SMT@Microsoft
Midwest Verification Day, Iowa, 2009

Leonardo de Moura
Microsoft Research
Verification/Analysis tools need some form of Symbolic Reasoning
Logic is “The Calculus of Computer Science” (Z. Manna).

High computational complexity
Applications

- Test case generation
- Verifying Compilers
- Predicate Abstraction
- Invariant Generation
- Type Checking
- Model Based Testing
Some Applications @ Microsoft

- The Spec# Programming System
- HAVOC
- Hyper-V Virtualization
- Terminator T-2
- VCC
- NModel
- SpecExplorer
- SAGE
- Vigilante
- F7

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

We want a trace where the loop is executed twice.

(y₀ > 0) and
(m₀ = x₀ % y₀) and
not (m₀ = 0) and
(x₁ = y₀) and
(y₁ = m₀) and
(m₁ = x₁ % y₁) and
(m₁ = 0)

x₀ = 2
y₀ = 4
m₀ = 2
x₁ = 4
y₁ = 2
m₁ = 0

SMT@Microsoft
Type checking

Signature:

\[ \text{div} : \text{int}, \{ x : \text{int} \mid x \neq 0 \} \rightarrow \text{int} \]

Call site:

\[
\text{if } a \leq 1 \text{ and } a \leq b \text{ then }
\text{return div}(a, b)
\]

Verification condition

\[ a \leq 1 \text{ and } a \leq b \text{ implies } b \neq 0 \]
Is formula $F$ satisfiable modulo theory $T$?

SMT solvers have specialized algorithms for $T$
Satisfiability Modulo Theories (SMT)

$b + 2 = c \text{ and } f(\text{read}(\text{write}(a,b,3), c-2)) \neq f(c-b+1)$
\[ b + 2 = c \quad \text{and} \quad f(\text{read}(\text{write}(a,b,3), c-2)) \neq f(c-b+1) \]
Satisfiability Modulo Theories (SMT)

\[ b + 2 = c \text{ and } f(\text{read}(\text{write}(a, b, 3), c-2)) \neq f(c - b + 1) \]
$b + 2 = c$ and $f(\text{read(\text{write}(a,b,3), c-2)}) \neq f(c-b+1)$
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
For most SMT solvers: \( F \) is a set of ground formulas

Many Applications

- Bounded Model Checking
- Test-Case Generation
An SMT Solver is a collection of Little Engines of Proof
Deciding Equality

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e, \ a \neq s \]
Deciding Equality

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e, \ a \neq s \]
Deciding Equality

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e, \ a \neq s \]
Deciding Equality

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e, \ a \neq s \]
Deciding Equality

\[ a = b, \quad b = c, \quad d = e, \quad b = s, \quad d = t, \quad a \neq e, \quad a \neq s \]
Deciding Equality

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e, \ a \neq s \]
Deciding Equality

\[ a = b, \quad b = c, \quad d = e, \quad b = s, \quad d = t, \quad a \neq e, \quad a \neq s \]
Deciding Equality

\[a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e, \ a \neq s\]
Deciding Equality

\[a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e, \ a \neq s\]
Deciding Equality

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e, \ a \neq s \]
Deciding Equality

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e, \ a \neq s \]
Deciding Equality

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e, \ a \neq s \]
Deciding Equality

\[ a = b, \; b = c, \; d = e, \; b = s, \; d = t, \; a \neq e, \; a \neq s \]
Deciding Equality

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e \]

Model
\[ |M| = \{ 0, 1 \} \]
\[ M(a) = M(b) = M(c) = M(s) = 0 \]
\[ M(d) = M(e) = M(t) = 1 \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ f(a, g(d)) \neq f(b, g(e)) \]

Congruence Rule:

\[ x_1 = y_1, \ldots, x_n = y_n \text{ implies } f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ f(a, g(d)) \neq f(b, g(e)) \]

Congruence Rule:

\[ x_1 = y_1, \ldots, \ x_n = y_n \text{ implies } f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ f(a, g(d)) \neq f(b, g(e)) \]

Congruence Rule:
\[ x_1 = y_1, \ldots, x_n = y_n \text{ implies } f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ f(a, g(d)) \neq f(b, g(e)) \]

Congruence Rule:

\[ x_1 = y_1, \ldots, x_n = y_n \implies f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ f(a, g(d)) \neq f(b, g(e)) \]

Congruence Rule:

\[ x_1 = y_1, \ ... , \ x_n = y_n \text{ implies } f(x_1, ..., x_n) = f(y_1, ..., y_n) \]
Deciding Equality + (uninterpreted) Functions

\[ a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ f(a, g(d)) \neq f(b, g(e)) \]

unsatisfiable
(fully shared) DAGs for representing terms
Union-find data-structure + Congruence Closure
O(n log n)
In practice, we need a combination of theory solvers.

Nelson-Oppen combination method.
Reduction techniques.
Model-based theory combination.
SAT (propositional checkers): Case Analysis

\[ p \lor q, \]
\[ p \lor \neg q, \]
\[ \neg p \lor q, \]
\[ \neg p \lor \neg q \]
SAT (propositional checkers): Case Analysis

\[ p \lor q, \]
\[ p \lor \neg q, \]
\[ \neg p \lor q, \]
\[ \neg p \lor \neg q \]

Assignment:
\[ p = false, \]
\[ q = false \]
SAT (propositional checkers): Case Analysis

\[ p \lor q, \]
\[ p \lor \neg q, \]
\[ \neg p \lor q, \]
\[ \neg p \lor \neg q \]

Assignment:
\[ p = \text{false}, \]
\[ q = \text{true} \]
SAT (propositional checkers): Case Analysis

Assignment:

\[ p = \text{true,}\]
\[ q = \text{false}\]
SAT (propositional checkers): Case Analysis

Assignment:
p = true,
q = true
DPLL

Partial model

Set of clauses

M | F
Guessing

\[ p \mid p \lor q, \neg q \lor r \]

\[ p, \neg q \mid p \lor q, \neg q \lor r \]
DPLLL

Deducing

\[ p \mid p \lor q, \neg p \lor s \]

\[ p, s \mid p \lor q, \neg p \lor s \]
Backtracking

\[ p, \neg s, q \mid p \lor q, s \lor q, \neg p \lor \neg q \]

\[ p, s \mid p \lor q, s \lor q, \neg p \lor \neg q \]
Modern DPLL

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

SMT@Microsoft
Efficient decision procedures for conjunctions of ground literals.

\[ a=b, \ a<5 \ | \ \neg a=b \lor f(a)=f(b), \ a < 5 \lor a > 10 \]
Theory Conflicts

\[ a = b, \ a > 0, \ c > 0, \ a + c < 0 \mid F \]

backtrack
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

\[ a = b, \ a < 5 \quad \text{|} \quad \ldots \quad a < 6 \quad \lor \quad f(a) = a \]

\[ a = b, \ a < 5, \ a < 6 \quad \text{|} \quad \ldots \quad a < 6 \quad \lor \quad f(a) = a \]
Decision Procedures must be:

- Incremental & Backtracking
- Theory Propagation
- Precise (theory) lemma learning

Given:
\[ a = b, \ a > 0, \ c > 0, \ a + c < 0 \ | \ F \]

Learn clause:
\[ \neg (a = b) \lor \neg (a > 0) \lor \neg (c > 0) \lor \neg (a + c < 0) \]

Imprecise!

Precise clause:
\[ \neg (a > 0) \lor \neg (c > 0) \lor \neg (a + c < 0) \]
For some theories, SMT can be reduced to SAT

Higher level of abstraction

\[ \text{bvmul}_{32}(a,b) = \text{bvmul}_{32}(b,a) \]
SMT x First-order provers

$F \cup T \rightarrow$ First-order Theorem Prover

$T$ may not have a finite axiomatization
SMT: Some Applications

- Test case generation
- Verifying Compiler
- Predicate Abstraction

Z3
SMT: Some Applications

- Verifying Compiler
- Predicate Abstraction
- Test case generation
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
Security is critical

- Security bugs can be very expensive:
  - Cost of each MS Security Bulletin: $600k to $Millions.
  - Cost due to worms: $Billions.
  - The real victim is the customer.

- Most security exploits are initiated via files or packets.
  - Ex: Internet Explorer parses dozens of file formats.

- Security testing: **hunting for million dollar bugs**
  - Write A/V
  - Read A/V
  - Null pointer dereference
  - Division by zero
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
Directed Automated Random Testing (DART)

- Run Test and Monitor
- Execution Path
- Path Condition
- Test Inputs
- Known Paths
- Constraint System
- Solve

Z3

New input
seed
PEX: Implements DART for .NET.

SAGE: Implements DART for x86 binaries.

YOGI: Implements DART to check the feasibility of program paths generated statically using a SLAM-like tool.

Vigilante: Partially implements DART to dynamically generate worm filters.
What is Pex?

Test input generator

- Pex starts from parameterized unit tests
- Generated tests are emitted as traditional unit tests
ArrayList: The Spec

ArrayList.Add Method

- Adds an object to the end of the ArrayList.
- **Namespace:** System.Collections
- **Assembly:** mscorlib (in mscorlib.dll)

**Remarks**

ArrayList accepts a null reference (Nothing in Visual Basic) as a valid value and allows duplicate elements.

- If `Count` already equals `Capacity`, the capacity of the ArrayList is increased by automatically reallocating the internal array, and the existing elements are copied to the new array before the new element is added.

- If `Count` is less than `Capacity`, this method is an O(1) operation. If the capacity needs to be increased to accommodate the new element, this method becomes an O(n) operation, where n is `Count`.

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

class ArrayList {
    object[] items;
    int count;

    ArrayList(int capacity) {
        if (capacity < 0) throw ...
        items = new object[capacity];
    }

    void Add(object item) {
        if (count == items.Length)
            ResizeArray();

        items[this.count++] = item;
    }
    ...

.LayoutFrame dll .NET Framework Class Library
Assembly:mscorlib (in mscorlib.dll)
class ArrayList {
    object[] items;
    int count;

    ArrayList(int capacity) {
        if (capacity < 0) throw ...;
        items = new object[capacity];
    }

    void Add(object item) {
        if (count == items.Length)
            ResizeArray();
        items[this.count++] = item;
    }

    ...
class ArrayList {
    object[] items;
    int count;

    ArrayList(int capacity) {
        if (capacity < 0) throw ...
        items = new object[capacity];
    }

    void Add(object item) {
        if (count == items.Length)
            ResizeArray();
        items[this.count++] = item;
    }
    ...
}

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

class ArrayList {
    object[] items;
    int count;

    ArrayList(int capacity) {
        if (capacity < 0) throw ...;
        items = new object[capacity];
    }

    void Add(object item) {
        if (count == items.Length)
            ResizeArray();
        items[this.count++] = item;
    }
...
class ArrayListTest {
[PexMethod]
void AddItem(int c, object item) {
    var list = new ArrayList(c);
    list.AddItem(item);
    Assert(list[0] == item); }
}

class ArrayList {
    object[] items;
    int count;

    ArrayList(int capacity) {
        if (capacity < 0) throw ...;
        items = new object[capacity];
    }

    void Add(object item) {
        if (count == items.Length) 
            ResizeArray();
        items[this.count++] = item; }
...
### ArrayList: Run 1, (0,null)

**Inputs** | **Observed Constraints**
---|---
(0,null) | !(c<0) && 0==c

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

```csharp
class ArrayList {
    object[] items;
    int count;

    ArrayList(int capacity) {
        if (capacity < 0) throw ...
        items = new object[capacity];
    }

    void Add(object item) {
        if (count == items.Length)
            ResizeArray();
        items[this.count++] = item;
    }
    ...
```
class ArrayListTest {
[PexMethod]
void AddItem(int c, object item) {
    var list = new ArrayList(c);
    list.Add(item);
    Assert(list[0] == item); }
}

class ArrayList {
    object[] items;
    int count;

    ArrayList(int capacity) {
        if (capacity < 0) throw ...;
        items = new object[capacity];
    }

    void Add(object item) {
        if (count == items.Length)
            ResizeArray();

        items[this.count++] = item; }

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

class ArrayList {
    object[] items;
    int count;

    ArrayList(int capacity) {
        if (capacity < 0) throw ...;
        items = new object[capacity];
    }

    void Add(object item) {
        if (count == items.Length)
            ResizeArray();
        items[this.count++] = item;
    }
    ...
}

<table>
<thead>
<tr>
<th>Constraints to solve</th>
<th>Inputs</th>
<th>Observed Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, null)</td>
<td>!(c&lt;0) &amp;&amp; 0==c</td>
<td></td>
</tr>
<tr>
<td>!(c&lt;0) &amp;&amp; 0!=c</td>
<td>(1, null)</td>
<td></td>
</tr>
</tbody>
</table>
class ArrayListTest {
[PexMethod]
void AddItem(int c, object item) {
    var list = new ArrayList(c);
    list.AddItem(item);
    Assert(list[0] == item);
}
}

class ArrayList {
    object[] items;
    int count;

    ArrayList(int capacity) {
        if (capacity < 0) throw ...;
        items = new object[capacity];
    }

    void Add(object item) {
        if (count == items.Length) 
            ResizeArray();
        items[this.count++] = item;
    }

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

class ArrayList {
    object[] items;
    int count;

    ArrayList(int capacity) {
        if (capacity < 0) throw ...;
        items = new object[capacity];
    }

    void Add(object item) {
        if (count == items.Length)
            ResizeArray();

        items[this.count++] = item;
    }

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

class ArrayList {
    object[] items;
    int count;

    ArrayList(int capacity) {
        if (capacity < 0) throw ...;
        items = new object[capacity];
    }

    void Add(object item) {
        if (count == items.Length)
            ResizeArray();

        items[this.count++] = item;
    }
...

<table>
<thead>
<tr>
<th>Constraints to solve</th>
<th>Inputs</th>
<th>Observed Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, null)</td>
<td>!(c&lt;0) &amp;&amp; 0==c</td>
<td></td>
</tr>
<tr>
<td>!(c&lt;0) &amp;&amp; 0!=c</td>
<td>(1, null)</td>
<td>!(c&lt;0) &amp;&amp; 0!=c</td>
</tr>
<tr>
<td>c&lt;0</td>
<td>(-1, null)</td>
<td></td>
</tr>
</tbody>
</table>

Z3
class ArrayListTest {  
[PexMethod]  
void AddItem(int c, object item) {  
    var list = new ArrayList(c);  
    list.Add(item);  
    Assert(list[0] == item); }  
}

class ArrayList {  
object[] items;  
int count;  
ArrayList(int capacity) {  
    if (capacity < 0) throw ...;  
    items = new object[capacity];  
}  
void Add(object item) {  
    if (count == items.Length)  
        ResizeArray();  
    items[this.count++] = item; }  
...

<table>
<thead>
<tr>
<th>Constraints to solve</th>
<th>Inputs</th>
<th>Observed Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,null)</td>
<td>!(c&lt;0) &amp;&amp; 0==c</td>
<td></td>
</tr>
<tr>
<td>!(c&lt;0) &amp;&amp; 0!=c</td>
<td>(1,null)</td>
<td>!(c&lt;0) &amp;&amp; 0!=c</td>
</tr>
<tr>
<td>c&lt;0</td>
<td>(-1,null)</td>
<td>c&lt;0</td>
</tr>
</tbody>
</table>

\[ c < 0 \rightarrow \text{true} \]
class ArrayListTest {
    [PexMethod]
    void AddItem(int c, object item) {
        var list = new ArrayList(c);
        list.Add(item);
        Assert(list[0] == item);
    }
}

class ArrayList {
    object[] items;
    int count;

    ArrayList(int capacity) {
        if (capacity < 0) throw ...;
        items = new object[capacity];
    }

    void Add(object item) {
        if (count == items.Length)
            ResizeArray();

        items[this.count++] = item;
    }

    ...
Rich Combination

Linear arithmetic

Bitvector

Arrays

Free Functions

Models

Model used as test inputs

∀-Quantifier

Used to model custom theories (e.g., .NET type system)

API

Huge number of small problems. Textual interface is too inefficient.
PEX ↔ Z3: Incrementality

- Pex “sends” several similar formulas to Z3.
- Plus: backtracking primitives in the Z3 API.
  - push
  - pop
- Reuse (some) lemmas.
**PEX ↔ Z3: Small models**

- **Given** a set of constraints $C$, find a model $M$ that minimizes the interpretation for $x_0, \ldots, x_n$.

- In the ArrayList example:
  
  Why is the model where $c = 2147483648$ less desirable than the model with $c = 1$?
  
  $$!(c < 0) \land 0 != c$$

- **Simple solution:**
  
  Assert $C$
  
  while satisfiable
  
  Peek $x_i$ such that $M[x_i]$ is big
  
  Assert $x_i < n$, where $n$ is a small constant

  Return last found model
**PEX ↔ Z3: Small models**

- **Given** a set of constraints $C$, find a model $M$ that **minimizes** the interpretation for $x_0, \ldots, x_n$.

- In the ArrayList example:
  - Why is the model where $c = 2147483648$ less desirable than the model with $c = 1$?
    
    $$!(c<0) \land 0! = c$$

- **Refinement:**
  - Eager solution stops as soon as the system becomes unsatisfiable.
  - A “bad” choice (peek $x_i$) may prevent us from finding a good solution.
  - Use **push** and **pop** to retract “bad” choices.
Apply DART to large applications (not units).
Start with well-formed input (not random).
Combine with generational search (not DFS).
- Negate 1-by-1 each constraint in a path constraint.
- Generate many children for each parent run.

SAGE
Zero to Crash in 10 Generations

- Starting with 100 zero bytes ...
- SAGE generates a crashing test for Media1 parser

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex Value</th>
<th>Binary Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000h</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>; ................</td>
</tr>
<tr>
<td>00000010h</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>; ................</td>
</tr>
<tr>
<td>00000020h</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>; ................</td>
</tr>
<tr>
<td>00000030h</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>; ................</td>
</tr>
<tr>
<td>00000040h</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>; ................</td>
</tr>
<tr>
<td>00000050h</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>; ................</td>
</tr>
<tr>
<td>00000060h</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>; ....</td>
</tr>
</tbody>
</table>

Generation 0 – seed file
Starting with 100 zero bytes ...

SAGE generates a crashing test for Media1 parser
Zero to Crash in 10 Generations

- Starting with 100 zero bytes ...
- SAGE generates a crashing test for Media1 parser

```
00000000h: 52 49 46 46 00 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF....*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00
```

Generation 2
Starting with 100 zero bytes ...

SAGE generates a crashing test for Media1 parser

Generation 3

| 00000000h | 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 00 ; RIFF...*** ..... |
| 00000010h | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................ |
| 00000020h | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................ |
| 00000030h | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................ |
| 00000040h | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................ |
| 00000050h | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................ |
| 00000060h | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................ |
|          | ; ....               |
Zero to Crash in 10 Generations

- Starting with 100 zero bytes ...
- SAGE generates a crashing test for Media1 parser

```
00000000h:  52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h:  00 00 00 00 73 74 72 68 00 00 00 00 00 00 00 00 ; ....strh........
00000040h:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h:  00 00 00 00
```

Generation 4
Starting with 100 zero bytes …

SAGE generates a crashing test for Media1 parser

Generation 5

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 00 76 69 64 73 ; ....strh... vids
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00 ; ....
```
Starting with 100 zero bytes ...

SAGE generates a crashing test for Media1 parser

Generation 6

<table>
<thead>
<tr>
<th>Address</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000h</td>
<td>52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 00 ; RIFF=...*** .....</td>
</tr>
<tr>
<td>00000010h</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ...................</td>
</tr>
<tr>
<td>00000020h</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ...................</td>
</tr>
<tr>
<td>00000030h</td>
<td>00 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; .....strh....vids</td>
</tr>
<tr>
<td>00000040h</td>
<td>00 00 00 00 00 73 74 72 66 00 00 00 00 00 00 00 00 ; .....strf........</td>
</tr>
<tr>
<td>00000050h</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ...................</td>
</tr>
<tr>
<td>00000060h</td>
<td>00 00 00 00</td>
</tr>
</tbody>
</table>

Generation 6
Zero to Crash in 10 Generations

- Starting with 100 zero bytes ...
- SAGE generates a crashing test for Media1 parser

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 00 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 00 00 00 00 28 00 00 00 ; ....strf....[
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00 ; ....
```

Generation 7
Zero to Crash in 10 Generations

Starting with 100 zero bytes ...

SAGE generates a crashing test for Media1 parser

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...***.....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 00 00 00 00 28 00 00 00 ; ....strf....(...
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 C9 9D E4 4E ; ............
00000060h: 00 00 00 00
```

Generation 8
Starting with 100 zero bytes ... 
SAGE generates a crashing test for Media1 parser

Generation 9
Zero to Crash in 10 Generations

- Starting with 100 zero bytes ...
- SAGE generates a crashing test for Media1 parser

| 00000000h: | 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** .... |
| 00000010h: | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................ |
| 00000020h: | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................ |
| 00000030h: | 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids |
| 00000040h: | 00 00 00 00 73 74 72 66 B2 75 76 3A 28 00 00 00 ; ....str²uv:(...) |
| 00000050h: | 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 ; ................ |
| 00000060h: | 00 00 00 00 ; .... |

Generation 10 – CRASH
SAGE is very effective at finding bugs.
Works on large applications.
Fully automated
Easy to deploy (x86 analysis – any language)
Used in various groups inside Microsoft
Powered by Z3.
Formulas are usually big conjunctions.

SAGE uses only the bitvector and array theories.

Pre-processing step has a huge performance impact.
- Eliminate variables.
- Simplify formulas.

Early unsat detection.
SMT: Some Applications

- Verifying Compiler
- Test case generation
- Predicate Abstraction

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
Verification architecture

Spec#

Spec# compiler

MSIL

Bytecode translator

Boogie

V.C. generator

Verification condition

Z3

VCC

HAVOC

C

C

Static program verifier (Boogie)
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.
VCC translates an annotated C program into a Boogie PL program.

A C-ish memory model
- Abstract heaps
- Bit-level precision

Microsoft Hypervisor: verification grand challenge.
**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
VCs have several Mb
Thousands of non ground clauses
Developers are willing to wait at most 5 min per VC
Main Challenge

- Quantifiers, quantifiers, quantifiers, ...
- Modeling the runtime
  \[ \forall h, o, f: \]
  \[ \text{IsHeap}(h) \land o \neq \text{null} \land \text{read}(h, o, \text{alloc}) = t \]
  \[ \Rightarrow \]
  \[ \text{read}(h, o, f) = \text{null} \lor \text{read}(h, \text{read}(h, o, f), \text{alloc}) = t \]
Quantifiers, quantifiers, quantifiers, ...

Modeling the runtime

Frame axioms

\[ \forall o, f: \]
\[ o \neq \text{null} \land \text{read}(h_0, o, \text{alloc}) = t \Rightarrow \]
\[ \text{read}(h_1, o, f) = \text{read}(h_0, o, f) \lor (o, f) \in M \]
Main Challenge

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

∀ i,j: i ≤ j ⇒ read(a,i) ≤ read(b,j)
Main Challenge

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

SMT@Microsoft
Challenge: modeling runtime

- Is the axiomatization of the runtime consistent?
- **False** implies everything
- Partial solution: SMT + Saturation Provers
- Found many bugs using this approach
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
Joint work with Y. Hamadi (MSRC) and C. Wintersteiger
Multi-core & Multi-node (HPC)
Different strategies in parallel
Collaborate exchanging lemmas
SMT: Some Applications

- Test case generation
- Verifying Compiler
- Predicate Abstraction
Overview

- [http://research.microsoft.com/slam/](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: \textit{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 $F$

- $\text{Implies}_F(e)$
  - Best Boolean function over $F$ that implies $e$.

- $\text{ImpliedBy}_F(e)$
  - Best Boolean function over $F$ that is implied by $e$.
  - $\text{ImpliedBy}_F(e) = \text{not} \text{Implies}_F(\text{not} \quad e)$
Computing $\text{Implies}_F(e)$

- minterm $m = l_1 \land ... \land l_n$, where $l_i = p_i$, or $l_i = \text{not } p_i$.
- $\text{Implies}_F(e)$: 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 $F$
  - $!x < y, !x=2$ implies $y > 1$
  - $x < y, !x=2$ implies $y > 1$
  - $!x < y, x=2$ implies $y > 1$
  - $x < y, x=2$ implies $y > 1$

$\text{Implies}_F(y > 1) = x < y \land x = 2$
Computing $\text{Implies}_F(e)$

- $F = \{ x < y, x = 2 \}$
- $e : y > 1$
- **Minterms over F**
  - $\neg x < y, \neg x = 2 \implies y > 1$ ✗
  - $x < y, \neg x = 2 \implies y > 1$ ✗
  - $\neg x < y, x = 2 \implies y > 1$ ✗
  - $x < y, x = 2 \implies y > 1$ ✓

$\text{Implies}_F(y > 1) = b_1 \land b_2$
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 Z3.
All-SAT
  Better (more precise) Predicate Abstraction
Unsatisfiable cores
  Why the abstract path is not feasible?
Fast Predicate Abstraction
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:

- $p_1, \neg p_2, p_3, p_4 \Rightarrow e$
- $p_1, \neg p_2, p_3, \neg p_4 \Rightarrow e$
- $p_1, p_2, p_3, \neg p_4 \Rightarrow e$
Other Microsoft clients

- Model programs (M. Veanes – MSRR)
- Termination (B. Cook – MSRC)
- Security protocols (A. Gordon and C. Fournet - MSRC)
- Business Application Modeling (E. Jackson - MSRR)
- Cryptography (R. Venki – MSRR)
- Verifying Garbage Collectors (C. Hawblitzel – MSRR)
- Model Based Testing (L. Bruck – SQL)
- Semantic type checking for D models (G. Bierman – MSRC)

More coming soon...
**Conclusion**

- SMT is hot at Microsoft.
- Many applications.
- Z3 is a new and **very efficient** SMT solver.

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

Thank You!