Satisfiability Modulo Theories: An Appetizer
SBMF 2009 - Gramado

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

Hyper-V

HAVOC

Terminator T-2

VCC

Microsoft Virtualization

SLAM

SAGE

F7

SpecExplorer

Satisfiability Modulo Theories: An Appetizer
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.

(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)

(x_0 = 2
y_0 = 4
m_0 = 2
x_1 = 4
y_1 = 2
m_1 = 0

Satisfiability Modulo Theories: An Appetizer
Signature:

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

Call site:

if \( a \leq 1 \) and \( a \leq b \) then

return \( \text{div}(a, b) \)

Verification condition

\( a \leq 1 \) and \( a \leq b \) implies \( b \neq 0 \)
Is formula $F$ satisfiable modulo theory $T$?

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

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

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

\[ b + 2 = c \quad \text{and} \quad f(\text{read}(\text{write}(a,b,3), c-2)) \neq f(c-b+1) \]
A Theory is a set of sentences

Alternative definition: A Theory is a class of structures
Z3 is a new solver developed at Microsoft Research. Development/Research driven by internal customers. Free for academic research.

Interfaces:

- C/C++
- .NET
- OCaml
- Text

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
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

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

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

Satisfiability Modulo Theories: An Appetizer
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

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

Satisfiability Modulo Theories: An Appetizer
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

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

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

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

a, b, c, s  
\( \text{d, e, t} \)  
g(d), g(e)  
f(a, g(d))  
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

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

Satisfiability Modulo Theories: An Appetizer

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

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

Satisfiability Modulo Theories: An Appetizer
Deciding Equality +
(uninterpreted) Functions

(fully shared) DAGs for representing terms
Union-find data-structure + Congruence Closure
O(n log n)
Deciding Polynomial Equations (over $\mathbb{C}$)

$x^2y - 1 = 0, \; xy^2 - y = 0, \; xz - z + 1 = 0$

Tool: Gröbner Basis
Deciding Polynomial Equations (over $\mathbb{C}$)

Polynomial Ideals:
Algebraic generalization of zeroness

$0 \in I$

$p \in I, q \in I$ implies $p + q \in I$

$p \in I$ implies $pq \in I$
The ideal generated by a finite collection of polynomials \( P = \{ p_1, \ldots, p_n \} \) is defined as:

\[
I(P) = \{ p_1 q_1 + \ldots + p_n q_n \mid q_1, \ldots, q_n \text{ are polynomials} \}
\]

\( P \) is called a basis for \( I(P) \).

Intuition:
For all \( s \in I(P) \),
\[
p_1 = 0, \ldots, p_n = 0 \text{ implies } s = 0
\]
Hilbert’s Weak Nullstellensatz

\[ p_1 = 0, \ldots, p_n = 0 \text{ is unsatisfiable over } \mathbb{C} \]

iff

\[ \mathfrak{l}(\{p_1, \ldots, p_n\}) \text{ contains all polynomials} \]

\[ 1 \in \mathfrak{l}(\{p_1, \ldots, p_n\}) \]
1st Key Idea: polynomials as rewrite rules.

\[ xy^2 - y = 0 \]
Becomes
\[ xy^2 \rightarrow y \]

The rewriting system is terminating but it is not confluent.

\[ xy^2 \rightarrow y, \quad x^2y \rightarrow 1 \]
2\textsuperscript{nd} Key Idea: Completion.

\[ xy^2 \rightarrow y, \quad x^2y \rightarrow 1 \]

Add polynomial:

\[ xy - y = 0 \]

\[ xy \rightarrow y \]
Deciding Polynomial Equations (over \( \mathbb{C} \))

\[
\begin{align*}
x^2y - 1 &= 0, & xy^2 - y &= 0, & xz - z + 1 &= 0 \\
x^2y \rightarrow 1, & xy^2 \rightarrow y, & xz \rightarrow z - 1 \\
x^2y \rightarrow 1, & xy^2 \rightarrow y, & xz \rightarrow z - 1, & xy \rightarrow y \\
x^2y \rightarrow 1, & xy^2 \rightarrow y, & xz \rightarrow z - 1, & xy \rightarrow y \\
x \rightarrow 1, & xy^2 \rightarrow y, & xz \rightarrow z - 1, & xy \rightarrow y \\
y \rightarrow 1, & x \rightarrow 1, & xz \rightarrow z - 1, & xy \rightarrow y \\
y \rightarrow 1, & x \rightarrow 1, & 1 = 0, & xy \rightarrow y
\end{align*}
\]
In practice, we need a combination of theory solvers.

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

Partial model

M | F

Set of clauses
Guessing (case-splitting)

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

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

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

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

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

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

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

Satisfiability Modulo Theories: An Appetizer
Efficient decision procedures for conjunctions of ground literals.

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

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

backtrack

Satisfiability Modulo Theories: An Appetizer
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

\[ a=b, \ a<5 \ | \ ... \ a<6 \ \lor \ f(a) = a \]

\[ a=b, \ a<5, \ a<6 \ | \ ... \ a<6 \ \lor \ f(a) = a \]
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:
\( \neg(a=b) \lor \neg(a > 0) \lor \neg(c > 0) \lor \neg(a + c < 0) \)

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

\[ \text{bvmul}_{32}(a,b) = \text{bvmul}_{32}(b,a) \]
Theorem

First-order Prover

$F \cup T$

$T$ may not have a finite axiomatization

Satisfiability Modulo Theories: An Appetizer
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

Satisfiability Modulo Theories: An Appetizer
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)

1. **Run Test and Monitor**
   - Test Inputs
   - Seed
   - New input

2. **Execution Path**

3. **Constraint System**
   - Solve

4. **Known Paths**

5. **Path Condition**

---

Satisfiability Modulo Theories: An Appetizer

Microsoft Research
PEX: Implements DART for .NET.

SAGE: Implements DART for x86 binaries.

YOGI: Implements DART to check the feasibility of program paths generated statically.

Vigilante: Partially implements DART to dynamically generate worm filters.

Satisfiability Modulo Theories: An Appetizer
Test input generator

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

Satisfiability Modulo Theories: An Appetizer
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; }
...
```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 Inputs
```
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) // 0 == c ➔ true
            ResizeArray();
        items[this.count++] = item;
    }
    ...

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

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.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.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></td>
</tr>
</tbody>
</table>
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.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></td>
<td></td>
</tr>
</tbody>
</table>
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 3, (-1, null)

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

```csharp
class ArrayListTest {
    [PexMethod]
    void AddItem(int c, object item) {
        var list = new ArrayList(c);
        list.AddItem(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;
    }
}```
Satisfiability Modulo Theories: An Appetizer

- 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.
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.
Starting with 100 zero bytes ...

SAGE generates a crashing test for Media1 parser

| 00000000h: | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; | .................. |
| 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 0 – seed file
Starting with 100 zero bytes ...

SAGE generates a crashing test for Media1 parser

```
00000000h: 52 49 46 46 00 00 00 00 00 00 00 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
```

Generation 1

Satisfiability Modulo Theories: An Appetizer
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
Zero to Crash in 10 Generations

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

<table>
<thead>
<tr>
<th>Generation 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF... *** ....</td>
</tr>
<tr>
<td>00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................</td>
</tr>
<tr>
<td>00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................</td>
</tr>
<tr>
<td>00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................</td>
</tr>
<tr>
<td>00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................</td>
</tr>
<tr>
<td>00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................</td>
</tr>
<tr>
<td>00000060h: 00 00 00 00</td>
</tr>
</tbody>
</table>
Starting with 100 zero bytes ...

SAGE generates a crashing test for Media1 parser

Generation 4
Starting with 100 zero bytes ... 

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

SAGE generates a crashing test for Media1 parser

Generation 6
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 ; ................
00000060h: 00 00 00 00
```

Generation 7
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
Starting with 100 zero bytes ...

SAGE generates a crashing test for Media1 parser

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.
Verifying Compilers

Annotated Program $\rightarrow$ Verification Condition $F$

pre/post conditions
invariants
and other annotations
class C {
    private int a, z;
    invariant z > 0

    public void M() {
        requires a != 0
        {
            z = 100/a;
        }
    }
}
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)
x := E
  - x := x + 1
  - x := 10

havoc x

assert P

assume P

S □ T

S ; T
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.
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 \( \text{wp}(S, Q) \)

to check \{P\} S \{Q\}, check \( P \Rightarrow P' \)
Weakest preconditions

- `wp(x := E, Q) = Q[E/x]`
- `wp(havoc x, Q) = (∀x • Q)`
- `wp(assert P, Q) = P ∧ Q`
- `wp(assume P, Q) = P → Q`
- `wp(S ; T, Q) = wp(S, wp(T, Q))`
- `wp(S □ T, Q) = wp(S, Q) ∧ wp(T, Q)`
Structured if statement

if E then S else T end =

assume E; S

assume ¬E; T
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

\[ \square \text{ assume } \neg \ E \]  

check that the loop invariant holds initially

“fast forward” to an arbitrary iteration of the loop

check that the loop invariant is maintained by the loop body

where x denotes the assignment targets of S
Verification conditions: Structure

∀ Axioms (non-ground)

Control & Data Flow

BIG and-or tree (ground)
**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
Challenge: annotation burden

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

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

Modeling the runtime

Frame axioms

\( \forall o, f: \)

\( o \neq \text{null} \land \text{read}(h_0, o, \text{alloc}) = t \Rightarrow \)

\( \text{read}(h_1, o, f) = \text{read}(h_0, o, f) \lor (o, f) \in M \)
Challenge

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

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

1. Heuristic quantifier instantiation
2. Combining SMT with Saturation provers
3. Complete quantifier instantiation
4. Decidable fragments
5. Model based quantifier instantiation
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

Satisfiability Modulo Theories: An Appetizer
Joint work with Y. Hamadi (MSRC) and C. Wintersteiger

Multi-core & Multi-node (HPC)

Different strategies in parallel

Collaborate exchanging lemmas
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!