Automated Testing and Debugging of SAT and QBF Solvers

Robert Brummayer, Florian Lonsing, and Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University Linz, Austria

Abstract. Robustness and correctness are essential criteria for SAT and QBF solvers. We develop automated testing and debugging techniques designed and optimized for SAT and QBF solver development. Our fuzz testing techniques are able to find critical solver defects that lead to crashes, invalid satisfying assignments and incorrect satisfiability results. Moreover, we show that sequential and concurrent delta debugging techniques are highly effective in minimizing failure-inducing inputs.

1 Introduction

Satisfiability solving has been shown to be a competitive problem solving technique that is used in many different domains such as verification, test case generation, scheduling, computational biology and artificial intelligence. For a recent survey on satisfiability solving we refer the reader to [8]. Recent advances of propositional satisfiability (SAT) solvers and quantified boolean formula (QBF) solvers are driven by competitions and real industrial applications such as formal hardware and software verification.

Essential criteria of satisfiability solvers are robustness and correctness. SAT and QBF solvers are used as core decision engines and the clients heavily depend on these important criteria. For instance, an incorrect SAT solver used as decision engine in a formal verification framework may lead to incorrect verification results, i.e. either the system may be spuriously proven to be correct or the verification framework generates a spurious counter-example. Moreover, wrong satisfying assignments (models) may be mapped to spurious verification counter-examples that hinder the overall verification process.

While a large part of current research focuses on speeding up SAT and QBF solving with various techniques such as improved decision heuristics and low-level optimizations, there are, to the best of our knowledge, no rigorous scientific publications about automated testing and debugging techniques for SAT and QBF solvers. This paper tries to improve this situation by introducing automated state-of-the-art testing and (multi-threaded) delta debugging techniques, designed and optimized for SAT and QBF solvers. Our experimental results are available at `http://fmv.jku.at/brummayer/fuzz-dd-sat-qbf.tar.7z`. Every tool is available at `http://fmv.jku.at/software/`.

O. Strichman and S. Szeider (Eds.): SAT 2010, LNCS 6175, pp. 44–57, 2010.

2 Fuzzing

Fuzzing is an automated negative testing technique, typically used in software security and quality assurance [45,46]. The original idea is to treat software as a black-box and repeatedly "attack" it with random inputs in order to find critical defects, e.g. buffer overflows. Fuzz testing methods such as "monkey testing" were already used around 1980 [46]. Miller, one of the fuzzing pioneers, demonstrated that fuzz testing could find many critical defects in UNIX applications [36]. The lack of a formal model and the brute force nature of the approach lead to the situation that papers about fuzz testing were often offended. Miller simply responded that he was just trying to find bugs [46], which is also exactly what we want to achieve with our fuzzing techniques, explicitly designed and optimized for SAT and QBF solvers.

The goal of previous work on random generation of SAT and QBF instances was to study the phase transition phenomenon [22,40,23,16] and to generate hard instances [27,48,1]. However, our work focuses on generating random instances in order to *find defects* in current state-of-the-art solver implementations. We propose to use *grammar-based black-box fuzzing* in order to test SAT and QBF solvers. A fuzzer repeatedly generates syntactically valid inputs. Solvers are treated as black-boxes, which makes our approach highly flexible. They are run on the generated inputs in order to detect critical defects such as segmentation faults and aborts. Moreover, reported satisfiability results are validated in order to find defects that lead to incorrect results and models.

One of the main success factors of fuzz testing is a high test throughput, e.g. testing a solver with five instances per hour is unlikely to be successful. Therefore, generating hard instances solely is counter-productive. On the other hand, trivial instances are unlikely to trigger interesting defects. Ideally, a fuzzer should be able to generate a variety of different inputs that lead to the execution of different paths in the tested solver. The majority of the generated instances should be easy to solve in order to maintain a sufficiently high test throughput. The combination of automation, diversity and high throughput makes fuzz testing an effective negative testing technique. Our experiments in section 2.3 show that this technique can be successfully applied to SAT and QBF solvers.

In the following we describe our novel fuzzing techniques for SAT and QBF solvers, implemented in our fuzzers `CNFuzz`, `FuzzSAT` and `QBFuzz`. Due to the probabilistic nature of fuzzing, our fuzzers use magic constants found through direct experimentation. All fuzzing approaches use a random number generator. We assume that picks during fuzzing are performed uniformly at random.

2.1 SAT Fuzzing

3SATGen. Easy to solve instances do not exercise solvers enough. Therefore, it is unlikely to find interesting defects with easy instances alone. However, as research [22] on the phase transition in random 3-SAT suggests, it is straightforward to write a random CNF generator that generates reasonable hard instances. In our view, this is an important application of [22].

Our 3SAT generator `3SATGen` is based on this technique and works as follows. First, the number of variables m is picked, typically between 10 and 400 variables. The next step is to determine a clause variable ratio r, which should be around the hardness threshold, typically between 3 and 5. Finally, $m \cdot r$ random ternary clauses are generated, where each literal is picked uniformly.

CNFuzz. Random 3-SAT formulas are lacking structure. However, the success of SAT solvers in industry seems to rely on their ability to use structure, at least implicitly, even though we do not actually know how to describe this connection in a more formal way. This rather vague argument implies that `3SAT` does not exercise all the interesting features of an industrial SAT solver. Therefore, we were looking for other ways to generate "more structured" instances. Our fuzzer `CNFuzz` enforces certain locality restrictions and thus generates instances that contain more internal structure than the simple 3SAT approach.

The CNF generated by `CNFuzz` consists of $l \in [1, 20]$ layers of maximum width $w \in [10, 70]$. Both numbers are picked randomly within these ranges. The i-th layer with $i \in [1, l]$ introduces $n_i \in [10, w]$ new variables, again chosen randomly. Each layer is associated with a separately "picked-clause-variable-ratio" $r_i \in [3, 4.5]$ from which the number $c_i = r_i w_i$ of clauses in layer i is calculated. Clauses are at least ternary, and with exponentially decreasing probability longer: $2/3$ are expected to be ternary, $1/3 \cdot 2/3$ of length 4, $(1/3)^2 \cdot 2/3$ of length 5, etc. Variables are picked either from the same or from smaller layers. The layer from which a variable is picked is determined in a similar way as the length of clauses. A variable from layer i is picked with probability $1/2$, from layer $i - 1$ with probability $1/4$, from layer $i - 2$ with probability $1/8$, etc., down to the first layer, which accumulates the remaining probabilities. As a further refinement of the iterative clause generation process, variables that have not been selected are preferred within the same layer.

FuzzSAT. The even more structured approach of `FuzzSAT` is based on the translation of boolean circuits into CNF. To be more precise, a directed acyclic graph (DAG) representation of a random boolean circuit is generated. The generated DAG is converted into CNF by using the Tseitin transformation [47] afterwards.

The boolean circuit DAG is constructed as follows. First, $v \in [1, 100]$ boolean input nodes are generated and inserted into a global set n, which is a container for all nodes generated during the construction process. Then, in the core routine of our DAG generation approach, we randomly select a boolean operator *op* from the set of operators $O = \{\text{AND, OR, XOR, IFF}\}$. Moreover, we select two operands o_1 and o_2 from n, negate each operand with probability $1/2$, generate the new operator node, and insert it into n. This process is repeated until each original input variable is referenced at least t times, where t is usually 1.

Then, we take the set r of all boolean roots, i.e. generated operators that are not referenced by other operators, and combine them to one boolean root as follows. We select a boolean operator *op* from O, select two operands r_1 and r_2 from r, negate each of them with probability $1/2$, remove r_1 and r_2 from r,

generate the new root, and insert it into r. This process is repeated until there is only one root left. Then, we perform the Tseitin transformation on this root.

Let c be the number of clauses generated so far and let p be a probability $\in [0.01, 0.1]$. Finally, $c \cdot p$ random clauses of varying size $s \in [2, 6]$ are added to the CNF in order to increase diversity. The size of the additional random clauses is picked for each clause individually.

2.2 QBF Fuzzing

A quantified boolean formula (QBF) $F = B_1 \dots B_n . \phi$ in *prenex conjunctive normal form (PCNF)* consists of a propositional formula ϕ in CNF over a set of variables V and a *quantifier prefix* $B_1 \dots B_n$. The quantifier prefix is a linearly ordered set of *blocks* B_i where $B_1 < \dots < B_n$, forming a partition on V.

A block B_i is *existential* ($q(B_i) = \exists$) if it is associated with an existential quantifier and *universal* ($q(B_i) = \forall$) otherwise. For two adjacent blocks B_i and B_{i+1} where $1 \leq i < n$, $q(B_i) \neq q(B_{i+1})$.

A clause C is *forall-reduced* [12] if for every literal $l \in C$ with $l \in B_i$ and $q(B_i) = \forall$ there is a literal $k \in C$ with $k \in B_j$ and $q(B_j) = \exists$ and $i < k$.

In the following we describe two different approaches for QBF fuzz testing. All generated formulas are in PCNF and contain forall-reduced clauses only.

Random QBF model. We have implemented a QBF fuzzer `BlocksQBF` which generates random QBFs in PCNF according to the model described in [14]. This model is an extension of an approach originally introduced in [13], which was further improved in [23].

The model [14] used in our fuzzer has the following parameters: the number of clauses n_c, the number of quantifier blocks n_b in the prefix $B_1 \dots B_{n_b}$, the number of variables $n_{v,1}, \dots, n_{v,i}, \dots, n_{v,n_b}$ in each block B_i for $1 \leq i \leq n_b$ and the number of literals $n_{l,1}, \dots, n_{l,i}, \dots, n_{l,n_b}$ taken from a block B_i to appear in each clause, where $n_{l,i} \leq n_{v,i}$ for $1 \leq i \leq n_b$. By convention, always $q(B_{n_b}) = \exists$, i.e. all clauses are forall-reduced by construction and have the same length.

We generate exactly n_c *distinct* clauses one after the other as follows. From each block B_i for $1 \leq i \leq n_b$ we select and negate exactly $n_{l,i}$ literals where $n_{l,i} \leq n_{v,i}$. Different from the description given in [14], complementary or duplicate literals in a clause are *always* discarded until a new literal is generated which can be added to the clause. This is possible as we never add more literals from a block B_i than there are variables in B_i (since $n_{l,i} \leq n_{v,i}$ for $1 \leq i \leq n_b$).

Newly generated clauses are added to the formula only if there is no duplicate clause already present. Otherwise the new clause is discarded and another attempt is carried out. This process continues until exactly n_c distinct clauses are generated, which is different from [14]. For improper parameter settings such as big n_c and very small $n_{v,i}$ it can be impossible to generate exactly n_c distinct clauses, but this was avoided in our experiments, where we used the following settings: $n_c = 160$, $n_b = 3$ (i.e. quantifier prefix $\exists\forall\exists$), block sizes $n_{v,1} = 15$, $n_{v,2} = 10$, $n_{v,3} = 25$ and $n_{l,1} = n_{l,2} = 2$, $n_{l,3} = 1$ literals taken from each block.

QBFuzz. The second QBF fuzzer `QBFuzz` we used in our experiments generates QBFs in PCNF which do not follow an exact model such as [13,23,14], leading to a higher diversity. The following parameters are *maximum* values: number of clauses n_c, number of variables n_v and number of blocks n_b. Further, minimum *min* and maximum number *max* of literals in a clause and the ratio $r \in [0, 1]$ of existential variables in the formulas and in each clause is specified.

Formulas according to the given setting are generated as follows: first a quantifier prefix is selected according to values of n_b, n_v and r, where the number of variables per block is selected at random. Next n_c clauses are generated of length $len \in [min, max]$ and each containing $r.len$ existential variables. Different from `BlocksQBF`, literals are selected from *any* block and are negated uniformly at random. As described above, duplicate and complementary literals are discarded. After generated clauses have been forall-reduced, duplicate clauses and unused variables are removed from the formula. We used the following settings in our experiments: $n_c = 80$, $n_v = 40$, $n_b = 15$, $min = 5$, $max = 15$ and $r = 0.4$.

2.3 Experiments

In order to evaluate our fuzzing techniques, we performed fuzz testing experiments with a selected subset of complete SAT solvers that participated in the SAT competition 2007 and 2009. Moreover, we fuzz tested several state-of-the-art QBF solvers. We ran our experiments under Ubuntu Linux on an Intel Core 2 Quad machine with 2.66 GHz and 8 GB RAM. Our fuzzing test framework used each of the four cores. The results of our fuzzing experiments with SAT solvers are shown in Tab. 1 and Tab. 2. The QBF results are shown in Tab. 3.

The results of our fuzz testing experiments with SAT solvers in Tab. 1 and Tab. 2 clearly show the overall effectiveness of our fuzz testing techniques. We were able to find serious defects such as segmentation faults, aborts, assertion failures, invalid models and incorrect results. We classified the defects into the following categories. Unexpected termination without providing a result was classified as an *error*. Cases where solvers reported an incorrect satisfiability status, i.e. a solver reported that an instance is unsatisfiable although the instance is provably satisfiable, were classified as *incorrect*. Finally, providing the correct satisfiability status but an invalid satisfying assignment was classified as *invalid model*, labeled *model* in tables Tab. 1 and Tab. 2. Notice that multiple observable failures may be caused by the same solver defect.

We used our tool `PrecoCheck` to validate models. Cases where we could not fully decide which satisfiability status is correct, e.g. some solvers claim that the instance is unsatisfiable and some others claim that instance is satisfiable, but provide an invalid model, did not occur. If all solvers agreed that the current instance is unsatisfiable, we did not further validate the unsatisfiability status as it is highly unlikely that all solvers are wrong.

We were able to find six defective solvers that participated in the SAT competition 2007. Notice that we did not test all solvers. We selected only a subset of the most competitive complete SAT solvers in order to demonstrate the effectiveness of our fuzzing techniques. Moreover, in order to keep our set of solvers

small, we did not test incomplete and portfolio-based solvers. Notice that our fuzz testing and delta debugging techniques can be applied to any kind of solver.

Five of the six SAT competition 2007 solvers shown in Tab. 1 have defects that lead to incorrect results, which we consider as the worst case that can happen. Incorrect results reported by the multi-threaded SAT solver MiraXTv3 are non-deterministic. Depending on the thread scheduling and the actual utilization of the individual processing cores, MiraXTv3 either reports that an instance is satisfiable or unsatisfiable. Moreover, our fuzzers detected that two SAT solvers generate invalid models. Notice that RSat respectively PicoSAT, were ranked first respectively second in the industrial category (satisfiable and unsatisfiable instances). Moreover, notice that March_ks was the second best solver in the random category (satisfiable and unsatisfiable instances).

Our fuzzing techniques were able to find three defective solvers that participated in the SAT competition 2009. We found critical defects causing segmentation faults in MiniSat-9z, the winner of the MiniSat hack track. Moreover, we found non-deterministic crashes in ManySat, which was the winner of the parallel solver application track. Finally, our fuzzer `FuzzSAT` was able to reveal that Mirch_hi, second best solver (SAT + UNSAT) and best solver (UNSAT) in the random track, sometimes generates invalid models.

None of the fuzzing techniques is clearly superior to the others, except that `CNFuzz` and `FuzzSAT` were able to find more varying defects as the simple 3SAT generator `3SATGen`. The restriction to 3SAT CNF instances may miss failures that occur if the input contains clauses of arbitrary size. Nevertheless, the 3SAT generator was still able to find defects in three of the six solvers, which is rather surprising as SAT solvers are typically tested with 3SAT instances. Interestingly, while `CNFuzz` was the only fuzzer that found defects in Barcelogic-fixed and incorrect results of Barcelogic, `FuzzSAT` was the only fuzzer that was able to

Table 1. Experimental results of fuzz testing SAT solvers from SAT competition 2007. The 3SAT generator `3SATGen` and our fuzzers `CNFuzz` and `FuzzSAT` generated 10000 CNF instances, respectively. We fuzz tested Barcelogic [9] and Barcelogic-fixed [9], CMUSAT [30], March_ks [26], MiniSat [18], MiraXTv3 [34], MXC [10], PicoSAT [7], RSat [41], Sat7 [32], SAT4J [4], Spear [2] and Tinisat [29]. All solver binaries were taken from the SAT competition 2007. Only solvers for which defects have been found are shown in the table. The testing time was about two hours for the 3SAT generator and `FuzzSAT`, respectively, and one hour for `CNFuzz`. For each solver and each CNF instance a time limit of thirty seconds was used.

	3SATGen			CNFuzz			FuzzSAT		
solver	error	incorrect	model	error	incorrect	model	error	incorrect	model
Barcelogic	0	0	0	1	**3**	1	1	0	1
Barcelogic-fixed	0	0	0	0	**1**	**1**	0	0	0
March_ks	**24**	2	0	5	0	0	2	2	0
MiraXTv3	26	7	0	91	**13**	0	**286**	2	0
PicoSAT	0	0	0	0	0	0	0	**2**	0
RSat	**56**	0	0	27	0	0	3	0	0

Table 2. Experimental results of fuzz testing SAT solvers from SAT competition 2009. The 3SAT generator `3SATGen` and our fuzzers `CNFuzz` and `FuzzSAT` generated 10000 CNF instances, respectively. We fuzz tested CirCUs [5], Clasp [20], Cumr_p [5], Glucose [5], LySATi [5], ManySAT [25], March_hi [26], MiniSat [18], MiniSat-9z [5], MXC [5], PicoSAT [7], PrecoSAT [5], RSat [5], SApperloT-base [5], SAT4J [4] and Varsat-industrial [28]. All solvers binaries were taken from the SAT competition 2009. Only solvers for which defects have been found are shown in the table. No discrepancies were found, i.e. all solvers agreed on the satisfiability status of each CNF instance. The testing time was about two hours for the 3SAT generator, three hours for `FuzzSAT` and one hour and thirty minutes for `CNFuzz`. For each solver and each instance a time limit of thirty seconds was used.

	3SATGen		CNFuzz		FuzzSAT	
solver	error	model	error	model	error	model
ManySat	2	0	56	0	**836**	0
March_hi	0	0	0	0	0	**24**
MiniSat-9z	2	0	58	0	**852**	0

Table 3. Experimental results of fuzz testing QBF solvers with `BlocksQBF` and `QBFuzz`. Both generated 10000 CNF instances, respectively. We fuzz tested an internal version of DepQBF [35], MiniQBF-090608 [43], QMRES [39], Quantor-3.0 [6], QuBE6.0 [24], QuBE6.5 [24], QuBE6.6 [24], Semprop-010604 [33], sKizzo-0.8.2 [3], SQBF-1.0 [44], Squolem-1.03 [31] and yQuaffle-021006 [53]. The fuzz testing time was one hour and fifteen minutes for `QBFuzz` and one hour and twenty minutes for `BlocksQBF`. Only solvers for which defects have been found are shown in the table. For each solver and each instance a time limit of thirty seconds was used.

	BlocksQBF		QBFuzz	
solver	error	incorrect	error	incorrect
Quantor	0	0	1	0
QuBE 6.0	0	684	5	7
QuBE 6.5	0	0	4	0
sKizzo	0	0	2	29
SQBF	0	0	35	0
yQuaffle	0	0	94	0

generate instances on which PicoSAT reports an incorrect satisfiability status. Moreover, the Barcelogic errors found by `CNFuzz` and `FuzzSAT` are different. Additionally, `FuzzSAT` was able to find an assertion failure of March_ks, which the other fuzzers were not able find. Our experimental results suggest that a portfolio of fuzzers should be used in order to find different solver defects. Notice that we listed the defects that each of our fuzzers were able to find in only *one* hour, which shows the impressing effectiveness of fuzz testing. Moreover, a portfolio of fuzzers could be run on a cluster for days or even weeks, which would strongly increase the probability of finding defects that could not be found so far.

The fuzz testing results for QBF solvers listed in Tab. 3 shows that also our QBF fuzzing techniques were able to find many critical defects in state-of-the-art QBF solvers. As validating QBF solver results is much harder than validating SAT solvers, we used a majority voting in order to determine the correct result. If at least 90% of the QBF solvers agreed on the satisfiability status, then all solvers reporting the opposite were classified as incorrect.

Our QBF fuzzer QBFuzz is clearly superior to the QBF generator `BlocksQBF`. The higher diversity of instances generated by `QBFuzz` enabled finding defects that `BlocksQBF` was not able to detect.

3 Delta Debugging

The overall goal of delta debugging [51,50,15,37] is to minimize failure-inducing inputs. Typically, minimized inputs simplify the debugging process as irrelevant input parts have been removed. In principle, delta debugging SAT and QBF solvers works as follows. First, the delta debugger runs the solver on the original failure-inducing input in order to observe the failure induced by the original input, e.g. the solver crashes or reports an incorrect satisfiability status. Then, the delta debugger repeatedly tries to simplify the failure-inducing input. After each simplification, the delta debugger runs the solver on the simplified input. If the solver shows the same observable behavior, the delta debugger treats the simplification as success and continues simplifying the reduced input. Otherwise, the delta debugger undoes the last simplification, and continues with other simplifications. The delta debugger repeats this process until a given time limit or fix-point is reached.

In general, it is not guaranteed that delta debugging generates a minimal failure-inducing input. However, this feature is rarely needed in practice. Instead, greedy minimization techniques are used to simplify the input as much and as fast as possible in order to generate a small failure-inducing input that can be used for effective debugging. In the following we present our delta debugging techniques for SAT and QBF.

3.1 SAT Delta Debugging

Our first CNF delta-debugger `cnfdd` is based on a variant of the algorithm described in [50]. With increasing granularity it iteratively tries to remove subsets of the whole clause set, without changing the exit code of the solver on the reduced formula. Eventually the delta-debugger will try to remove individual clauses. Thus `cnfdd` applied to solving an unsatisfiable instance, using a sound SAT solver of course, simply simulates a binary search for minimal unsatisfiable cores. In contrast to [50], complements of subsets are not considered to be removed, and `cnfdd` is also not restarted after a successful removal of a subset of clauses. This makes `cnfdd` greedier than the original DDMIN approach [51]. These changes lead to a reduction of the actual number of calls to the SAT solver during delta debugging, leading to improved performance.

However, and this is a key insight, only removing clauses, will just make the formula easier to satisfy. This will rarely lead to sufficient overall reduction. It is essential, to also strengthen the formula, of course without removing the failure. Our current version tries to remove individual literal occurrences, which is rather costly and an opportunity for future improvement. After this phase of removing individual literals, and if at least one literal was removed, the delta debugger tries to reduce the variable range, and the whole procedure is restarted.

There is also a multi-threaded version `mtcnfdd` which tries to remove clauses and literals in parallel. In the clause removal phase all sets of clauses of the current granularity are split into as many parts as threads are available. Each thread checks in parallel whether some subsets of the clauses of its part can be removed. For the clause removal phase, the worker threads are synchronized after all subsets of the current granularity have been tried. Successful removals are merged sequentially by the master thread, starting with the local view of a thread that was able to remove the largest number of clauses. In the literal removal phase, which is far less frequently successful than clause removal, clauses are split among the threads as well. Successful literal removal attempts will be tried to be merged immediately. They become permanent if the attempt of a worker thread to merge its reduced local view with the global view succeeded. Otherwise the global view takes precedence and is copied as local view.

3.2 QBF Delta Debugging

Our tool `qbfdd` is a highly configurable delta debugger for QBF instances in PCNF. It supports different variants of delta debugging strategies such as the original DDMIN [51] approach (default), DDMIN with complements only [50], and a simple strategy based on one-by-one elimination. Similar to `cnfdd`, it tries to remove subsets of the whole clause set. Then, it tries to remove individual literals. Optionally, it can move variables between quantifier sets, which may enable further simplifications. If any simplification was possible, the delta debugger continues with a new simplification round, and terminates otherwise.

3.3 Experiments

We ran our experiments on the same hardware as our fuzzing experiments. The results of our delta debugging experiments for SAT solvers are shown in Tab. 4. The experimental results clearly show the overall effectiveness of our delta debugging techniques in shrinking failure-inducing CNF instances. With the exception of RSat, our delta debugger could eliminate huge parts of the original failure-inducing parts. In the case of PicoSAT the delta debugger was able to shrink the original failure-inducing CNF instances containing more than one thousand clauses to a tiny CNF with only a few clauses as shown in Fig. 1. The defects found for RSat, which are aborts and segmentation faults, could not be minimized significantly. This in contrast to delta debugging crash-inducing instances of other solvers. For instance, segmentation faults for MiniSat-9z could be delta debugged efficiently with an average reduction of 98.8%. Therefore, we

suppose that the defects of RSat are non-trivial and cannot be triggered by a small CNF easily. For example, the failures could need a minimum number of unit propagations in order to occur.

During our experimental evaluation we observed that RSat and March_hi sometimes needed an unexpected long time (several hours) to solve instances generated during delta debugging. For instance, March_hi generated an invalid solution for the original failure-inducing instance almost immediately, but it needed hours to solve simplified instances proposed by the delta debugger. In order to speed up delta debugging, we used a time limit as proposed in [11]. We used a time limit of ten seconds to each call to Rsat and March_hi during delta debugging. If the solver exceeds the limit the delta debugger simply treats this case as if the current failure-inducing input does not lead to the same observable failure as the original input, i.e. the current simplification was not successful.

Our multi-threaded delta debugger `mtcnfdd` clearly outperforms our single-threaded delta debugger `cnfdd`. It is significantly faster on the failure-inducing instances of Barcelogic, Barcelogic-fixed, March_hi and RSat. Moreover, `mtcnfdd` tends to generate smaller instances than `cnfdd`.

Notice that we did not show experimental results of delta debugging failure-inducing inputs for ManySAT and MiraXTv3 as they showed non-deterministic behavior. For instance, MiraXTv3 reported different satisfiability results when all four cores of our computer were utilized. Due to space constraints we omit our preliminary results on delta debugging non-deterministic solvers.

In order to delta debug incorrect results, we used MiniSAT from SAT competition 2009 for SAT and Qube6.6 for QBF as reference solvers. The delta debugger calls a wrapper script instead of calling the incorrect solver directly. The script calls the reference solver and the incorrect solver on the current instance proposed by the delta debugger. If both solver agree on the satisfiability status, the script returns 1, and 0 otherwise. The possibility of calling scripts instead of solvers directly makes our delta debuggers highly flexible. Optionally, satisfiability results could be validated with techniques as proposed in [21,52].

In order to illustrate the success of our delta debuggers, we show some selected examples of minimized instances in Fig. 1. PicoSAT from SAT competition 2007 prints the solution `1 2 3 -4` for the first instance shown left, although it is obviously unsatisfiable. March_ks from SAT competition 2007 prints the solution `1 2 3` for the second unsatisfiable example. Moreover, it claims that the solution has been verified, which shows the demand for external checking tools such as [52] for the unsatisfiable case. The QBF solver yQuaffle aborts with an assertion failure when run on the third instance. QuBE 6.0 claims that the fourth instance (shown right) is satisfiable although it contains a universal unit clause.

3.4 Related Work

The work most closely related is [11]. The authors showed that fuzz testing and delta debugging techniques can be successfully applied to Satisfiability Modulo Theories (SMT) solvers. In this paper, we introduce techniques that have been explicitly designed and optimized for pure SAT and QBF. The SMT-LIB

Table 4. Experimental results of delta debugging SAT solvers from SAT competition 2007 and 2009. We evaluated our single-threaded delta debugger `cnfdd` and our multi-threaded delta debugger `mtcnfdd`, configured to use six threads. From left to right, the table shows the solver name (`solver`), the number of failure-inducing files (`files`), the number of bug classes (`classes`), the average delta debugging time (`time`) in seconds, the average file size (`size`) of the reduced instances in bytes and the average file size reduction (`red`) achieved by the delta debugger. Notice that the delta debugging time includes the time needed for the solver calls. We used a time limit of three hours for delta debugging each CNF instance. The delta debugger `cnfdd` exceeded this time limit three times (one instance of march_hi and two instances of RSat). The multi-threaded delta debugger `mtcnfdd` exceeded the time limit two times (the same RSat instances as `cnfdd`). Moreover, we used a time limit of ten seconds for each call to RSat and March_hi during delta debugging.

			cnfdd			mtcnfdd		
solver	files	classes	time	size	red	time	size	red
Barcelogic	7	4	39	432	95.8%	**20**	**378**	**96.4**%
Barcelogic-fixed	2	2	41	361	99.0%	**29**	**360**	99.0%
March_hi	24	1	638	**1982**	**88.4**%	**277**	2507	85.4%
March_ks	35	3	4	147	97.8%	**3**	**130**	**98.0**%
MiniSat-9z	912	1	<1	10	98.8%	<1	10	98.8%
PicoSAT	2	1	2	**39**	99.8%	2	40	99.8%
RSat	86	2	1478	17068	32.5%	**762**	**16971**	**32.9**%

Table 5. Experimental results of delta debugging QBF solvers. The columns have the same meaning as in Tab. 4. The delta debugging time includes the time needed for the solver calls. In order to effectively delta debug failure-inducing inputs on which SQBF crashed almost immediately, we used a time limit of two seconds for each call to SQBF during delta debugging.

	qbfdd				
solver	files	classes	time	size	red
Quantor	1	1	35	446	83.0%
QuBE 6.0	696	2	150	33	99.0%
QuBE 6.5	4	1	84	363	83.8%
sKizzo	31	2	330	497	76.2%
SQBF	35	1	57	289	86.7%
yQuaffle	94	1	26	31	98.8%

format [42] is much more complex than the DIMACS and QDIMACS format as it supports specifying formulas in several fragments of first order logic. However, in contrast to the flat CNF in SAT and QBF instances, the structural information in SMT-LIB instances can be used to apply Hierarchical Delta Debugging [37] (HDD), which is hardly possible in SAT as hierarchical information is typically lost during the translation to CNF.

```
c Picosat07
p cnf 4 4
-2 -1 0
-2 1 0
2 0
-3 -4 0
```

```
c March_ks07
p cnf 3 5
1 0
2 -1 -3 0
-2 -1 3 0
-3 -2 0
3 2 0
```

```
c yQuaffle09
p cnf 1 2
e 1
1 0
-1 0
```

```
c QuBE6.0
p cnf 2 2
a 1 0
e 2 0
2 0
1 0
```

Fig. 1. Examples of delta debugged failure-inducing inputs for SAT and QBF

Freeman mentions in his thesis [19] that he uses a 3SAT generator to test his SAT solver. However, to the best of our knowledge, there does not exist any rigorous scientific publication about automated testing and debugging SAT and QBF solvers. Nevertheless, there are a few publications that treat the problem of validating solvers. For instance, in [21,52,49] the authors instrument DPLL-based [17] solvers in order to verify unsatisfiability claims by checking traces. Recent work focuses on QBF solver validation with the help of certificates [38,31].

4 Conclusion

Essential criteria of SAT and QBF solvers are robustness and correctness. We have demonstrated that our fuzzing techniques were able to find critical defects that lead to crashes, incorrect results and invalid models in state-of-the-art SAT and QBF solvers. In particular, our fuzzers detected critical defects in top-ranked solvers at the SAT competition 2007 and 2009. Therefore, we propose to use fuzz testing in an extra qualification phase in SAT and QBF competitions in order to increase the reliability of competition results. Moreover, we showed that our delta debugging techniques are very effective in minimizing failure-inducing inputs for SAT and QBF solvers. All tools are available as open source and provide support for automated testing and debugging of SAT and QBF solvers.

Acknowledgements. We would like to thank T. Hribernig, M. Preiner and A. Niemetz for the implementations of `mtcnfdd`, `QBFuzz` and `qbfdd`.

References

1. Ansótegui, C., Béjar, R., Fernández, C., Mateu, C.: Generating Hard SAT/CSP Instances Using Expander Graphs. In: AAAI (2008)
2. Babić, D.: Exploiting Structure for Scalable Software Verification. PhD thesis, University of British Columbia, Vancouver, Canada (2008)
3. Benedetti, M.: sKizzo: A Suite to Evaluate and Certify QBFs. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 369–376. Springer, Heidelberg (2005)
4. Le Berre, D.: SAT4J: Bringing the Power of SAT Technology to the Java Platform, `http://www.sat4j.org`

5. Le Berre, D., Roussel, O., Simon, L.: SAT 2009 Competitive Events Booklet - Preliminary Version. In: SAT competition solver descriptions (September 2009), http://www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf
6. Biere, A.: Resolve and Expand. In: SAT (Selected Papers) (2004)
7. Biere, A.: PicoSAT Essentials. JSAT 4 (2008)
8. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. IOS Press, Amsterdam (2009)
9. Bofill, M., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: The Barcelogic SMT Solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 294–298. Springer, Heidelberg (2008)
10. Bregman, D., Mitchell, D.: The SAT solver MXC, Version 0.5., SAT competition solver description (2007)
11. Brummayer, R., Biere, A.: Fuzzing and Delta-Debugging SMT Solvers. In: SMT. ACM International Conference Proceedings Series. ACM, New York (2009)
12. Kleine Büning, H., Karpinski, M., Flögel, A.: Resolution for Quantified Boolean Formulas. Inf. Comput. 117(1), 12–18 (1995)
13. Cadoli, M., Schaerf, M., Giovanardi, A., Giovanardi, M.: An Algorithm to Evaluate Quantified Boolean Formulae and Its Experimental Evaluation. J. Autom. Reasoning 28 (2002)
14. Chen, H., Interian, Y.: A Model for Generating Random Quantified Boolean Formulas. In: IJCAI (2005)
15. Claessen, K., Hughes, J.: QuickCheck: a Lightweight Tool for Random Testing of Haskell Programs. ICFP 35(9) (2000)
16. Creignou, N., Daudé, H., Egly, U., Rossignol, R.: New Results on the Phase Transition for Random Quantified Boolean Formulas. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 34–47. Springer, Heidelberg (2008)
17. Davis, M., Logemann, G., Loveland, D.: A Machine Program for Theorem-Proving. ACM Commun. 5 (1962)
18. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
19. Freeman, J.W.: Improvements to Propositional Satisfiability Search Algorithms. PhD thesis, Depart. of Comp. and Inf. Science, University of Pennsylvania (1995)
20. Gebser, M., Kaufmann, B., Schaub, T.: The Conflict-Driven Answer Set Solver clasp: Progress Report. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 509–514. Springer, Heidelberg (2009)
21. Van Gelder, A.: Extracting (Easily) Checkable Proofs from a Satisfiability Solver that Employs both Preorder and Postorder Resolution. In: AMAI (2002)
22. Gent, I., Walsh, T.: The SAT Phase Transition. In: ECAI (1994)
23. Gent, I., Walsh, T.: Beyond NP: the QSAT phase transition. In: AAAI/IAAI (1999)
24. Giunchiglia, E., Narizzano, M., Tacchella, A.: QuBE++: An Efficient QBF Solver. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 201–213. Springer, Heidelberg (2004)
25. Hamadi, Y., Jabbour, S.: ManySAT: a Parallel SAT Solver. JSAT 6 (2009)
26. Heule, M.: SmArT Solving: Tools and Techniques for Satisfiability Solvers. PhD thesis, TU Delft (2008)
27. Horie, S., Watanabe, O.: Hard instance generation for SAT. CoRR, cs.CC/9809117 (1998)
28. Hsu, E., McIlraith, S.: VARSAT: Integrating Novel Probabilistic Interference Techniques with DPLL Search. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 377–390. Springer, Heidelberg (2009)
29. Huang, J.: TINISAT in SAT Competition 2007. SAT competition solver description (2007)

30. Jain, H., Clarke, E.: SAT Solver Descriptions: CMUSAT-Base and CMUSAT. SAT competition solver description (2007)
31. Jussila, T., Biere, A., Sinz, C., Kröning, D., Wintersteiger, C.: A First Step Towards a Unified Proof Checker for QBF. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 201–214. Springer, Heidelberg (2007)
32. Kern, C., Khaleghi, M., Kugele, S., Schallhart, C., Tautschnig, M., Weis, A.: SAT 7 - Engineering a Modular SAT-Solver. SAT competition solver description (2007)
33. Letz, R.: Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas. In: Egly, U., Fermüller, C. (eds.) TABLEAUX 2002. LNCS (LNAI), vol. 2381, p. 160. Springer, Heidelberg (2002)
34. Lewis, M., Schubert, T., Becker, B.: Multithreaded SAT Solving. In: Asia and South Pacific DAC (2007)
35. Lonsing, F.: DepQBF 0.1 Source Code (2010), `http://fmv.jku.at/depqbf/`
36. Miller, B., Koski, D., Lee, C., Maganty, V., Murthy, R., Natarajan, A., Steidl, J.: Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. Technical Report CS-TR-1995-1268, University of Wisconsin, Madison (1995)
37. Misherghi, G., Su, Z.: HDD: Hierarchical Delta Debugging. In: ICSE, pp. 142–151. ACM, New York (2006)
38. Narizzano, M., Peschiera, C., Pulina, L., Tacchella, A.: Evaluating and Certifying QBFs: A Comparison of State-of-the-Art Tools. AI Commun. 22 (2009)
39. Pan, G., Vardi, M.: Symbolic Decision Procedures for QBF. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 453–467. Springer, Heidelberg (2004)
40. Pennock, D., Stout, Q.: Exploiting a Theory of Phase Transitions in Three-Satisfiability Problems. In: AAAI/IAAI, vol. 1 (1996)
41. Pipatsrisawat, K., Darwiche, A.: RSat 2.0: SAT Solver Description. Technical Report D–153, Automated Reasoning Group, CSD, UCLA (2007)
42. Ranise, S., Tinelli, C.: The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa (2006)
43. Samulowitz, H.: MiniQBF Solver (2010), `http://miniqbf.spaces.live.com/`
44. Samulowitz, H., Bacchus, F.: Using SAT in QBF. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 578–592. Springer, Heidelberg (2005)
45. Sutton, M., Greene, A., Amini, P.: Fuzzing - Brute Force Vulnerability Discovery. Pearson Ed., London (2007)
46. Takanen, A., Demott, J., Miller, C.: Fuzzing for Software Security Testing and Quality Assurance. Artech House (2008)
47. Tseitin, G.: On the Complexity of Proofs in Propositional Logics. Automation of Reasoning: Classical Papers in Computational Logic 1967-1970 2 (1983)
48. Xu, K., Boussemart, F., Hemery, F., Lecoutre, C.: A Simple Model to Generate Hard Satisfiable Instances. CoRR, abs/cs/0509032 (2005)
49. Yu, Y., Malik, S.: Validating the Result of a Quantified Boolean Formula (QBF) Solver: Theory and Practice. In: Asia and South Pacific DAC (2005)
50. Zeller, A.: Why Programs Fail. A Guide to Systematic Debugging. Morgan Kaufmann, San Francisco (2005)
51. Zeller, A., Hildebrandt, R.: Simplifying and Isolating Failure-Inducing Input. IEEE Transactions on Software Engineering 28(2), 183–200 (2002)
52. Zhang, L., Malik, S.: Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. In: DATE 2003 (2003)
53. Zhang, L., Malik, S.: Towards Symmetric Treatment of Conflicts And Satisfaction in Quantified Boolean Satisfiability Solver. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, p. 200. Springer, Heidelberg (2002)