Satisfiability Modulo Theories: A Calculus of Computation
PUC, Rio de Janeiro, 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

Satisfiability Modulo Theories: A Calculus of Computation
Some Applications @ Microsoft

- HAVOC
- Terminator T-2
- VCC
- Hyper-V
- SAGE
- SpecExplorer
- NModel
- SpecExplorer
- Vigilante
- Yogi
- F7

Satisfiability Modulo Theories: A Calculus of Computation
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

Satisfiability Modulo Theories: A Calculus of Computation
Type checking

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

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 $T$. 
b + 2 = c \text{ and } f(read(write(a,b,3), c-2) \neq f(c-b+1)
\[ b + 2 = c \] and 
\[ f(\text{read}(\text{write}(a, b, 3), c-2)) \neq f(c-b+1) \]
\[ b + 2 = c \quad \text{and} \quad f(\text{read(write}(a,b,3), c-2) \neq f(c-b+1) \]
$b + 2 = c$ and $f(read(write(a,b,3), c-2)) \neq f(c-b+1)$
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:

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

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

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

Satisfiability Modulo Theories: A Calculus of Computation
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

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

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

Unsatisfiable

Satisfiability Modulo Theories: A Calculus of Computation
$a = b, \ b = c, \ d = e, \ b = s, \ d = t, \ a \neq e$

Model

$|M| = \{ 0, 1 \}$

$M(a) = M(b) = M(c) = M(s) = 0$

$M(d) = M(e) = M(t) = 1$
Deciding Equality + (uninterpreted) Functions

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

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

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:

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

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

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

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

\[ f(a,g(d)), f(b,g(e)) \]

\[ a, b, c, s \quad d, e, t \quad g(d), g(e) \]

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

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

Satisfiability Modulo Theories: A Calculus of Computation
Deciding Equality + (uninterpreted) Functions

(fully shared) DAGs for representing terms
Union-find data-structure + Congruence Closure
$O(n \log n)$
Deciding Equality + (uninterpreted) Functions

Many theories can be reduced to Equality + Uninterpreted functions

Arrays, Sets
Lists, Tuples
Inductive Datatypes
Deciding Difference Arithmetic

\[ a - b \leq 2 \]

\[ x := x + 1 \]

\[ x_1 = x_0 + 1 \]

\[ x_1 \leq x_0 + 1, \ x_1 \geq x_0 + 1 \]

\[ x_1 - x_0 \leq 1, \ x_0 - x_1 \leq -1 \]
Deciding Difference Arithmetic

\[ a - b \leq 2, \quad b - c \leq -3, \quad b - d \leq -1, \quad d - a \leq 4, \quad c - a \leq 0 \]
a - b ≤ 2,  b - c ≤ -3,  b - d ≤ -1,  d - a ≤ 4,  c - a ≤ 0
a - b ≤ 2,  b - c ≤ -3,  b - d ≤ -1,  d - a ≤ 4,  c - a ≤ 0
Deciding Difference Arithmetic

\[ a - b \leq 2, \quad b - c \leq -3, \quad b - d \leq -1, \quad d - a \leq 4, \quad c - a \leq 0 \]
Deciding Difference Arithmetic

\[ a - b \leq 2, \quad b - c \leq -3, \quad b - d \leq -1, \quad d - a \leq 4, \quad c - a \leq 0 \]
Deciding Difference Arithmetic

\[ a - b \leq 2, \quad b - c \leq -3, \quad b - d \leq -1, \quad d - a \leq 4, \quad c - a \leq 0 \]
Deciding Difference Arithmetic

\[ a - b \leq 2, \quad b - c \leq -3, \quad b - d \leq -1, \quad d - a \leq 4, \quad c - a \leq 0 \]

Unsatisfiable iff Negative Cycle
Shortest Path Algorithms

Bellman-Ford: $O(nm)$
Floyd-Warshall: $O(n^3)$
## Other Little Engines

<table>
<thead>
<tr>
<th>Linear Arithmetic</th>
<th>Simplex, Fourier Motzkin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-linear (complex) arithmetic</td>
<td>Grobner Basis</td>
</tr>
<tr>
<td>Non-linear (real) arithmetic</td>
<td>Cylindrical Algebraic Decomposition</td>
</tr>
<tr>
<td>Equational theories</td>
<td>Knuth-Bendix Completion</td>
</tr>
<tr>
<td>Bitvectors</td>
<td>Bit-blasting, rewriting</td>
</tr>
</tbody>
</table>
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

Assignment:

\[
\begin{align*}
p & \lor q, \\
p & \lor \neg q, \\
\neg p & \lor q, \\
\neg p & \lor \neg q
\end{align*}
\]

\[
\begin{align*}
p & = \text{false}, \\
q & = \text{false}
\end{align*}
\]
SAT (propositional checkers): Case Analysis

Assignment:
\[ p = \text{false,} \]
\[ q = \text{true} \]
Satisfiability Modulo Theories: A Calculus of Computation

SAT (propositional checkers): Case Analysis

$p \lor q,$
$p \lor \neg q,$
$\neg p \lor q,$
$\neg p \lor \neg q$

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

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

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

Partial model

Set of clauses

M | F
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

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

\[ p, s \ | \ p \lor q, s \lor q, \neg p \lor \neg q \]
Efficient indexing (two-watch literal)
Non-chronological backtracking (backjumping)
Lemma learning
...

Satisfiability Modulo Theories: A Calculus of Computation
Efficient decision procedures for conjunctions of ground literals.

\[ a=b, a<5 \mid \neg a=b \lor f(a)=f(b), \quad a < 5 \lor a > 10 \]
a=b, a > 0, c > 0, a + c < 0 | F

backtrack
Naïve recipe?

SMT Solver = DPLL + Decision Procedure

Standard question:

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

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

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

$T$ may not have a finite axiomatization

Satisfiability Modulo Theories: A Calculus of Computation
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: A Calculus of Computation
Hunting for Security Bugs.

- Two main techniques used by “black hats”:
  - Code inspection (of binaries).
  - **Black box fuzz testing.**

- **Black box** fuzz testing:
  - A form of black box random testing.
  - Randomly *fuzz* (=modify) a well formed input.
  - Grammar-based fuzzing: rules to encode how to fuzz.

- **Heavily** used in security testing
  - At MS: several internal tools.
  - Conceptually simple yet effective in practice
Directed Automated Random Testing (DART)

Run Test and Monitor

Execution Path

Path Condition

Test Inputs

Known Paths

Constraint System

Satisfiability Modulo Theories: A Calculus of Computation
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: A Calculus of Computation
Test input generator

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

.NET Framework Class Library
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.
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.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.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); }
}

item == item ➞ true

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; }

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

    Constraints to solve | Inputs | Observed Constraints
----------------------|-------|---------------------
                            (0,null)   !(c<0) && 0==c
                            (1,null)   !(c<0) && 0!=c
**ArrayList: Run 2, (1, null)**

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

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

```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 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; }  
...  
  
[Constraints to solve]  

<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>
<tr>
<td>!(c&lt;0) &amp;&amp; 0!=c</td>
<td>(1,null)</td>
</tr>
<tr>
<td>c&lt;0</td>
<td>(-1,null)</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 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; } ...
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.
Zero to Crash in 10 Generations

- 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

Generation 1

Satisfiability Modulo Theories: A Calculus of Computation
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

Generation 3

```
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 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
```

Satisfiability Modulo Theories: A Calculus of Computation
Starting with 100 zero bytes ...

SAGE generates a crashing test for Media1 parser

Generation 4

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000h</td>
<td>52 49 46 46 3D 00 00 00 ** ** ** 20 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 73 74 72 68 00 00 00 00 00 00 00 00 ; ................</td>
</tr>
<tr>
<td>00000040h</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................</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>

Satisfiability Modulo Theories: A Calculus of Computation
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 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000033h: 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
```

Generation 5

*Satisfiability Modulo Theories: A Calculus of Computation*
**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>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000h</td>
<td>52 49 46 46 3D 00 00 00 ** ** ** 20 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 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 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 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 68 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

Generation 8

Satisfiability Modulo Theories: A Calculus of Computation
Starting with 100 zero bytes ...

SAGE generates a crashing test for Media1 parser

Generation 9

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 [0:1] 00 00 00 ; ....................
00000060h: 00 00 00 00

Satisfiability Modulo Theories: A Calculus of Computation
Zero to Crash in 10 Generations

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

Generation 10 – CRASH

Satisfiability Modulo Theories: A Calculus of Computation
SAGE (cont.)

- 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;
        }
    }
}
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),
- SMT
Command language

- \( x := E \)
  - \( x := x + 1 \)
  - \( x := 10 \)

- havoc \( x \)

- \( S ; T \)

- assert \( P \)

- assume \( P \)

- \( S \models T \)
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 \textit{weakest precondition} of S with respect to Q, written \( \text{wp}(S, Q) \).

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

- \( \text{wp}( x := E, Q ) = Q[ E / x ] \)
- \( \text{wp}( \text{havoc} x, Q ) = (\forall x \bullet Q) \)
- \( \text{wp}( \text{assert} P, Q ) = P \land Q \)
- \( \text{wp}( \text{assume} P, Q ) = P \implies Q \)
- \( \text{wp}( S ; T, Q ) = \text{wp}( S, \text{wp}( T, Q )) \)
- \( \text{wp}( S \Box T, Q ) = \text{wp}( S, Q ) \land \text{wp}( T, Q ) \)
Structured if statement

\[
\text{if } E \text{ then } S \text{ else } T \text{ end } = \\
\text{assume } E; \; S \\
\text{else} \\
\text{assume } \neg E; \; T
\]
While loop with loop invariant

while E
    invariant J
    do
        S
    end

where x denotes the assignment targets of S

check that the loop invariant holds initially

assert J;

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

“fast forward” to an arbitrary iteration of the loop

check that the loop invariant is maintained by the loop body
Verification conditions: Structure

∀ Axioms (non-ground)

BIG and-or tree (ground)

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

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

- Automatic generation of: Loop Invariants
- Houdini-style automatic annotation generation
Modeling the runtime

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

Modeling the runtime

Frame axioms

\( \forall o, f: \)

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

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

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

\[ \forall i, j: i \leq j \implies read(a, i) \leq read(b, j) \]
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!

Satisfiability Modulo Theories: A Calculus of Computation
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 timing out”
- This also happens with SAT solvers (NP-complete)
- In our case, the problems are undecidable
- Partial solution: parallelization

Satisfiability Modulo Theories: A Calculus of Computation
Joint work with Y. Hamadi (MSRC) and C. Wintersteiger
Multi-core & Multi-node (HPC)
Different strategies in parallel
Collaborate exchanging lemmas
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
SMT solvers use heuristic quantifier instantiation.

E-matching (matching modulo equalities).

Example:

\[ \forall x: f(g(x)) = x \{ f(g(x)) \} \]

\[ a = g(b), \]
\[ b = c, \]
\[ f(a) \neq c \]
SMT solvers use heuristic quantifier instantiation.

E-matching (matching modulo equalities).

Example:
\[ \forall x: f(g(x)) = x \{ f(g(x)) \} \]
\[ a = g(b), \]
\[ b = c, \]
\[ f(a) \neq c \]

Equalities and ground terms come from the partial model \( M \)
E-matching: why do we use it?

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

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

<table>
<thead>
<tr>
<th>Problem</th>
<th>Indexing Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast retrieval</td>
<td>E-matching code trees</td>
</tr>
<tr>
<td>Incremental E-Matching</td>
<td>Inverted path index</td>
</tr>
</tbody>
</table>
E-matching code trees

Trigger:
\[ f(x_1, g(x_1, a), h(x_2), b) \]

Instructions:
1. \( \text{init}(f, 2) \)
2. \( \text{check}(r_4, b, 3) \)
3. \( \text{bind}(r_2, g, r_5, 4) \)
4. \( \text{compare}(r_1, r_5, 5) \)
5. \( \text{check}(r_6, a, 6) \)
6. \( \text{bind}(r_3, h, r_7, 7) \)
7. \( \text{yield}(r_1, r_7) \)

Similar triggers share several instructions.

Combine code sequences in a code tree

Satisfiability Modulo Theories: A Calculus of Computation
Tight integration: DPLL + Saturation solver.
Inference rule:

\[
\begin{array}{c}
C_1 \\ \vdots \\ C_n \\
\hline
C
\end{array}
\]

DPLL(\(\Gamma\)) is parametric.

Examples:
- Resolution
- Superposition calculus
- ...
DPLL(\(\Gamma\))

Saturation Solver  \arrow{ground literals}  DPLL + Theories

\arrow{ground clauses}