SMT @ Microsoft

Johannes Kepler University
Linz, Austria 2008

Leonardo de Moura and Nikolaj Bjørner
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 solver + Theories*
  - Some problems are more naturally expressed in SMT.
  - More automation.

Satisfiability Modulo Theories (SMT)

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

Arithmetic

Array Theory

Uninterpreted Functions

SMT-Solvers & SMT-Lib & SMT-Comp

- SMT-Solvers:

  Argo-Lib, Ario, Barcelogic, CVC, CVC Lite, CVC3, ExtSAT, Fx7, Fx8, Harvey, HTP, ICS, Jat, MathSAT, Sateen, Simplify, Spear, STeP, STP, SVC, TSAT, 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

Applications

- Test-case generation.
  - *Pex, Yogi, Vigilante, Sage*
- Verifying compiler.
  - *Spec#, VCC, Havoc*
  - ESC/Java
- Model Checking & Predicate Abstraction.
  - *SLAM/SDV, Yogi*
- Bounded Model Checking (BMC) & *k*-induction
- Planning & Scheduling
- Equivalence Checking.

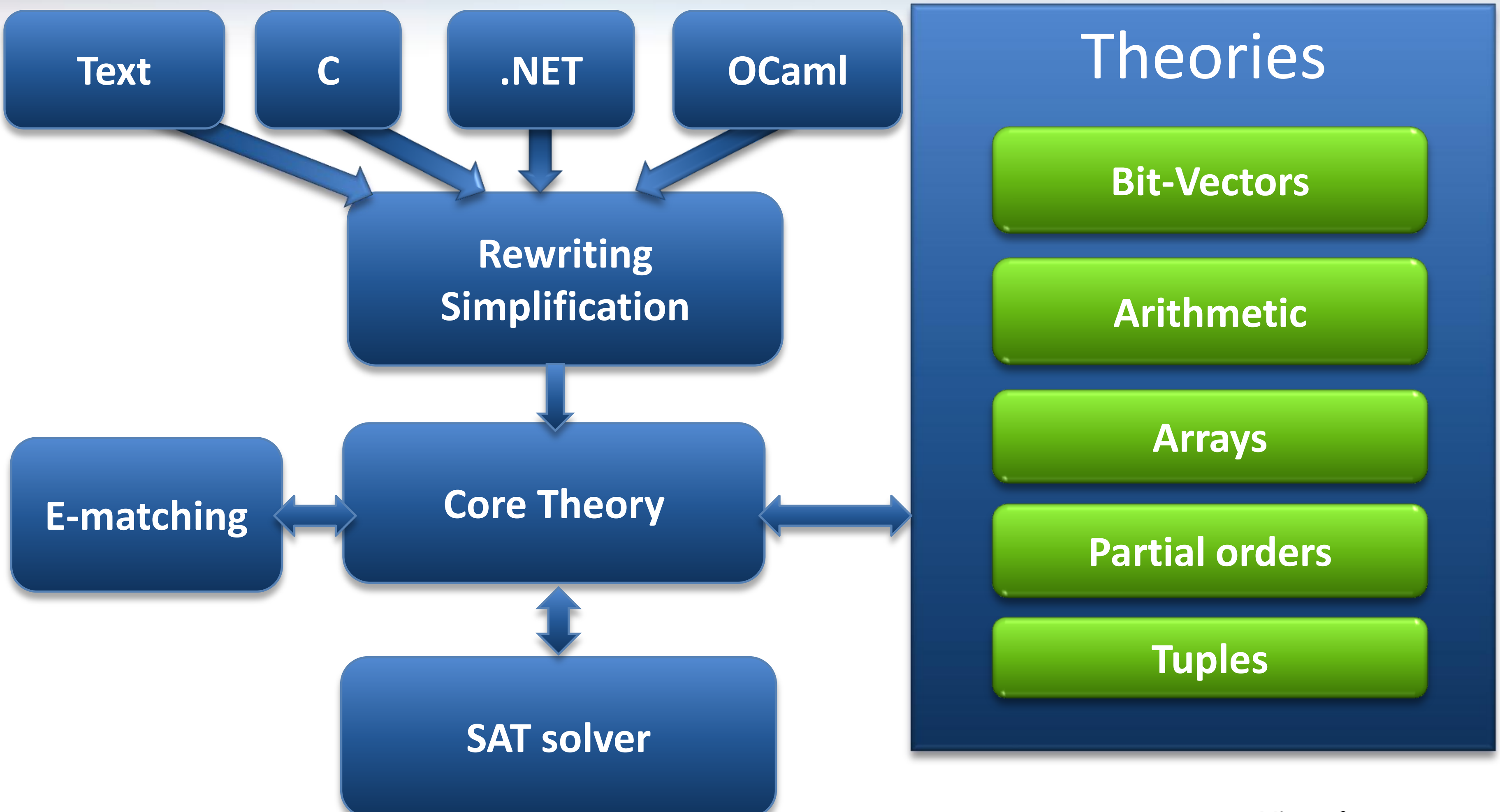
Z3: An Efficient SMT Solver

- *Z3 is a new SMT solver developed at Microsoft.*
- Version 0.1 competed in SMT-COMP'07.
- Version 1.2 is the latest release.
- Free for academic research.
- It is used in several program analysis, verification, test-case generation projects at Microsoft.
- http://research.microsoft.com/projects/z3

Z3: Main features

- Linear real and integer arithmetic.
- Fixed-size bit-vectors
- Uninterpreted functions
- Extensional arrays
- Quantifiers
- Model generation
- Several input formats (Simplify, SMT-LIB, Z3, Dimacs)
- Extensive API (C/C++, .Net, OCaml)

Z3: Core System Components



Z3: Some Technical goodies

- Model-based Theory Combination
  - *How to efficiently combine theory solvers?*
  - Use models to control Theory Combination.
- E-matching abstract machine
  - Term indexing data-structures for incremental matching modulo equalities.
- Relevancy propagation
  - Use Tableau advantages with DPLL engine

Test-case generation

Overview

- 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.

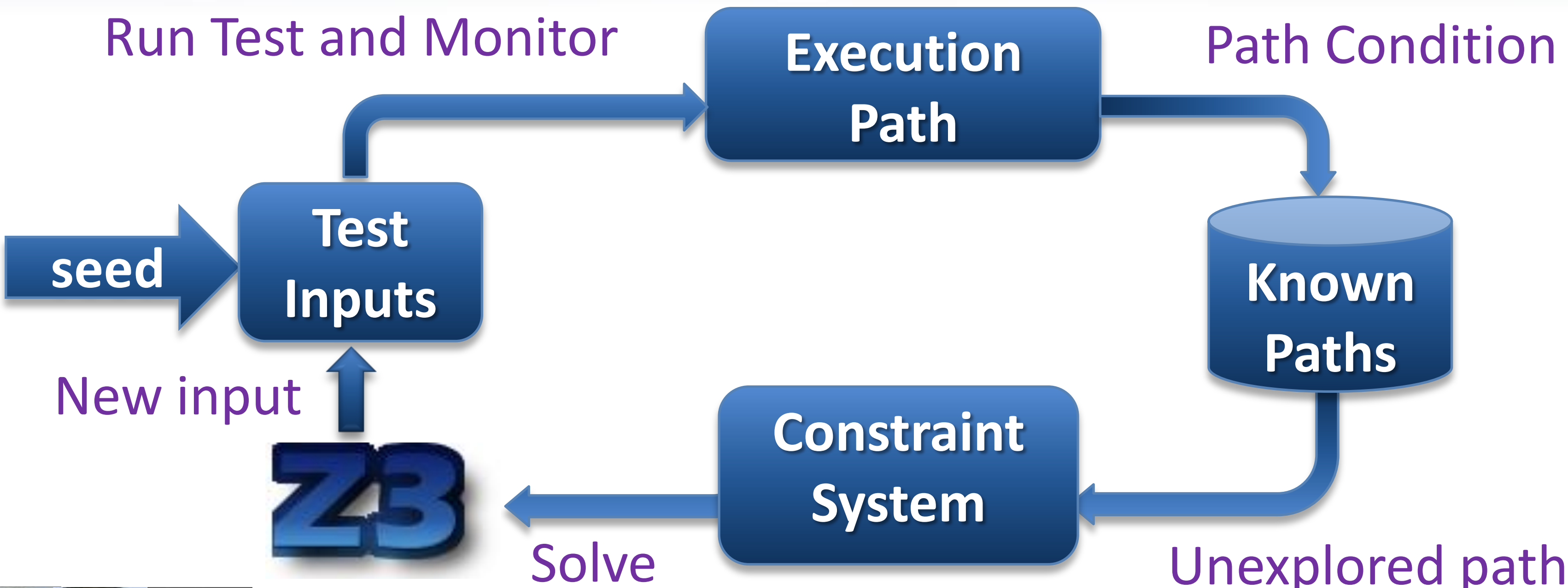
Security is Critical

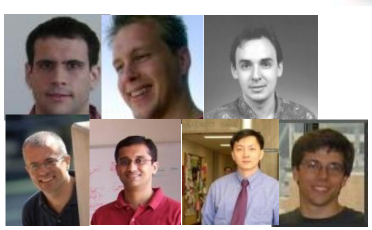
- Security bugs can be very expensive:
  - Cost of each MS Security Bulletin: $600K to $Millions.
  - Cost due to worms (Slammer, CodeRed, etc.): $Billions.
  - *The real victim is the customer.*
- Most security exploits are initiated via files or packets:
  - Ex: Internet Explorer parses dozens of files 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

Test case generation @ Microsoft





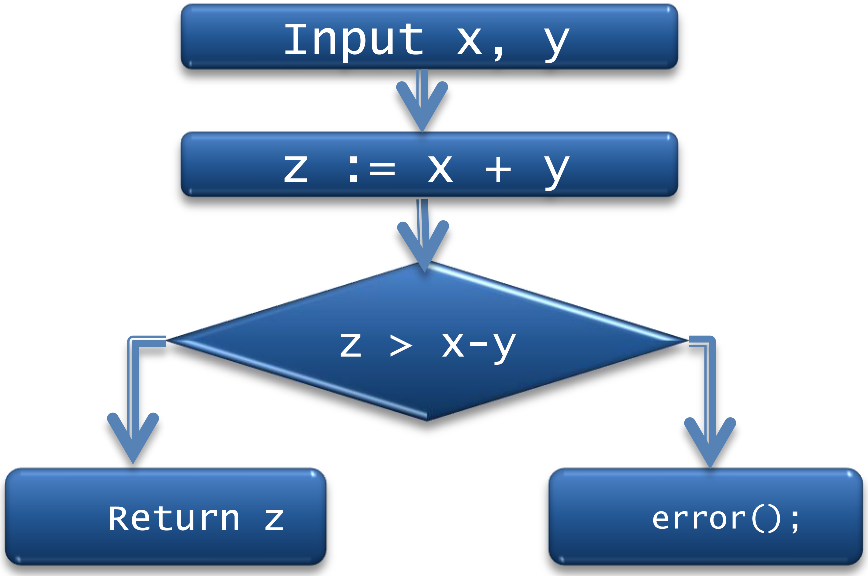
Pex

Yogi

Vigilante

Nikolai Tillmann, Peli de Halleux, Patrice Godefroid
Aditya Nori, Jean Philippe Martin, Miguel Castro,
Manuel Costa, Lintao Zhang

Example



Solve: $z = x + y \wedge z \leq x - y$

$x = 1, y = 2$

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

$x = -2, y = -3$

Microsoft Research

Z3 & Test case generation

- Formulas may be a big conjunction
  - Pre-processing step
  - Eliminate variables and simplify input format
- Incremental: solve several similar formulas
  - New constraints are asserted.
  - **push** and **pop**: (user) backtracking
  - Lemma reuse

Z3 & Test case generation

- "Small Models"
  - Given a formula *F*, 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
- True Arithmetic × Machine Arithmetic

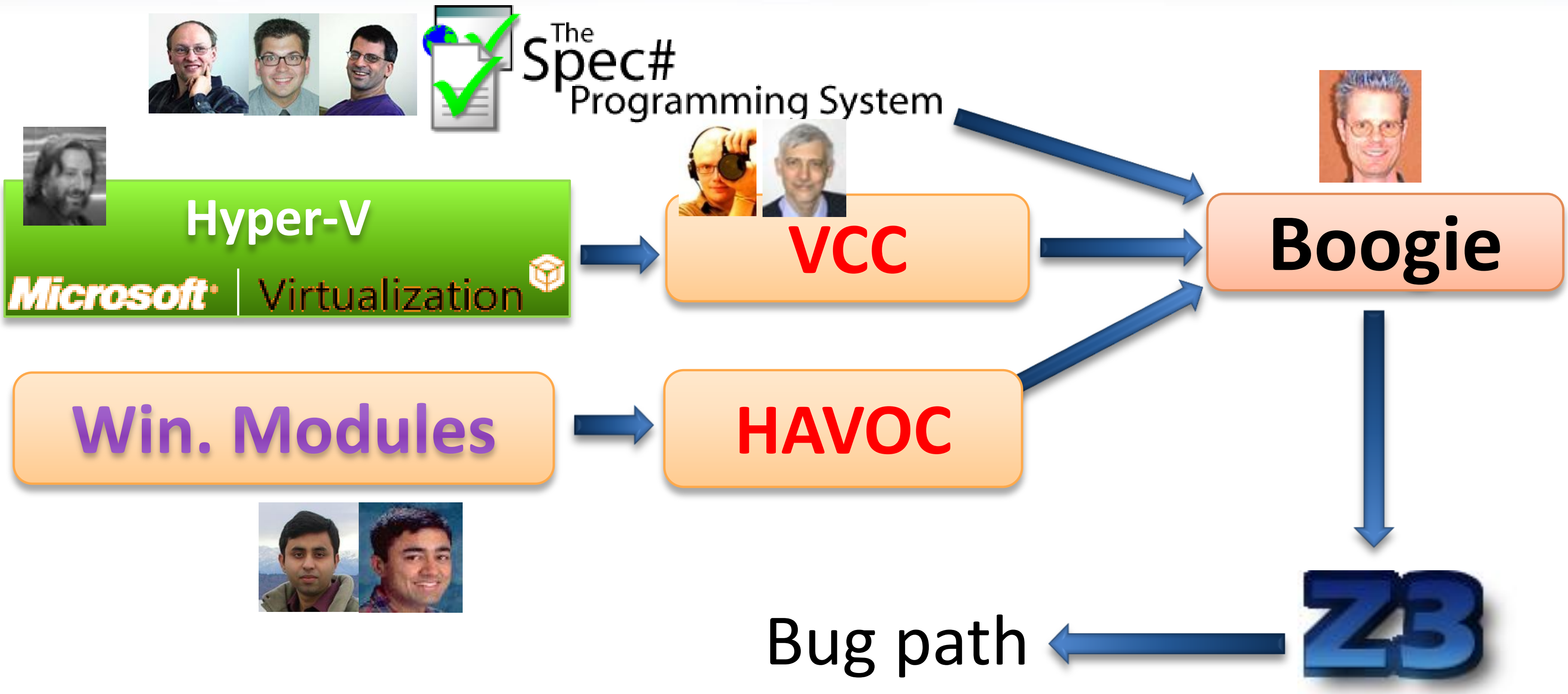
Verifying Compilers

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.

Tony Hoare 2004

Program Verification @ Microsoft

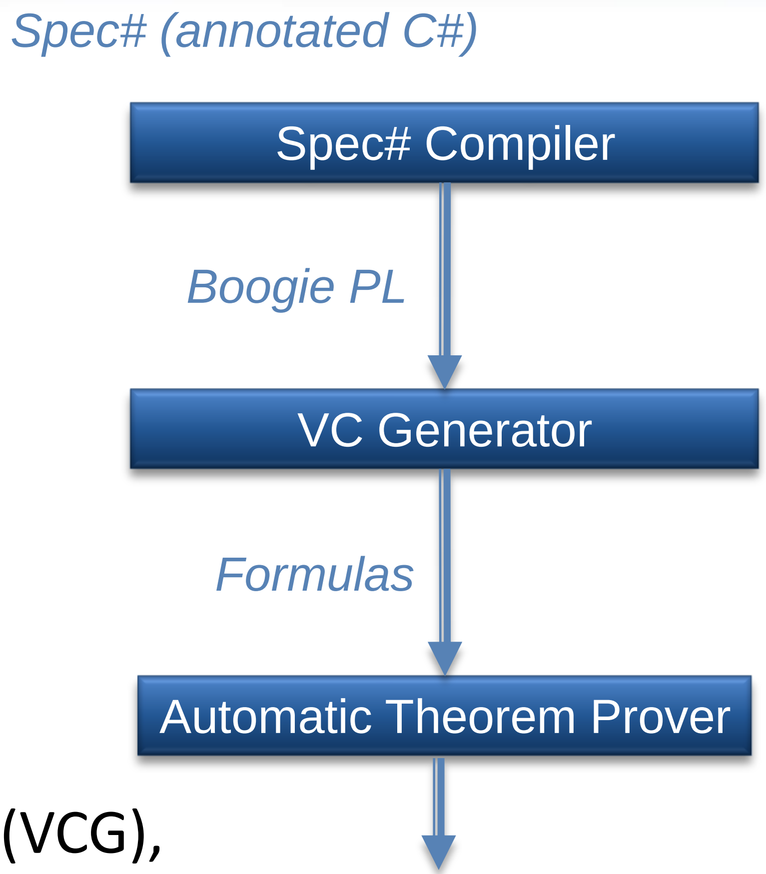


Rustan Leino, Mike Barnet, Michal Moskal, Shaz Qadeer,
Shuvendu Lahiri, Herman Venter, Peter Muller,
Wolfram Schulte, Ernie Cohen

Microsoft Research

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

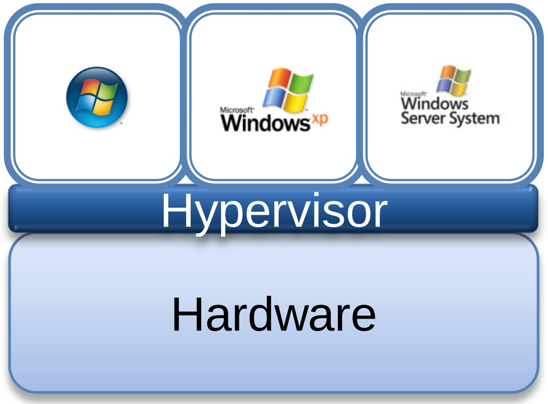
- Spec# (annotated C#) ⇒Boogie PL ⇒Formulas
- Example:

```
class C {
        private int a, z;
        invariant z > 0;
        public void M()
                requires a != 0;
                { z = 100/a; }
}
```

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

Microsoft Hypervisor

- **Meta OS**: small layer of software between hardware and OS.
- **Mini**: 60K lines of non-trivial concurrent systems C code.
- **Critical**: simulates a number of virtual x64 machines
- **Trusted**: a grand verification challenge.



Microsoft Research

A Verifying 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.

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 & Z3

- Quantifiers, quantifiers, quantifiers, …
  - Modeling the runtime
  - Frame axioms ("what didn't change")
  - Users provided assertions (e.g., the array is sorted)
  - Prototyping decision procedures (e.g., reachability, heaps, …)
- *Solver must be fast in satisfiable instances.*
- Trade-off between precision and performance.
- *Candidate (Potential) Models*

Heuristic Quantifier Instantiation

- Semantically, $\forall x_1, \ldots, x_n.F$ is equivalent to the infinite conjunction $\beta_1(F) \wedge \beta_2(F) \wedge \ldots$
- Solvers use heuristics to select from this infinite conjunction those instances that are "relevant".
- The key idea is to treat an instance (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.

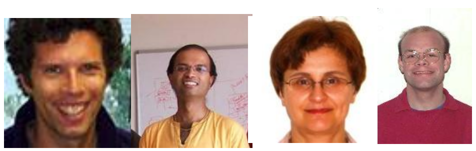
E-matching: Example

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

Quantifiers in Z3

- Z3 uses a E-matching abstract machine.
  - Patterns → 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.

Static Driver Verifier



Ella Bounimova, Vlad Levin, Jakob Lichtenberg,
Tom Ball, Sriram Rajamani, Byron Cook

Overview

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

  c2bp C program → 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 in $B(P, F)$ are true in 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 is implied by *e.*
  - $ImpliedBy_F\ (e) = not\ Implies_F\ (not\ e)$

Computing $Implies_F(e)$

- minterm $m = l_1 \wedge \ldots \wedge l_n$, where $l_i = p_i$, or $l_i = not\ p_i$.
- $Implies_F\ (e)$: 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 $m \Rightarrow e$.
- 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.

SDV & Z3

- Z3 is part of SDV 2.0 (Windows 7)
- All-SAT
  - Fast Predicate Abstraction
- Unsatisfiable cores:
  - Why the path is not feasible?

Demo

More Applications

- Bounded model-checking of model programs 
- Termination 
- Security protocols  
- Business application modeling 
- Cryptography 
- Model Based Testing (SQL-Server)
- *Your killer-application here*

Future/Current Work

- Coming soon (Z3 2.0):
  - Proofs & Unsat cores
  - Superposition Calculus
  - Decidable Fragments
  - Machine Learning
  - Non linear arithmetic (Gröbner Bases)
  - Inductive Datatypes
  - Improved Array & Bit-vector theories
- Several performance improvements
- More “customers” & Applications

Conclusions

- Formal verification is hot at Microsoft.
- Z3 is a new SMT solver from Microsoft Research.
- Z3 is used in several projects.
- Z3 is freely available for academic research:
  - http://research.microsoft.com/projects/z3

Microsoft®

Your potential. Our passion.™

Your SMT problem. Our joy.