# Automated Testing and Debugging of SAT and QBF Solvers

Robert Brummayer, Florian Lonsing and Armin Biere

JKU Linz, Austria

$13^{th}$ International Conference on
Theory and Applications of Satisfiability Testing
July 13, 2010
Edinburgh, Scotland, UK

# Outline

# Motivation (1/2)

- SAT and QBF solvers used as core decision engines
  - Verification, automatic test generation, scheduling, model checking, SMT solvers, ...
- Main requirements
  - Correctness
    - Satisfiability status
    - Model resp. unsatisfiability proof
  - Robustness
    - No crashes when run on syntactically valid inputs
  - Speed
- Clients heavily depend on these criteria
  - Incorrect solver may lead to incorrect overall result
  - Crashing solver may crash the client as well
  - Solver should compute the result as fast as possible
    - High user expectations although problem is NP-complete

# Motivation (2/2)

- Demand for fast solvers led to various improvements
  - Improved algorithms
    - Clause learning, rapid restarts, literal watching, $\cdots$
    - Well understood from a theoretical point of view
  - Low-level implementation and optimization details
    - Error prone engineering
    - Developer secrets
- How do we make sure that our implementations are correct?
- Traditional testing approaches
  - Unit testing / regression testing
    - Tedious task of generating test cases manually
  - Testing solver against benchmark suite
    - Limited set of highly specific benchmarks
- Complement traditional testing with fuzz testing

# Fuzz Testing

- Fuzz Testing (Miller et al.)
  - automated negative testing technique
- Grammar-based black-box fuzz testing
  - Treat solver as a black-box
  - However, use domain knowledge to implement fuzzer
- Fuzzer: random instance generator
  - Random instances must be syntactically valid
  - Ideally, we want high diversity to cover many corner cases
- Repeatedly "attack" solver with random instances
  - Check satisfiability result
  - Check model resp. unsatisfiability proof
- High throughput as a success factor
  - Random instances should not be "too" hard to solve
  - However, trivial instances are unlikely to reveal critical defects

# Random 3SAT

- Random 3SAT fuzzer `3SATGen`
- Pick number of variables $m$
- Pick clause variable ratio $r$
- Generate $m \cdot r$ random ternary clauses
  - Pick each variable uniformly and negate it with probability $1/2$
  - Avoid generating trivial clauses
  - Avoid picking the same literal multiple times within a clause
- Trivial to implement
- However, random 3SAT instances lack structure
  - Generated instances are unlikely to reveal interesting defects

# Random Layered CNF

- Use domain knowledge
  - Often, SAT solvers use input structure
  - Layered CNF fuzzer `CNFuzz`
- Pick a number of layers $l$ of maximum width $w$
- Each $i^{th}$ layer introduces $f$ fresh variables
- Each layer has a separate ratio $r$ from which the number of clauses is computed
- Clauses are at least ternary
  - Probability of larger clauses decreases exponentially
  - Probability of picking variables from previous layers decreases exponentially

# Random CNF by Circuit Translation

- Use domain knowledge
  - Industrial SAT solvers are optimized for CNFs generated by circuit translation, e.g. Tseitin translation
  - Fuzzer that builds circuit and translates it to CNF: `FuzzSAT`
- Generate $v$ boolean variables and insert them into set $n$
- Pick op $\in \{\text{AND, OR, XOR, IFF}\}$, pick $o_1$ and $o_2$ from $n$, negate both with probability $1/2$, generate new operator node and insert it into $n$
- Repeat until each input variable is referenced at least $t$ times
- Combine roots to one boolean root
- Translate root to CNF by Tseitin translation
- Finally, add random clauses of varying size to increase diversity and difficulty

# Experiments for SAT Solvers from SAT-COMP'07

| | 3SATGen | | | CNFuzz | | | FuzzSAT | | |
|---|---|---|---|---|---|---|---|---|---|
| solver | err | inc | mod | err | inc | mod | err | inc | mod |
| Blogic | 0 | 0 | 0 | 1 | **3** | 1 | 1 | 0 | 1 |
| Blogic-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 |

- err = error, e.g. segfault
- inc = incorrect satisfiability status
- mod = invalid model
- Tested solvers
  - Barcelogic, Barcelogic-fixed, CMUSAT, March_ks, MiniSat, MiraXTv3, MXC, PicoSAT, RSat, Sat7, SAT4J, Spear, Tinisat.

# Experiments for SAT Solvers from SAT-COMP'09

| | 3SATGen | | CNFuzz | | FuzzSAT | |
|---|---|---|---|---|---|---|
| solver | err | mod | err | mod | err | mod |
| ManySat | 2 | 0 | 56 | 0 | **836** | 0 |
| March_hi | 0 | 0 | 0 | 0 | 0 | **24** |
| MiniSat-9z | 2 | 0 | 58 | 0 | **852** | 0 |

- err = error, e.g. segfault
- mod = invalid model
- Tested solvers
  - CirCUs, Clasp, Cumr_p, Glucose, LySATi, ManySAT, Marchi_hi, MiniSat, MiniSat-9z, MXC, PicoSAT, PrecoSAT, RSat, SApperloT, SAT4J, Varsat-industrial

# Random QBF model

- Fuzzer that uses theoretical model by Chen and Interian to generate random QBFs: `BlocksQBF`
- Model bears similarities to random SAT
  - study threshold behavior
- All clauses are for-all reduced by construction
- All clauses have the same length
- Each clause is free of complementary and duplicate literals
- Duplicate clauses are discarded

# QBFuzz

- Observation: exact model limits diversity
- Configurable fuzzer `QBFuzz`
- First, a quantifier prefix is generated
  - Number of variables is picked for each block
- Clauses are of varying length
- Literals are selected from any block
  - Complementary and duplicate literals are discarded
- For-all reduction
- Discard duplicate clauses
- Eliminate unused variables

# Fuzz Testing Experiments

| | 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 |

- Tested solvers
  - DepQBF, MiniQBF-090608, QMRES, Quantor-3.0, QuBE6.0, QuBE6.5, QuBE6.6, Semprop-010604, sKizzo-0.8.2, SQBF-1.0, Squolem-1.03, yQuaffle-021006

# Delta Debugging

- Delta Debugging (Zeller et al.)
  - Technique to automatically shrink failure-inducing instances
- Delta debugger (DD) runs solver on failure-inducing instance $\phi$ to obtain golden exit code
- DD tries to simplify the failure-inducing instance greedily
- DD calls solver with a simplified instance $\phi'$
  - Exit code $=$ golden exit code, success, continue simplifying $\phi'$
  - Exit code $\neq$ golden exit code, failure, try other simplification
- Use wrapper script instead of calling solver directly
  - Script determines if the observable behavior is equal
    - For example, grep for a specific error message

# SAT Delta Debugging

- Fuzz testing and Delta Debugging has been shown to be effective for SMT
  - Delta debugging for SMT uses the explicit structure of the instance (hierarchical delta debugging)
  - However, SAT instances are flat, i.e. structure is implicit
- Delta debugger `cnfdd` handles instance as set of clauses
- Uses an even greedier version of DDMIN (Zeller et al.)
  - Greedier version reduces the number of calls to SAT solver
- Delta debugger executes 2 alternating phases
  - With increasing granularity, try to remove sets of clauses
  - Try to remove individual literals
    - Rather costly, but necessary for sufficient overall reduction
- Delta Debugging is run until fix-point is reached
  - Or some threshold is reached, .e.g. time limit

# Multi-threaded SAT Delta Debugging

- Delta Debugging algorithm offers parallelism
- Multi-threaded SAT delta debugger `mtcnfdd`
- Simplification attempts can be done in parallel
- Clause removal phase
  - Divide set of clauses into $g$ sets, where $g$ is current granularity
  - Try to remove individual sets in parallel
    - Independent parallel calls to SAT solver
  - Merge successful simplifications after each round
    - Start with smallest reduced formula
- Literal removal phase
  - Clauses are split among threads
  - However, successful literal removals are merged immediately

# SAT Delta Debugging Experiments

| | | | cnfdd | | | mtcnfdd | | |
|---|---|---|---|---|---|---|---|---|
| solver | files | c | time | size | red | time | size | red |
| Blogic | 7 | 4 | 39 | 432 | 95.8% | **20** | **378** | **96.4%** |
| Blogic-fix | 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%** |

- c = number of observable classes, e.g. segfault, incorrect, $\cdots$
- time = average delta debugging time in seconds
- size = average number of bytes of delta debugged instance
- red = overall reduction achieved by delta debugger

# QBF Delta Debugging

- QBF delta debugging similar to SAT delta debugging
- QBF delta debugger executes 2 alternating phases
  - With increasing granularity, try to remove sets of clauses
  - Try to remove individual literals
- However, as a third phase, moving variables between quantifier sets may trigger further simplifications
  - Potential subject for further research
- Configurable delta debugger `qbfdd`
  - Different strategies for clause removal
  - Moving variables between quantifier sets (optionally)

# QBF Delta Debugging Experiments

| qbfdd | | | | | |
|---|---|---|---|---|---|
| solver | files | c | 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% |

- c = number of observable classes, e.g. segfault, incorrect, $\cdots$
- time = average delta debugging time in seconds
- size = average number of bytes of delta debugged instance
- red = overall reduction achieved by delta debugger

# Conclusions

- Fuzz Testing is an effective automated negative testing technique for SAT and QBF solvers
  - Our experiments found critical defects
  - We propose to use fuzz tests in a qualification round of further SAT and QBF competitions
- Delta debugging techniques are effective for automatically reducing failure-inducing instances for SAT and QBF solvers
- All tools available as open source
  - "Attack" your own solvers!
- Acknowledgements
  - We would like to thank T. Hribernig, M. Preiner and A. Niemetz for implementing `mtcnfdd`, `QBFuzz` and `qbfdd`.