SMT Solvers in Program Analysis and Verification

Nikolaj Bjørner and Leonardo de Moura

T 3 – August 10
Satisfiability Modulo Theories solvers in Program Analysis and Verification

Leonardo de Moura and Nikolaj Bjørner
Microsoft Research

Tutorial overview
- Appetizers
- SMT solving
- Applications
- Applications at Microsoft Research
- Background
  - Basics, DPLL(⊕), Equality, Arithmetic, DPLL(T), Arrays, Matching
- Z3 – An Efficient SMT solver

Domains from programs
- Bits and bytes
  \[0 = ((x - 1) \& x) \iff x = 00100000\ldots00\]
- Numbers
  \[x + y = y + x\]
- Arrays
  \[\text{read(write}(a,i,4),i) = 4\]
- Records
  \[\text{mkpair}(x,y) = \text{mkpair}(z,u) \implies x = z\]
- Heaps
  \[n \rightarrow^* n' \land m = \text{cons}(a,n) \implies m \rightarrow^* n'\]
- Data-types
  \[\text{car}(\text{cons}(x,nil)) = x\]
- Object inheritance
  \[B < A \land C < B \implies C < A\]

Satisfiability Modulo Theories (SMT)
\[x + 2 = y \implies [\text{if read}(\text{write}(a,x,3),y-2)] = [\text{if}(y-x+1)]\]

Applications Appetizer
Some takeaways from Applications

- SMT solvers are used in several applications:
  - Program Verification
  - Program Analysis
  - Program Exploration
  - Software Modeling
- SMT solvers are
  - directly applicable, or
  - disguised beneath a transformation
- Theories and quantifiers supply abstractions
  - Replace ad-hoc, often non-scalable, solutions

Program Verification

- Hypothetical diagram with various tools and concepts such as VCC, Boogie, Win. Modules, and Z3.

Test case generation

- Diagram illustrating test case generation process:
  - Execution Path
  - Path Condition
  - Known Paths
  - New Inputs
  - Unexplored path

Static Driver Verifier

- Z3 is part of SDV 2.0 (Windows 7)
- It is used for:
  - Predicate abstraction (c2bp)
  - Counter-example refinement (newton)

More applications

- Bounded model-checking of model programs
- Termination
- Security protocols, F#/7
- Business application modeling
- Cryptography
- Model Based Testing (SQL-Server)
- Verified garbage collectors
Program Exploration with Pex

Nikolai Tillmann, Pel: de Halleux

http://research.microsoft.com/Pex

What is Pex
- Test input generator
  - Pex starts from parameterized unit tests
  - Generated tests are emitted as traditional unit tests
- Dynamic symbolic execution framework
  - Analysis of .NET instructions (bytecode)
  - Instrumentation happens automatically at JIT time
  - Using SMT-solver Z3 to check satisfiability and generate models = test inputs

Array List: The Spec

Array List: AddItem Test

Array List: Starting Pex...

Array List: Run 1, (0,null)
ArrayList: Run 1, (0,null)

```csharp
class ArrayListTest { 
    [PexMethod]
    void AddItem(int c, object item) {
        var list = new ArrayList();
        list.Add(item);
        Assert(list[0] == item); } 
} 
```

- Inputs: (0,null)
- Observed Constraints: ![Observations](image1)

ArrayList: Run 1, (0,null)

```csharp
class ArrayListTest { 
    [PexMethod]
    void AddItem(int c, object item) {
        var list = new ArrayList();
        list.Add(item);
        Assert(list[0] == item); } 
} 
```

- Inputs: (0,null)
- Observed Constraints: ![Observations](image2)

ArrayList: Run 1, (0,null)

```csharp
class ArrayListTest { 
    [PexMethod]
    void AddItem(int c, object item) {
        var list = new ArrayList();
        list.Add(item);
        Assert(list[0] == item); } 
} 
```

- Inputs: (0,null)
- Observed Constraints: ![Observations](image3)

ArrayList: Run 1, (0,null)

```csharp
class ArrayListTest { 
    [PexMethod]
    void AddItem(int c, object item) {
        var list = new ArrayList();
        list.Add(item);
        Assert(list[0] == item); } 
} 
```

- Inputs: (0,null)
- Observed Constraints: ![Observations](image4)

ArrayList: Picking the next branch to cover

```csharp
class ArrayListTest { 
    [PexMethod]
    void AddItem(int c, object item) {
        var list = new ArrayList();
        list.Add(item);
        Assert(list[0] == item); } 
} 
```

- Constraints to solve: ![Constraints](image5)
- Inputs: (0,null)
- Observed Constraints: ![Observations](image6)

ArrayList: Solve constraints using SMT solver

```csharp
class ArrayListTest { 
    [PexMethod]
    void AddItem(int c, object item) {
        var list = new ArrayList();
        list.Add(item);
        Assert(list[0] == item); } 
} 
```

- Constraints to solve: ![Constraints](image7)
- Inputs: (0,null)
- Observed Constraints: ![Observations](image8)

ArrayList: Run 2, (1,null)

```csharp
class ArrayListTest { 
    [PexMethod]
    void AddItem(int c, object item) {
        var list = new ArrayList();
        list.Add(item);
        Assert(list[0] == item); } 
} 
```

- Constraints to solve: ![Constraints](image9)
- Inputs: (0,null)
- Observed Constraints: ![Observations](image10)
### ArrayList: Pick new branch

```csharp
class ArrayListTest {
    [PexMethod]
    void AddItem(int c, object item) {
        var list = new ArrayList(c);
        list.AddItem();
        Assert(list[0] == item);
    }
}
```

### ArrayList: Run 3, (-1, null)

```csharp
class ArrayListTest {
    [PexMethod]
    void AddItem(int c, object item) {
        var list = new ArrayList(c);
        list.AddItem();
        Assert(list[0] == item);
    }
}
```

---

**Pex** – Test more with less effort

- Reduce testing costs
- Automated analysis, reproducible results
- Produce more secure software
- White-box code analysis
- Produce more reliable software
- Analysis based on contracts written as code

---

**White box testing in practice**

**How to test this code?**

(Real code from .NET base class libraries.)

---

© 2007 Microsoft Corporation. All rights reserved. Microsoft, Windows, Windows Vista and other product names are or may be registered trademarks and/or trademarks in the U.S. and/or other countries. The information herein is for informational purposes only and represents the current view of Microsoft Corporation as of the date of this presentation. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information provided after the date of this presentation.

MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS PRESENTATION.
White box testing in practice

Test Input Generation tomorrow

Test Input Generation by Dynamic Symbolic Execution

Constraint System

Test Inputs

Execution Path

Known Paths

Result: small test suite, high code coverage

Finds only real bugs

No false warnings

Test Input Generation by Dynamic Symbolic Execution

Constraint System

Test Inputs

Execution Path

Known Paths

Result: small test suite, high code coverage

Finds only real bugs

No false warnings
Test Input Generation by Dynamic Symbolic Execution

- Test Inputs
- Constraint System
- Execution Path
- Known Paths

Result: small test suite, high code coverage
Finds only real bugs
No false warnings

Automatic Test Input Generation: Whole-program, white box code analysis

- Test Inputs
- Constraint System
- Execution Path
- Known Paths

Result: small test suite, high code coverage
Finds only real bugs
No false warnings

Constraint Solving: Preprocessing

- Independent constraint optimization + Constraint caching (similar to EXE)
- Idea: Related execution paths give rise to “similar” constraint systems
- Example: Consider \( x > y \land z > 0 \) vs. \( x > y \land z \leq 0 \)
- If we already have a cached solution for a “similar” constraint system, we can reuse it
  - \( x=1, y=0, z=1 \) is solution for \( x > y \land z > 0 \)
  - we can obtain a solution for \( x > y \land z \leq 0 \) by reusing old solution of \( x > y \): \( x=1, y=0 \)
  - combining with solution of \( z \leq 0; z=0 \)

Constraint Solving: Z3

- Rich Combination: Solvers for uninterpreted functions with equalities, linear integer arithmetic, bitvector arithmetic, arrays, tuples
- Formulas may be a big conjunction
  - Pre-processing step
  - Eliminate variables and simplify input format
- Universal quantifiers
  - Used to model custom theories, e.g., .NET type system
- Model generation
  - Models used as test inputs
- Incremental solving
  - Given a formula \( F \), find a model \( M \), that minimizes the value of the variables \( x_0, \ldots, x_n \)
- Push / Pop of contexts for model minimization
- Programmatic API
  - For small constraint systems, text through pipes would add huge overhead

Monitoring by Code Instrumentation

- Record concrete values
- (The real C# compiler output is actually more complicated.)
Spec# and Boogie

Rustan Leino & Mike Barnett

Spec# (annotated C#)
Spec# Compiler
Boogie PL
VC Generator
Formulas
Z3

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 Z3

Basic verifier architecture

Source language
Intermediate verification language
Verification condition
(logical formula)

Verification architecture

Spec# compiler
MSIL
Bytecode translator
Boogie
V.C. generator
Inference engine
VC Generator
Formulas
Z3

Spec# compiler
C
vcc
HAVOC
Dafny verifier
Dafny

Modeling execution traces

terminates
diverges
goes wrong
States and execution traces

- **State**
  - Cartesian product of variables \((x: \text{int}, y: \text{int}, z: \text{bool})\)

- **Execution trace**
  - Nonempty finite sequence of states
  - Infinite sequence of states
  - Nonempty finite sequence of states followed by special error state

Command language

- \(x := E\)
- \(x := x + 1\)
- \(x := 10\)
- \(\text{havoc } x\)
- \(\text{assert } P\)
- \(\text{assume } P\)
- \(\neg P\)

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 \(P\) and \(Q\), what is the largest \(S'\) satisfying \(\{ P \} S' \{ Q \}\)?
  - to check \(\{ P \} S \{ Q \}\), check \(S \subseteq S'\)

  - **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

- \( \wp(x := E, Q) = Q[E/x] \)
- \( \wp(\text{havoc } x, Q) = (\forall x \cdot Q) \)
- \( \wp(\text{assert } P, Q) = P \land Q \)
- \( \wp(\text{assume } P, Q) = P \implies Q \)
- \( \wp(S ; T, Q) = \wp(S, \wp(T, Q)) \)
- \( \wp(S \triangleright T, Q) = \wp(S, Q) \land \wp(T, Q) \)

Structured if statement

- \( \text{if } E \text{ then } S \text{ else } T \text{ end } = \)
  - \( \text{assume } E; S \)
  - \( \text{assume } \neg E; T \)

Dijkstra's guarded command

- \( \text{if } E \rightarrow S \mid F \rightarrow T \text{ fi } = \)
  - \( \text{assert } E \lor F; \)
  - \( (\)
  - \( \text{assume } E; S \)
  - \( \)
  - \( \text{assume } F; T \)
  - \( ) \)

Picking any good value

- \( \text{assign } x \text{ such that } P = \)
  - \( \text{havoc } x; \text{ assume } P \)
  - \( P \)
  - \( \neg P \)
  - \( ; \)
  - \( = \)
  - \( \text{assign } x \text{ such that } x^*x = y \)

Procedures

- A \textit{procedure} is a user-defined command
- \textit{procedure} \( M(x, y, z) \) returns \((r, s, t)\)
  - requires \(P\)
  - modifies \(g, h\)
  - ensures \(Q\)

Procedure example

- \textit{procedure} \( \text{Inc}(n) \) returns \((b)\)
  - requires \(0 \leq n\)
  - modifies \(g\)
  - ensures \(g = \text{old}(g) + n\)
Procedures

- A **procedure** is a user-defined command
- procedure **M**(x, y, z) returns (r, s, t)
  requires P
  modifies g, h
  ensures Q
- call a, b, c := M(E, F, G)
  = x' := E; y' := F; z' := G;
  assert P';
  g0 := g; h0 := h;
  havoc g, h, r', s', t';
  assume Q';
  a := r'; b := s'; c := t'

Procedures implementations

- procedure **M**(x, y, z) returns (r, s, t)
  requires P
  modifies g, h
  ensures Q
- implementation **M**(x, y, z) returns (r, s, t) is
  = assume P';
  g0 := g; h0 := h;
  S;
  assert Q'
  where
  - g0, h0 are fresh names
  - Q' is Q with g0, h0 for old(g), old(h)

While loop with loop invariant

while E
  invariant J
  do
  S
  end
  = assert J;
  havoc x; assume J;
  ( assume E; S; assert J; assume false
  □ assume ¬E
  )

Properties of the heap

- introduce:
  function **IsHeap**(HeapType) returns (bool);
- introduce:
  axiom (∀ h: HeapType, o: Ref, f: Field Ref •
  **IsHeap**(h) ∧ o ≠ null ∧ h[ o, alloc]
  ⇒ h[ o, f] = null ∨ h[ h[ o,f], alloc ]);
- introduce: assume **IsHeap**(Heap)
  after each Heap update; for example:
  Tr[[ E.x := F ]] =
  assert ...; Heap[...]; := ...;
  assume **IsHeap**(Heap)

Properties of the heap

- introduce:
  method **M**(x: X) returns (y: Y)
  requires P; modifies S; ensures Q;
  ( Stmt )
- procedure **M**(this: Ref, x: Ref) returns (y: Ref);
  free requires **IsHeap**(Heap);
  free requires this ≠ null ∧ Heap[ this, alloc];
  free requires x = null ∨ Heap[ x, alloc];
  free requires Df[[ P ]];
  requires Df[[ S ]];
  modifies Heap;
  ensures Df[[ Q ]];
  ensures (∀ o: Ref, f: Field •
  o ≠ null ∧ old[Heap][ o, alloc] ⇒ Heap[ o, f] = old[Heap][ o, f] ∨
  (o, f) ∉ old[Tr[[ S ]]];)
  free ensures **IsHeap**(Heap);
  free ensures y = null ∨ Heap[ y, alloc];
  free ensures (Yo: Ref • old[Heap][ o, alloc] ⇒ Heap[ o, alloc]);
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.

The Static Driver Verifier
SLAM

Example
Do this code obey the locking rule?

```c
Do()
  KeAcquireSpinLock();
  npacketsOld = npackets;
  if (request) {
    request = request->Next;
    KeReleaseSpinLock();
  } else {
    while (npackets != npacketsOld);
  }
  KeReleaseSpinLock();
```

Example
Model checking
Boolean program

```c
Do()
  KeAcquireSpinLock();
  if(*)
    KeReleaseSpinLock();
  )
  while (*);
  KeReleaseSpinLock();
```
Observations about SLAM

- Automatic discovery of invariants
  - driven by property and a finite set of (false) execution paths
  - predicates are not invariants, but observations
  - abstraction + model checking computes inductive invariants (boolean combinations of observations)

- A hybrid dynamic/static analysis
  - newton executes path through C code symbolically
  - c2bp+bebop explore all paths through abstraction

- A new form of program slicing
  - program code and data not relevant to property are dropped
  - non-determinism allows slices to have more behaviors

Syntactic Sugar

```c
if (e) {
  S1;
} else {
  S2;
} 
S3;
```

goto L1, L2;

L1: assume(e);
S1;
goto L3;
L2: assume(!e);
S2;
goto L3;
L3: S3;
```
Predicate Abstraction: \texttt{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 Assignments via WP

- Statement \( y = y + 1 \) and \( F = \{ y < 4, y < 5 \} \)
  - \( \{ y < 4 \}, \{ y < 5 \} = ((\neg y < 5) || \neg(y < 4)) \ ? \text{false} : *), \{ y < 4 \} \)
- \( \text{WP}(x = e, Q) = Q[x \to e] \)
- \( \text{WP}(y = y + 1, y < 5) = (y < 5)[y \to y + 1] = (y + 1 < 5) = (y < 4) \)

WP Problem

- \( \text{WP}(s, p_i) \) is not always expressible via \( \{ p_1, \ldots, p_n \} \)
- **Example:**
  - \( F = \{ x == 0, x == 1, x < 5 \} \)
  - \( \text{WP}(x = x + 1, x < 5) = x < 4 \)

Abstracting Expressions via \( F \)

- \( \text{Implies}_F(e) \)
  - Best Boolean function over \( F \) that implies \( e \).
- \( \text{ImpliedBy}_F(e) \)
  - Best Boolean function over \( F \) that is implied by \( e \).
  - \( \text{ImpliedBy}_F(e) = \neg \text{Implies}_F(\neg e) \)

Implies \( F(e) \) and ImpliedBy \( F(e) \)

- **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 \( F(e) \)

- minterm \( m = l_1 \land \ldots \land l_n \), where \( l_i = p_i \) or \( l_i = \neg p_i \)
- \( \text{Implies}_F(e) \): 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 $\text{Implies}_F(e)$

- $F = \{ x < y, x = 2 \}$
- $e : y > 1$

Minterms over $F$
- $\neg x < y, \neg x = 2 \implies y > 1$
- $x < y, \neg x = 2 \implies y > 1$
- $\neg x < y, x = 2 \implies y > 1$
- $x < y, x = 2 \implies y > 1$

$\text{Implies}_F(y > 1) = \neg x < y, x = 2$

Assigning Example

Statement: $y = y + 1$  
Predicates: $\{x == y\}$

Weakest Precondition:
$WP(y = y + 1, x==y) = x == y + 1$

$\text{Implies}_F(x == y + 1) = \text{false}$
$\text{Implies}_F(x != y + 1) = x == y$

Abstraction of $y = y + 1$
$\{x == y\} = \{x == y\}$ ? false : *;

Abstracting Assignments

- if $\text{Implies}_F(WP(s, p_i))$ is true before $s$ then
  $p_i$ is true after $s$
- if $\text{Implies}_F(WP(s, \neg p_i))$ is true before $s$ then
  $p_i$ is false after $s$

$\{p_i\} = \text{Implies}_F(WP(s, p_i))$ ? true : $\text{Implies}_F(WP(s, \neg p_i))$ ? false : *;

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.

Abstracting Assumes

- $WP(\text{assume}(e), Q) = e \implies Q$
- $\text{assume}(e)$ is abstracted to:
  $\text{assume}(\text{ImpliedBy}_F(e))$
- Example:
  $F = \{x == 2, x < 5\}$
  $\text{assume}(x < 2)$ is abstracted to:
  $\text{assume}(x < 5) \&\& \{x == 2\}$

A Verifying C Compiler

Ernie Cohen, Michal Moskal, Herman Venter, Wolfram Schulte
+ Microsoft Aachen + Verisoft Saarbrücken
**Microsoft Hypervisor**

- **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 grand verification challenge

---

**What is to be verified?**

- **Source code**: C + x64 assembly

- **Sample verifiable slices**:
  - **Safety**: Basic memory safety
  - **Functionality**: Hypervisor simulates a number of virtual x64 machines.
  - **Utility**: Hypervisor services guest OS with available resources.

---

**HAVOC's Architecture**

- C program
- Front End
- BoundedPL program
- Verification Conditions
- Warning

---

**Heaps and Shapes**

Representative shape graph in Windows Kernel component

Doubly linked lists in Windows Kernel code
Precise and expressive heap reasoning

- Pointer Arithmetic
  \[ q = \text{CONTAINING_RECORD}(p, IRP , \text{link}) = (\text{IRP} \cdot ((\text{char}*)p - (\text{char}*)(&((\text{IRP} \cdot 0)\rightarrow \text{link}))) \]

- Transitive Closure
  \[ \text{Reach}(\text{next}, u) = \{ u, u\rightarrow \text{next}, u\rightarrow \text{next}\rightarrow \text{next}, \ldots \} \]

Efficient logic for program verification

- Logic with Reach, Quantifiers, Arithmetic
- Expressive
- Careful use of quantifiers
- Efficient logic
- Only NP-complete

DASH Algorithm

- Main workhorse: test case generation
- Use counterexamples from current abstraction to "extend frontier" and generate tests
- When test case generation fails, use this information to "refine" abstraction at the frontier
- Use only aliases that happen on the tests!

Annotation Language & Logic

- Procedure contracts
- requires, ensures, modifies
- Arbitrary C expressions
- program variables, resources
- Boolean connectives
- quantifiers
- Can express a rich set of contracts
- API usage (e.g. lock acquire/release)
- Synchronization protocols
- Memory safety
- Data structure invariants (linked list)
- Challenge:
  - Retain efficiency
  - Decidable fragments

Logic with Reach, Quantifiers, Arithmetic

- Expressive
- Careful use of quantifiers
- Efficient logic
- Only NP-complete

Combining Random Testing with Model Checking

Aditya Nori, Sriram Rajamani, Nels E. Beckman, Nori, Rajamani, Rob Simmons

Example

\begin{verbatim}
void lock(int x) {
  20:  sfx(x = 0);
  25:  assert(1);
  30:  x = 1;
}

void unlock(int x) {
  20:  sfx(x = 1);
  25:  assert(0);
}
\end{verbatim}
Example

void prove_me(int y)
{
  1: do {
  2:    lock();
  3:    x = y;
  4:    if (+) {
  5:      unlock();
  6:      y = y + 1;
  7:    } while (x\neq y);
  8:    unlock();
  }

Template-based refinement

\( p = (\text{lock.state} = \text{L}) \)

Symbolic execution + Theorem Proving

void prove_me(int y)
{
  1: do {
  2:    lock();
  3:    x = y;
  4:    if (+) {
  5:      unlock();
  6:      y = y + 1;
  7:    } while (x\neq y);
  8:    unlock();
  }

Input: Program P
Property \( \psi \)

Symbolic memory

\( x \)

Constriants

\( \{ x = y \} \)

\( \text{lock.state(e)} = \text{L} \)
Example

\begin{verbatim}
void prove_me(int y)
{
    do {
        y = y;
        if (x) {
            unlock();
            y = y + 1;
        }
    } while (x > y);
    unlock();
}
\end{verbatim}

Proof!

Correct, the program is

Yogi’s solver interface

Representation
- \( L \)
  - program locations.
- \( R \subseteq L \times L \)
  - Control flow graph
- State: \( L \rightarrow \text{Formula set} \)
  - Symbolic state: each location has set of disjoint formulas

Theorem proving needs
- Facts about pointers:
  * \( *x = x \)
- Subsumption checks:
  \( \varphi \Rightarrow \text{WP}(l, \psi) \)
  \( \varphi \Rightarrow \neg \text{WP}(l, \psi) \)
- Structure sharing
  - Similar formulas in different states
- Simplification
  - Collapse/Reduce formulas

Better Bug Reporting with Better Privacy

Miguel Castro, Manuel Costa, Jean-Philippe Martin
ASPL08’08
See also: Vigilante – Internet Worm
Containment: Miguel Castro, Manuel Costa, Lintao Zhang

Finding the buffer overflow

Privacy: measure distance between original crash input and new input
Program Termination
Byron Cook

http://www.foment.net/byron/fsharp.shtml

A complete method for the synthesis of linear ranking functions. Podelski & Rybalchenko; VMCAI 04

Does this program Terminate?

\[ x > 0 \land y > 0 \land x' = x - 1 \land y' > y \]

```
while (x > 0 && y > 0) {
    x = x - 1;
    y = y + 1 + z*;
}
```

```
x > 0
x' \geq x - 1
x' \leq x - 1
y > 0
y' > y
```

\[ 0x' + 0y' + -1x + 0y + 1 \leq 0 \]
\[ 1x' + 0y' + -1x + 0y + 1 \leq 0 \]
\[ -1x' + 0y' + 1x + 0y + -1 \leq 0 \]
\[ 0x' + 0y' + 0x + -1y + 1 \leq 0 \]
\[ 0x' + -1y' + 0x + 1y + 1 \leq 0 \]

Rank function synthesis

\[ 0x' + 0y' + -1x + 0y + 1 \leq 0 \]
\[ 1x' + 0y' + -1x + 0y + 1 \leq 0 \]
\[ -1x' + 0y' + 1x + 0y + -1 \leq 0 \]
\[ 0x' + 0y' + 0x + -1y + 1 \leq 0 \]
\[ 0x' + -1y' + 0x + 1y + 1 \leq 0 \]

Can we find \( f, b \) such that the inclusion holds?

\[ \subseteq \]
\[ f(x, y) \geq f(x', y') \geq b \]

That is:

\[ f(x', y') + -f(x, y) + 1 \leq 0 \]
\[ -f(x', y') + b \leq 0 \]

Rank function synthesis

Search over linear templates:

\[ f(a, b) \triangleq c_1 a + c_2 b \]
\[ -f(a, b) \triangleq c_1 a + c_2 b \]
\[ c_1 = -1c_3 \]
\[ c_2 = -1c_4 \]

Find \( c_1, c_2, c_3, c_4 \)

Search over linear templates:

\[ f(a, b) \triangleq c_1 a + c_2 b \]
\[ -f(a, b) \triangleq c_1 a + c_2 b \]
\[ c_1 = -1c_3 \]
\[ c_2 = -1c_4 \]
Rank function synthesis

\[ \exists c_1, c_2, c_3, c_4, \forall x, y, x', y' \]

\[
\begin{align*}
0x' + 0y' + \text{1} - 1z' + 0y + 1 & \leq 0 \\
1x' + 0y' + \text{1} - 1z' + 0y + 1 & \leq 0 \\
-1x' + 0y' + 0z' + \text{1} - 1z' + 0y + 1 & \leq 0 \\
0x' + -1y' + 0z' + \text{1} - 1z' + 0y + 1 & \leq 0 \\
\end{align*}
\]

\[ \Rightarrow \]

Search over linear templates:

\[
\begin{align*}
f(a, b) & \triangleq c_1a + c_2b \\
f(a, b) & \triangleq c_3a + c_4b \\
c_1 & = -1c_3 \\
c_2 & = -1c_4 \\
\end{align*}
\]

Instead solve:

\[
\begin{align*}
a_1 + a_2 + a_3 + a_4 & = \lambda_1 \\
a_2 + a_3 + a_5 & = \lambda_2 \\
\end{align*}
\]

Farkas' lemma. \[ R \Rightarrow \psi \iff \text{there exist real multipliers } \lambda_1, \ldots, \lambda_5 \geq 0 \text{ such that } \]

\[
c_1 = \sum_{i=1}^{3} \lambda_i a_i, \quad \ldots, \quad c_4 = \sum_{i=4}^{5} \lambda_i a_i, \quad 1 \leq (\sum_{i=3}^{5} \lambda_i a_i)
\]

Solver: Dual Simplex for Th(LRA).

See Byron Cook's blog for an # program that produces input to Z3
How to find loop invariant \( I \)?

1. Assume \( I \) is of the form \( \sum a_i x_i \leq b \).
2. Simplified problem: \( \forall I, x, I \Rightarrow I(x) \)
   \[ \exists I \forall x (I(x) \land c(x) \land S(x, x') \Rightarrow I(x')) \]
   \[ \neg c(x) \land I(x) \Rightarrow Post(x) \]

- Assume \( I \) is of the form \( \sum a_i x_i \leq b \)
- Simplified problem: \( \forall I, x, I \Rightarrow I(x) \)

### Loop invariants ⇒ Existential

- Original: \( \exists I \forall x \varphi(I, x) \)
- Relaxed: \( \exists A, b \forall x \varphi(A, x, b) \)
- Farkas': \( \forall x (Ax \leq 0 \Rightarrow bx \leq 0) \)
  \[ \Rightarrow \exists \lambda, \lambda_1, \ldots, \lambda_m (b = \lambda + \sum \lambda_i a_i) \]
- Existential: Problem: contains multiplication
  \[ \exists A, b, \lambda \varphi_2(A, b, \lambda) \]

### Loop invariants ⇒ SMT solving

- Original: \( \exists I \forall x \varphi(I, x) \)
- Existential: \( \exists A, b \exists \lambda \varphi_2(A, b, \lambda) \)
- Bounded: \( \exists A, b, p_1, p_2, p_\varphi_2(A, b) \quad \text{[}	ext{ite}(p_1, A, 0) + \text{ite}(p_2, 2, 0) + \text{ite}(p_1, 1, 0)] \text{]}
- Or: Bit-vectors: \( \exists A, b, \lambda : \text{BitVec}[8], \varphi_2(A, b, \lambda) \)

### Bit-vector multiplication

- For each sub-term \( A \times B \)
  - Replace by fresh vector \( OUT \)
  - Create circuit for: \( OUT = A \times B \)
  - Convert circuit into clauses:
    - For each internal gate
      - Create fresh propositional variable
      - Represent gate as clause
    \[ (\text{Out}[0], \neg A[0], \neg B[0]), (A[0], \neg \text{Out}[0], \neg B[0]), \neg \text{Out}[0], \ldots \]

---

8/4/2008

© 2007 Microsoft Corporation. All rights reserved. Microsoft, Windows, Windows Vista and other product names are or may be registered trademarks and/or trademarks in the U.S. and/or other countries. The information herein is for informational purposes only and represents the current view of Microsoft Corporation as of the date of this presentation. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information provided after the date of this presentation.

MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS PRESENTATION.
Tableau goes outside in, DPLL inside out
Relevancy propagation: If DPLL sets \( \varphi \lor \psi \) to \text{true}, \( \theta \) is marked as relevant, then first of \( \varphi \), \( \psi \) to be set to \text{true} gets marked as relevant.
Used for circuit gates and for quantifier matching

**Tableau + DPLL = Relevancy Propagation**

**Digression: Bit-vectors and Z3**

Proper abstraction:
- Concrete reachable states: \( \text{CR: } L \rightarrow \wp(S) \)
- Abstract reachable states: \( \text{AR: } L \rightarrow A \)
- Connections:
  - \( \sqcup : A \times A \rightarrow A \)
  - \( \gamma : A \rightarrow \wp(S) \)
  - \( \alpha : S \rightarrow A \)
  - \( \alpha : \wp(S) \rightarrow A \) where \( \alpha(S) = \sqcup \{ \alpha(s) \mid s \in S \} \)

**Concrete reachable states:**
- \( \text{CR } \ell x \leftarrow \Theta x \land \ell = \ell_{\text{init}} \)
- \( \text{CR } \ell x \leftarrow \text{CR } \ell_0 x_0 \land R \ell_0 x_0 \land \ell \)

**Abstract reachable states:**
- \( \text{AR } \ell x \leftarrow \alpha(\Theta(x)) \land \ell = \ell_{\text{init}} \)
- \( \text{AR } \ell x \leftarrow \alpha(\gamma(\text{AR } \ell_0 x_0) \land R \ell_0 x_0 \land \ell) \)

**Why? Fewer (finite) abstract states**

**Abstract using SMT**

Abstract reachable states:
- \( \text{AR } \ell_{\text{init}} \leftarrow \alpha(\Theta) \)
Find interpretation \( M \):
- \( M = \gamma(\text{AR } \ell_0 x_0) \land R \ell_0 x_0 \land \ell \land \neg \gamma(\text{AR } \ell x) \)
Then:
- \( \text{AR } \ell \leftarrow \text{AR } \ell \sqcup \alpha(x^\wedge) \)
### Abstraction: Linear congruences

- States are linear congruences:
  \[ A \mathbf{V} = \mathbf{b} \mod 2^m \]
  - \( \mathbf{V} \) is set of program variables.
  - \( A \) matrix, \( \mathbf{b} \) vector of coefficients \([0..2^m-1]\)

### Example

\[ \ell_2: \ y \leftarrow x; \ c \leftarrow 0; \]
\[ \ell_1: \text{while } y \neq 0 \ do \ y \leftarrow y \& (y-1); \ c \leftarrow c + 1 \]  

- When at \( \ell_2 \):
  - \( y \) is 0.
  - \( c \) contains number of bits in \( x \).

### Abstraction: Linear congruences

- States are linear congruences:
  \[ \gamma \left[ \begin{array}{c} 2 \\ 1 \\ 1 \end{array} \right] \mathbf{x}_0 = \left[ \begin{array}{c} 1 \\ 3 \end{array} \right] \mod 2^1 \]  
  \[ 2x_0 + 3x_1 = 1 \mod 2^1 \land x_0 + x_1 = 3 \mod 2^1 \]  

As Bit-vector constraints (SMTish syntax):

\[
(\text{and} \ 
(= \ (	ext{bvadd} \ (\text{bvmul} \ 010 \ x_0) \ (\text{bvmul} \ 011 \ x_2)) \ 001) \\
(= \ (	ext{bvadd} \ x_0 \ x_2) \ 011)
)
\]

### Abstraction: Linear congruences

- \( a(x = 1, y = 2) \uplus \left[ \begin{array}{c} 1 \\ 0 \end{array} \right] \ x = \left[ \begin{array}{c} 1 \\ 2 \end{array} \right] \)  
- \( (A \mathbf{V} = \mathbf{b} \mod 2^m) \) \( \cup \) \( (A' \mathbf{V} = \mathbf{b}' \mod 2^m) \)

- Combine:
  \[ \begin{bmatrix}
  1 & 1 & 0 & 0 & 0 & 0 \\
  -b & 0 & A & 0 & 0 & 0 \\
  0 & -b' & 0 & A' & 0 & 0 \\
  0 & 0 & -I & -I & I & 0 \\
  0 & 0 & 0 & 0 & 0 & 0
  \end{bmatrix}
  \]

- Triangulate (Seidl & Olm)
- Project on \( x \)

### Goal: Model Based Development

Integration with symbolic analysis techniques at design time – smart model debugging
- Theorem proving
- Model checking
- Compositional reasoning
- Domain specific front ends
  - Different subareas require different adaptations
  - Model programs provide the common framework

**Motivating example**
- SMB2 Protocol Specification
- Sweet spot for model-based testing and verification.

Sample protocol document for SMB2 (a network file protocol)
Symbolic Reachability

Invisage Checker

Given a model program $P$ step bound $k$ and reachability condition $\varphi$

\[ \text{Reach}(P, \varphi, k) \triangleq I_P \land (\bigwedge_{a \leq c \leq k} P(a)) \land (\bigvee_{i \leq k} \varphi(i)) \]

\[ P(i) \triangleq \bigvee_{f \in \mathcal{F}} \left( \text{action}[i] = f(f_2[i], \ldots, f_n[i]) \land G^k[i][i] \right) \]

\[ \bigwedge_{v \in \mathcal{V}} v[i + 1] = v(i) \land \bigwedge_{v \in \mathcal{V} \setminus \mathcal{I}} v[i + 1] = v(i) \]

Array model programs and quantifier elimination

- Array model programs use only maps with integer domain sort.
- For normalizable comprehensions universal quantifiers can be eliminated using a decision procedure for the array property fragment [Bradley et. al, VMCAI 06]

Implementation using the SMT solver Z3

- Set comprehensions are introduced through skolem constant definitions using support for quantifiers in Z3
- Elimination of quantifiers is partial.
- Model is refined if a spurious model is found by Z3.
- A spurious model may be generated by Z3 if an incomplete heuristic is used during quantifier elimination.

A different example: Adaptive Planning with Finite Horizon Lookahead

Model program:

Verifying Garbage Collectors - Automatically and fast

Goal: safely run untrusted code

**Context**

Singularity
- Safe micro-kernel
- all services and drivers in processes
- Software isolated processes (SIPs)
- All untrusted code is verified safe
- Processes and kernels isolated at execution
- Communication via channels
- channel behavior is specified and checked
- in-kernel APIs
- Working research prototype
- not Windows replacement
- fast and efficient communication
- channel behavior is specified and checked
- processes and kernel sealed at execution
- some unsafe code in trusted runtime
- all user code is verifiably safe
- 95% written in C#

**Garbage Collectors**

Mark&Sweep
- MarkSweep
- Copying GC
- Verify small garbage collectors
- more automated than interactive provers
- automate ideas from type systems for regions

**Mark-and-and-copying collectors**

**Garbage collector properties**

- safety: gc does no harm
- type safety
- gc turns well-typed heap into well-typed heap
- graph isomorphism
- concrete graph represents abstract graph
- effectiveness
- after gc, unreachable objects reclaimed
- termination
- efficiency

**Proving safety**

**Controlling quantifier instantiation**

Idea: use marker

**Function:**

```
function (expand false) T(i:int) returns (bool) { true }
```

Relativize quantifiers using marker

function Gcinv(Color:int,int, $absAbs:int,int, $absAbsMem:int,int,int, Mem:int,int,int) returns (bool) {
  WellFormed($stAbs)
  & & (forall Int: $c(i) T(i) == memAdd(i) ==>
    Objinv(i, $stAbs, $absAbsMem, Mem)
    & & G <= Color(i) & & Color(i) < 4
    & & (Black(Color[i]) == True & & WhiteColor(Mem[i])) & & WhiteColor(Mem[i]))
  & & ($stAbs(i) == NO_ABS ==>
    Unalloc(Color[i]))
}
```

© 2007 Microsoft Corporation. All rights reserved. Microsoft, Windows, Windows Vista and other product names are or may be registered trademarks and/or trademarks in the U.S. and/or other countries.

The information herein is for informational purposes only and represents the current view of Microsoft Corporation as of the date of this presentation. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information provided after the date of this presentation.

MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS PRESENTATION.
Controlling quantifier instantiation

Insert markers to enable triggers

```plaintext
procedure Mark(ptr:int)
  requires GcIn(Color, $toAbs, $AbsMem, Mem);
  requires memAddr(ptr) & $Tp;
  requires $Stabs(ptr) != NO_ABS;
  modifies Color;
  ensures GcIn(Color, $toAbs, $AbsMem, Mem);
  ensures (forall i:int::{T(i) == Color[i] == old(Color)[i]});
  ensures !White(Color[ptr]);
  {
    if (White(Color[ptr])) {
      Color[ptr] := 2; // make gray
      call Mark(Mem[ptr,0]);
      call Mark(Mem[ptr,1]);
      Color[ptr] := 3; // make black
    }
  }
```

Refinement Types for Secure Implementations

http://research.microsoft.com/F7

Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, Sergio Maffeis
Csf 2008

Verifying protocol reference implementations

- Executable code has more details than models
- Executable code has better tool support: types, compilers, testing, debuggers, libraries, verification
- Using dependent types: integrate cryptographic protocol verification as a part of program verification
- Such predicates can also represent security-related concepts like roles, permissions, events, compromises, access rights, ...

Example: access control for files

- **Un-trusted code** may call a trusted library
- **Trusted code** expresses security policy with assumes and asserts
- Each policy violation causes an assertion failure
- F7 statically prevents any assertion failures by typing

```plaintext
type facts = CanRead of string | CanWrite of string
let read file = assert(CanRead(file)); ...
let delete file = assert(CanWrite(file)); ...
let pwd = "C:/etc/passwd"
let tmp = "C:/temp/temp"
assume CanWrite(mp)
assume \x. CanWrite(x) → CanRead(x)
let untrusted() = let v1 = read tmp in // ok
let v2 = read pwd in //CanRead(pwd) // assertion fails
```

Access control with refinement types

```plaintext
val read: file:string(CanRead(file)) → string
val delete: file:string(CanDelete(file)) → unit
val publish: file:string → unit(Public(file))
```

- Pre-conditions express access control requirements
- Post-conditions express results of validation
- F7 type checks partially trusted code to guarantee that all preconditions (and hence all asserts) hold at runtime

Models for Domain Specific Languages with FORMULA & BAM

Ethan Jackson

FORTE 08
Designing Complex Systems Requires Multiple Abstractions

Many Modeling Styles are Used to Build Abstractions

A Notorious Problem: How Do We Compose Abstractions?

Search for satisfying instances are Reduced to Z3

FORMULA is a CLP Language for Specifying, Composing, and Analyzing Abstractions

Selected Background on SMT
Pre-requisites and notation

Language: Signatures
- A signature Σ is a finite set of:
  - Function symbols:
    \( \Sigma_F = \{ f, g, \ldots \} \)
  - Predicate symbols:
    \( \Sigma_P = \{ P, Q, =, \text{true}, \text{false}, \ldots \} \)
  - And an arity function:
    \( \Sigma \rightarrow \mathbb{N} \)
  - Function symbols with arity 0 are constants
  - A countable set \( V \) of variables
  - disjoint from \( \Sigma \)

Language: Terms
- The set of terms \( T(\Sigma_F, V) \) is the smallest set formed by the syntax rules:
  \[
  t \in T \ ::= \begin{cases} v & v \in V \\ f(t_1, \ldots, t_n) & f \in \Sigma_F, t_1, \ldots, t_n \in T \end{cases}
  \]
- Ground terms are given by \( T(\Sigma_F, \emptyset) \)

Language: Atomic Formulas
- \( a \in \text{Atoms} \ ::= P(t_1, \ldots, t_n) \)
  \( P \in \Sigma_P \), \( t_1, \ldots, t_n \in T \)

An atom is ground if \( t_1, \ldots, t_n \in T(\Sigma_F, \emptyset) \)

Literals are (negated) atoms:
- \( l \in \text{Literals} \ ::= a \mid \neg a \)
  \( a \in \text{Atoms} \)

Language: Quantifier free formulas
- The set \( \text{QFF}(\Sigma, V) \) of quantifier free formulas is the smallest set such that:
  \[
  \varphi \in \text{QFF} \ ::= \begin{cases} a \in \text{Atoms} & \text{atoms} \\ \neg \varphi & \text{negations} \\ \varphi \leftrightarrow \varphi' & \text{bi-implications} \\ \varphi \land \varphi' & \text{conjunction} \\ \varphi \lor \varphi' & \text{disjunction} \\ \varphi \rightarrow \varphi' & \text{implication} \end{cases}
  \]
## Language: Formulas

- The set of **first-order formulas** are obtained by adding the formation rules:

\[
\varphi ::= \ldots \\
| \forall x . \varphi \quad \text{universal quant.} \\
| \exists x . \varphi \quad \text{existential quant.}
\]

- **Free** (occurrences) of variables in a formula are those not bound by a quantifier.
- A **sentence** is a first-order formula with no free variables.

## Models (Semantics)

- A model \( M \) is defined as:
  - Domain \( S \): set of elements.
  - Interpretation, \( f^M: S^n \rightarrow S \) for each \( f \in \Sigma_f \) with \( \text{arity}(f) = n \)
  - Interpretation \( P^M \subseteq S^n \) for each \( P \in \Sigma_P \) with \( \text{arity}(P) = n \)
  - Assignment \( \lambda^M \in S \) for every variable \( x \in V \)

- A formula \( \varphi \) is true in a model \( M \) if it evaluates to true under the given interpretations over the domain \( S \).

- \( M \) is a model for the theory \( T \) if all sentences of \( T \) are true in \( M \).

## T-Satisfiability

- A formula \( \varphi(x) \) is **T-satisfiable in a theory** \( T \) if there is a model of \( DC(T \cup \exists x \varphi(x)) \). That is, there is a model \( M \) for \( T \) in which \( \varphi(x) \) evaluates to true.

- Notation:

\[
M \models_T \varphi(x)
\]

## T-Validity

- A formula \( \varphi(x) \) is **T-valid in a theory** \( T \) if \( \forall x \varphi(x) \in T \). That is, \( \varphi(x) \) evaluates to true in every model \( M \) of \( T \).

- **T-validity**:

\[
\models_T \varphi(x)
\]

## Checking validity

- Checking the validity of \( \varphi \) in a theory \( T \):

\[
\varphi \text{ is T-valid} \\
\quad \Rightarrow T\text{-unsat: } \neg \varphi \\
\quad \Rightarrow T\text{-unsat: } \forall x \exists y \forall z \exists u . \varphi \quad \text{ (prenex of } \neg \varphi) \\
\quad \Rightarrow T\text{-unsat: } \forall x \forall z . \varphi[f(x),g(x,z)] \quad \text{ (skolemize)} \\
\quad \Leftarrow T\text{-unsat: } \varphi[f(a_1),g(a_1,b_1)] \land \ldots \quad \text{ (instatiate)} \\
\quad \quad \land \varphi[f(a_n),g(a_n,b_n)] \quad \Rightarrow \text{ if compactness} \\
\quad \Rightarrow T\text{-unsat: } \phi_1 \lor \ldots \lor \phi_m \quad \text{ (DNF)}
\]

where each \( \phi_i \) is a conjunction.
Checking Validity – the morale

- Theory solvers must minimally be able to
- check unsatisfiability of conjunctions of literals.

Clauses – CNF conversion

We want to only work with formulas in Conjunctive Normal Form CNF.

\[ \varphi : x = 5 \iff (y < 3 \lor z = x) \] is not in CNF.

Clauses – CNF conversion

\[ \text{cnf}(\varphi) = \text{let } (q, F) = \text{cnf'}(\varphi) \text{ in } q \land F \]
\[ \text{cnf'}(a) = (a, \text{true}) \]
\[ \text{cnf'}(\varphi \land \varphi') = \text{let } (q, F_1) = \text{cnf'}(\varphi) \]
\[ (r, F_2) = \text{cnf'}(\varphi') \]
\[ p = \text{fresh Boolean variable} \]
\[ \text{in } \]
\[ (p, F_1 \land F_2 \land (\neg p \lor q) \land (\neg p \lor r) \land (\neg p \lor \neg q \lor \neg r)) \]

Exercise: \text{cnf'}(\varphi \lor \varphi'), \text{cnf'}(\varphi \iff \varphi'), \text{cnf'}(\neg \varphi)

Clauses - CNF

- Main properties of basic CNF
  - Result \( F \) is a set of \textit{clauses}.
  - \( \varphi \) is \( T \)-satisfiable iff \( \text{cnf}(\varphi) \) is.
  - \( \text{size}(\text{cnf}(\varphi)) \leq 4(\text{size}(\varphi)) \)
  - \( \varphi \iff \exists p_{aux} \text{cnf}(\varphi) \)
**DPLL - classique**

- Incrementally build a model $M$ for a CNF formula $F$ (set of clauses).
- Initially $M$ is the empty assignment
- **Propagate** $M$: $M(r) \leftarrow \text{false}$ if $(p \lor q \lor \neg r) \in F$, $M(p) = \text{false}$, $M(q) = \text{true}$
- **Decide** $M(p) \leftarrow \text{true}$ or $M(p) \leftarrow \text{false}$, if $p$ is not assigned.
- **Backtrack**: $M(r) \leftarrow \text{false}$ if $(p \lor q \lor \neg r) \in F$, $M(p) = \text{false}$, $M(q) = M(r) = \text{true}$, (e.g. $M \models \neg C$)

**Modern DPLL – as transitions**

- **Conflict** $M \parallel F \Rightarrow M \parallel F \parallel C$ if $C \in F$, $M \models \neg C$
- **Learn** $M \parallel F \parallel C \Rightarrow M \parallel F \parallel C \parallel C$ i.e., add $C$ to $F$
- **Resolve** $M^p C \lor \neg p \parallel F \parallel C \lor \neg p \Rightarrow M \parallel F \parallel C \lor C'$
- **Skip** $M^p \parallel F \parallel C \Rightarrow M \parallel F \parallel C$ if $\neg l \in C$
- **Backjump** $M^l M' \parallel F \parallel C \Rightarrow M \parallel \neg C \parallel F$
  - if $\neg l \in C$ and $M'$ does not intersect with $\neg C$

**DPLL(E)**

- Congruence closure just checks satisfiability of conjunction of literals.
- How does this fit together with Boolean search DPLL?
- DPLL builds partial model $M$ incrementally
  - Use $M$ to build $C'$
  - After every **Decision** or **Propagate**, or
  - When $F$ is propositionally satisfied by $M$.
  - Check that disequalities are satisfied.

**Modern DPLL – as transitions**

- **Maintain states** of the form:
  - $M \parallel F$ - during search
  - $M \parallel F \parallel C$ – for backjumping
  - $M$ a partial model, $F$ are clauses, $C$ is a clause.
- **Decide** $M \parallel F \Rightarrow M^d \parallel F$ if $l \in F \setminus M$
  - $d$ is a decision marker
- **Propagate** $M \parallel F \Rightarrow M^C \parallel F$
  - if $l \in C \in F$, $C = (C' \lor l)$, $M \models \neg C'$

**E - conflicts**

Recall **Conflict**:

- **Conflict** $M \parallel F \Rightarrow M \parallel F \parallel C$ if $C \in F$, $M \models \neg C$

A version more useful for theories:

- **Conflict** $M \parallel F \Rightarrow M \parallel F \parallel C$ if $C \subseteq \neg M \models C$
**E - conflicts**

Example

- \( M = \text{fff}(a) = a, \ g(b) = c, \ \text{ffff}(a) = a, \ a \neq f(a) \)
- \( \rightarrow C = \text{fff}(a) = a, \ \text{ffff}(a) = a, \ a \neq f(a) \)
- \( \models \text{fff}(a) \neq a \lor \text{ffff}(a) \neq a \lor a = f(a) \)

Use \( C \) as a conflict clause.

**Approaches to linear arithmetic**

Fourier-Motzkin:
- Quantifier elimination procedure
  \[ \exists x (t \leq ax \land t' \leq bx \land c \leq t') \iff ct \leq at' \land ct' \leq bt' \]
- Polynomial for difference logic.
- Generally: exponential space, doubly exponential time.

Simplex:
- Worst-case exponential, but
- Time-tried practical efficiency.
- Linear space

**Nelson-Oppen procedure**

Initial state: \( L \) is set of literals over \( \Sigma_1 \cup \Sigma_2 \)

Purify: Preserving satisfiability, convert \( L \) into \( L' = L_1 \cup L_2 \) such that
\[ L_1 \in T(\Sigma_1, V), \ L_2 \in T(\Sigma_2, V) \]
So \( L_1 \cap L_2 = V_{\text{shared}} \subseteq V \)

Interaction:
- Guess a partition of \( V_{\text{shared}} \)
  - Express the partition as a conjunction of equalities.
  - Example: \( \{ x_1 \}, \{ x_2, x_3 \}, \{ x_4 \} \)
  - Represented as:
    \[ \psi : x_1 \neq x_2 \land x_1 \neq x_4 \land x_2 \neq x_3 \land x_3 = x_4 \]

Component Procedures:
- Use solver 1 to check satisfiability of \( L_1 \land \psi \)
- Use solver 2 to check satisfiability of \( L_2 \land \psi \)

**NO – reduced guessing**

- Instead of guessing, we can often deduce the equalities to be shared.

- Interaction: \( T_1 \land L_1 \models x = y \) then add equality to \( \psi \).

- If theories are convex, then we can:
  - Deduce all equalities.
  - Assume every thing not deduced is distinct.
- Complexity: \( O(n^4 \times T_1(n) \times T_2(n)) \).
Model-based combination

- Reduced guessing is only complete for convex theories.
- Deducing all implied equalities may be expensive.
  - Example: Simplex – no direct way to extract from just bounds and β
- But: backtracking is pretty cheap nowadays:
  - If β(x) = β(y), then x, y are equal in arithmetical component.
  - Backjumping is cheap with modern DPLL:
    - If β(x) = β(y), then x, y are equal in arithmetical model.
    - So let’s add x = y to ψ, but allow to backtrack from guess.
    - In general: if M₁ is the current model
      - M₁ ⊨ x = y then add literal (x = y)

Theory of arrays

- Functions: Σᵢ = { read, write }
- Predicates: Σᵢ = { = }
- Convention a[i] means: read(a, i)

- Non-extensional arrays Tᵢ:
  - ∀a, i, v . write(a, i, v)[i] = v
  - ∀a, i, j, v . i ≠ j ⇒ write(a, i, v)[j] = a[j]

- Extensional arrays: T_EA = Tᵢ +
  - ∀a, b. (∀i. a[i] = b[i]) ⇒ a = b

Decision procedures for arrays

- Let L be literals over Σᵢ = { read, write }
- Find M such that: M ⊨ Tᵢ L

- Basic algorithm, reduce to E:
  - for every sub-term read(a, i), write(b, j, v) in L
    - i ≠ j ∧ a = b ⇒ read(write(b, j, v), i) = read(a, i)
    - read(write(b, j, v), j) = v
  - Find M_E such that
    - M_E ⊨ L ∧ AssertedAxioms

Quantifiers and E-graph matching
**DPLL(QT) – cute quantifiers**

- We can use DPLL(T) for $\varphi$ with quantifiers.
- Treat quantified sub-formulas as atomic predicates.
- In other words, if $\forall x. \psi(x)$ is a sub-formula of $\varphi$, then introduce fresh $p$. Solve instead
  
  $\varphi[\forall x. \psi(x) \leftarrow p]$

**DPLL(QT)**

- Suppose DPLL(T) sets $p$ to false
  
  - $\Rightarrow$ any model $M$ for $\varphi$ must satisfy:
    
    $M \models \neg \forall x. \psi(x)$
  
  - $\Rightarrow$ for some $sk_x$:
    
    $M \models \neg \psi(sk_x)$
  
  - In general:
    
    $\models \neg p \rightarrow \neg \psi(sk_x)$

**DPLL(QT)**

- Suppose DPLL(T) sets $p$ to true
  
  - $\Rightarrow$ any model $M$ for $\varphi$ must satisfy:
    
    $M \models \forall x. \psi(x)$
  
  - $\Rightarrow$ for every term $t$:
    
    $M \models \psi(t)$
  
  - In general:
    
    $\models p \rightarrow \psi(t)$
  
  For every term $t$.

**DPLL(QT) with E-matching**

- $\models p \rightarrow \psi(t)$ For every term $t$.
  
  Approach:
  
  - Add patterns to quantifiers
  - Search for instantiations in $E$-graph.

  $\forall a,i,v \{ \text{write}(a,i,v) \}. \text{read}(\text{write}(a,i,v),i) = v$

**DPLL(QT) with E-matching**

- $\models p \rightarrow \psi(t)$ For every term $t$.
  
  Approach:
  
  - Add patterns to quantifiers
  - Search for pattern matches in $E$-graph.

  $\forall a,i,v \{ \text{write}(a,i,v) \}. \text{read}(\text{write}(a,i,v),i) = v$

  - Add equality every time there is a write(b,j,w) term in $E$. 
Z3 - An Efficient SMT Solver

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)

Supporting material

Example: C API
```c
for (n = 2; n <= 5; n++)
    printf("%d
", n);
bool_type = Z3_mk_bool_type(ctx);
array_type = Z3_mk_array_type(ctx, bool_type, bool_type);
/* create arrays */
for (i = 0; i < n; i++)
    Z3_mk_symbol( ctx, i);
all = Z3_mk_const( ctx, n, array_type);
/* assert distinct a[0], ..., a[n-1] */
d = Z3_mk_distinct( ctx, n, all);
printf("\n\n Sorry, a[0] == a[%d]
", i);
/* assert in unsat */
if (Z3_check(ctx, m) == Z3_TRUE)
    printf("\n\n Sorry, \n\n Z3 returned unsat.
\n");
Z3_del_context(ctx);
```

Theories
- Core Theory
- SAT solver
- Bit-Vectors
- Arithmetic
- Arrays
- Partial orders
- Tuples

Core System Components
- Text
- C
- .NET
- OCaml
- Rewriting
- Simplification
- E-matching
- Core Theory
- SAT solver

Given arrays:
- bool a1[bool];
- bool a2[bool];
- bool a3[bool];
- bool a4[bool];

All can be distinct.

Add:
- bool a5[bool];

Two of a1...a5 must be equal.
Example: SMT-LIB

```
benchmark integer-linear-arithmetic
  status sat
  logic QF_LIA
  extrasorts (x1 int) (x2 int) (x3 int)
  (x4 int) (x5 int)
  (x6 int)
  (y int) (z int)
  (x1 (* x1 x2) 1)
  (= x1 (* x1 x2) 1)
  (= x1 x2)
  (= x2 x3)
  (= x2 (* 6 x3))

benchmark array
  logic QF_AUFLIA
  status sat
  extrasorts (a Array) (b Array) (c Array)
  extrasorts (l int) (j int)
  formula (and
    (= (store a i) v)
    (= (store a j) w)
    (select b j) w
    (= select c i) v
    (not (= b c))
    formula
  )

SMT-LIB syntax – basics

- **Logics:**
  - QF_UF – Un-interpreted functions. Built-in sort \( U \).
  - QF_AUFLIA – Arrays and Integer linear arithmetic.

- **Built-in Sorts:**
  - Int, Array (of Int to Int)

- **Built-in Predicates:**
  - \( <, >, \leq, \geq \).

- **Built-in Functions:**
  - +, *, select, store.

- **Constants:** 0, 1, 2, ...

```

SMT-LIB – encodings

Q: There is no built-in function for `max` or `min`. How do I encode it?

- `(max x y)` is the same as `(ite (> x y) x y)`

- Also: replace `(max x y)` by fresh constant `max_x_y` add assumptions:
  - assumption (implies (`>` x y) (\( \text{max}_x_y \))
  - assumption (implies (`<=$ x y) (\( \text{max}_x_y \))

Q: Encode the predicate (even \( n \)), that is true when \( n \) is even.

```

Quantifiers

Quantified formulas in SMT-LIB:

```
- `fmla ::= (forall bound* fmla) | (exists bound* fmla)`

- `Bound ::= (id sort-id)`

- Q: I want \( f \) to be an injective function. Write an axiom that forces \( f \) to be injective.

- Patterns: guiding the instantiation of quantifiers (Lecture 5)

```

```
Using the Z3 (managed) API

```

```

© 2007 Microsoft Corporation. All rights reserved. Microsoft, Windows, Windows Vista and other product names are or may be registered trademarks and/or trademarks in the U.S. and/or other countries. The information herein is for informational purposes only and represents the current view of Microsoft Corporation as of the date of this presentation. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information provided after the date of this presentation.

MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS PRESENTATION.
Using the Z3 (managed) API

```
let (=>) x y = z3.MkEq(x,y)
let (==>) x y = z3.MkImplies(x,y)
let && x y = z3.MkAnd(x,y)
let f x = z3.MkApp(f_decl,x)
let a = z3.MkType("a")
let f_decl = z3.MkFuncDecl("f",a,a)
let x = z3.MkConst("x",a)
let neg x = z3.MkNot(x)

let fmla1 = ((x == f(f(f(f(f(f x)))))) && (x == f(f(f x)))) ==> (x == (f x))
do check (neg fmla1)
```

Declaring Z3 shortcuts, constants and functions

Proving a theorem

```
let (===) x y = z3.MkEq(x,y)
let (==>) x y = z3.MkImplies(x,y)
let && x y = z3.MkAnd(x,y)
let neg x = z3.MkNot(x)
let a = z3.MkType("a")
let f_decl = z3.MkFuncDecl("f",a,a)
let x = z3.MkConst("x",a)
let f x = z3.MkApp(f_decl,x)
```

Enumerating models

```
Enumerator:
```

Representing the problem

```
enum void Test() {
    Config par = new Config();
    par.SetParamValue("MODEL", "true");
    z3 = new TypeSafeContext(par);
    intT = z3.MkIntType();
    i1 = z3.MkConst("i1", intT);
    i2 = z3.MkConst("i2", intT);
    i3 = z3.MkConst("i3", intT);
    z3.AssertCnstr(Num(2) < i1 & i1 <= Num(5));
    z3.AssertCnstr(Num(1) < i2 & i2 <= Num(7));
    z3.AssertCnstr(Num(-1) < i3 & i3 <= Num(17));
    z3.AssertCnstr(Num(0) <= i1 + i2 + i3 & Eq(i2 + i3, i1));
    z3.AssertCnstr(!block);
    model.Dispose();
}
```

Maximize:

```
int Maximize(TermAst a, int lo, int hi) {
    while (lo < hi) {
        int mid = (lo+hi)/2;
        Console.WriteLine("lo: {0}, hi: {1}, mid: {2}", lo, hi, mid);
        z3.Push();
        z3.AssertCnstr(Num(mid+1) <= a & a <= Num(hi));
        TypeSafeModel model = null;
        if (LBool.True == z3.CheckAndGetModel(ref model)) {
            lo = model.GetNumericalValueInt(model.Eval(a));
            model.Dispose();
        }
        else hi = mid;
    }
    return hi;
}
```

Push, Pop

```
enum Push, Pop
```

However, $i$ is not unique. To see why, we may look at the following equation:

```
2 < i_1 <= 5     
1 < i_2 <= 7     
-1 < i_3 <= 17    
0 <= i_1 + i_2 + i_3 
\quad \land  
\quad i_2 + i_3 = i_1
```

But we only care about different $i_j$.

```
enum void Enumerate() {
    TypeSafeModel model = null;
    while (LBool.True == z3.CheckAndGetModel(ref model)) {
        model.Display(Console.Out);
        int v1 = model.GetNumeralValueInt(model.Eval(i1));
        TermAst block = Eq(Num(v1), i1);
        Console.WriteLine("Block {0}", block);
        z3.AssertCnstr(!block);
        model.Dispose();
    }
}
```

```
enum int Maximize(TermAst a, int lo, int hi) {
    while (lo < hi) {
        int mid = (lo+hi)/2;
        Console.WriteLine("lo: {0}, hi: {1}, mid: {2}", lo, hi, mid);
        z3.Push();
        z3.AssertCnstr(Num(mid+1) <= a & a <= Num(hi));
        TypeSafeModel model = null;
        if (LBool.True == z3.CheckAndGetModel(ref model)) {
            lo = model.GetNumericalValueInt(model.Eval(a));
            model.Dispose();
        } else hi = mid;
    }
    return hi;
}
```

Push, Pop – but reuse search

```
enum int Maximize(TermAst a, int lo, int hi) {
    while (lo < hi) {
        int mid = (lo+hi)/2;
        Console.WriteLine("lo: {0}, hi: {1}, mid: {2}", lo, hi, mid);
        z3.Push();
        z3.AssertCnstr(Num(mid+1) <= a & a <= Num(hi));
        TypeSafeModel model = null;
        if (LBool.True == z3.CheckAndGetModel(ref model)) {
            lo = model.GetNumericalValueInt(model.Eval(a));
            model.Dispose();
        } else hi = mid;
    }
    return hi;
}
```