SMT@Microsoft

Intel 2007

Leonardo de Moura and Nikolaj Bjørner

{leonardo, nbjorner}@microsoft.com.

Microsoft Research

Introduction

- Industry tools rely on powerful verification engines.
  - Boolean satisfiability (SAT) solvers.
  - Binary decision diagrams (BDDs).
- *Satisfiability Modulo Theories (SMT)*
  - The next generation of verification engines.
  - *SAT solvers + Theories*
    - Arithmetic
    - Arrays
    - Uninterpreted Functions
  - Some problems are more naturally expressed in SMT.
  - More automation.

SMT: Examples

$$x + 2 = y \Rightarrow f(\textit{read}(\textit{write}(a, x, 3), y - 2)) = f(y - x + 1)$$

$$f(f(x) - f(y)) \neq f(z), x + z \leq y \leq x \Rightarrow z < 0$$

SMT-Solvers & SMT-Lib & SMT-Comp

- SMT-Solves:

  Ario, Barcelogic, CVC, CVC Lite, CVC3, ExtSAT, Fx7,

  Harvey, HTP, ICS, Jat, MathSAT, Sateen, Simplify, Spear,

  STeP, STP, SVC, TSAT, UCLID, Yices, Zap, *Z3 (Microsoft)*

- SMT-Lib: library of benchmarks

  `http://www.smtlib.org`

- SMT-Comp: annual SMT-Solver competition.

  `http://www.smtcomp.org`

Z3: An Efficient SMT Solver

- *Z3 is a new SMT solver developed at Microsoft Research.*
- Version 0.1 competed in SMT-COMP'07.
  - 4 first places.
  - 7 second places.
- Version 1.0 was released last week.
- Free for non-commercial use.
- Managed (.Net) & Unmanaged (C/C++) APIs are available.
- `http://research.microsoft.com/projects/z3`
- More features coming soon.

Applications

- Test-case generation.
  - *Pex, SAGE, Yogi, and Vigilante.*
- Verifying Compiler.
  - *Spec#/Boogie, VCC/Boogie, and HAVOC/Boogie*
  - ESC/Java
- Model Checking & Predicate Abstraction.
  - *SLAM/SDV and Yogi.*
- Bounded Model Checking (BMC) & k-induction.
- Planning & Scheduling.
- Equivalence checking.

Roadmap

- Test-case generation
- Verifying Compiler
- Model Checking & Predicate Abstraction.
- Future

Test-case generation

- Test (correctness + usability) is 95% of the deal:
  - Dev/Test is 1-1 in products.
  - Developers are responsible for unit tests.
- Tools:
  - Annotations and static analysis (SAL, ESP)
  - File Fuzzing
  - *Unit test case generation:*

    program analysis tools, automated theorem proving.

Security is Critical

- Security bugs can be very expensive:
  - Cost of each MS Security Bulletin: \$600K to \$Millions.
  - Cost due to worms (Slammer, CodeRed, Blaster, etc.): \$Billions.
  - *The real victim is the customer.*
- Most security exploits are initiated via files or packets:
  - Ex: Internet Explorer parses dozens of files formats.
- Security testing: *hunting for million-dollar bugs*
  - Write A/V (always exploitable),
  - Read A/V (sometimes exploitable),
  - NULL-pointer dereference,
  - Division-by-zero (harder to exploit but still DOS attack), ...

Hunting for Security Bugs

- Two main techniques used by *"black hats"*:
  - Code inspection (of binaries).
  - *Black box fuzz testing*.
- **Black box** fuzz testing:
  - A form of black box random testing.
  - Randomly *fuzz* (=modify) a well formed input.
  - Grammar-based fuzzing: rules to encode how to fuzz.
- **Heavily** used in security testing
  - At MS: several internal tools.
  - Conceptually simple yet effective in practice

    Has been instrumental in weeding out 1000 of bugs during development and test.

Automatic Code-Driven Test Generation

Given program with a set of input parameters.

Generate inputs that maximize code coverage.

Automatic Code-Driven Test Generation

Given program with a set of input parameters.

Generate inputs that maximize code coverage.

Example:

Input x, y

$z = x + y$

If $z > x - y$ Then

    Return z

Else

    Error

Automatic Code-Driven Test Generation

Given program with a set of input parameters.

Generate inputs that maximize code coverage.

Example:

Input x, y

$z = x + y$

If $z > x - y$ Then

  Return z

Else

  Error

Solve $z = x + y \land z > x - y$

Automatic Code-Driven Test Generation

Given program with a set of input parameters.

Generate inputs that maximize code coverage.

Example:

Input x, y

$z = x + y$

If $z > x - y$ Then

Return z

Else

Error

Solve $z = x + y \wedge z > x - y$

$\Longrightarrow x = 1, y = 1$

Automatic Code-Driven Test Generation

Given program with a set of input parameters.

Generate inputs that maximize code coverage.

Example:

Input x, y

$z = x + y$

If $z > x - y$ Then

  Return z

Else

  Error

Solve $z = x + y \wedge \neg(z > x - y)$

Automatic Code-Driven Test Generation

Given program with a set of input parameters.

Generate inputs that maximize code coverage.

Example:

Input x, y

$z = x + y$

If $z > x - y$ Then

Return z

Else

Error

Solve $z = x + y \wedge \neg(z > x - y)$

$\Longrightarrow x = 1, y = -1$

Method: Dynamic Test Generation

- *Run* program with *random* inputs.
- *Collect constraints* on inputs.
- *Use SMT solver* to generate new inputs.
- Combination with randomization: DART (Godefroid-Klarlund-Sen-05)

Method: Dynamic Test Generation

- *Run* program with *random* inputs.
- *Collect constraints* on inputs.
- *Use SMT solver* to generate new inputs.
- Combination with randomization: DART (Godefroid-Klarlund-Sen-05)

Repeat while finding new *execution paths*.

DARTish projects at Microsoft

- *SAGE* (CSE) implements DART for x86 binaries and merges it with "fuzz" testing for finding security bugs.
- *PEX* (MSR-Redmond FSE Group) implements DART for .NET binaries in conjunction with "parameterized-unit tests" for unit testing of .NET programs.
- *YOGI* (MSR-India) implements DART to check the feasibility of program paths generated statically using a SLAM-like tool.
- *Vigilante* (MSR Cambridge) partially implements DART to dynamically generate worm filters.

Inital Experiences with SAGE

25+ security bugs and counting. (most missed by blackbox fuzzers)

- OS component X

  4 new bugs: “This was an area that we heavily fuzz tested in Vista”.

- OS component Y

  Arithmetic/stack overflow in y.dll

- Media format A

  Arithmetic overflow; DOS crash in previously patched component

- Media format B & C

  Hard-to-reproduce uninitialized-variable bug

Pex

- Pex monitors the execution of .NET application using the CLR profiling API.
- Pex dynamically checks for violations of programming rules, e.g. resource leaks.
- Pex suggests code snippets to the user, which will prevent the same failure from happening again.
- *Very instrumental in exposing bugs in .NET libraries.*

Test-case generation & SMT

- Formulas are usually a big conjunction.
- Incrementality: solve several similar formulas.
- "Small models".
- Arithmetic $\times$ Machine Arithmetic.

Test-case generation & SMT

- Formulas are usually a big conjunction.
  - Pre-processing step.
  - Eliminate variables and simplify input formula.
  - *Significant performance impact*.
- Incrementality: solve several similar formulas.
- “Small models”.
- Arithmetic $\times$ Machine Arithmetic.

Test-case generation & SMT

- Formulas are usually a big conjunction.
- Incrementality: solve several similar formulas.
  - *Z3 is incremental*: new constraints can be asserted.
  - **push** and **pop**: (user) backtracking.
  - Reuse (some) lemmas.
- "Small models".
- Arithmetic $\times$ Machine Arithmetic.

Test-case generation & SMT

- Formulas are usually a big conjunction.
- Incrementality: solve several similar formulas.
- "Small models".
  - **Given** a set of constraints C, find a model M that *minimizes* the value of the variables $x_0, \ldots, x_n$.
- Arithmetic $\times$ Machine Arithmetic.

Test-case generation & SMT

- Formulas are usually a big conjunction.
- Incrementality: solve several similar formulas.
- "Small models".
  - **Given** a set of constraints C, find a model M that *minimizes* the value of the variables $x_0, \ldots, x_n$.
  - **Eager (cheap) Solution:**

    Assert C.

    While satisfiable

        Peek x_i such that $M[x_i]$ is big

        Assert $x_i < c$, where c is a small constant

    Return last found model
- Arithmetic $\times$ Machine Arithmetic.

Test-case generation & SMT

- Formulas are usually a big conjunction.
- Incrementality: solve several similar formulas.
- "Small models".
  - **Given** a set of constraints C, find a model M that *minimizes* the value of the variables $x_0, \ldots, x_n$.
  - **Refinement:**
    - Eager solution stops as soon as the context becomes unsatisfiable.
    - A "bad" choice (peek x_i) may prevent us from finding a good solution.
    - Use **push** and **pop** to retract "bad" choices.
- Arithmetic $\times$ Machine Arithmetic.

Test-case generation & SMT

- Formulas are usually a big conjunction.
- Incrementality: solve several similar formulas.
- “Small models”.
- Arithmetic $\times$ Machine Arithmetic.
  - *Precision* $\times$ *Performance*.
  - SAGE has flags to abstract expensive operations.

Roadmap

- Test-case generation
- Verifying Compiler
- Model Checking & Predicate Abstraction.
- Future

The Verifying Compiler

A verifying compiler uses *automated reasoning to check the correctness* of a program that is compiles.

Correctness is specified by *types, assertions, . . . and other redundant annotations* that accompany the program.

Hoare 2004

Spec# Approach for a Verifying Compiler

- *Source Language*
  - C# + goodies = Spec#
- *Specifications*
  - method contracts,
  - invariants,
  - field and type annotations.
- *Program Logic*
  - Dijkstra's weakest preconditions.
- *Automatic Verification*
  - type checking,
  - verification condition generation (VCG),
  - automatic theorem proving (SMT)

Spec# Approach for a Verifying Compiler

- Spec# (annotated C#) $\Longrightarrow$ Boogie PL $\Longrightarrow$ Formulas
- Example:

class C $\{$

private int a, z;

invariant z $>$ *0*

public void M()

requires a != 0

$\{$ z = 100/a; $\}$

$\}$

- Weakest preconditions:
  - *wp*$(S; T, Q) =$ *wp*$(S,$ *wp*$(T, Q))$
  - *wp*$($*assert* $C, Q) = C \wedge Q$

Microsoft Hypervisor

- **Meta OS:** small layer of software between hardware and OS.
- **Mini:** 60K lines of non-trivial concurrent systems C code.
- **Critical:** must *guarantee isolation*.
- **Trusted:** a grand verification challenge.

What is to be verified?

- Source code: C + x64 assembly.
- Sample verifiable slices:
  - **Safety:** Basic memory safety
  - **Security:** OS isolation
  - **Utility:** Hypervisor services guest OS with available resources.

Tool: A Verified C Compiler

- VCC translates an *annotated C program* into a *Boogie PL* program.
- Boogie generates verification conditions:
  - Z3 (automatic)
  - Isabelle (interactive)
- A C-ish memory model
  - Abstract heaps
  - Bit-level precision
- The verification project has very recently started.
- It is a multi-man multi-year effort.
- More news coming soon.

Tool: HAVOC

- A tool for specifying and checking properties of systems software written in C.
- It also translates annotated C into Boogie PL.
- It allows the expression of *richer properties about the program heap and data structures* such as linked lists and arrays.
- HAVOC is being used to specify and check:
  - Complex locking protocols over heap-allocated data structures in Windows.
  - Properties of collections such as IRP queues in device drivers.
  - Correctness properties of custom storage allocators.

Verifying Compilers & SMT

- *Quantifiers, Quantifiers, . . .*
  - Modeling the runtime.
  - Frame axioms (“what didn’t change”).
  - User provided assertions and invariants (e.g., the array is sorted).
  - Prototyping decision procedures (e.g., reachability, partial orders, . . .).
- Since first-order logic is undecidable, satisfiability is not solvable for arbitrary quantified formulas.
- Z3: pragmatic approach
  - *Heuristic Quantifier Instantiation.*
  - E-matching (i.e., matching modulo equalities).

Heuristic Quantifier Instantiation

- Semantically, $\forall x_1, \ldots, x_n.F$ is equivalent to the infinite conjunction $\bigwedge_\beta \beta(F)$.
- Solvers use heuristics to select from this infinite conjunction those instances that are “relevant”.
- The key idea is to treat an instance $\beta(F)$ as relevant whenever it contains enough terms that are represented in the solver state.
- Non ground terms p from F are selected as *patterns*.
- *E-matching* (matching modulo equalities) is used to find instances of the patterns.
- Example: $f(a, b)$ matches the pattern $f(g(x), x)$ if $a = g(b)$.

E-matching

- E-matching is NP-hard.
- The number of matches can be exponential.
- It is not refutationally complete.
- In practice:
  - Indexing techniques for fast retrieval.
  - Incremental E-matching.

E-matching: example

- $\forall x. f(g(x)) = x$
- Pattern: $f(g(x))$
- Atoms: $a = g(b), b = c, f(a) \neq c$
- $\longrightarrow$ *instantiate* $f(g(b)) = b$

Quantifiers in Z3

- Z3 uses a E-matching abstract machine.
  - Patterns $\rightsquigarrow$ code sequence.
  - Abstract machine executes the code.
- *Z3 uses new algorithms that identify matches on E-graphs incrementally and efficiently.*
  - E-matching code trees.
  - Inverted path index.
- Z3 garbage collects clauses, together with their atoms and terms, that were useless in closing branches.

Roadmap

- Test-case generation
- Verifying Compiler
- Model Checking & Predicate Abstraction.
- Future

SLAM: device driver verification

- `http://research.microsoft.com/slam/`
- *SLAM/SDV* is a software model checker.
- Application domain: *device drivers*.
- Architecture

  c2bp C program $\rightsquigarrow$ boolean program (*predicate abstraction*).

  bebop Model checker for boolean programs.

  newton Model refinement (*check for path feasibility*)
- SMT solvers are used to perform predicate abstraction and to check path feasibility.
- c2bp makes several calls to the SMT solver. The formulas are relatively small.

Predicate Abstraction: c2bp

- **Given** a C program P and $F = \{p_1, \ldots, p_n\}$.
- **Produce** a boolean program $B(P, F)$
  - Same control flow structure as P.
  - Boolean variables $\{b_1, \ldots, b_n\}$ to match $\{p_1, \ldots, p_n\}$.
  - Properties true of $B(P, F)$ are true of P.
- Each p_i is a pure boolean expression.
- Each p_i represents set of states for which p_i is true.
- Performs modular abstraction.

Abstracting Expressions via F

- *Implies*$_F(e)$
  - Best boolean function over F that implies e
- *ImpliedBy*$_F(e)$
  - Best boolean function over F that implied by e
  - *ImpliedBy*$_F(e) = \neg$*Implies*$_F(\neg e)$

Computing Implies$_F(e)$

- minterm $m = l_1 \wedge \ldots \wedge l_n$, where $l_i = p_i$, or $l_i = \neg p_i$.
- *Implies*$_F(e)$ is the disjunction of all minterms that imply e.
- Naive approach
  - Generate all 2^n possible minterms.
  - For each minterm m, use SMT solver to check validity of $m \implies e$.
- Many possible optimizations.

Newton

- Given an error path p in the boolean program B.
- Is p a feasible path of the corresponding C program?
  - Yes: found a bug.
  - No: find predicates that explain the infeasibility.
- Execute path symbolically.
- Check conditions for inconsistency using SMT solver.

Roadmap

- Test-case generation
- Verifying Compiler
- Model Checking & Predicate Abstraction.
- Future

Future work

- Proof production
- Interpolants
- Quantifier elimination
  - Z3 already supports Fourier-Motzkin elimination.
- New theories:
  - Reachability
  - Sets
  - Partial orders
  - Better support for non-linear arithmetic

Proof production

- It is not required by internal projects.
  - May be useful in Hypervisor.
- Requested by potential external users (e.g., Cambridge).
- *Useful for debugging (and optimizing) Z3.*
- Our approach:
  - Goal: reasonable performance when proof production is enabled.
  - Store a compact representation of deduction steps in a log.
  - Use a post-processor to transform log into a proof.

Interpolants

- Requested several times by internal & external users.
- Useful for predicate abstraction (i.e., predicate discovery).

Conclusion

- Formal verification is hot at Microsoft.
- Main applications:
  - Test-case generation.
  - Verifying compiler.
  - Model Checking & Predicate Abstraction.
- Z3 is a new SMT solver.

  `http://research.microsoft.com/projects/z3`