SAT & SMT

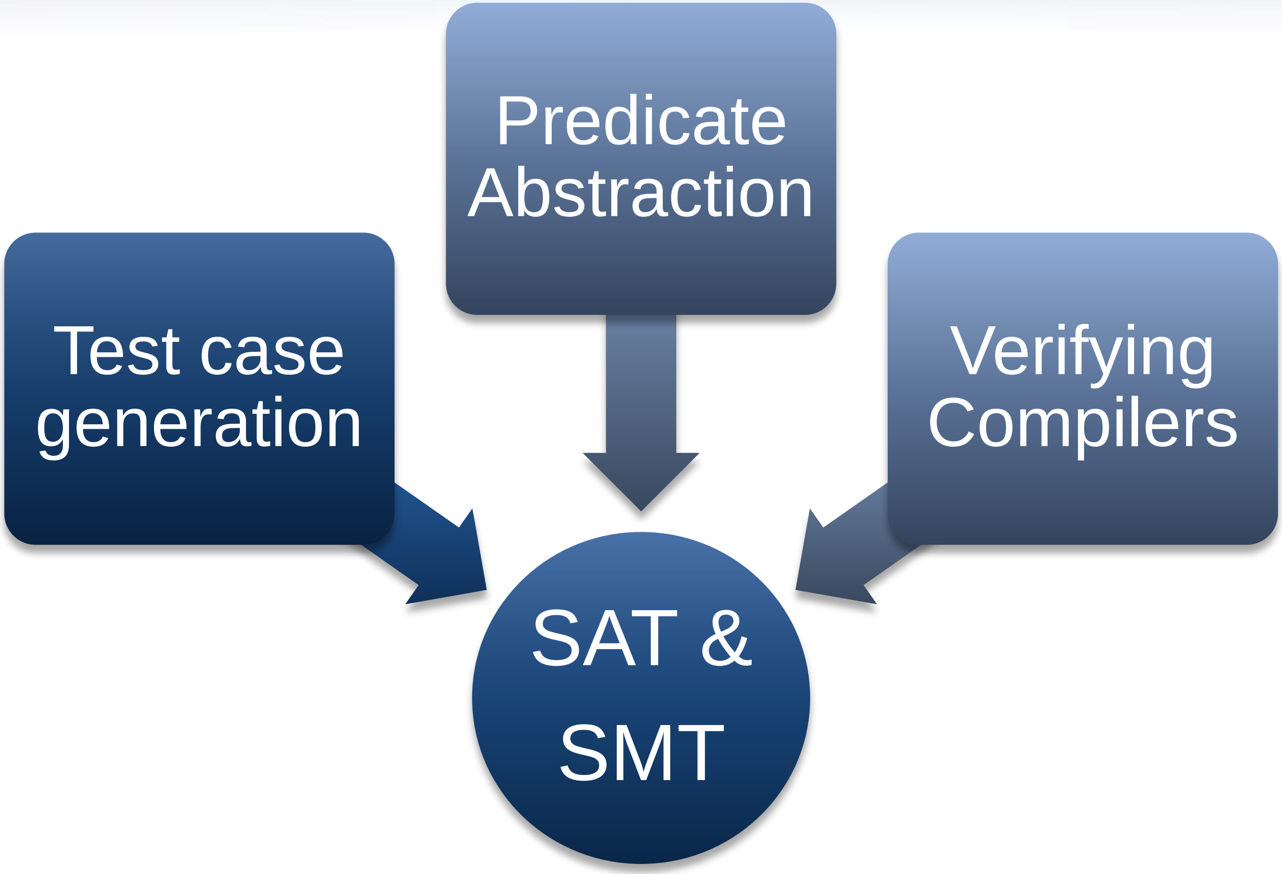
Constraint Satisfaction in

# Software Verification & Testing

NSF Workshop on Symbolic Computation for Constraint Satisfaction - 2008

Leonardo de Moura
Microsoft Research

# SAT & SMT in Software Verification & Testing



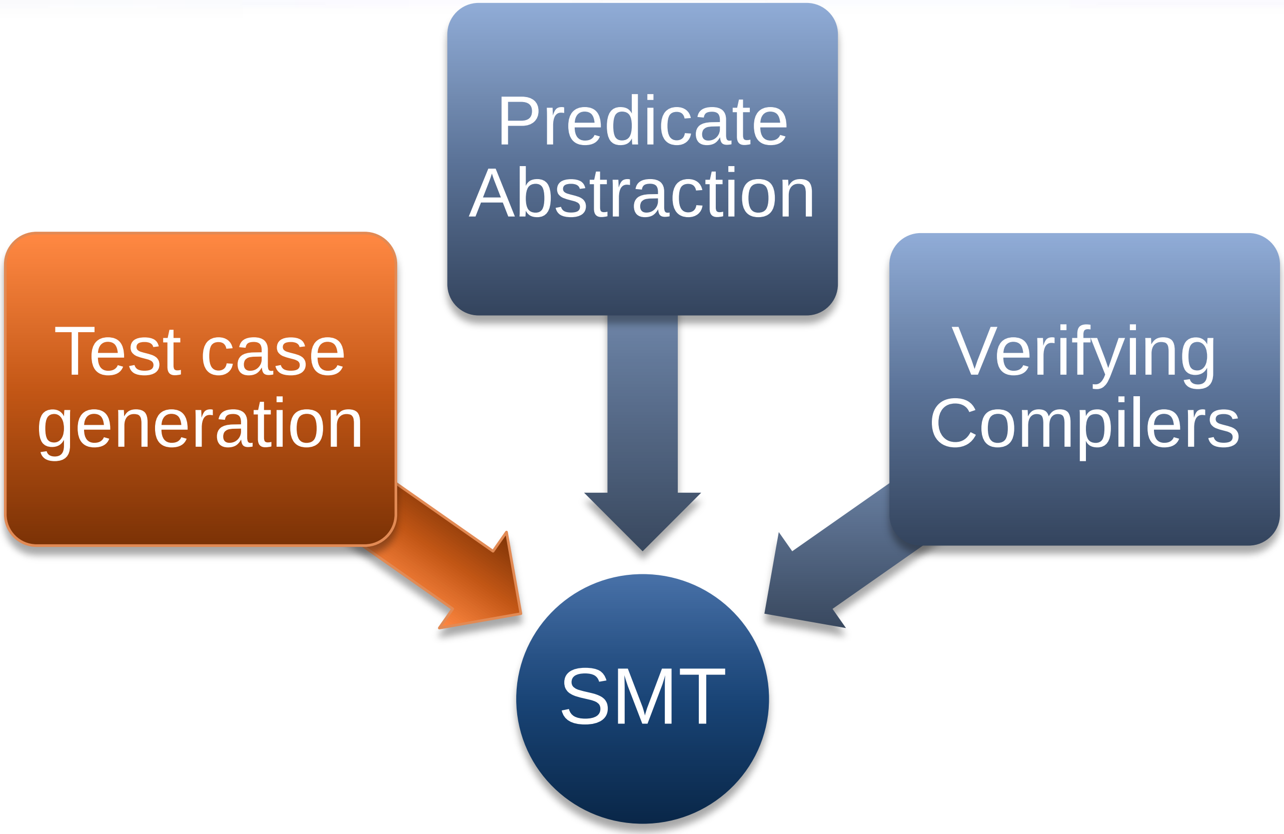



Microsoft Research

# Satisfiability Modulo Theories (SMT)


- Arithmetic
- Bit-vectors
- Arrays
- …

# SMT in Software Verification & Testing






Microsoft Research

# Test case generation

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

SSA

$(y_0 > 0)$ and

$(m_0 = x_0 \% y_0)$ and

not $(m_0 = 0)$ and

$(x_1 = y_0)$ and

$(y_1 = m_0)$ and

$(m_1 = x_1 \% y_1)$ and

$(m_1 = 0)$

SMT Solver

model

$x_0 = 2$

$y_0 = 4$

$m_0 = 2$

$x_1 = 4$

$y_1 = 2$

$m_1 = 0$

We want a trace where the loop is executed twice.

# 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:
  - 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.
- 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



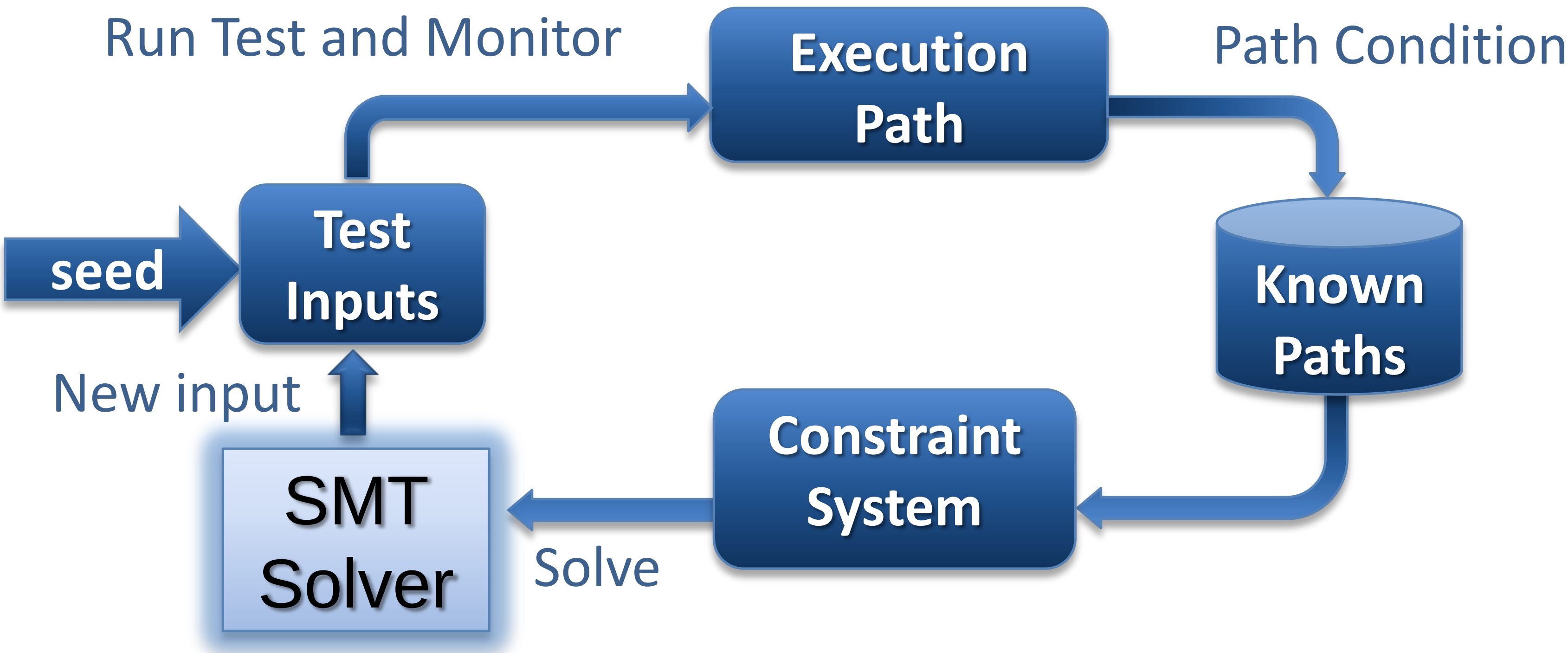
Microsoft Research

# 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)

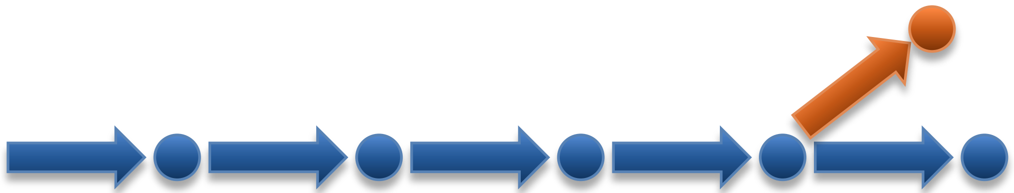





Microsoft Research

# Constraint System Generation

- Unit x Application
- Trade off between performance and precision
- Execution path mutation:




- Concretization

  x = y * z ➔ x = 128 (if y = 2 and z = 64 in the existing path)



Microsoft Research

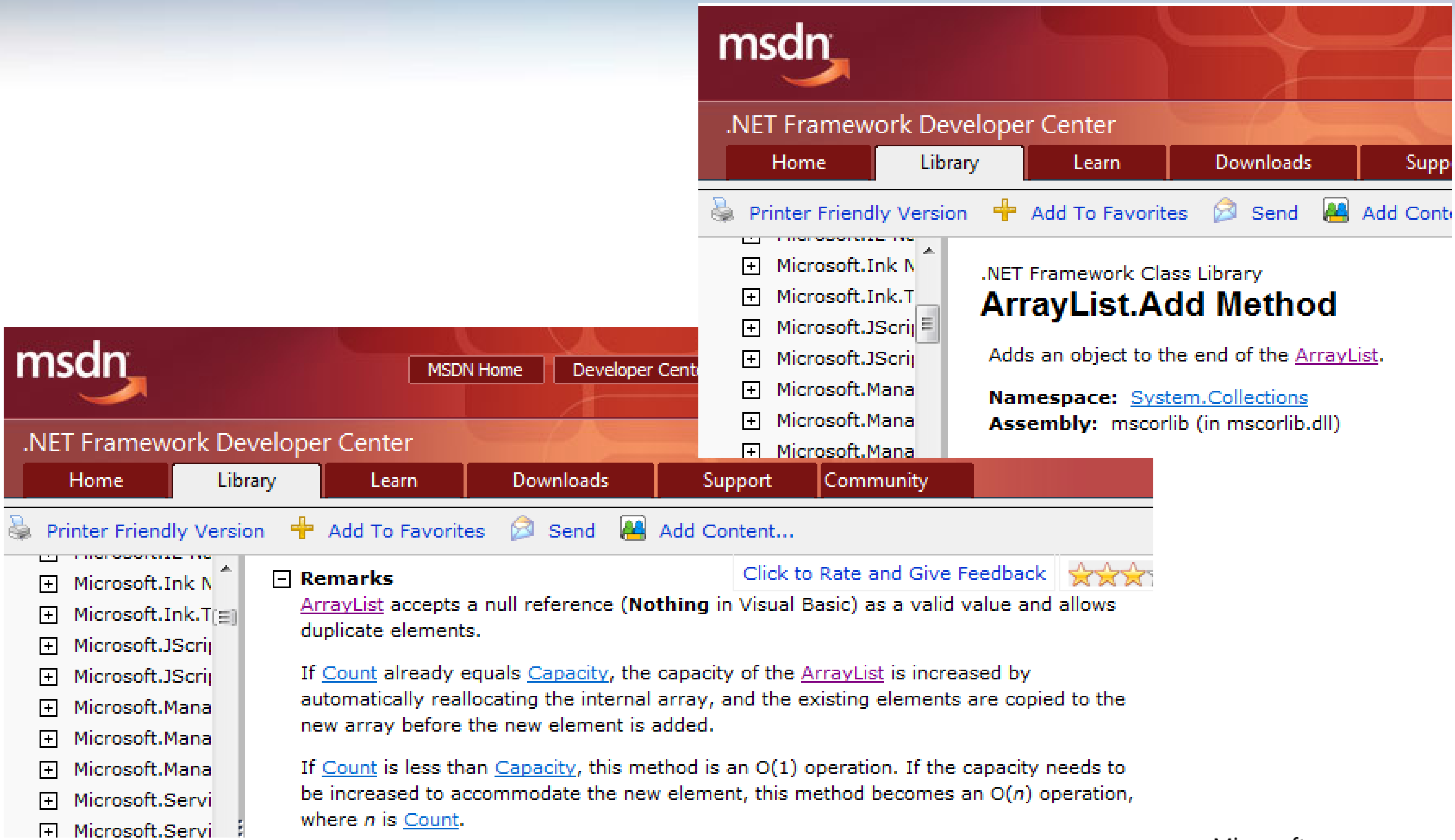
# DARTish projects at Microsoft

| | |
|---|---|
| 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
- Visual Studio Plugin

# 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.

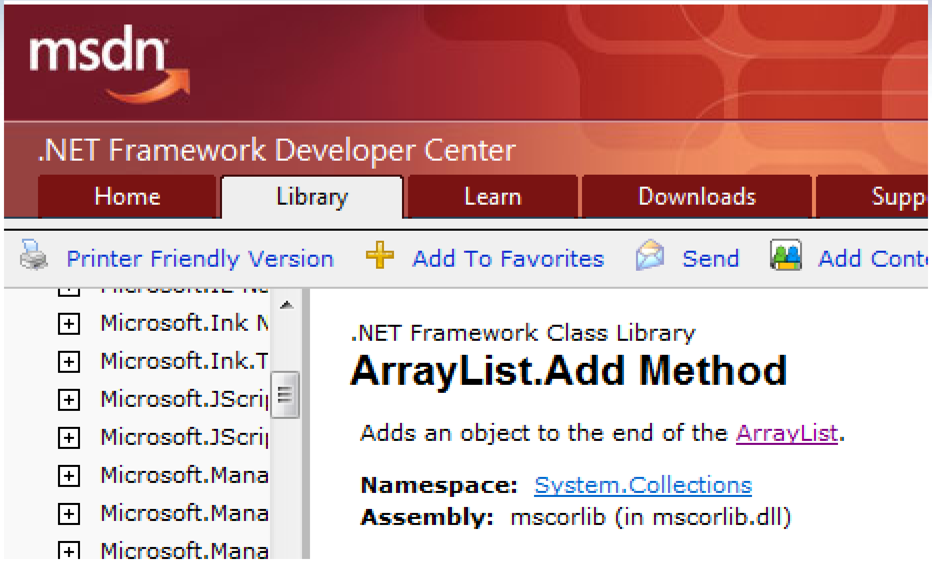
If Count already equals Capacity, the capacity of the ArrayList is increased by automatically reallocating the internal array, and the existing elements are copied to the new array before the new element is added.

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.

# ArrayList: AddItem Test

```
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; }
...
```



# ArrayList: Starting Pex…

```
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; }
...
```

| Inputs |
|---|
| |

Microsoft Research

# ArrayList: Run 1, (0,null)

```
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; }
...
```

| Inputs |
|---|
| (0,null) |

Microsoft Research

# ArrayList: Run 1, (0,null)

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

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

```
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; }
...
```

c < 0 → false

# ArrayList: Run 1, (0,null)

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

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

```
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; }
...
```

0 == c → true

Microsoft Research

# ArrayList: Run 1, (0,null)

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

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

```
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; }
...
```

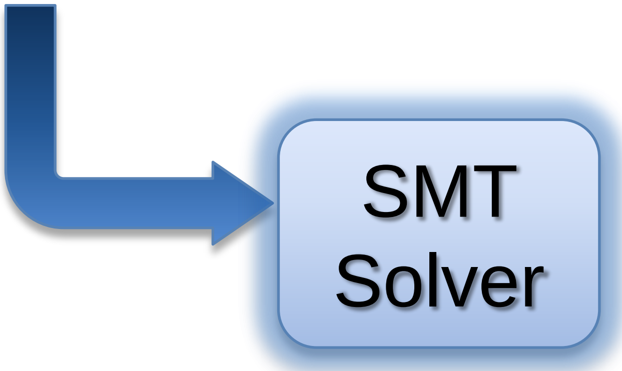
Microsoft Research

# ArrayList: Picking the next branch to cover

```
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; }
...
```

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

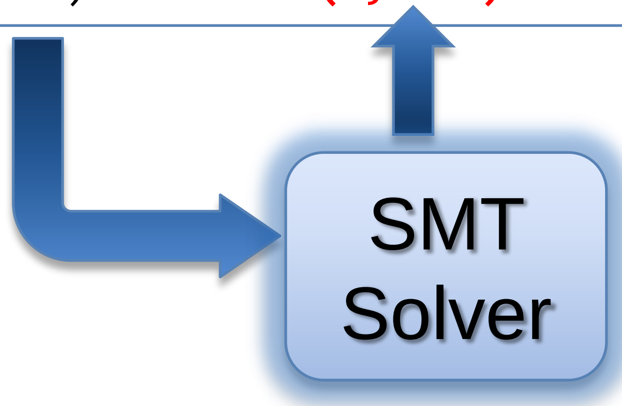


# ArrayList: Solve constraints using SMT solver

```
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; }
...
```

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




Microsoft Research

# ArrayList: Run 2, (1, null)

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

| Constraints to solve | Inputs | Observed Constraints |
|---|---|---|
| | (0,null) | !(c<0) && 0==c |
| !(c<0) && 0!=c | (1,null) | !(c<0) && 0!=c |

```
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; }
...
```

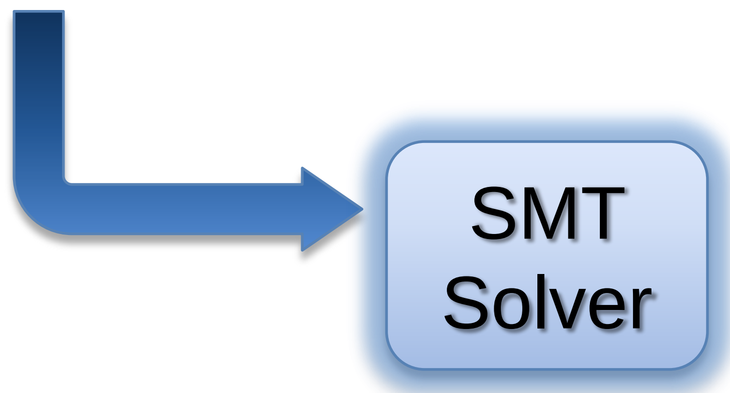
0 == c → false

# ArrayList: Pick new branch

```
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; }
...
```

| Constraints to solve | Inputs | Observed Constraints |
|---|---|---|
| | (0,null) | !(c<0) && 0==c |
| !(c<0) && 0!=c | (1,null) | !(c<0) && 0!=c |
| c<0 | | |



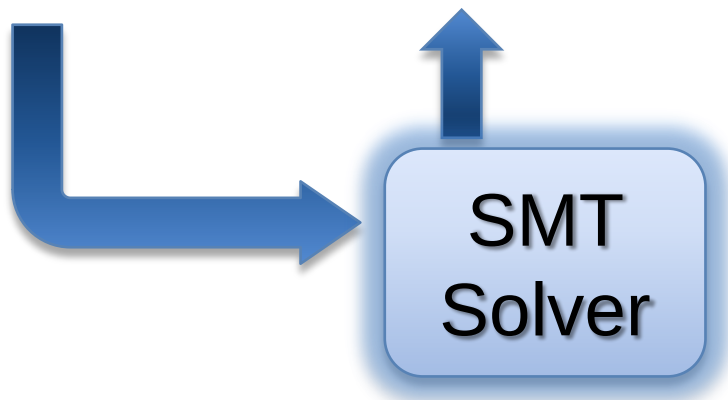

Microsoft Research

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

```
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; }
...
```

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




Microsoft Research

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

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

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

```
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; }
...
```

c < 0 → true

Microsoft Research

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

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

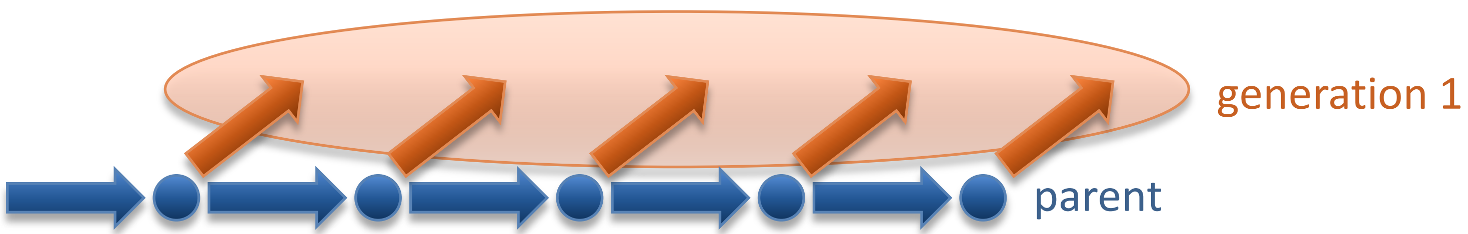
| Constraints to solve | Inputs | Observed Constraints |
|---|---|---|
| | (0,null) | !(c<0) && 0==c |
| !(c<0) && 0!=c | (1,null) | !(c<0) && 0!=c |
| c<0 | (-1,null) | c<0 |

```
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; }
...
```

Microsoft Research

# SAGE

- 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.



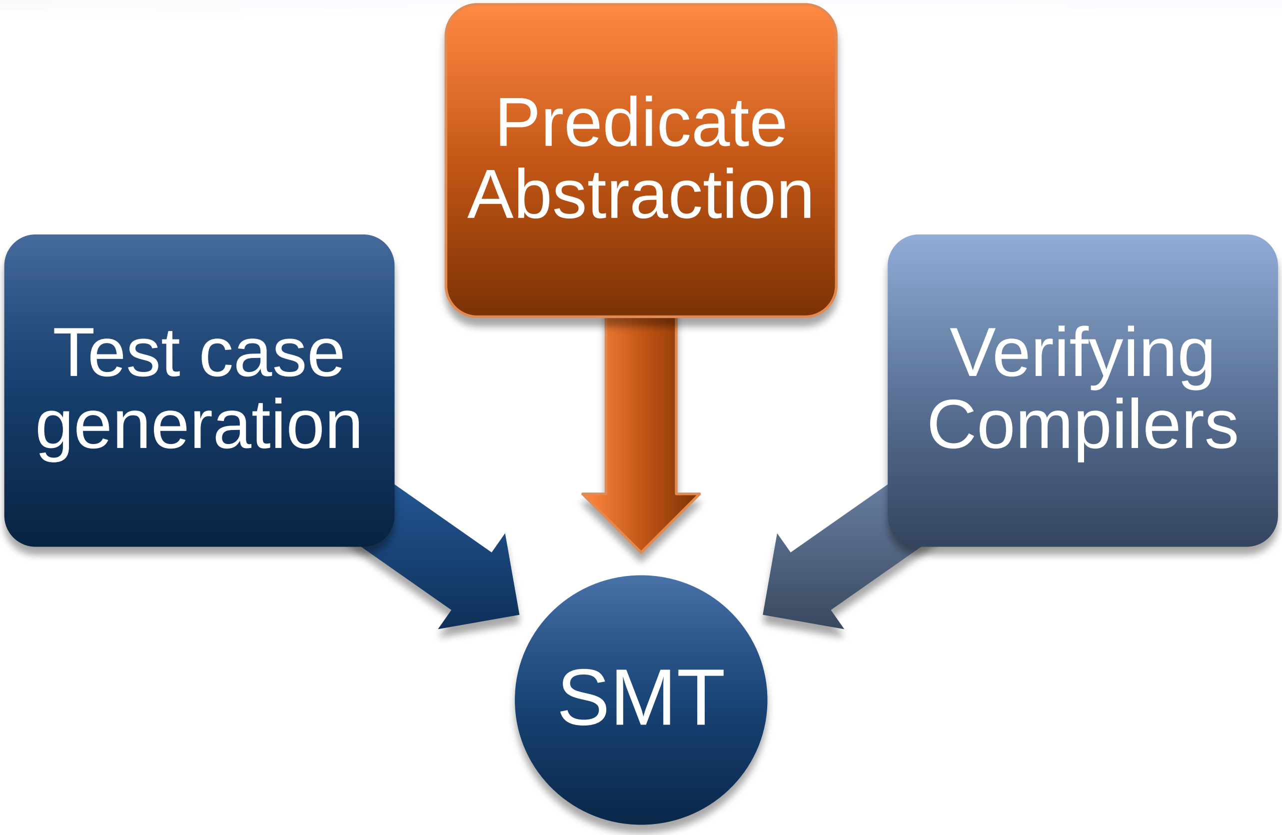
# SAGE (cont.)

- 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
- Found > 100 security bugs in Windows 7
- Gold medal in an internal file fuzzing competition
- **Powered by SMT**

# Challenges

- Trade off between precision and performance
- Scalability
- Machine arithmetic (aka Bitvectors)
- Floating point arithmetic. FP operations are:
  - Concretized in SAGE
  - Approximated using rational numbers in Pex

# SMT in Software Verification & Testing



# Overview

- *SLAM/SDV* is a software model checker.
- Application domain: *device drivers*.
- Architecture:

  **c2bp** C program → boolean program (*predicate abstraction).*

  **bebop** Model checker for boolean programs.

  **newton** Model refinement (check for path feasibility)
- SMT solvers are used to perform predicate abstraction and to check path feasibility.
- c2bp makes several calls to the **SMT solver**. The formulas are relatively small.

# Predicate Abstraction: *c2bp*

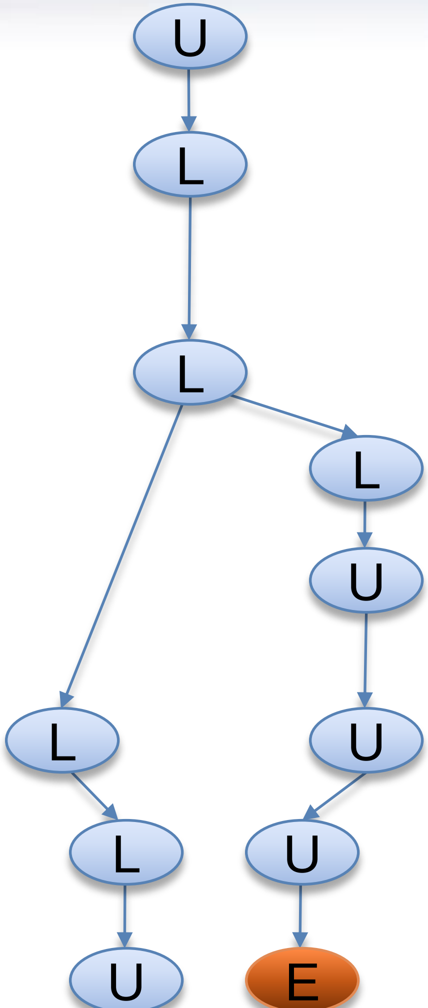
- **Given** a C program $P$ and $F = \{p_1, \ldots, p_n\}$.
- **Produce** a Boolean program $B(P, F)$
  - Same control flow structure as P.
  - Boolean variables $\{b_1, \ldots, b_n\}$ to match $\{p_1, \ldots, p_n\}$.
  - Properties true in $B(P, F)$ are true in $P$.
- Each $p_i$ is a pure Boolean expression.
- Each $p_i$ represents set of states for which $p_i$ is true.
- Performs modular abstraction.

# Example

Do this code obey the looking rule?

```
do {
   KeAcquireSpinLock();

   nPacketsOld = nPackets;

   if(request){
        request = request->Next;
        KeReleaseSpinLock();
        nPackets++;
   }
} while (nPackets != nPacketsOld);

KeReleaseSpinLock();
```

# Example

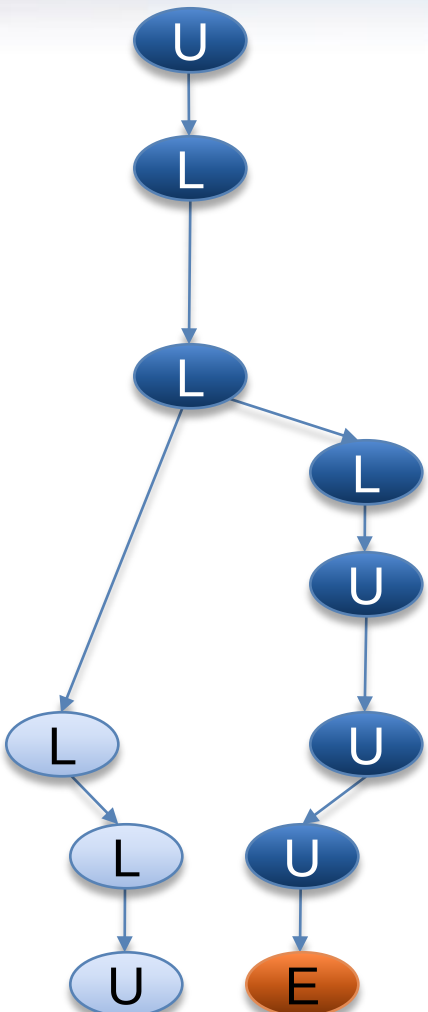



```
do {
  KeAcquireSpinLock();

  if(*){

    KeReleaseSpinLock();

  }
} while (*);

KeReleaseSpinLock();
```

Model checking Boolean program

# Example

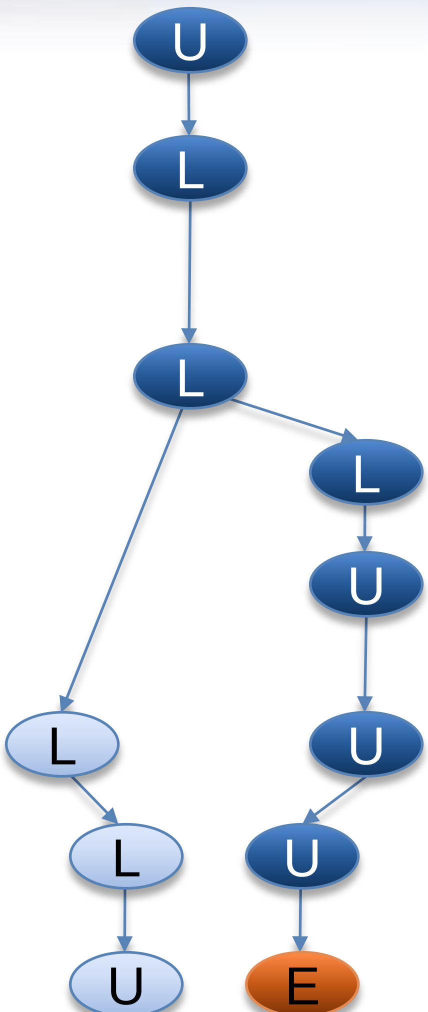
Is error path feasible?




```
do {
   KeAcquireSpinLock();

   nPacketsOld = nPackets;

   if(request){
       request = request->Next;
       KeReleaseSpinLock();
       nPackets++;
   }
} while (nPackets != nPacketsOld);

KeReleaseSpinLock();
```

# Example

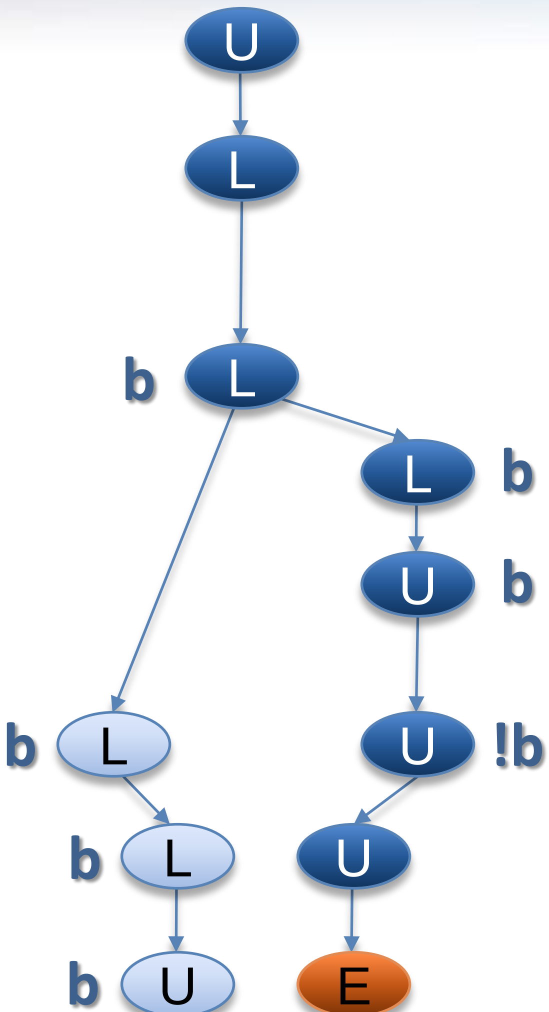
Add new predicate to
Boolean program
b: (nPacketsOld == nPackets)




```
do {
   KeAcquireSpinLock();

   b = true;

   if(request){
      request = request->Next;
      KeReleaseSpinLock();
      b = b ? false : *;
   }
} while (!b);

KeReleaseSpinLock();
```

# Example

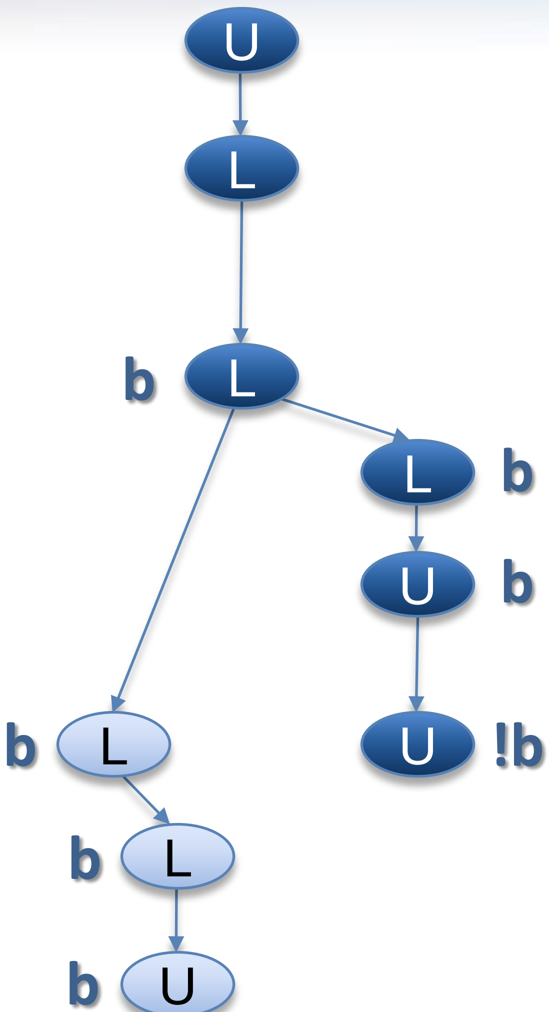
Model Checking
Refined Program
b: (nPacketsOld == nPackets)




```
do {
   KeAcquireSpinLock();

   b = true;

   if(*){

      KeReleaseSpinLock();
      b = b ? false : *;
   }
} while (!b);

KeReleaseSpinLock();
```

# Example

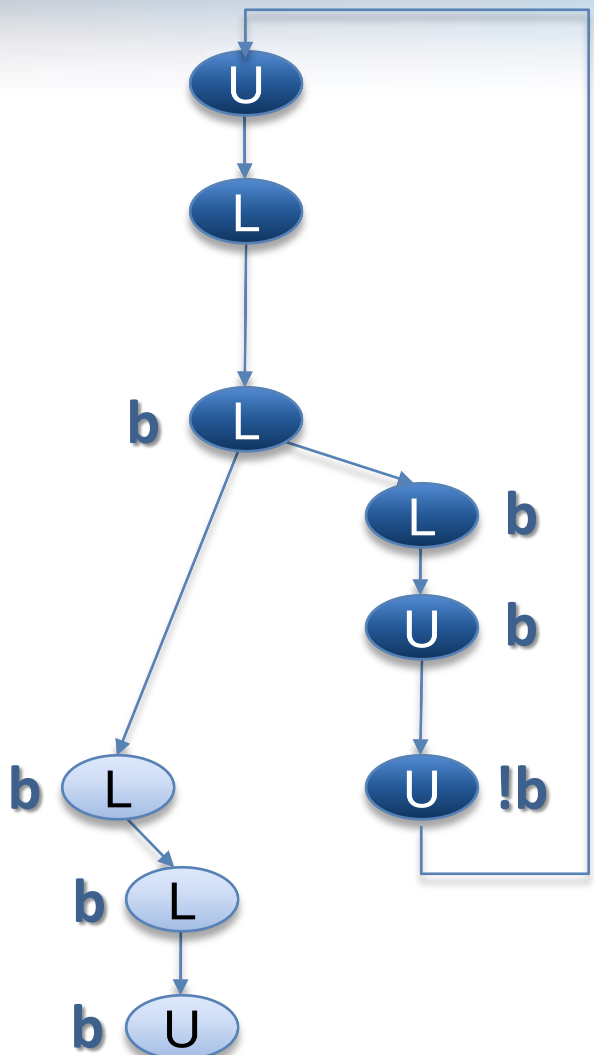
Model Checking
Refined Program
b: (nPacketsOld == nPackets)




```
do {
   KeAcquireSpinLock();

   b = true;

   if(*){

      KeReleaseSpinLock();
      b = b ? false : *;
   }
} while (!b);

KeReleaseSpinLock();
```

# Example

Model Checking
Refined Program
b: (nPacketsOld == nPackets)




```
do {
   KeAcquireSpinLock();

   b = true;

   if(*){

      KeReleaseSpinLock();
      b = b ? false : *;
   }
} while (!b);

KeReleaseSpinLock();
```

# Abstracting Expressions via *F*

- $Implies_F\ (e)$
  - Best Boolean function over *F* that implies *e.*
- $ImpliedBy_F\ (e)$
  - Best Boolean function over *F* that is implied by *e.*
  - $ImpliedBy_F\ (e) = not\ Implies_F\ (not\ e)$

# Computing $Implies_F(e)$

- minterm $m = l_1 \wedge \ldots \wedge l_n$, where $l_i = p_i$, or $l_i = not\ p_i$.
- $Implies_F\ (e)$: disjunction of all minterms that imply $e$.
- Naive approach
  - Generate all $2^n$ possible minterms.
  - For each minterm $m$, use **SMT solver** to check validity of $m \Rightarrow e$.
- Many possible optimizations

# Computing $Implies_F(e)$

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

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

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

# Newton

- Given an error path *p* in the Boolean program *B*.
- Is *p* a feasible path of the corresponding C program?
  - Yes: found a bug.
  - No: find predicates that explain the infeasibility.
- Execute path symbolically.
- Check conditions for inconsistency using SMT Solver.



Microsoft Research

# Beyond Satisfiability

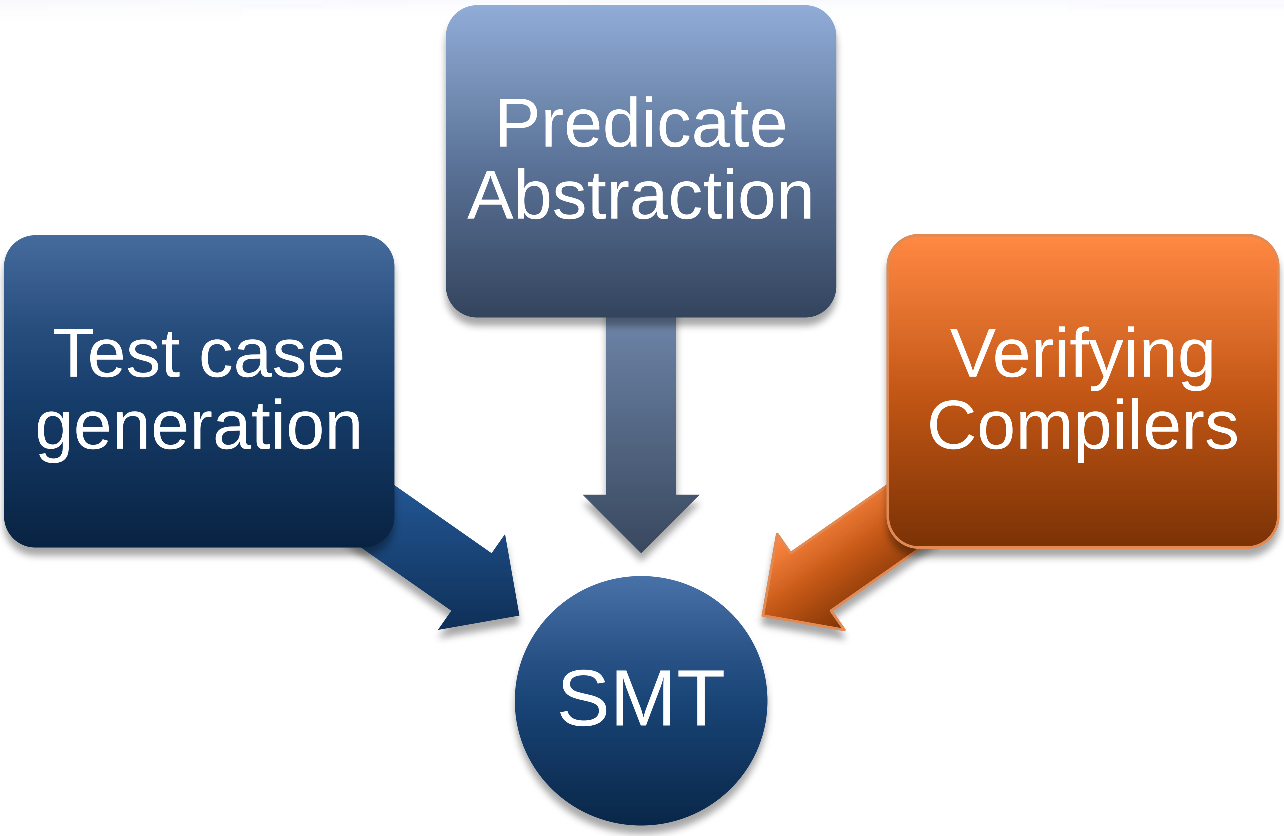
- All-SAT
  - Better (more precise) Predicate Abstraction
- Unsatisfiable cores
  - Why the abstract path is not feasible?
  - Fast Predicate Abstraction
- Interpolants

# Unsatisfiable cores

- Let $S$ be an unsatisfiable set of formulas.
- $S' \subseteq$ S is an unsatisfiable core of S if:
  - $S'$ is also unsatisfiable, and
  - There is no $S'' \subset S'$ that is also unsatisfiable.
- Computing 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:
    - $p_1, \neg p_2, p_3, p_4 \Rightarrow e$
    - $p_1, \neg p_2, p_3, \neg p_4 \Rightarrow e$
    - $p_1, p_2, p_3, \neg p_4 \Rightarrow e$

Microsoft Research

# SMT in Software Verification & Testing






Microsoft Research

# Verifying Compilers

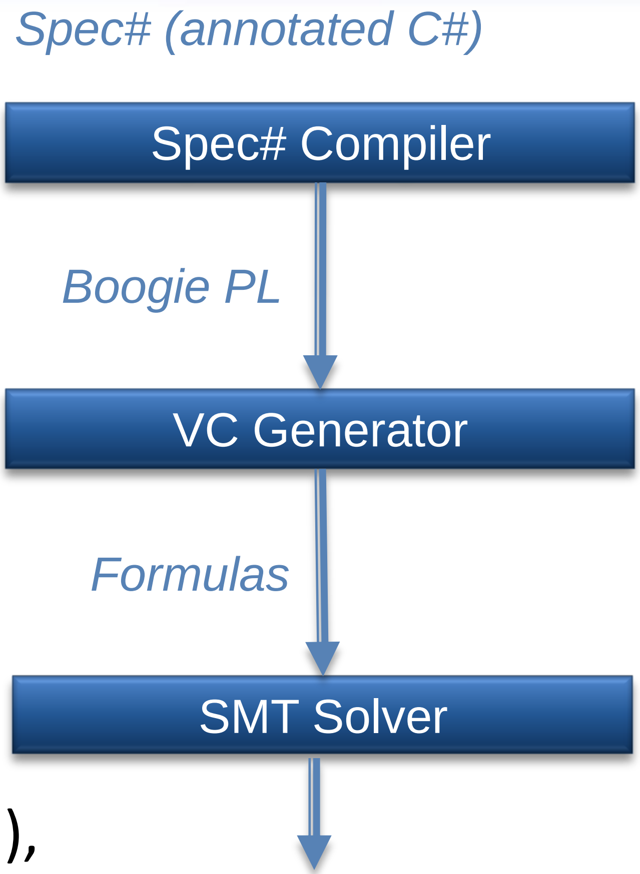
A verifying compiler uses *automated reasoning* to check the correctness of a program that is compiles.

Correctness is specified by *types, assertions, . . . and other redundant annotations* that accompany the program.

Tony Hoare 2004

# 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**




Microsoft Research

# Spec#: Example

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

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

Microsoft Research

# A Verifying C Compiler

- 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.

# Main Challenge

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

$\forall$ h,o,f:

$$\text{IsHeap}(h) \land o \neq \text{null} \land \text{read}(h, o, \text{alloc}) = t$$

$$\Rightarrow$$

$$\text{read}(h,o, f) = \text{null} \lor \text{read}(h, \text{read}(h,o,f),\text{alloc}) = t$$

# Main Challenge

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

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

# Main Challenge

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

$\forall$ i,j: i $\leq$ j $\Rightarrow$ read(a,i) $\leq$ 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

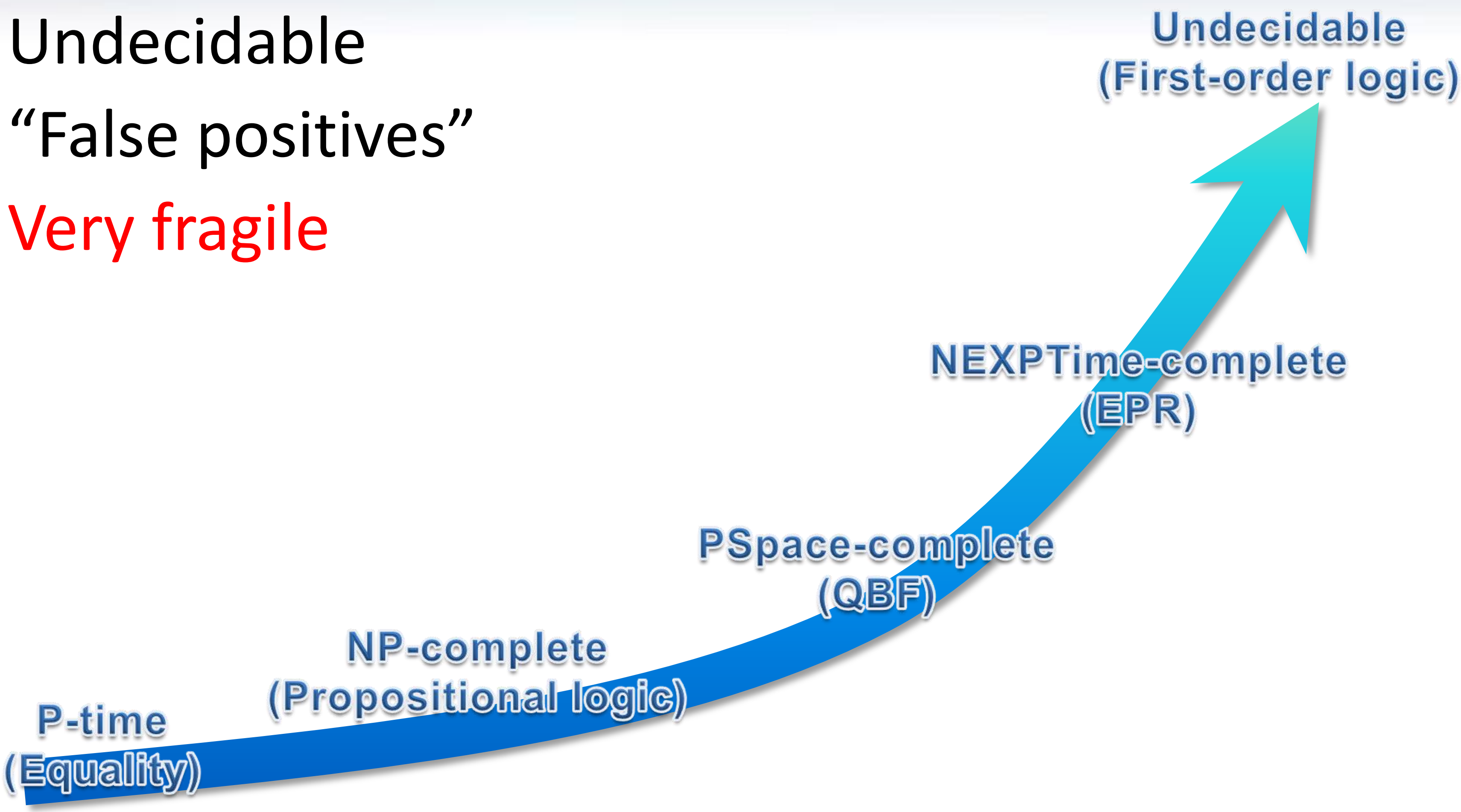
# Main Challenge

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

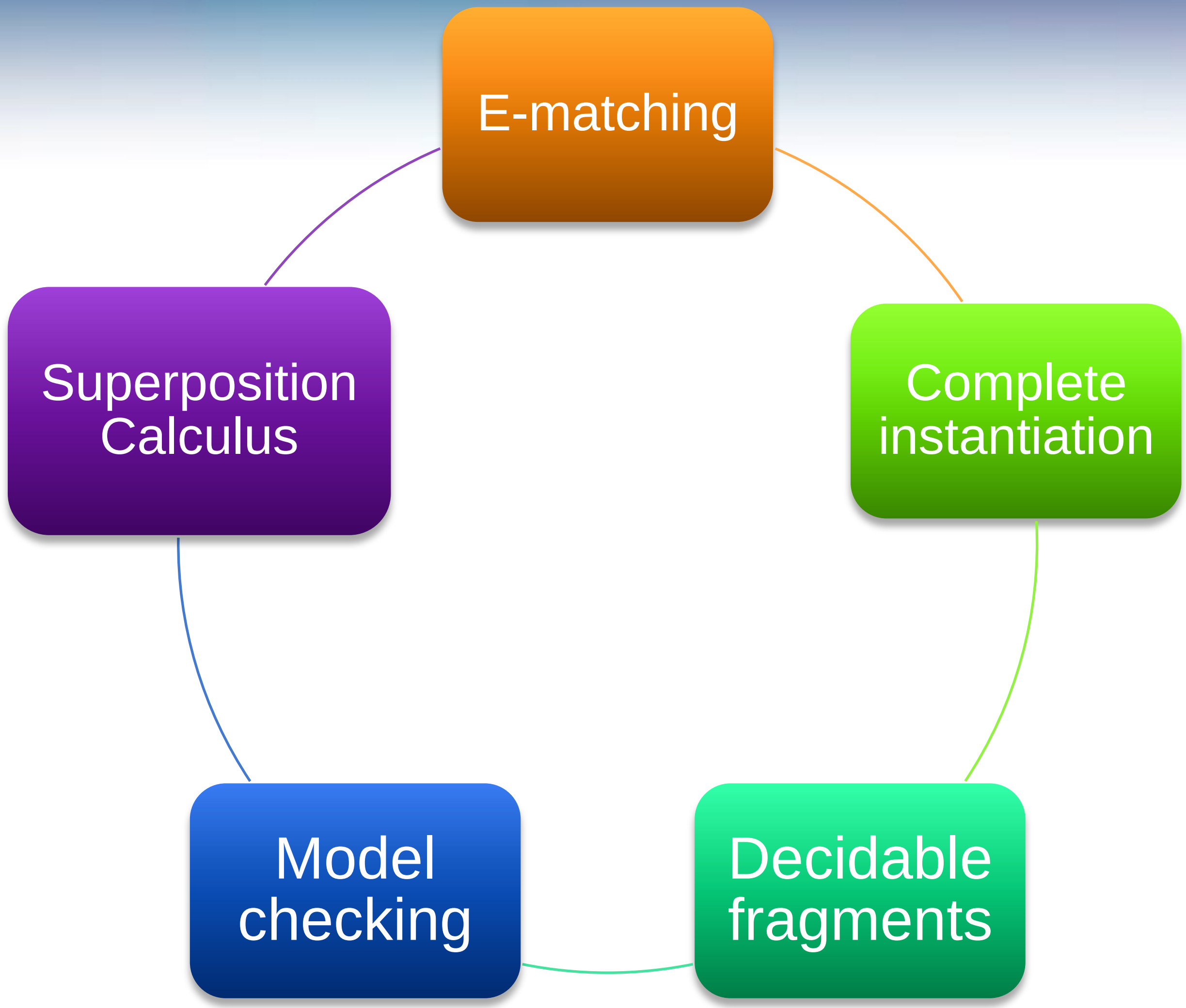
**We want to find bugs!**

# Quantifiers & SMT

- Undecidable
- “False positives”
- Very fragile



# Quantifiers & SMT: approaches



# Conclusion

- SMT is hot at Microsoft (> 15 projects)
- Many applications
  - Cryptography
  - Scheduling
  - Optimization
- Many challenges

**Thank You!**