Applications and Challenges in Satisfiability Modulo Theories

WING/ETAPS 2009 – York

Leonardo de Moura
Microsoft Research

Symbolic Reasoning

Verification/Analysis tools
need some form of
Symbolic Reasoning

Applications

- Test case generation
- Verifying Compilers
- Predicate Abstraction
- Invariant Generation
- Type Checking
- Model Based Testing

Some Applications @ Microsoft

The Spec# Programming System

HAVOC

FormuLa

Hyper-V
Microsoft Virtualization

Terminator T-2

VCC

SLAM

Yogi

NModel

Vigilante

SpecExplorer

Pex

F7

SAGE

Test case generation

```
unsigned GCD(x, y) {
  requires(y > 0);
  while (true) {
    unsigned m = x % y;
    if (m == 0) return y;
    x = y;
    y = m;
  }
}
```

SSA

$(y_0 > 0)$ and

$(m_0 = x_0 \% y_0)$ and

not $(m_0 = 0)$ and

$(x_1 = y_0)$ and

$(y_1 = m_0)$ and

$(m_1 = x_1 \% y_1)$ and

$(m_1 = 0)$

Solver

$x_0 = 2$

$y_0 = 4$

$m_0 = 2$

$x_1 = 4$

$y_1 = 2$

$m_1 = 0$

We want a trace where the loop is executed twice.

Satisfiability Modulo Theories (SMT)

Is formula *F* satisfiable modulo theory *T* ?

SMT solvers have specialized algorithms for *T*

Satisfiability Modulo Theories (SMT)

$$b + 2 = c \;\; and \;\; f(read(write(a,b,3), c\text{-}2) \neq f(c\text{-}b+1)$$

Satisfiability Modulo Theories (SMT)

$$\boxed{b + 2 = c} \text{ and } f(read(write(a,b,\boxed{3}), \boxed{c-2}) \neq f(\boxed{c-b+1})$$

Arithmetic

Satisfiability Modulo Theories (SMT)

$$b + 2 = c \ \ and \ \ f(\boxed{read}(\boxed{write}(a,b,3), c\text{-}2) \neq f(c\text{-}b+1)$$

Array Theory

Satisfiability Modulo Theories (SMT)

$$b + 2 = c \text{ and } \boxed{f}(read(write(a,b,3), c-2)) \neq \boxed{f}(c-b+1)$$

Uninterpreted Functions

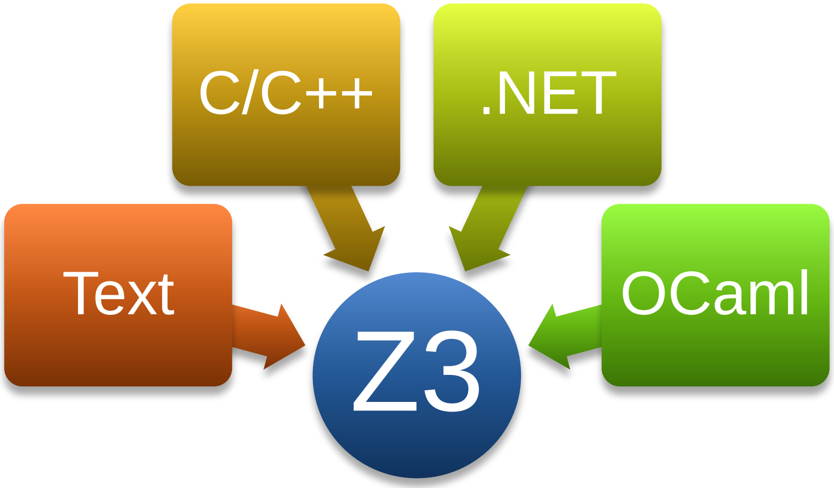
Theories

- ***A Theory is a set of sentences***

- Alternative definition:
  A Theory is a class of structures

SMT@Microsoft: Solver

- Z3 is a new solver developed at Microsoft Research.
- Development/Research driven by internal customers.
- Free for academic research.
- Interfaces:



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

SMT x SAT

For some theories, SMT can be reduced to SAT

Higher level of abstraction

$$\mathrm{bvmul}_{32}(a,b) = \mathrm{bvmul}_{32}(b,a)$$

Ground formulas

*For most SMT solvers: **F is a set of ground formulas***

Many Applications

Bounded Model Checking

Test-Case Generation

DPLL

M | F

Partial model

Set of clauses

DPLL

- Guessing (case-splitting)

$$p \mid p \vee q, \neg q \vee r$$

$$\Downarrow$$

$$p, \neg q \mid p \vee q, \neg q \vee r$$

DPLL

- Deducing

$$p \mid p \vee q, \neg p \vee s$$

$$\Downarrow$$

$$p, s \mid p \vee q, \neg p \vee s$$

DPLL

- Backtracking

$$p, \neg s, q \mid p \vee q, s \vee q, \neg p \vee \neg q$$

$$\Downarrow$$

$$p, s \mid p \vee q, s \vee q, \neg p \vee \neg q$$

Microsoft Research

Modern DPLL

- Efficient indexing (two-watch literal)
- Non-chronological backtracking (backjumping)
- Lemma learning
- …

Solvers = DPLL + Decision Procedures

- Efficient decision procedures for conjunctions of ground atoms.

$a=b, a<5 \mid \neg a=b \vee f(a)=f(b), \quad a < 5 \vee a > 10$

Efficient algorithms

Difference Logic	Belmann-Ford
Uninterpreted functions	Congruence closure
Linear arithmetic	Simplex

Theory Conflicts

a=b, a > 0, c > 0, a + c < 0 | F

backtrack

Naïve recipe?

SMT Solver = DPLL + Decision Procedure

Standard question:

Why don't you use CPLEX for handling linear arithmetic?

Efficient SMT solvers

Decision Procedures must be:

Incremental & Backtracking

Theory Propagation

$$a=b,\ a<5 \mid \ldots\ a<6 \vee f(a) = a$$

$$a=b,\ a<5,\ a<6 \mid \ldots\ a<6 \vee f(a) = a$$

Efficient SMT solvers

Decision Procedures must be:

- Incremental & Backtracking
- Theory Propagation
- Precise (theory) lemma learning
  - $a=b,\ a > 0,\ c > 0,\ a + c < 0 \mid F$
  - Learn clause:
  - $\neg(a=b) \vee \neg(a > 0) \vee \neg(c > 0) \vee \neg(a + c < 0)$
  - Imprecise!
  - Precise clause:
  - $\neg a > 0 \vee \neg c > 0 \vee \neg a + c < 0$

Microsoft Research

Verifying Compilers

Annotated Program → Verification Condition *F*

pre/post conditions
invariants
and other annotations

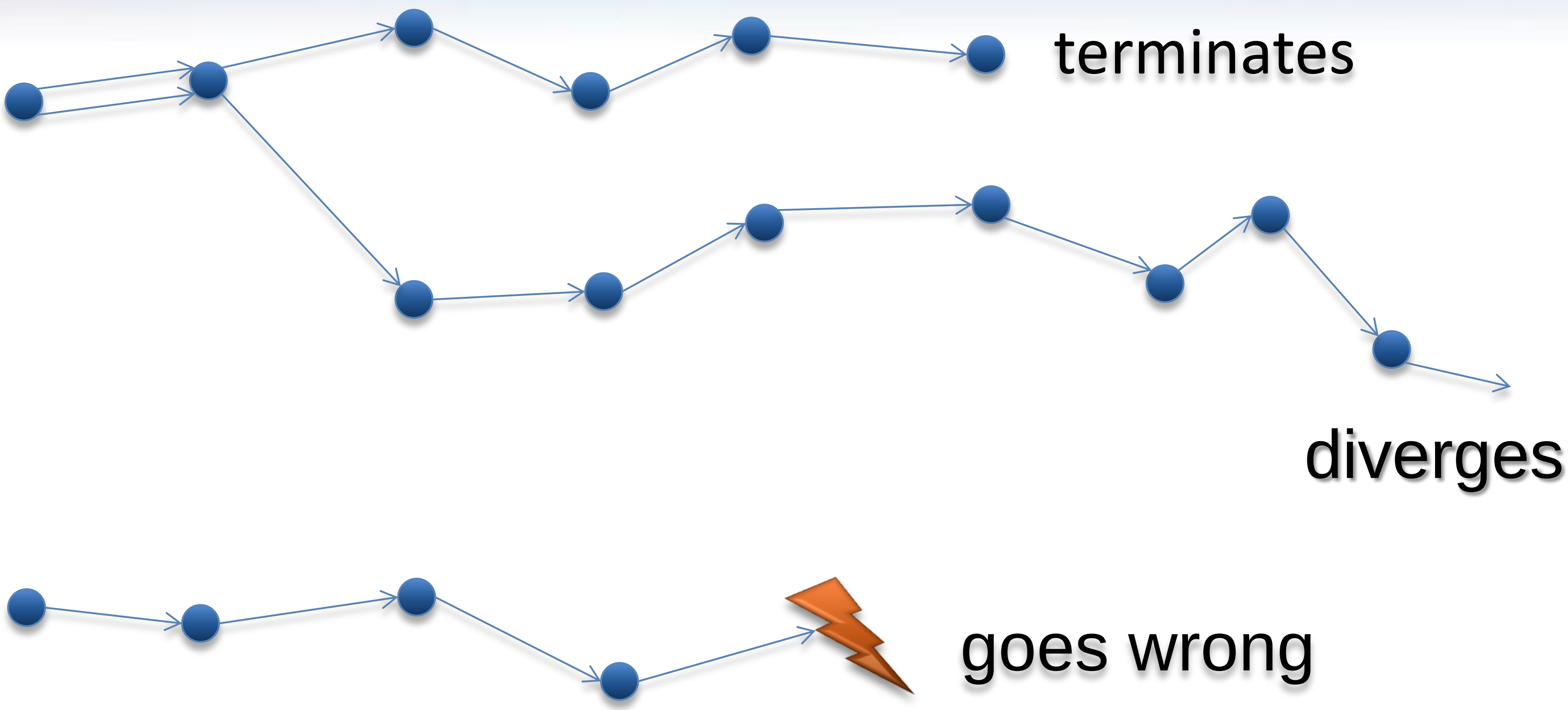
Annotations: Example

```
class C {
    private int a, z;
    invariant z > 0

    public void M()
        requires a != 0
        {
            z = 100/a;
        }
}
```

Microsoft Research

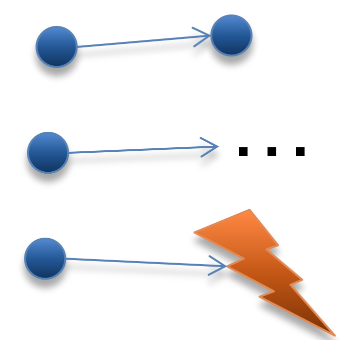
Modeling execution traces



States and execution traces

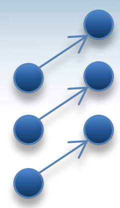
- State
  - Cartesian product of variables
- Execution trace
  - Nonempty finite sequence of states
  - Infinite sequence of states
  - Nonempty finite sequence of states followed by special error state

(x: int, y: int, z: bool)

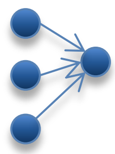


Command language

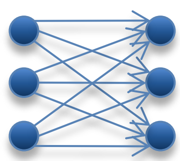
- x := E
  - x := x + 1



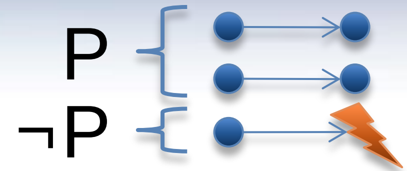
  - x := 10



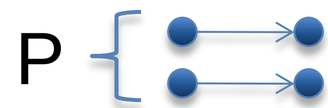
- havoc x



- assert P

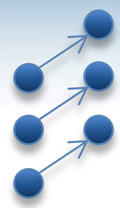


- assume P

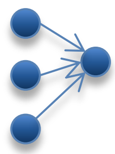


Command language

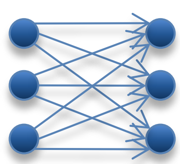
- x := E
  - x := x + 1



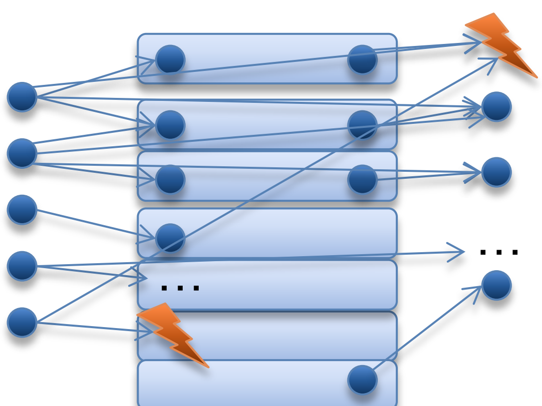
  - x := 10



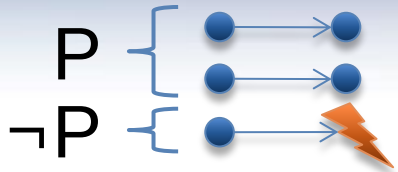
- havoc x



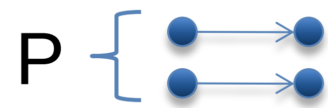
- S ; T



- assert P

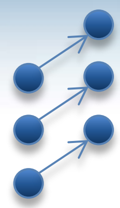


- assume P

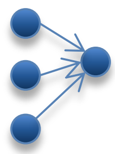


Command language

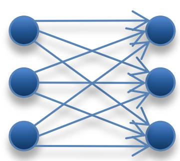
- x := E
  - x := x + 1



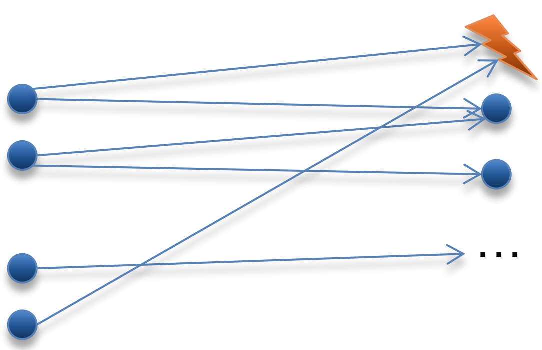
  - x := 10



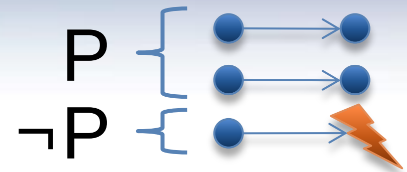
- havoc x



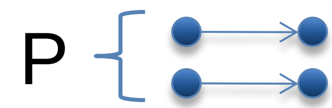
- S ; T



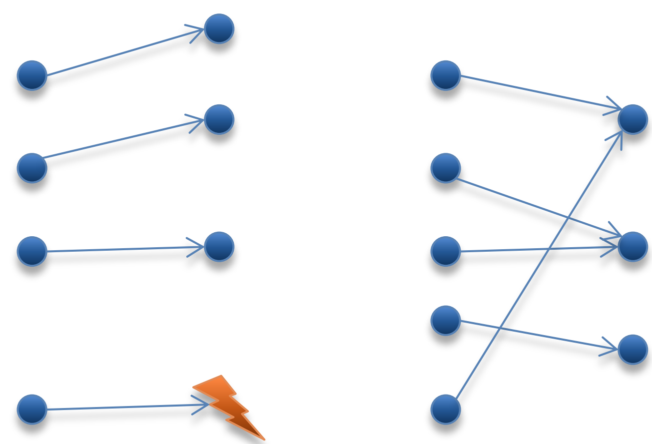
- assert P



- assume P



- S □ T



Reasoning about execution traces

- Hoare triple $\{ P \} \; S \; \{ Q \}$ says that

  every terminating execution trace of S that starts in a state satisfying P

  - does not go wrong, and
  - terminates in a state satisfying Q

Reasoning about execution traces

- Hoare triple $\{ P \} \; S \; \{ Q \}$ says that

  every terminating execution trace of S that starts in a state satisfying P

  - does not go wrong, and
  - terminates in a state satisfying Q
- Given S and Q, what is the weakest P’ satisfying {P’} S {Q} ?
  - P' is called the *weakest precondition* of S with respect to Q, written wp(S, Q)
  - to check {P} S {Q}, check $P \Rightarrow P'$

Weakest preconditions

- wp(x := E, Q) = Q[E / x]
- wp(havoc x, Q) = $(\forall x \bullet Q)$
- wp(assert P, Q) = $P \wedge Q$
- wp(assume P, Q) = $P \Rightarrow Q$
- wp(S ; T, Q) = wp(S, wp(T, Q))
- wp(S $\square$ T, Q) = wp(S, Q) $\wedge$ wp(T, Q)

Structured if statement

if E then S else T end =

assume E; S

$\Box$

assume ¬E; T

Dijkstra's guarded command

if E → S | F → T fi =

```
assert E ∨ F;
(
  assume E;  S
  □
  assume F;  T
)
```

While loop with loop invariant

```
while E
    invariant J
do
    S
end
```

where x denotes the assignment targets of S

check that the loop invariant holds initially

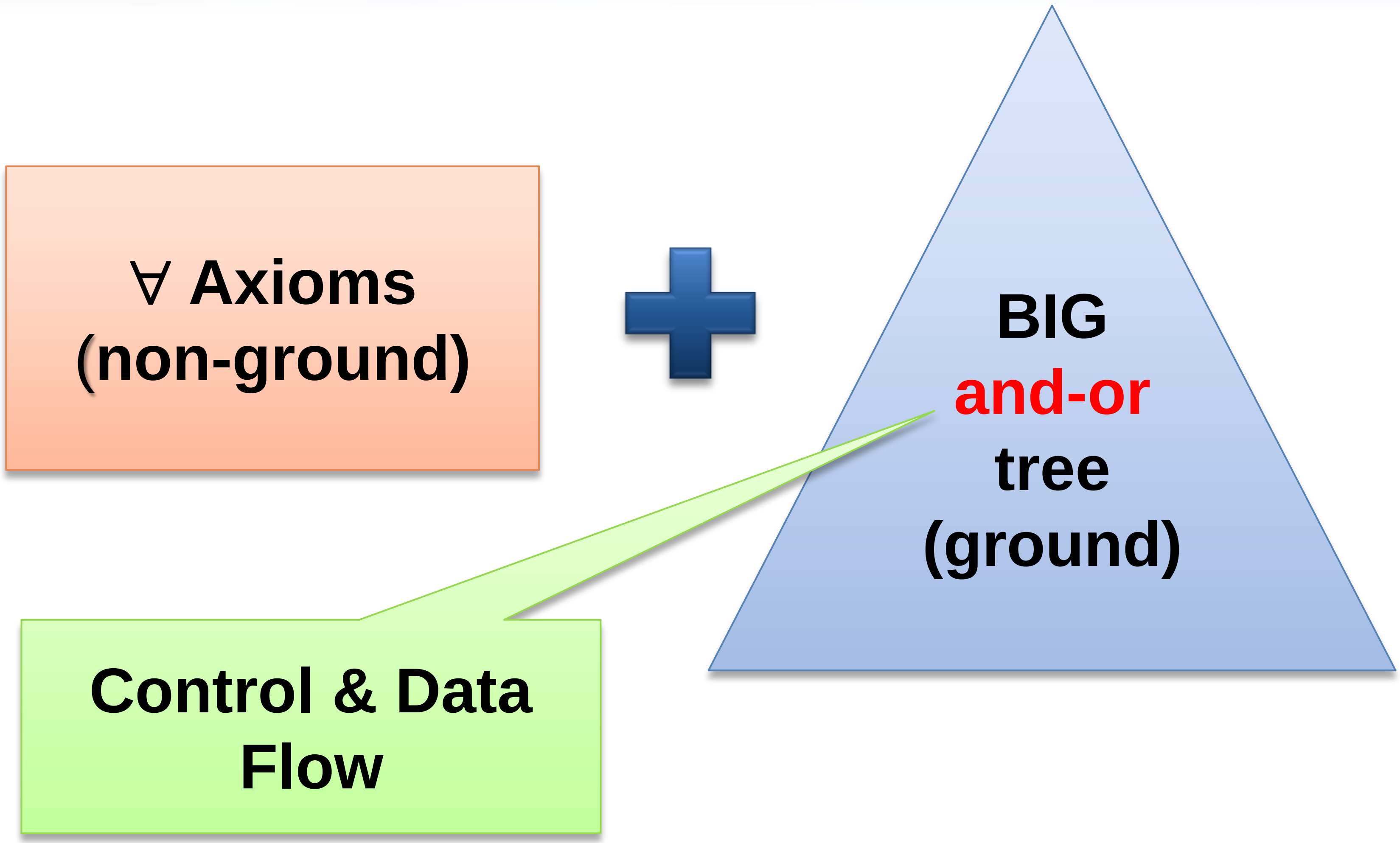
```
=   assert J;
    havoc x;  assume J;
    (   assume E;  S;  assert J;  assume false
    □   assume ¬E
    )
```

“fast forward” to an arbitrary iteration of the loop

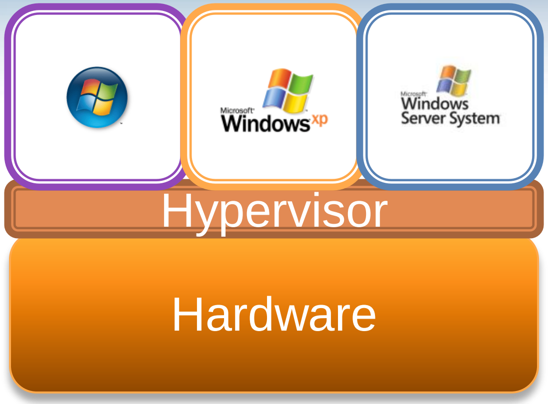
check that the loop invariant is maintained by the loop body

Microsoft Research

Verification conditions: Structure



Hypervisor: A Manhattan Project



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

Hypervisor: Some Statistics

- VCs have several Mb
- Thousands of non ground clauses
- Developers are willing to wait at most 5 min per VC

Challenge: annotation burden

- Partial solutions
  - Automatic generation of: Loop Invariants
  - Houdini-style automatic annotation generation

Challenge

- Quantifiers, quantifiers, quantifiers, …
- Modeling the runtime

$$\forall\, h,o,f: \quad \mathit{IsHeap}(h) \wedge o \neq \mathit{null} \wedge \mathit{read}(h, o, \mathit{alloc}) = t \Rightarrow \mathit{read}(h,o,f) = \mathit{null} \vee \mathit{read}(h, \mathit{read}(h,o,f),\mathit{alloc}) = t$$

Challenge

- Quantifiers, quantifiers, quantifiers, …
- Modeling the runtime
- Frame axioms

$$\forall\, o, f:$$
$$o \neq null \wedge read(h_0, o, alloc) = t \Rightarrow$$
$$read(h_1,o,f) = read(h_0,o,f) \vee (o,f) \in M$$

Challenge

- Quantifiers, quantifiers, quantifiers, …
- Modeling the runtime
- Frame axioms
- User provided assertions

$$\forall\, i,j\colon i \leq j \Rightarrow \mathrm{read}(a,i) \leq \mathrm{read}(b,j)$$

Microsoft Research

Challenge

- Quantifiers, quantifiers, quantifiers, …
- Modeling the runtime
- Frame axioms
- User provided assertions
- Theories
  - $\forall$ x: p(x,x)
  - $\forall$ x,y,z: p(x,y), p(y,z) $\Rightarrow$ p(x,z)
  - $\forall$ x,y: p(x,y), p(y,x) $\Rightarrow$ x = y

Challenge

- Quantifiers, quantifiers, quantifiers, …
- Modeling the runtime
- Frame axioms
- User provided assertions
- Theories
- Solver must be fast in satisfiable instances.

We want to find bugs!

Bad news

There is no sound and refutationally complete procedure for linear integer arithmetic + free function symbols

Many Approaches

Heuristic quantifier instantiation

Combining SMT with Saturation provers

Complete quantifier instantiation

Decidable fragments

Model based quantifier instantiation

E-matching & Quantifier instantiation

- SMT solvers use heuristic quantifier instantiation.
- E-matching (matching modulo equalities).
- Example:

$\forall\ x\colon f(g(x)) = x\ \{\ f(g(x))\ \}$

$a = g(b)$,

$b = c$,

$f(a) \neq c$

Trigger

E-matching & Quantifier instantiation

- SMT solvers use heuristic quantifier instantiation.
- E-matching (matching modulo equalities).
- Example:

$\forall\ x: f(g(x)) = x\ \{ f(g(x)) \}$

$a = g(b)$,

$b = c$,

$f(a) \neq c$

$\xrightarrow{x=b}$ $f(g(b)) = b$

Equalities and ground terms come from the partial model M

E-matching: why do we use it?

- Integrates smoothly with DPLL.
- Efficient for most VCs
- Decides useful theories:
  - Arrays
  - Partial orders
  - …

Efficient E-matching

- E-matching is NP-Hard.
- In practice

Problem	Indexing Technique
Fast retrieval	E-matching code trees
Incremental E-Matching	Inverted path index

E-matching code trees

Trigger:

f(x1, g(x1, a), h(x2), b)

Compiler

Instructions:

1. init(f, 2)
2. check(r4, b, 3)
3. bind(r2, g, r5, 4)
4. compare(r1, r5, 5)
5. check(r6, a, 6)
6. bind(r3, h, r7, 7)
7. yield(r1, r7)

Similar triggers share several instructions.

Combine code sequences in a code tree

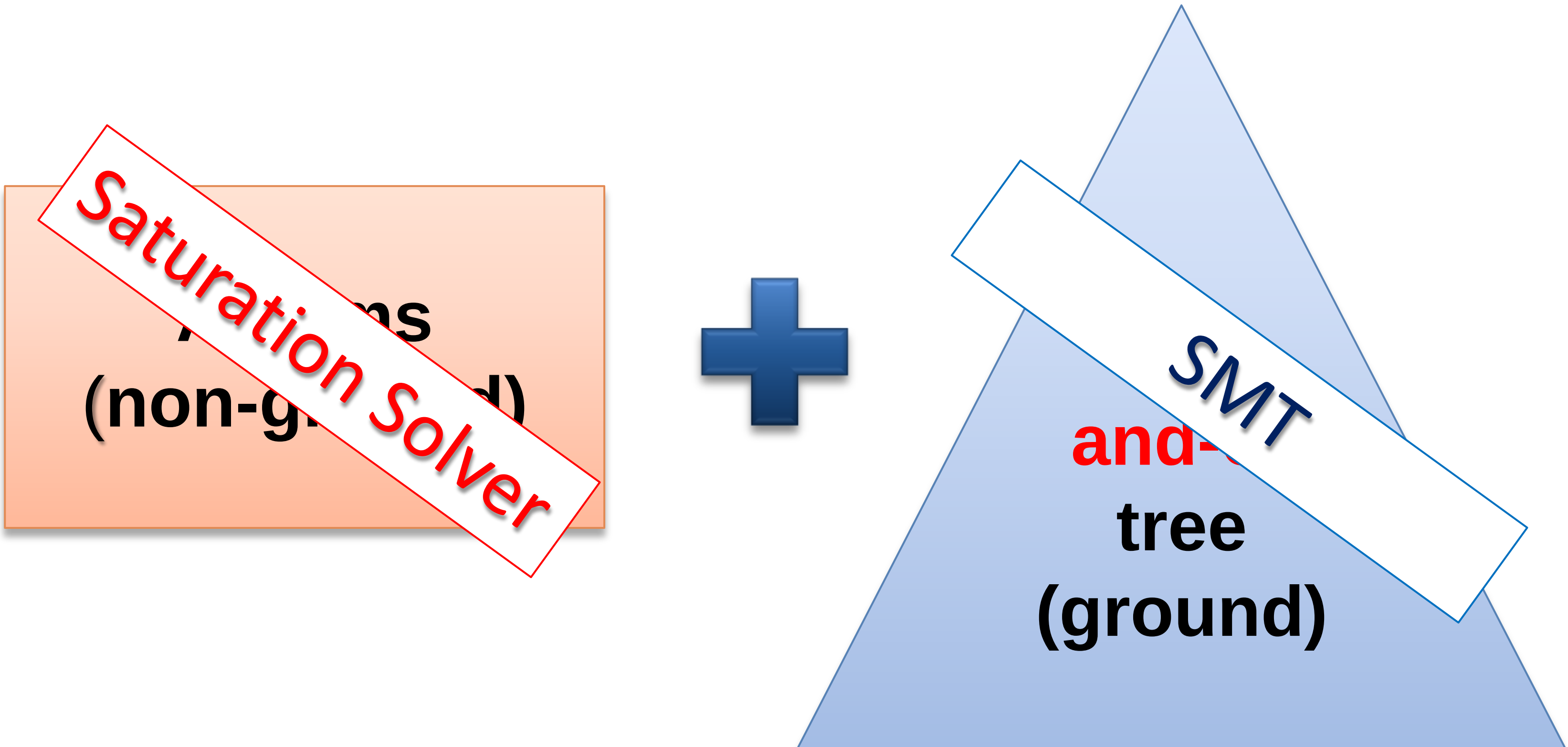
Challenge: modeling runtime

- Is the axiomatization of the runtime consistent?
- False implies everything
- E-matching doesn't work

  No ground terms to instantiate clauses
- Partial solution: SMT + Saturation Provers
- Found many bugs using this approach

DPLL(Γ)

- Tight integration: DPLL + Saturation solver.



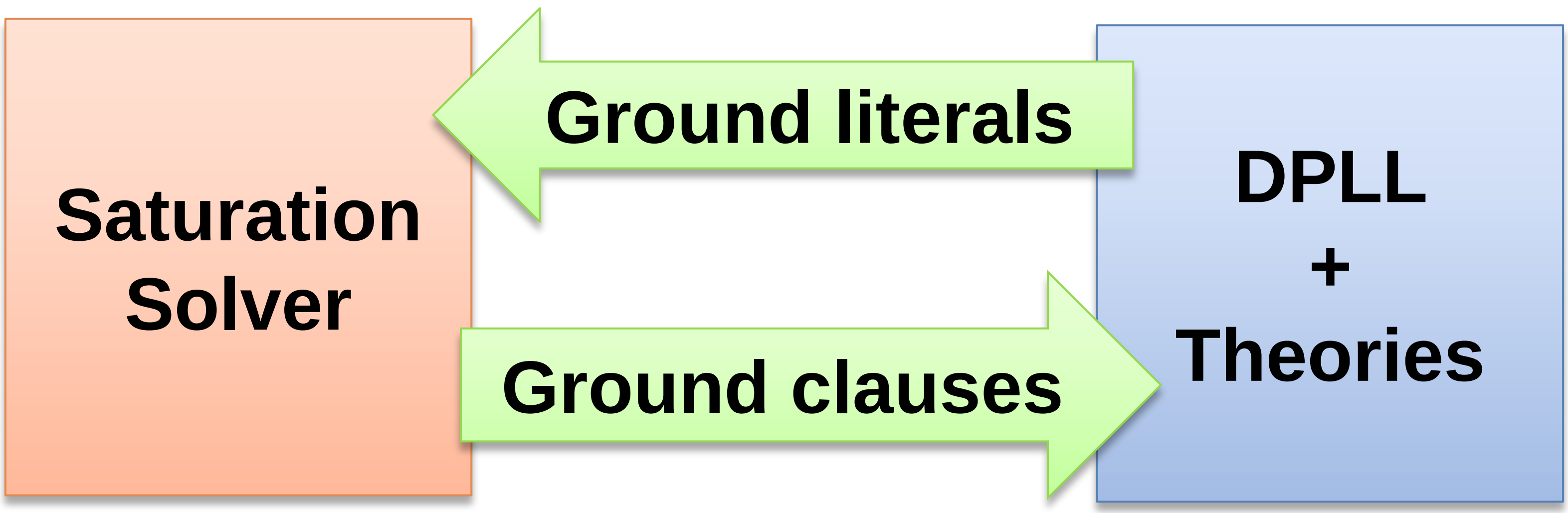
DPLL(Γ)

- Inference rule:

$$\frac{C_1 \quad \ldots \quad C_n}{C}$$

- DPLL(Γ) is parametric.
- Examples:
  - Resolution
  - Superposition calculus
  - …

DPLL(Γ)


Saturation Solver
Ground literals
Ground clauses
DPLL
+
Theories

Challenge: Robustness

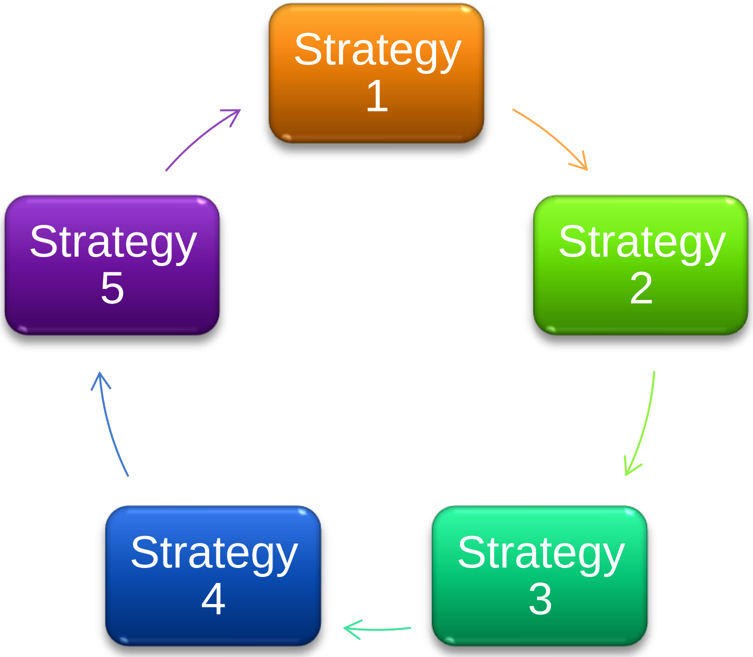
- Standard complain

  "I made a small modification in my Spec, and Z3 is timingout"

- This also happens with SAT solvers (NP-complete)
- In our case, the problems are undecidable
- Partial solution: parallelization

Parallel Z3

- Joint work with Y. Hamadi (MSRC) and C. Wintersteiger
- Multi-core & Multi-node (HPC)
- Different strategies in parallel
- Collaborate exchanging lemmas



Microsoft Research

Challenge: Non-Linear Arithmetic

- Non-linear arithmetic is necessary for verifying embedded and hybrid systems
- Non-linear integer arithmetic is undecidable
- Many approaches for non linear real arithmetic
  - Cylindrical Algebraic Decomposition

    Doubly exponential procedure
  - Grobner Basis + “extensions”
  - Heuristics

Predicate Abstraction & Invariant Generation

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.

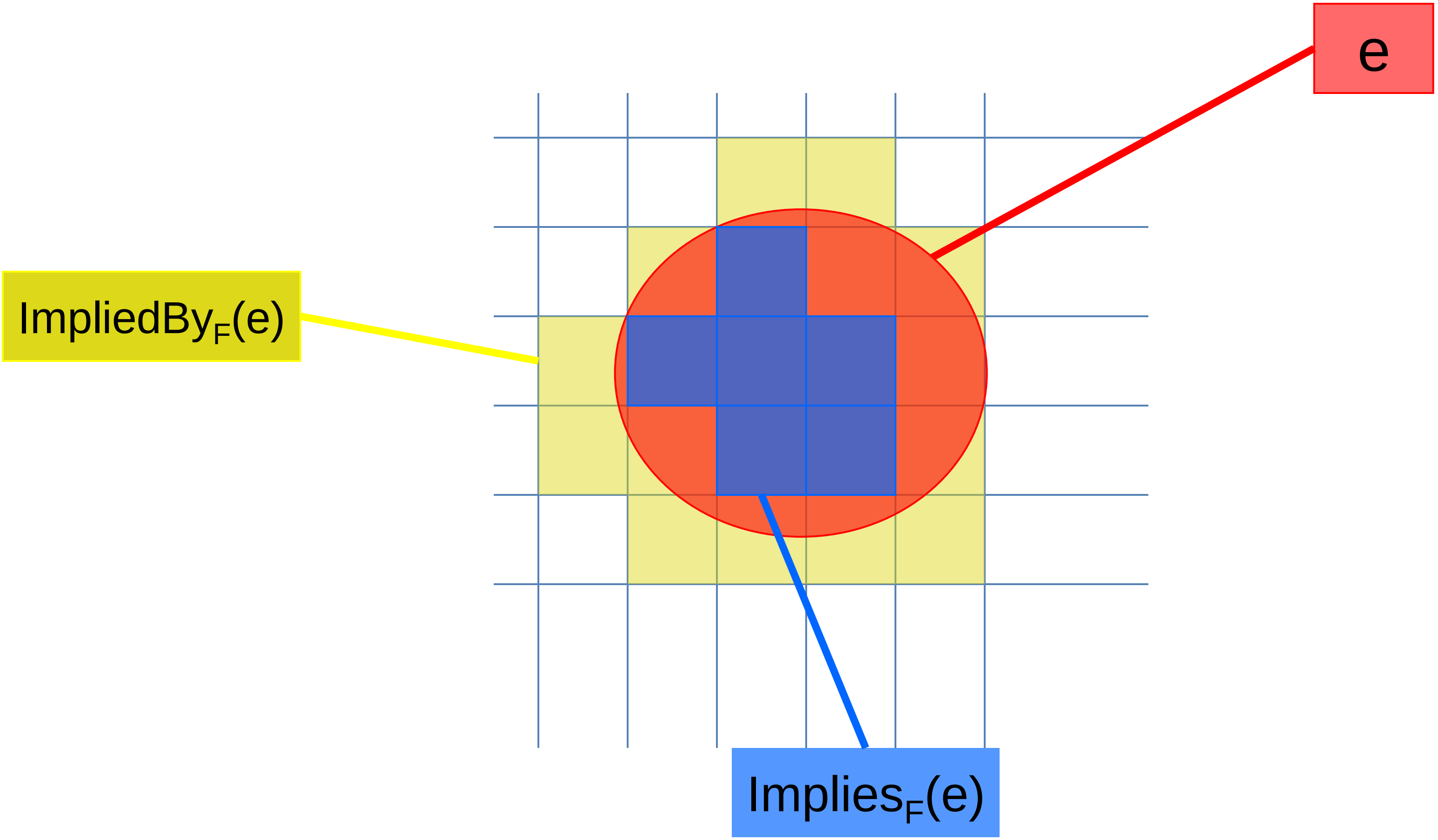
Predicate Abstraction: *c2bp*

- **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.*
  - $ImpliedBy_F\ (e) = not\ Implies_F\ (not\ e)$

$\text{Implies}_F(e)$ and $\text{ImpliedBy}_F(e)$



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

Computing $Implies_F(e)$

- F = { x < y, x = 2}
- *e* : y > 1
- Minterms over F
  - !x<y, !x=2 implies y>1
  - x<y, !x=2 implies y>1
  - !x<y, x=2 implies y>1
  - x<y, x=2 implies y>1

$Implies_F(y>1) = x<y \wedge x=2$

$Implies_F(y>1) = b_1 \wedge b_2$

Microsoft Research

Challenge: Rich API

- All-SAT
  - Better (more precise) Predicate Abstraction
- Unsatisfiable cores
  - Why the abstract path is not feasible?
  - Fast Predicate Abstraction

Unsatisfiable cores

- Let S be an unsatisfiable set of formulas.
- $S' \subseteq S$ is an unsatisfiable core of S if:
  - S' is also unsatisfiable, and
  - There is not $S'' \subset S'$ that is also unsatisfiable.
- Computing $\text{Implies}_F(e)$ with $F = \{p_1, p_2, p_3, p_4\}$
  - Assume $p_1, p_2, p_3, p_4 \Rightarrow e$ is valid
  - That is $p_1, p_2, p_3, p_4, \neg e$ is unsat
  - Now assume $p_1, p_3, \neg e$ is the unsatisfiable core
  - Then it is unnecessary to check:
    - $p_1, \neg p_2, p_3, p_4 \Rightarrow e$
    - $p_1, \neg p_2, p_3, \neg p_4 \Rightarrow e$
    - $p_1, p_2, p_3, \neg p_4 \Rightarrow e$

Loop invariants

Θ

while (c) {
    S
}
Post

$$\exists I \forall x \underbrace{\begin{bmatrix} \Theta(x) \Rightarrow I(x) \\ I(x) \wedge c(x) \wedge S(x,x') \Rightarrow I(x') \\ \neg c(x) \wedge I(x) \Rightarrow Post(x) \end{bmatrix}}_{\varphi_1(I,x)}$$

How to find loop invariant *I* ?

Template based approach

- I is a Boolean combination of $F = \{p_1, \dots, p_n\}$
- Unknown invariant on the LHS constraints how weak I can be

  $I(x) \wedge \neg c(x) \Rightarrow Post(x) \qquad I(x) \Rightarrow \neg c(x) \Rightarrow Post(x)$

- Unknown invariant on the RHS constraints how strong I can be

  $\Theta(x) \Rightarrow I(x)$

- More details: Constraint-based Invariant Inference over Predicate Abstraction, S. Gulwani et al, VMCAI 2009

Bit-precise test case generation

Test case generation

```
unsigned GCD(x, y) {
  requires(y > 0);
  while (true) {
     unsigned m = x % y;
      if (m == 0) return y;
      x = y;
      y = m;
  }
}
```

SSA

$(y_0 > 0)$ and

$(m_0 = x_0 \% y_0)$ and

not $(m_0 = 0)$ and

$(x_1 = y_0)$ and

$(y_1 = m_0)$ and

$(m_1 = x_1 \% y_1)$ and

$(m_1 = 0)$

Solver

$x_0 = 2$

$y_0 = 4$

$m_0 = 2$

$x_1 = 4$

$y_1 = 2$

$m_1 = 0$

Challenge: Machine Arithmetic

- Most solvers use bit-blasting
- **bvmul_{32}(a,b)** is converted into a multiplier circuit
- Solvers may run out of memory
- "Smart" algorithms are usually less efficient than bit-blasting

Challenge: Floating Point Arithmetic

- I'm unaware of any SMT solver for floating point arithmetic
- Approximate using Reals

  Unsound!

  Incomplete!

Conclusion

- Logic as a platform
- Most verification/analysis tools need symbolic reasoning
- SMT is a hot area
- Many applications & challenges
- http://research.microsoft.com/projects/z3

Thank You!