl. Conf. on Tests & Proofs (TAP'13)*, LNCS, page 17 pages. Springer, 2013. To be published.

[3] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[4] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In Aarti Gupta and Darti Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

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

[6] Robert Brummayer and Matti Järvisalo. Testing and debugging techniques for answer set solver development. *TPLP*, 10(4-6):741–758, 2010.

[7] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated Testing and Debugging of SAT and QBF Solvers. In Ofer Strichman and Stefan Szeider, editors, *SAT*, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2010.

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

[9] Ghassan Misherghi and Zhendong Su. HDD: hierarchical Delta Debugging. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *ICSE*, pages 142–151. ACM, 2006.

[10] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.