Satisfiability Modulo Theories solvers in Program Analysis and Verification

Leonardo de Moura and Nikolaj Bjørner
Microsoft Research

Tutorial overview

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

Microsoft Research

SMT Appetizer

Domains from programs

- Bits and bytes $0 = ((x-1) \,\&\, x) \Leftrightarrow x = 00100000..00$
- Arithmetic $x + y = y + x$
- Arrays $read(write(a, i, 4), i) = 4$
- Records $mkpair(x, y) = mkpair(z, u) \Rightarrow x = z$
- Heaps $n \rightarrow^{*} n' \wedge m = cons(a, n) \Rightarrow m \rightarrow^{*} n'$
- Data-types $car(cons(x, nil)) = x$
- Object inheritance $B <: A \wedge C <: B \Rightarrow C <: A$

Satisfiability Modulo Theories (SMT)

$$x+2=y \Rightarrow f(read(write(a,x,3),y-2)) = f(y-x+1)$$

Arithmetic | Arrays | Free Functions

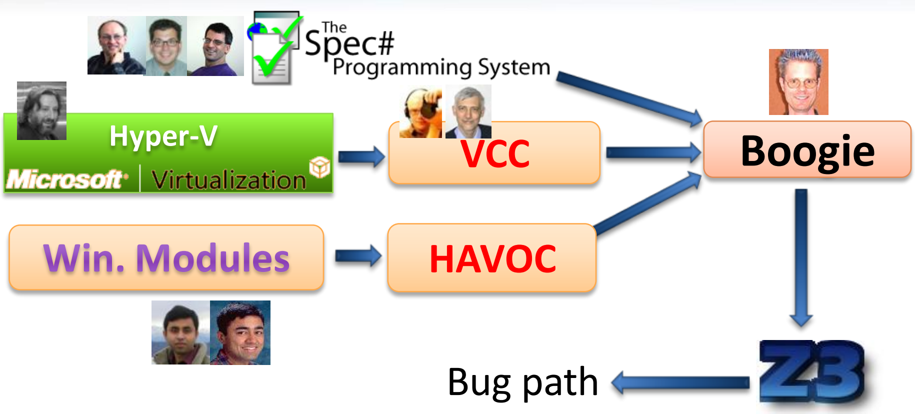
Microsoft Research

Applications Appetizer

Some takeaways from *Applications*

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

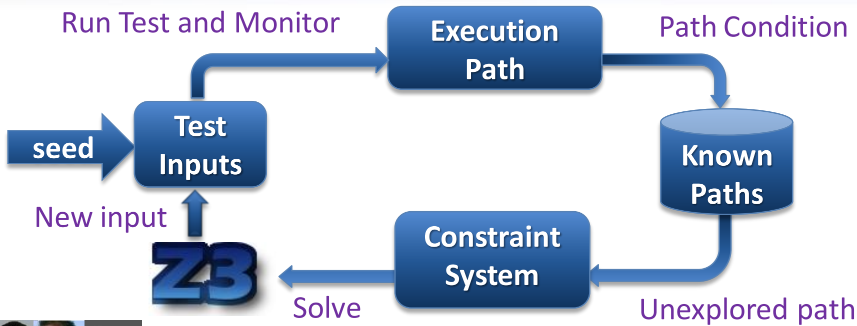
Program Verification

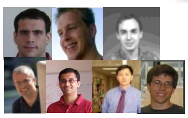


Rustan Leino, Mike Barnet, Michal Moskal, Shaz Qadeer,
Shuvendu Lahiri, Herman Venter, Peter Muller,
Wolfram Schulte, Ernie Cohen

Microsoft Research

Test case generation





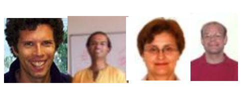
Pex Yogi Vigilante

Nikolai Tillmann, Peli de Halleux, Patrice Godefroid
Aditya Nori, Jean Philippe Martin, Miguel Castro,
Manuel Costa, Lintao Zhang

Static Driver Verifier

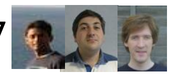
- Z3 is part of SDV 2.0 (Windows 7)
- It is used for:
  - Predicate abstraction (c2bp)
  - Counter-example refinement (newton)

SLAM



Ella Bounimova, Vlad Levin, Jakob Lichtenberg,
Tom Ball, Sriram Rajamani, Byron Cook

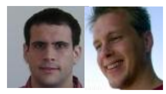
More applications

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

Microsoft Research

Applications

Program Exploration with Pex



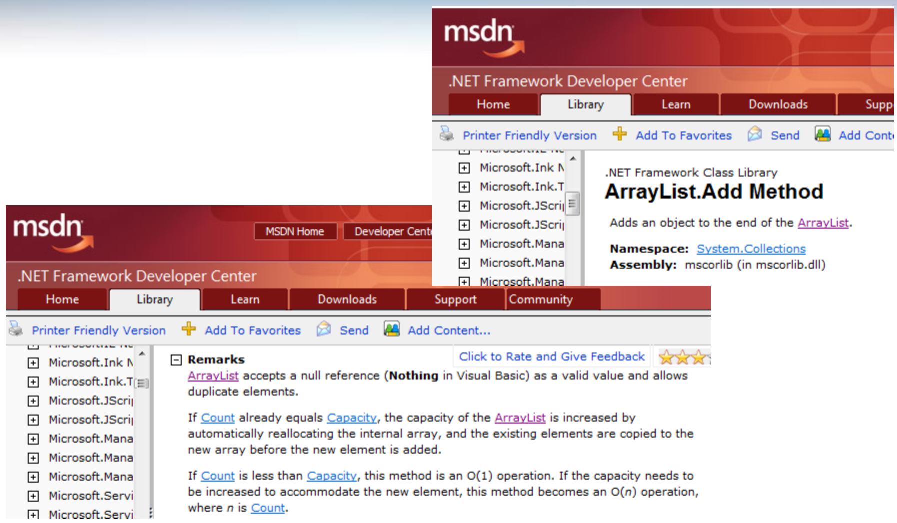
Nikolai Tillmann, Peli de Halleux

http://research.microsoft.com/Pex

What is Pex

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

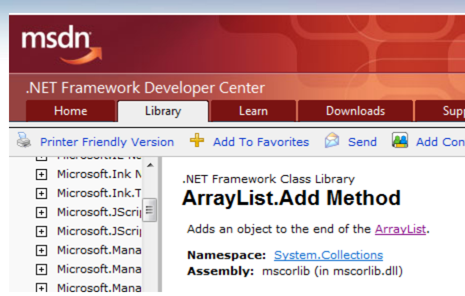
ArrayList: The Spec



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

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)

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

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); }
}
```

```
item == item  →  true
```

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; }
...
```

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); }
}
```

Constraints to solve	Inputs	Observed Constraints
	(0,null)	!(c<0) && 0==c
!(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; }
...
```

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)**	

Z3
Constraint solver

Z3 has decision procedures for
- Arrays
- Linear integer arithmetic
- Bitvector arithmetic
- ...
- (Everything but floating-point numbers)

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); }
}
```

```
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  →  false
      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

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); }
}
```

Constraints to solve	Inputs	Observed Constraints
	(0,null)	!(c<0) && 0==c
!(c<0) && 0!=c	(1,null)	!(c<0) && 0!=c
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; }
...
```

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)	

```
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)

```
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

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; }
...
```

Pex – Test more with less effort

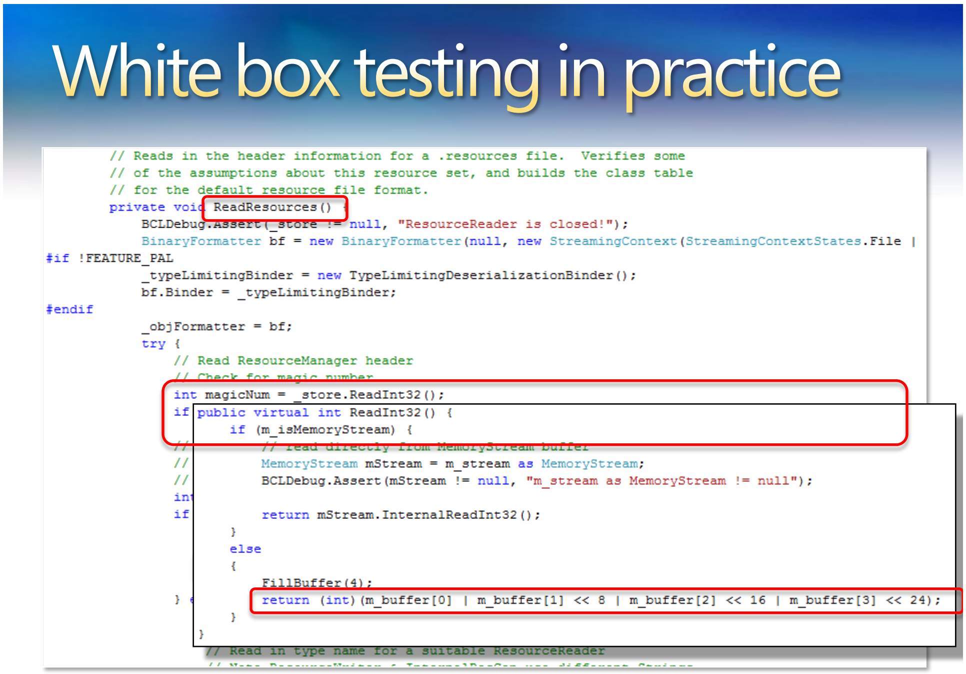
- Reduce testing costs
  - Automated analysis, reproducible results
- Produce more secure software
  - White-box code analysis
- Produce more reliable software
  - Analysis based on contracts written as code

White box testing in practice

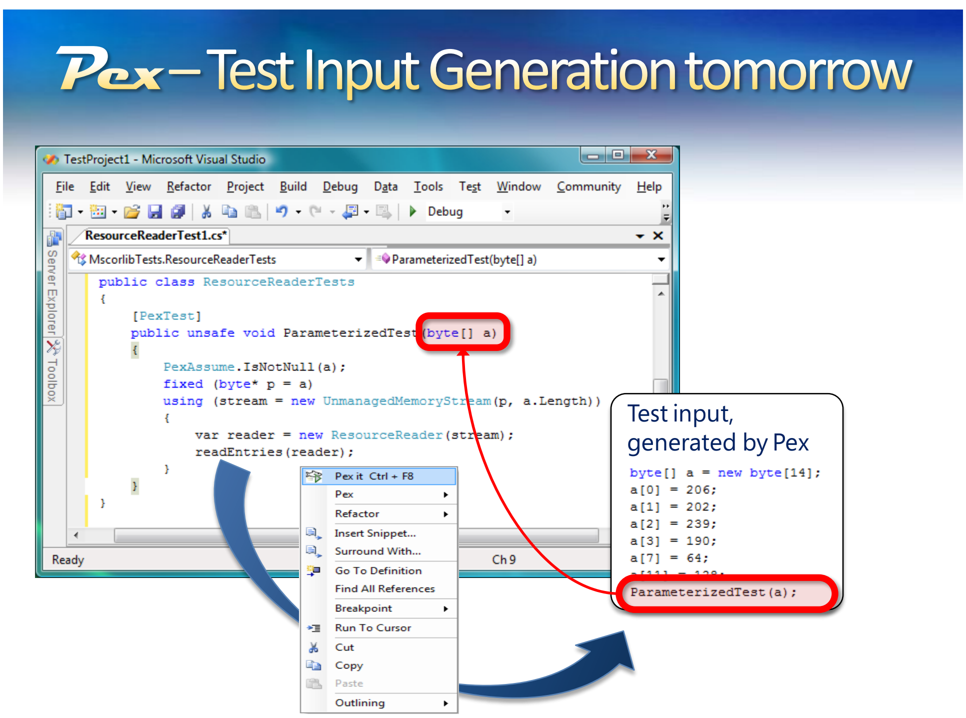
How to test this code?

(Real code from .NET base class libraries.)

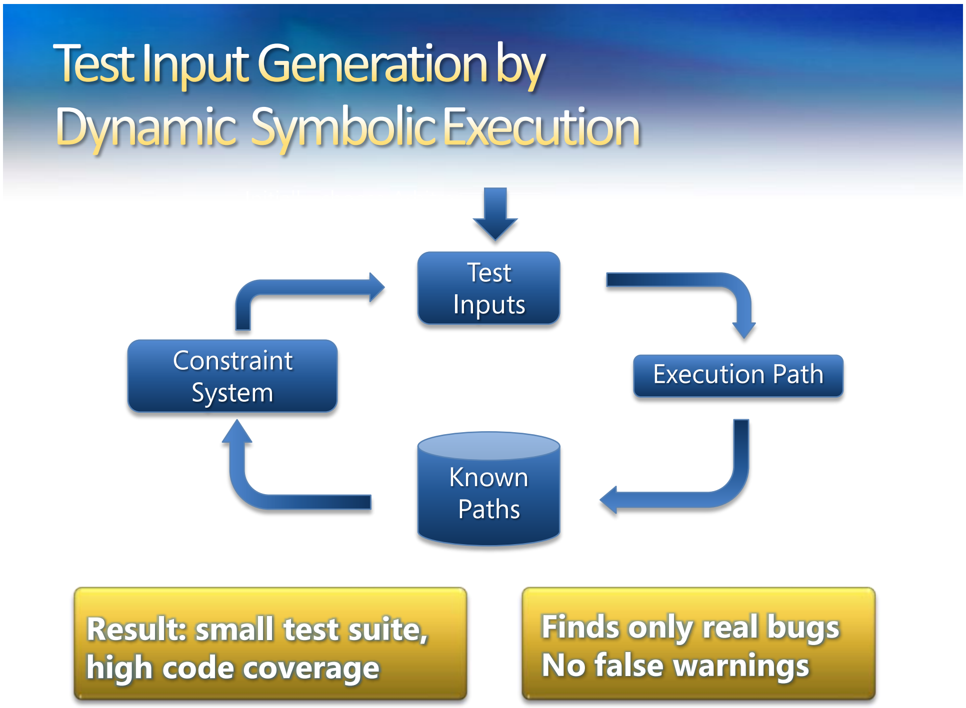
```
[SecurityPermissionAttribute(SecurityAction.LinkDemand, Flags=SecurityPermissionFlag.SerializationFormatter)]
public ResourceReader(Stream stream)
{
    if (stream==null)
        throw new ArgumentNullException("stream");
    if (!stream.CanRead)
        throw new ArgumentException(Environment.GetResourceString("Argument_StreamNotReadable"));

    _resCache = new Dictionary<String, ResourceLocator>(FastResourceComparer.Default);
    _store = new BinaryReader(stream, Encoding.UTF8);
    // We have a faster code path for reading resource files from an assembly.
    _ums = stream as UnmanagedMemoryStream;

    BCLDebug.Log("RESMGRFILEFORMAT", "ResourceReader .ctor(Stream).  UnmanagedMemoryStream: "+(_ums!=null));
    ReadResources();
}
```

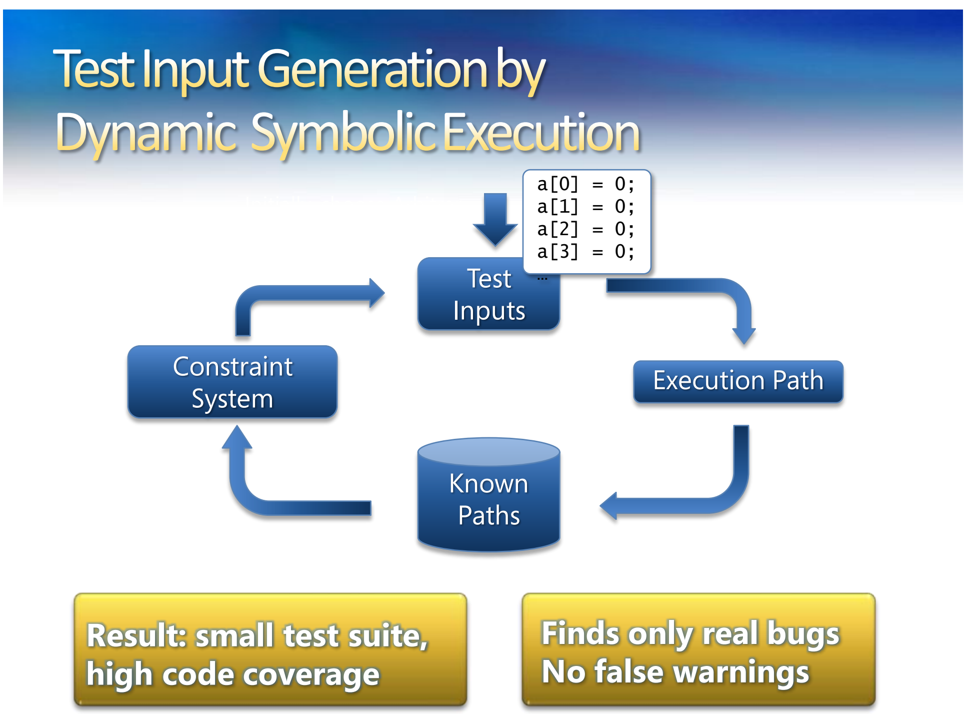


White box testing in practice
// Reads in the header information for a .resources file. Verifies some
// of the assumptions about this resource set, and builds the class table
// for the default resource file format.
private void ReadResources()
BCLDebug.Assert(_store != null, "ResourceReader is closed!");
BinaryFormatter bf = new BinaryFormatter(null, new StreamingContext(StreamingContextStates.File |
#if !FEATURE_PAL
_typeLimitingBinder = new TypeLimitingDeserializationBinder();
bf.Binder = _typeLimitingBinder;
#endif
_objFormatter = bf;
try {
// Read ResourceManager header
int magicNum = _store.ReadInt32();
public virtual int ReadInt32() {
if (m_isMemoryStream) {
MemoryStream mStream = m_stream as MemoryStream;
BCLDebug.Assert(mStream != null, "m_stream as MemoryStream != null");
return mStream.InternalReadInt32();
}
else
{
FillBuffer(4);
return (int)(m_buffer[0] | m_buffer[1] << 8 | m_buffer[2] << 16 | m_buffer[3] << 24);
}
}

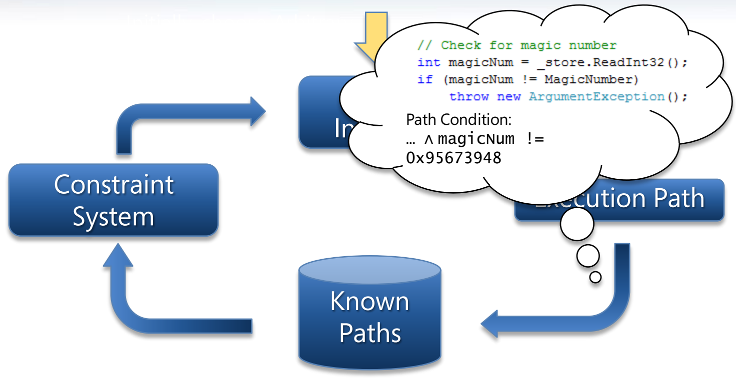


Pex – Test Input Generation tomorrow
TestProject1 - Microsoft Visual Studio
File Edit View Refactor Project Build Debug Data Tools Test Window Community Help
Debug
ResourceReaderTest1.cs*
MscorlibTests.ResourceReaderTests
ParameterizedTest(byte[] a)
public class ResourceReaderTests
{
[PexTest]
public unsafe void ParameterizedTest(byte[] a)
{
PexAssume.IsNotNull(a);
fixed (byte* p = a)
using (stream = new UnmanagedMemoryStream(p, a.Length))
{
var reader = new ResourceReader(stream);
readEntries(reader);
}
}
}
Ready
Ch 9
Pex it Ctrl + F8
Pex
Refactor
Insert Snippet...
Surround With...
Go To Definition
Find All References
Breakpoint
Run To Cursor
Cut
Copy
Paste
Outlining
Test input, generated by Pex
byte[] a = new byte[14];
a[0] = 206;
a[1] = 202;
a[2] = 239;
a[3] = 190;
a[7] = 64;
ParameterizedTest(a);


Test Input Generation by
Dynamic Symbolic Execution
Test
Inputs
Constraint
System
Execution Path
Known
Paths
Result: small test suite,
high code coverage
Finds only real bugs
No false warnings


Test Input Generation by
Dynamic Symbolic Execution
a[0] = 0;
a[1] = 0;
a[2] = 0;
a[3] = 0;
…
Test
Inputs
Constraint
System
Execution Path
Known
Paths
Result: small test suite,
high code coverage
Finds only real bugs
No false warnings

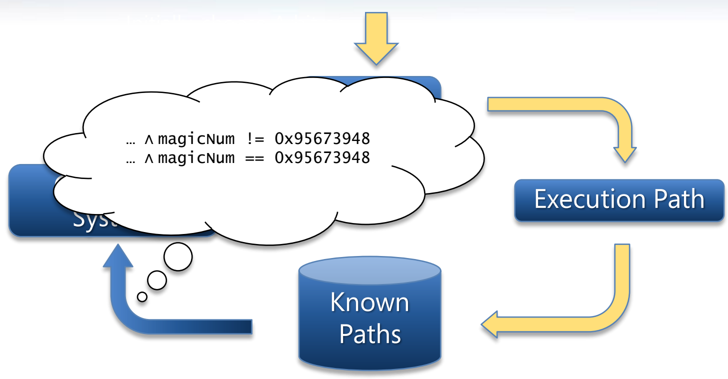
Test Input Generation by Dynamic Symbolic Execution



Result: small test suite, high code coverage

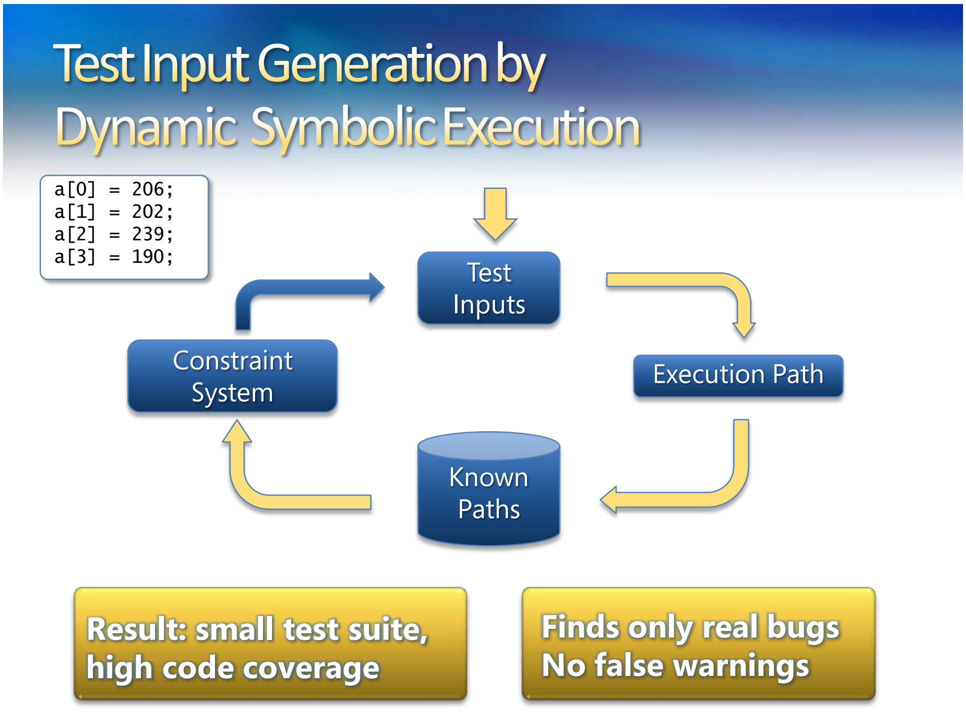
Finds only real bugs
No false warnings

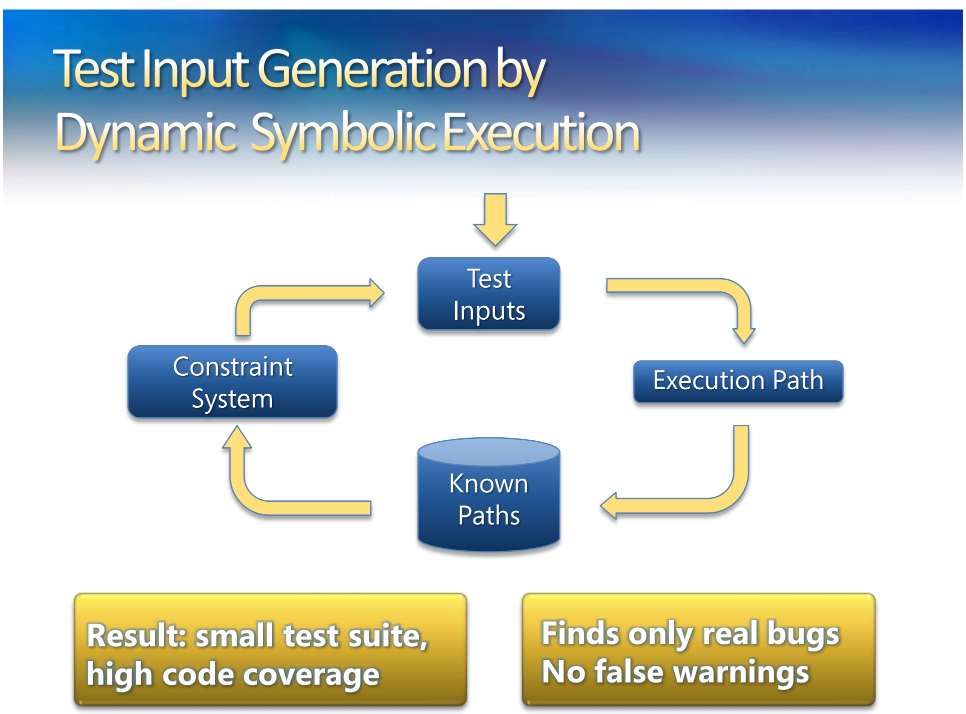
Test Input Generation by Dynamic Symbolic Execution



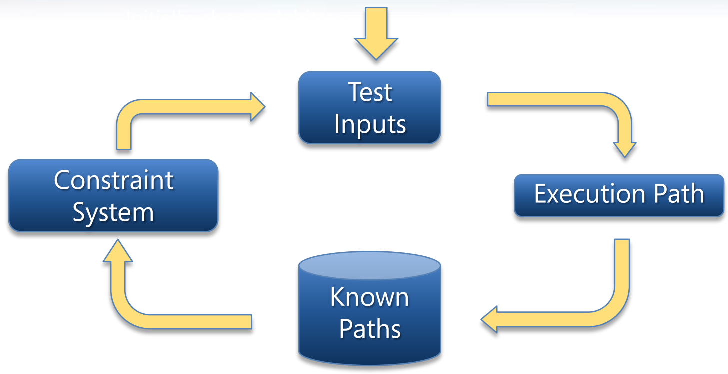
Result: small test suite, high code coverage

Finds only real bugs
No false warnings


Test Input Generation by
Dynamic Symbolic Execution
a[0] = 206;
a[1] = 202;
a[2] = 239;
a[3] = 190;
Test Inputs
Constraint System
Execution Path
Known Paths
Result: small test suite, high code coverage
Finds only real bugs
No false warnings


Test Input Generation by
Dynamic Symbolic Execution
Test Inputs
Constraint System
Execution Path
Known Paths
Result: small test suite, high code coverage
Finds only real bugs
No false warnings

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



Result: small test suite, high code coverage

Finds only real bugs
No false warnings

Constraint Solving: Preprocessing

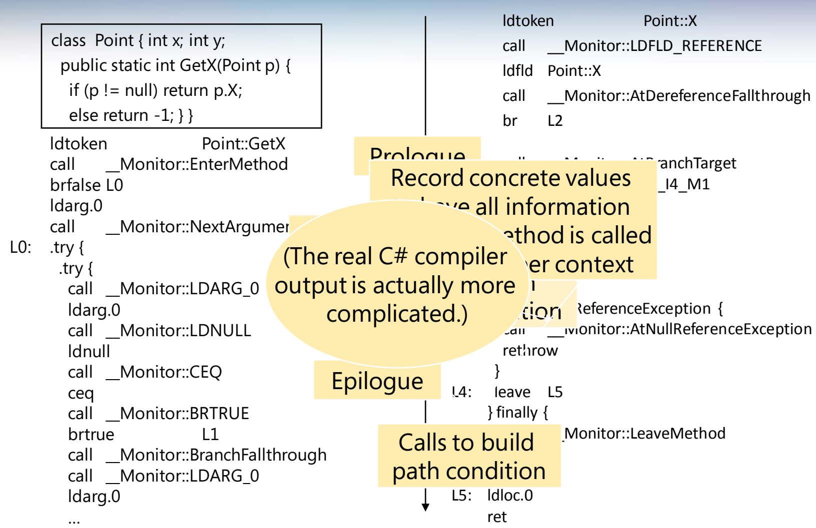
Independent constraint optimization + Constraint caching (similar to EXE)

- Idea: Related execution paths give rise to "similar" constraint systems
- Example: Consider x>y ∧ z>0 vs. x>y ∧ z<=0
- If we already have a cached solution for a "similar" constraint system, we can reuse it
  - x=1, y=0, z=1 is solution for x>y ∧ z>0
  - we can obtain a solution for x>y ∧ z<=0 by
    - reusing old solution of x>y: x=1, y=0
    - combining with solution of z<=0: z=0

Constraint Solving: Z3

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

Monitoring by Code Instrumentation



Spec# and Boogie


Rustan Leino & Mike Barnett

Verifying Compilers

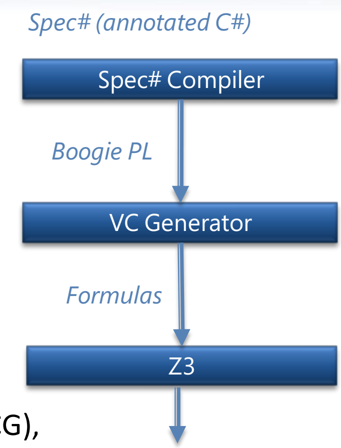
A verifying compiler uses *automated reasoning* to check the correctness of a program that is compiles.

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),
  - automatic theorem proving Z3



Microsoft Research

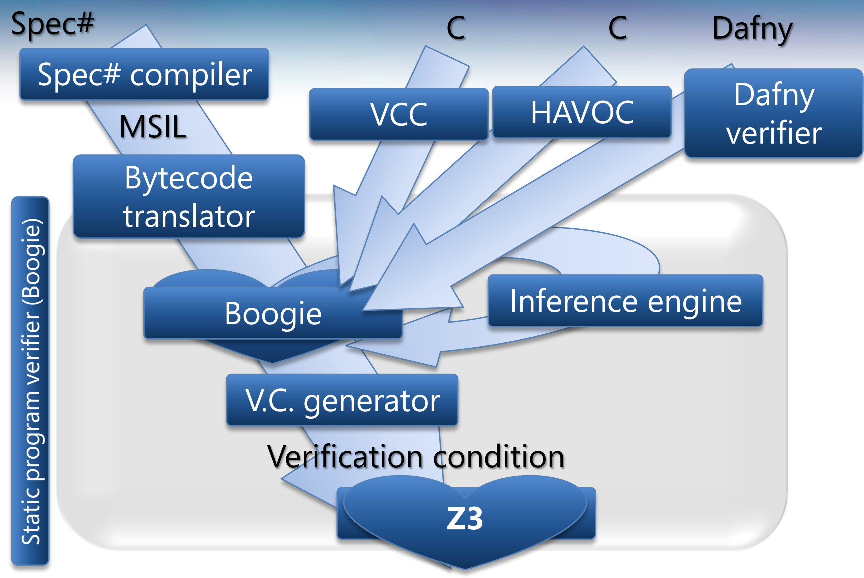
Basic verifier architecture

Source language

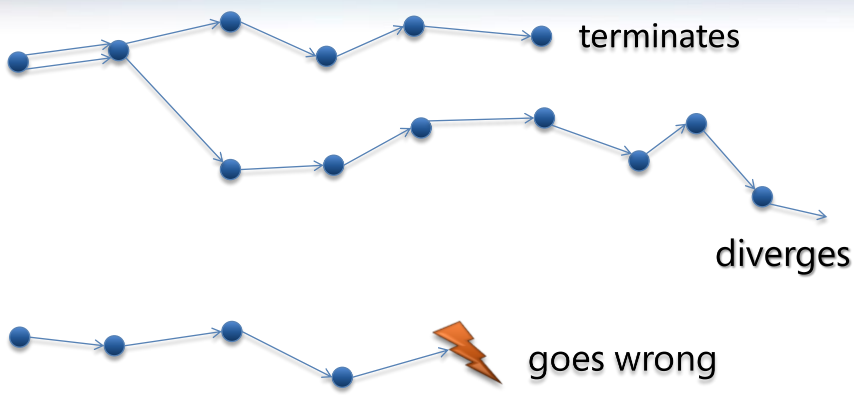
Intermediate verification language

Verification condition
(logical formula)

Verification architecture

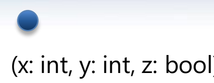


Modeling execution traces

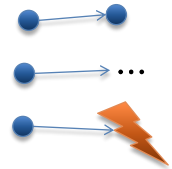


States and execution traces

- 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

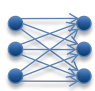


Command language

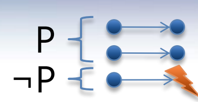
- x := E
  - x := x + 1

  - x := 10

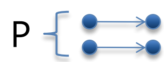
- havoc x



- assert P



- assume P

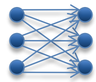


Command language

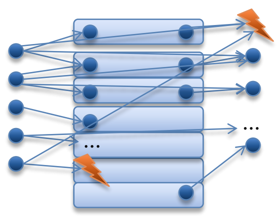
- x := E
  - x := x + 1

  - x := 10

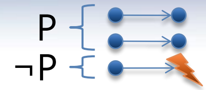
- havoc x



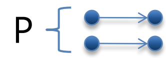
- S ; T



- assert P



- assume P

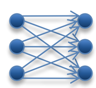


Command language

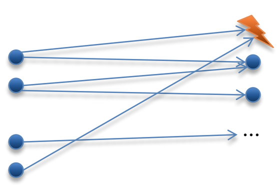
- x := E
  - x := x + 1

  - x := 10

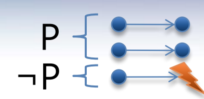
- havoc x



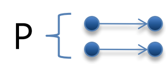
- S ; T



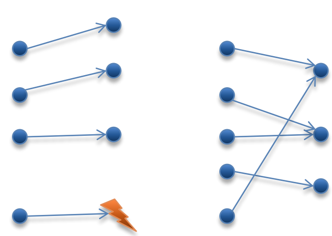
- assert P



- assume P



- S □ T



Reasoning about execution traces

- Hoare triple { P } S { Q } says that

  every terminating execution trace of S that starts in a state satisfying P
  - does not go wrong, and
  - terminates in a state satisfying Q

Reasoning about execution traces

- Hoare triple { P } S { Q } says that

  every terminating execution trace of S that starts in a state satisfying P
  - does not go wrong, and
  - terminates in a state satisfying Q
- Given S and Q, what is the weakest P’ satisfying {P’} S {Q} ?
  - P' is called the *weakest precondition* of S with respect to Q, written wp(S, Q)
  - to check {P} S {Q}, check $P \Rightarrow P'$

Weakest preconditions

- wp(x := E, Q) = Q[E / x]
- wp(havoc x, Q) = $(\forall x \bullet Q)$
- wp(assert P, Q) = $P \wedge Q$
- wp(assume P, Q) = $P \Rightarrow Q$
- wp(S ; T, Q) = wp(S, wp(T, Q))
- wp(S □ T, Q) = wp(S, Q) $\wedge$ wp(T, Q)

Structured if statement

if E then S else T end =

assume E; S

□

assume $\neg$E; T

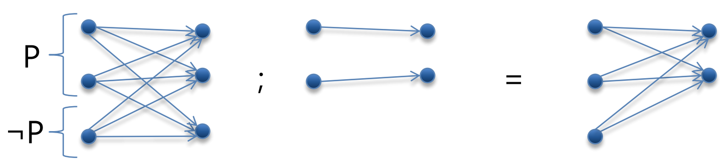
Dijkstra's guarded command

if E → S | F → T fi =

assert E ∨ F;
(
  assume E; S
  □
  assume F; T
)

Picking any good value

assign x such that P =
  havoc x; assume P



assign x such that x*x = y

Procedures

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

Procedure example

- procedure Inc(n) returns (b)
  requires 0 ≤ n
  modifies g
  ensures g = old(g) + n

Procedures

- A *procedure* is a user-defined command
- procedure M(x, y, z) returns (r, s, t)
  requires P
  modifies g, h
  ensures Q
- call a, b, c := M(E, F, G)
  = x′ := E; y′ := F; z′ := G;
  assert P′;
  g0 := g; h0 := h;
  havoc g, h, r′, s′, t′;
  assume Q′;
  a := r′; b := s′; c := t′

where
- x′, y′, z′, r′, s′, t′, g0, h0 are fresh names
- P′ is P with x′,y′,z′ for x,y,z
- Q′ is Q with x′,y′,z′,r′,s′,t′,g0,h0 for x,y,z,r,s,t, old(g), old(h)

Microsoft Research

Procedure implementations

- procedure M(x, y, z) returns (r, s, t)
  requires P
  modifies g, h
  ensures Q
- implementation M(x, y, z) returns (r, s, t) is S
  = assume P;
  g0 := g; h0 := h;
  S;
  assert Q′

where
- g0, h0 are fresh names
- Q′ is Q with g0,h0 for old(g), old(h)

syntactically check that S assigns only to g,h

Microsoft Research

While loop with loop invariant

```
while E
    invariant J
do
    S
end
```

where x denotes the assignment targets of S

```
=   assert J;                                      ← check that the loop invariant holds initially
    havoc x;  assume J;                            } "fast forward" to an arbitrary iteration of the loop
    (   assume E;  S;  assert J;  assume false
    □   assume ¬E
    )
```

↑ check that the loop invariant is maintained by the loop body

Microsoft Research

Properties of the heap

- introduce:

  axiom (∀ h: HeapType, o: Ref, f: Field Ref •
      o ≠ null ∧ h[o, alloc]
      ⇒
      h[o, f] = null ∨ h[h[o,f], alloc]);

Properties of the heap

- introduce:

  function IsHeap(HeapType) returns (bool);
- introduce:

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

Methods

- method M(x: X) returns (y: Y)
  requires P; modifies S; ensures Q;
  { Stmt }
- procedure M(this: Ref, x: Ref) returns (y: Ref);
  free requires IsHeap(Heap);
  free requires this ≠ null ∧ Heap[this, alloc];
  free requires x = null ∨ Heap[x, alloc];
  requires Df[[P]] ∧ Tr[[P]];
  requires Df[[S]];
  modifies Heap;
  ensures Df[[Q]] ∧ Tr[[Q]];
  ensures (∀⟨α⟩ o: Ref, f: Field α •
  o ≠ null ∧ old(Heap)[o,alloc] ⇒
  Heap[o,f] = old(Heap)[o,f] ∨
  (o,f) ∈ old(Tr[[S]]));
  free ensures IsHeap(Heap);
  free ensures y = null ∨ Heap[y, alloc];
  free ensures (∀o: Ref • old(Heap)[o,alloc] ⇒ Heap[o,alloc]);

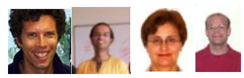
Spec# Chunker.NextChunk translation

```
procedure Chunker.NextChunk(this: ref where $IsNotNull(this, Chunker)) returns ($result: ref where $IsNotNull($result, System.String));
  // in-parameter:  target object
  free requires $Heap[this, $allocated];
  requires ($Heap[this, $ownerFrame] == $PeerGroupPlaceholder || !($Heap[$Heap[this, $ownerRef], $inv] <: $Heap[this, $ownerFrame]) ||
     $Heap[$Heap[this, $ownerRef], $localinv] == $BaseClass($Heap[this, $ownerFrame])) && (forall $pc: ref :: $pc != null && $Heap[$pc, $allocated]
     && $Heap[$pc, $ownerRef] == $Heap[this, $ownerRef] && $Heap[$pc, $ownerFrame] == $Heap[this, $ownerFrame] ==> $Heap[$pc, $inv] ==
     $typeof($pc) && $Heap[$pc, $localinv] == $typeof($pc));
  // out-parameter:  return value
  free ensures $Heap[$result, $allocated];
  ensures ($Heap[$result, $ownerFrame] == $PeerGroupPlaceholder || !($Heap[$Heap[$result, $ownerRef], $inv] <: $Heap[$result, $ownerFrame]) ||
     $Heap[$Heap[$result, $ownerRef], $localinv] == $BaseClass($Heap[$result, $ownerFrame])) && (forall $pc: ref :: $pc != null && $Heap[$pc,
     $allocated] && $Heap[$pc, $ownerRef] == $Heap[$result, $ownerRef] && $Heap[$pc, $ownerFrame] == $Heap[$result, $ownerFrame] ==>
     $Heap[$pc, $inv] == $typeof($pc) && $Heap[$pc, $localinv] == $typeof($pc));
  // user-declared postconditions
  ensures $StringLength($result) <= $Heap[this, Chunker.ChunkSize];
  // frame condition
  modifies $Heap;
  free ensures (forall $o: ref, $f: name :: { $Heap[$o, $f] } $f != $inv && $f != $localinv && $f != $FirstConsistentOwner && (!IsStaticField($f) ||
     !IsDirectlyModifiableField($f)) && $o != null && old($Heap)[$o, $allocated] && (old($Heap)[$o, $ownerFrame] == $PeerGroupPlaceholder ||
     !(old($Heap)[old($Heap)[$o, $ownerRef], $inv] <: old($Heap)[$o, $ownerFrame]) || old($Heap)[old($Heap)[$o, $ownerRef], $localinv] ==
     $BaseClass(old($Heap)[$o, $ownerFrame])) && old($o != this || !(Chunker <: DeclType($f)) || !$IncludedInModifiesStar($f)) && old($o != this || $f
     != $exposeVersion) ==> old($Heap)[$o, $f] == $Heap[$o, $f]);
  // boilerplate
  free requires $BeingConstructed == null;
  free ensures (forall $o: ref :: { $Heap[$o, $localinv] } { $Heap[$o, $inv] } $o != null && !old($Heap)[$o, $allocated] && $Heap[$o, $allocated] ==>
     $Heap[$o, $inv] == $typeof($o) && $Heap[$o, $localinv] == $typeof($o));
  free ensures (forall $o: ref :: { $Heap[$o, $FirstConsistentOwner] } old($Heap)[old($Heap)[$o, $FirstConsistentOwner], $exposeVersion] ==
     $Heap[old($Heap)[$o, $FirstConsistentOwner], $exposeVersion] ==> old($Heap)[$o, $FirstConsistentOwner] == $Heap[$o,
     $FirstConsistentOwner]);
  free ensures (forall $o: ref :: { $Heap[$o, $localinv] } { $Heap[$o, $inv] } old($Heap)[$o, $allocated] ==> old($Heap)[$o, $inv] == $Heap[$o, $inv] &&
     old($Heap)[$o, $localinv] == $Heap[$o, $localinv]);
  free ensures (forall $o: ref :: { $Heap[$o, $allocated] } old($Heap)[$o, $allocated] ==> $Heap[$o, $allocated]) && (forall $ot: ref :: { $Heap[$ot,
     $ownerFrame] } { $Heap[$ot, $ownerRef] } old($Heap)[$ot, $allocated] && old($Heap)[$ot, $ownerFrame] != $PeerGroupPlaceholder ==>
     old($Heap)[$ot, $ownerRef] == $Heap[$ot, $ownerRef] && old($Heap)[$ot, $ownerFrame] == $Heap[$ot, $ownerFrame]) &&
     old($Heap)[$BeingConstructed, $NonNullFieldsAreInitialized]  == $Heap[$BeingConstructed, $NonNullFieldsAreInitialized];
```

Z3 & Program Verification

- Quantifiers, quantifiers, quantifiers, …
  - Modeling the runtime
  - Frame axioms ("what didn't change")
  - Users provided assertions (e.g., the array is sorted)
  - Prototyping decision procedures (e.g., reachability, heaps, …)
- *Solver must be fast in satisfiable instances.*
- Trade-off between precision and performance.
- *Candidate (Potential) Models*

The Static Driver Verifier
SLAM



Ella Bounimova, Vlad Levin, Jakob Lichtenberg,
Tom Ball, Sriram Rajamani, Byron Cook

Overview

- *http://research.microsoft.com/slam/*
- *SLAM/SDV* is a software model checker.
- Application domain: *device drivers.*
- Architecture:

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

  bebop Model checker for boolean programs.

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

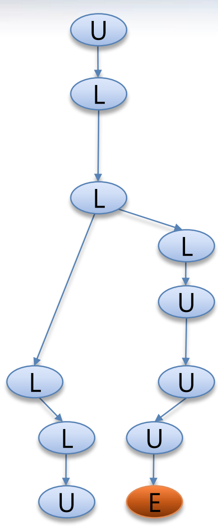
Example

Do this code obey the looking rule?

```
do {
  KeAcquireSpinLock();

  nPacketsOld = nPackets;

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

KeReleaseSpinLock();
```

Example

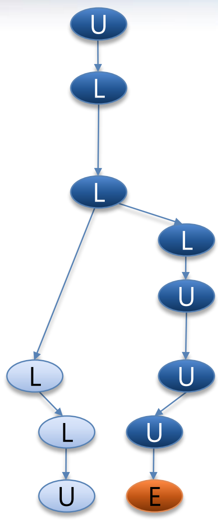
Model checking Boolean program




```
do {
  KeAcquireSpinLock();



  if(*){

      KeReleaseSpinLock();

  }
} while (*);

KeReleaseSpinLock();
```

Example

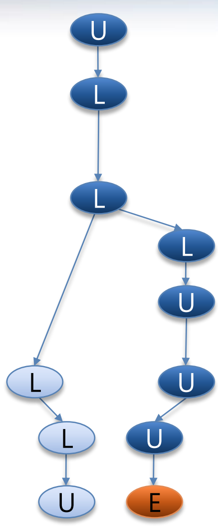
Is error path feasible?




```
do {
  KeAcquireSpinLock();

  nPacketsOld = nPackets;

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

KeReleaseSpinLock();
```

Example

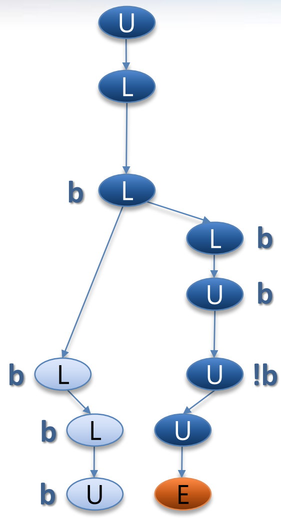
Add new predicate to Boolean program
b: (nPacketsOld == nPackets)




```
do {
  KeAcquireSpinLock();

  b = true;

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

KeReleaseSpinLock();
```

Example

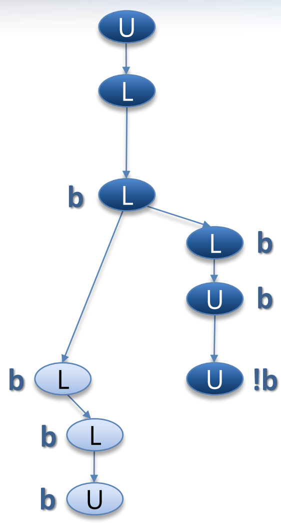
Model Checking
Refined Program
b: (nPacketsOld == nPackets)




```
do {
   KeAcquireSpinLock();

   b = true;

   if(*){

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

KeReleaseSpinLock();
```

Example

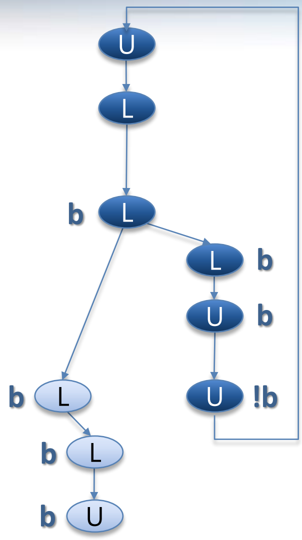
Model Checking
Refined Program
b: (nPacketsOld == nPackets)




```
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();
```

Observations about SLAM

- Automatic discovery of invariants
  - driven by property and a finite set of (false) execution paths
  - predicates are ***not*** invariants, but *observations*
  - abstraction + model checking computes inductive invariants (boolean combinations of observations)
- A hybrid dynamic/static analysis
  - newton executes path through C code symbolically
  - c2bp+bebop explore all paths through abstraction
- A new form of program slicing
  - program code and data not relevant to property are dropped
  - non-determinism allows slices to have more behaviors

Syntatic Sugar

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

```
goto L1, L2;

    L1:  assume(e);
    S1;
    goto L3;

    L2: assume(!e);
    S2;
    goto L3;

L3: S3;
```

Predicate Abstraction: *c2bp*

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

Abstracting Assignments via WP

- Statement y=y+1 and F={ y<4, y<5 }
  - {y<4}, {y<5} = ((!{y<5} || !{y<4}) ? false : *), {y<4})

- WP(x=e,Q) = Q[e/x]
- WP(y=y+1, y<5) =

  (y<5) [y+1/y] =

  (y+1<5) =

  (y<4)

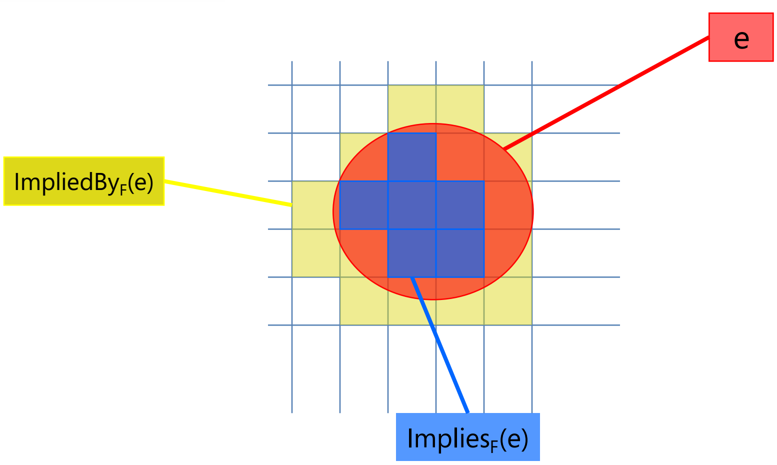
WP Problem

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

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)*

$Implies_F$(e) and $ImpliedBy_F$(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 ∧ x=2

***Implies$_F$*(y>1)** = $b_1 \wedge b_2$

Abstracting Assignments

- if $Implies_F(WP(s, p_i))$ is true before s then
  - p_i is true after s

- if $Implies_F(WP(s, !p_i))$ is true before s then
  - p_i is false after s

$\{p_i\}$ = $Implies_F(WP(s, p_i))$? true :
$Implies_F(WP(s, !p_i))$? false
: *;

Assignment Example

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

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

$Implies_F$(x==y+1) = false
$Implies_F$(x!=y+1) = x==y

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

Abstracting Assumes

- WP(assume(e), Q) = e implies Q
- assume(e) is abstracted to:

  assume(ImpliedBy_F(e))
- Example:

  F = {x==2, x<5}

  assume(x < 2) is abstracted to:

  assume(!{x==2} && {x<5})

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.

Z3 & Static Driver Verifier

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

Microsoft Research

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 not $S'' \subset S'$ 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

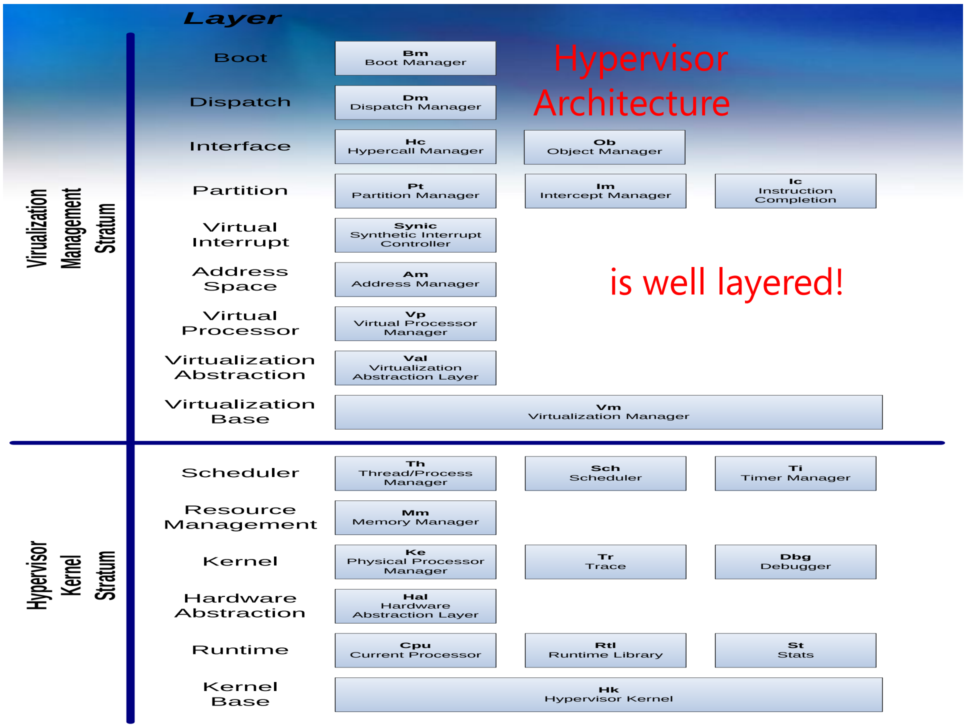
Microsoft Research

A Verifying C Compiler

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

Microsoft Hypervisor

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



What is to be verified?

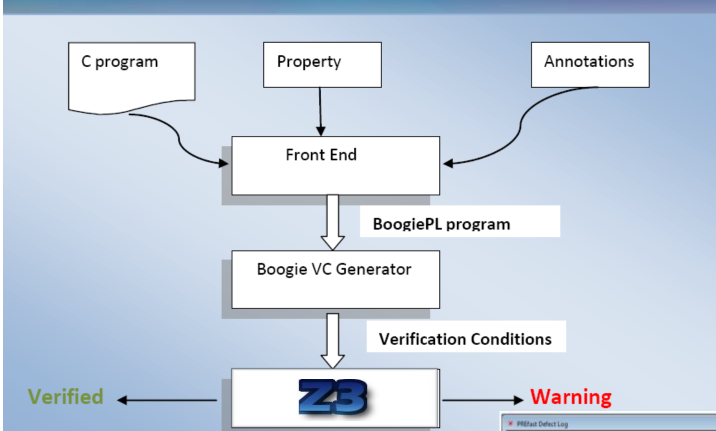
- Source code
  - C + x64 assembly

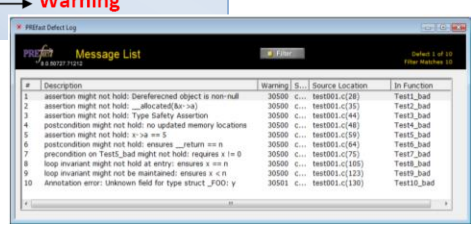
- Sample verifiable slices:
  - **Safety**: Basic memory safety
  - **Functionality**: Hypervisor simulates a number of virtual x64 machines.
  - **Utility**: Hypervisor services guest OS with available resources.

HAVOC
Verifying Windows Components

Lahiri & Qadeer, POPL'08,
Also: Ball, Hackett, Lahiri, Qadeer, MSR-TR-08-82.

HAVOC's Architecture

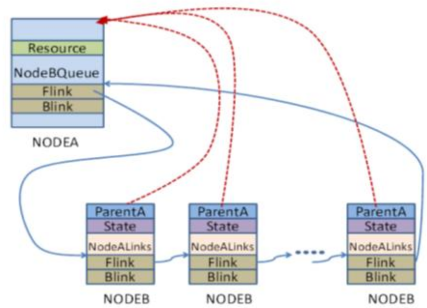




Heaps and Shapes

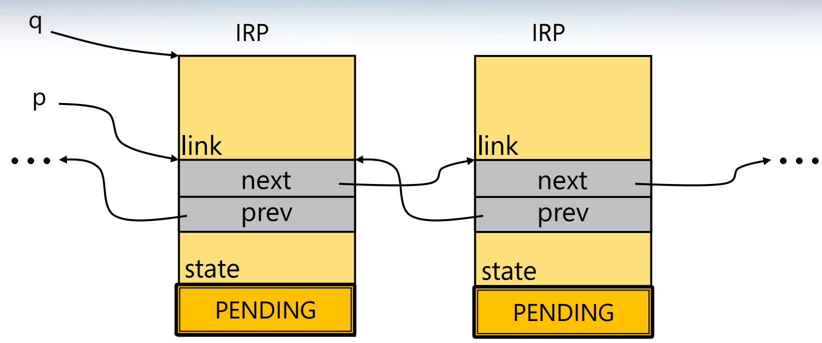
```
typedef struct _LIST_ENTRY{
  struct _LIST_ENTRY *Flink, *Blink;
} LIST_ENTRY, *PLIST_ENTRY;

typedef struct _NODEA{
  PERESOURCE Resource;
  LIST_ENTRY NodeBQueue;
  ...
} NODEA, *PNODEA;

typedef struct _NODEB{
  PNODEA     ParentA;
  ULONG State;
  LIST_ENTRY NodeALinks;
  ...
} NODEB, *PNODEB;

#define CONTAINING_RECORD(addr, type, field)\
   ((type *)((PCHAR)(addr) -\
             (PCHAR)(&((type *)0)->field)))
```

Doubly linked lists in Windows Kernel code



Representative shape graph
in Windows Kernel component

Precise and expressive heap reasoning



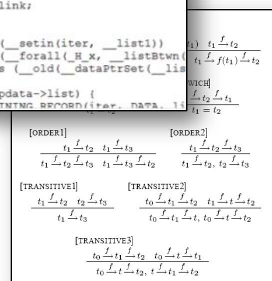
- Pointer Arithmetic
  q = CONTAINING_RECORD(p, IRP, link)
  = (IRP *) ((char*)p – (char*)(&(((IRP *)0)→link)))
- Transitive Closure
  Reach(next, u) ≡ {u, u->next, u->next->next, …}
  forall (x, Reach(next,p), CONTAINING_RECORD(x, IRP, link)->state == PENDING)

Annotation Language & Logic

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

```
__requires (NodeA != NULL)
..
__ensures  ((*PNodeB)->ParentA == NodeA)
__modifies (PNodeB)
void CompCreateNodeB
    (PNODEA NodeA,  PNODEB *PNodeB);
```

```
__requires (__forall(_H_x, __list1, __dataPtr(_H_
__requires (__setin(__head, __list1))
__ensures (__forall(_H_x, __list2, __initializedD
__modifies (__dataPtrSet(__list1))

void InitializeList() {
  LIST_ENTRY *iter;

  iter = pdata->list.Flink;

  __loop_invariant(
        __loop_assert (__setin(iter, __list1))
        __loop_assert (__forall(_H_x, __listBtwn(
        __loop_modifies (__old(__dataPtrSet(__lis
              )
     while (iter != &pdata->list) {
   PDATA elem = CONTAINING_RECORD(iter, DATA, li
```



Efficient logic for program verification

[REFLEXIVE]
$$\frac{}{t \xrightarrow{f} t}$$

[STEP]
$$\frac{f(t)}{t \xrightarrow{f} f(t)}$$

[REACH]
$$\frac{f(t_1) \quad t_1 \xrightarrow{f} t_2}{t_1 = t_2 \quad t_1 \xrightarrow{f} f(t_1) \xrightarrow{f} t_2}$$

[CYCLE]
$$\frac{f(t_1) = t_1 \quad t_1 \xrightarrow{f} t_2}{t_1 = t_2}$$

[SANDWICH]
$$\frac{t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_1}{t_1 = t_2}$$

[ORDER1]
$$\frac{t_1 \xrightarrow{f} t_2 \quad t_1 \xrightarrow{f} t_3}{t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3 \quad t_1 \xrightarrow{f} t_3 \xrightarrow{f} t_2}$$

[ORDER2]
$$\frac{t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3}{t_1 \xrightarrow{f} t_2,\ t_2 \xrightarrow{f} t_3}$$

[TRANSITIVE1]
$$\frac{t_1 \xrightarrow{f} t_2 \quad t_2 \xrightarrow{f} t_3}{t_1 \xrightarrow{f} t_3}$$

[TRANSITIVE2]
$$\frac{t_0 \xrightarrow{f} t_1 \xrightarrow{f} t_2 \quad t_1 \xrightarrow{f} t \xrightarrow{f} t_2}{t_0 \xrightarrow{f} t_1 \xrightarrow{f} t,\ t_0 \xrightarrow{f} t \xrightarrow{f} t_2}$$

[TRANSITIVE3]
$$\frac{t_0 \xrightarrow{f} t_1 \xrightarrow{f} t_2 \quad t_0 \xrightarrow{f} t \xrightarrow{f} t_1}{t_0 \xrightarrow{f} t \xrightarrow{f} t_2,\ t \xrightarrow{f} t_1 \xrightarrow{f} t_2}$$

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

Encoding using quantifiers and triggers

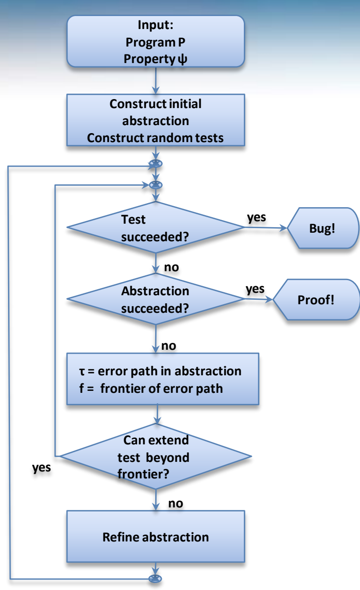
```
// transitive2
axiom(forall f: [int]int, x: int, y: int, z: int, w: int :: {ReachBe
} ReachBetween(f, x, y, z) && ReachBetween(f, y, w, z) ==> ReachBetw
);

// transitive3
axiom(forall f: [int]int, x: int, y: int, z: int, w: int ::
       {ReachBetween(f, x, y, z), ReachBetween(f, x, w, y)}
          ReachBetween(f, x, y, z) && ReachBetween(f, x, w, y) ==>
            ReachBetween(f, x, w, z) && ReachBetween(f, w, y, z));
```

Microsoft Research

Combining Random Testing with Model Checking

Aditya Nori, Sriram Rajamani,
ISSTA08: Proofs from Tests. Nels E. Beckman, Nori, Rajamani, Rob Simmons

DASH Algorithm



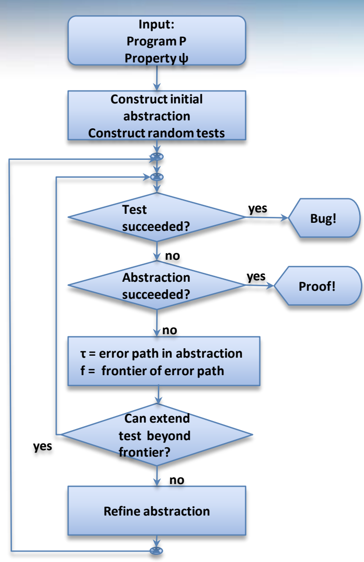
- Main workhorse: test case generation
- Use counterexamples from current abstraction to "extend frontier" and generate tests
- When test case generation fails, use this information to "refine" abstraction at the frontier
  - Use only aliases that happen on the tests!

Example

```
struct ProtectedInt
{
  int *lock;
  int *y;
};


void lock(int *x)
{
23:  if(*x != 0)
24:    error();
25:  *x = 1;
}


void unlock(int *x)
{
26:  if(*x != 1)
27:    error();
28:  *x = 0;
}
```

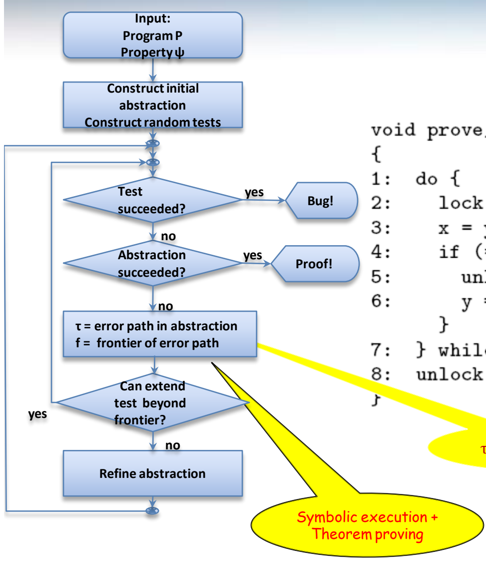
```
void LockUnlock(struct ProtectedInt *pi,
          int *lock1, int *lock2, int x)
{
1:  int do_return = 0;
2:  if(pi->lock == lock1 ){
3:    do_return = 1;
4:    pi->lock = lock2;
    }
5:  else if(pi->lock == lock2) {
6:    do_return = 1;
7:    pi->lock = lock1;
    }
  //initialize all locks to be unlocked
8:  *(pi->lock) = 0;
9:  *lock1 = 0;
10: *lock2 = 0;

11: if( do_return ) return;
12:   else {
13:      do {
14:         lock(pi->lock);
15:         if(*lock1 ==1 || *lock2 ==1)
16:            error();
17:          x = *(pi->y);
18:          if ( NonDet() ) {
19:               (*(pi->y))++;
20:               unlock(pi->lock);
             }
21:      } while(x != *(pi->y));
      }
22:   unlock(pi->lock);
}
```

Example




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

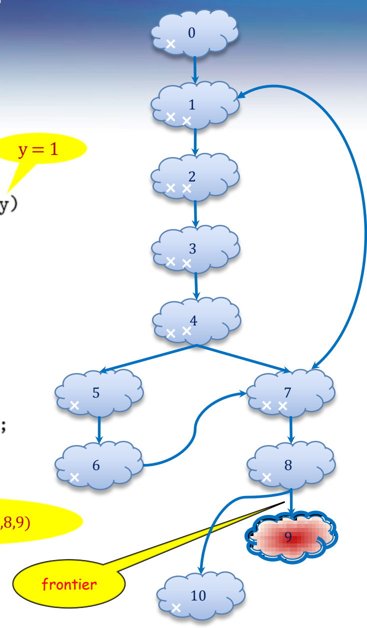
Example



y = 1

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

τ=(0,1,2,3,4,7,8,9)



Symbolic execution + Theorem Proving

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

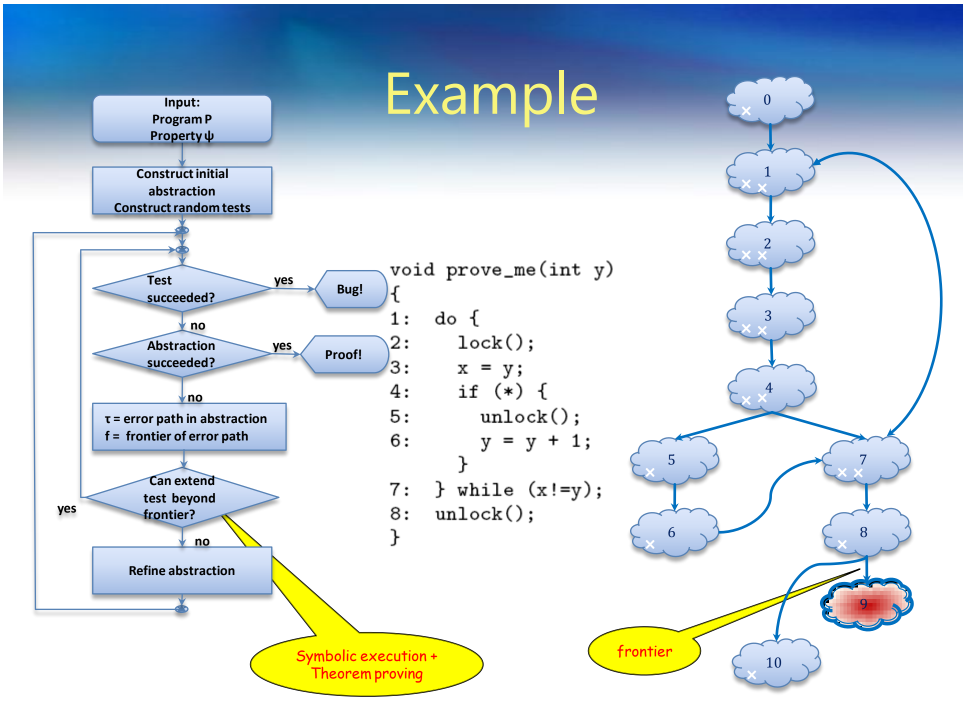
symbolic memory

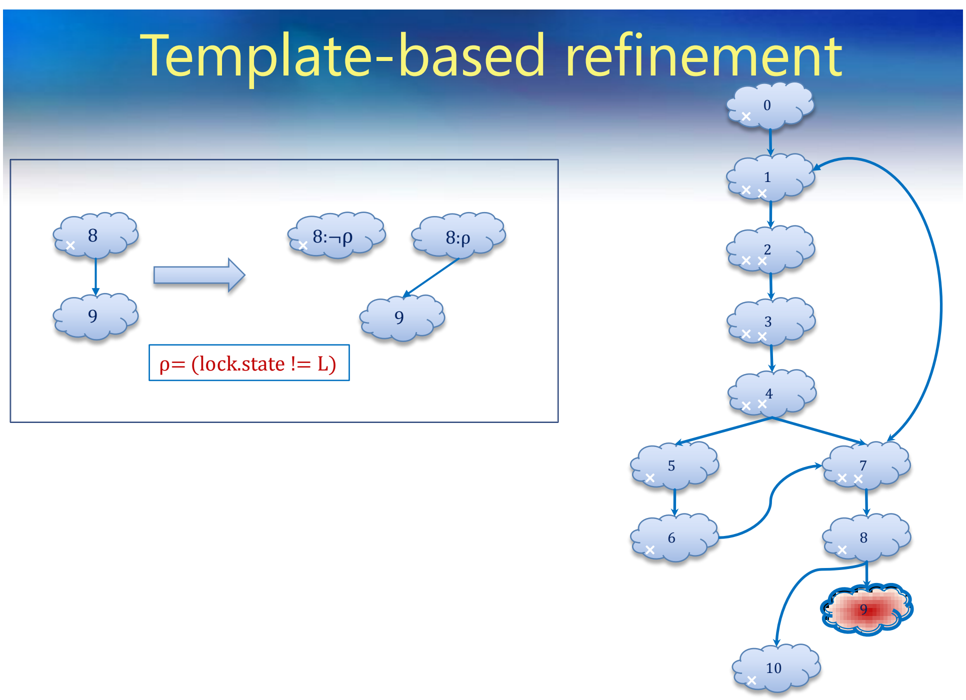
y	y_0
lock.state	L
x	y_0

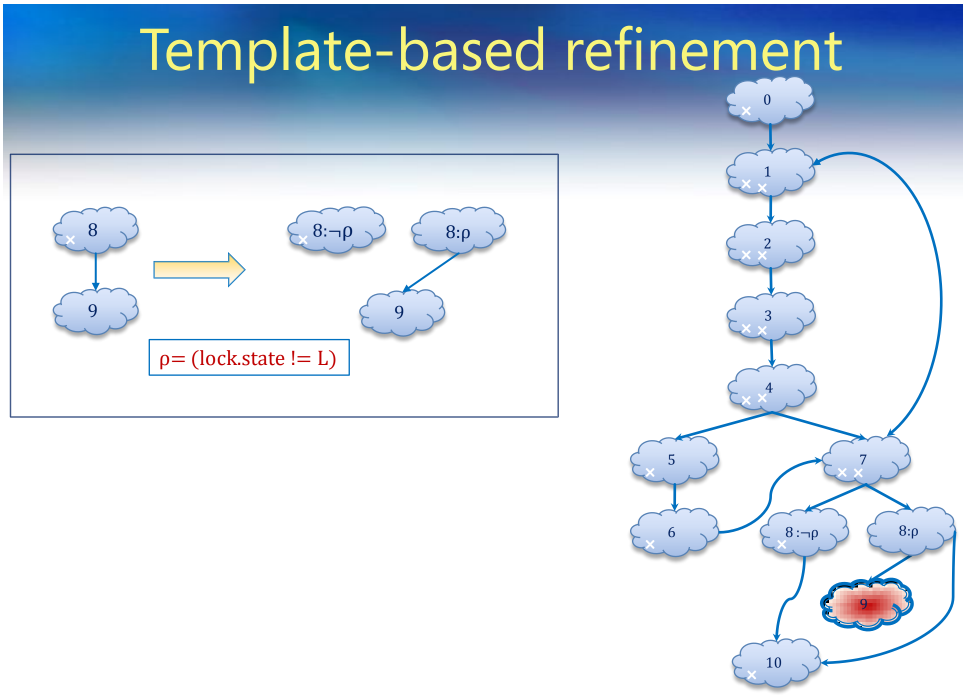
constraints

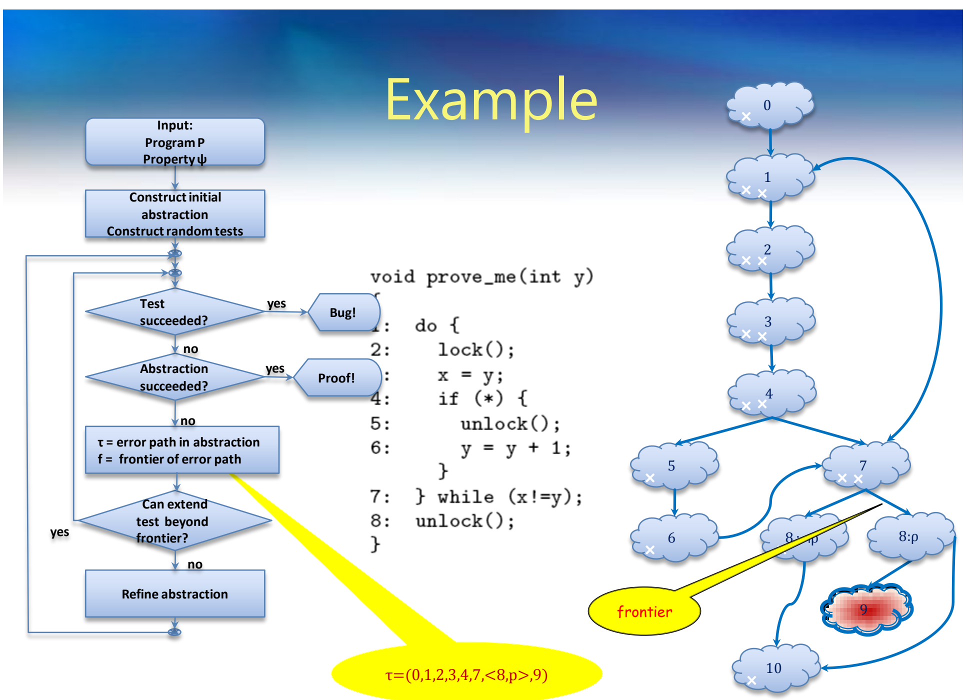
$(x = y) = (y_0 = y_0) = T$
$(lock.state \,!= L) = (L \,!= L) = F$

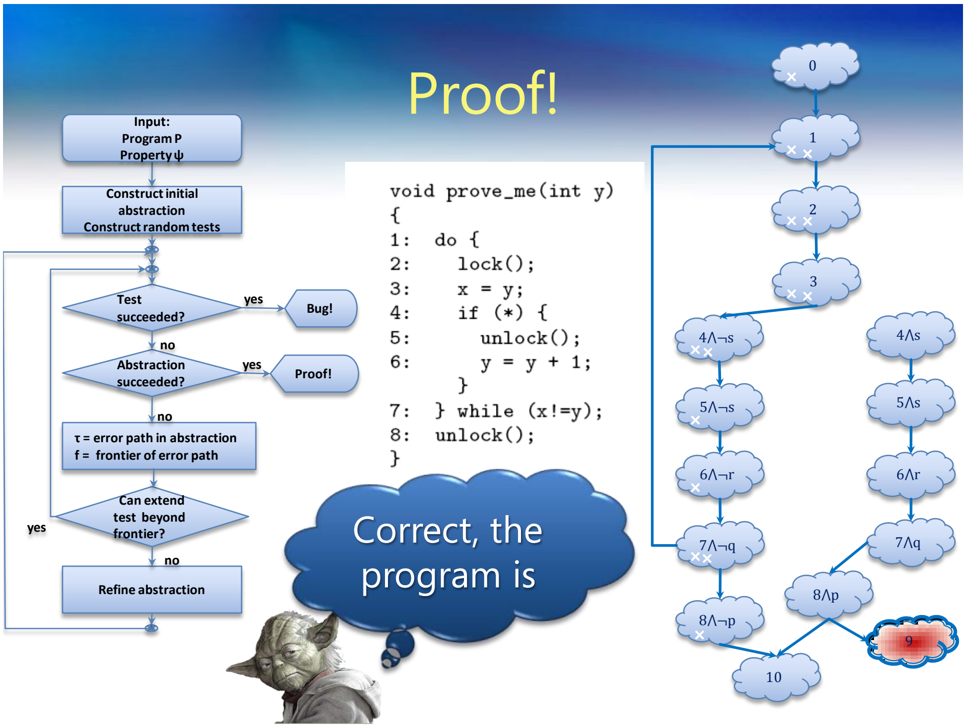
τ=(0,1,2,3,4,7,8,9)


Example
Input:
Program P
Property ψ
Construct initial abstraction
Construct random tests
Test succeeded?
yes
Bug!
no
Abstraction succeeded?
yes
Proof!
no
τ = error path in abstraction
f = frontier of error path
Can extend test beyond frontier?
yes
no
Refine abstraction
Symbolic execution + Theorem proving
void prove_me(int y)
{
1: do {
2: lock();
3: x = y;
4: if (*) {
5: unlock();
6: y = y + 1;
}
7: } while (x!=y);
8: unlock();
}
0
1
2
3
4
5
6
7
8
9
10
frontier


Template-based refinement
8
9
8:¬ρ
8:ρ
9
ρ= (lock.state != L)
0
1
2
3
4
5
6
7
8
9
10


Template-based refinement
8
9
8:¬ρ
8:ρ
9
ρ= (lock.state != L)
0
1
2
3
4
5
7
6
8:¬ρ
8:ρ
9
10


Example
Input:
Program P
Property ψ
Construct initial
abstraction
Construct random tests
Test
succeeded?
yes
Bug!
no
Abstraction
succeeded?
yes
Proof!
no
τ = error path in abstraction
f = frontier of error path
Can extend
test beyond
frontier?
yes
no
Refine abstraction
void prove_me(int y)
{
1: do {
2: lock();
3: x = y;
4: if (*) {
5: unlock();
6: y = y + 1;
}
7: } while (x!=y);
8: unlock();
}
τ=(0,1,2,3,4,7,<8,p>,9)
frontier
0
1
2
3
4
5
7
6
8:p
9
10



Yogi's solver interface

Representation

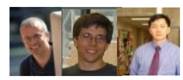
- L
  - program locations.
- $R \subseteq L \times L$
  - Control flow graph
- State: $L \rightarrow$ *Formula* ***–set***
  - Symbolic state: each location has set of *disjoint* formulas

Theorem proving needs

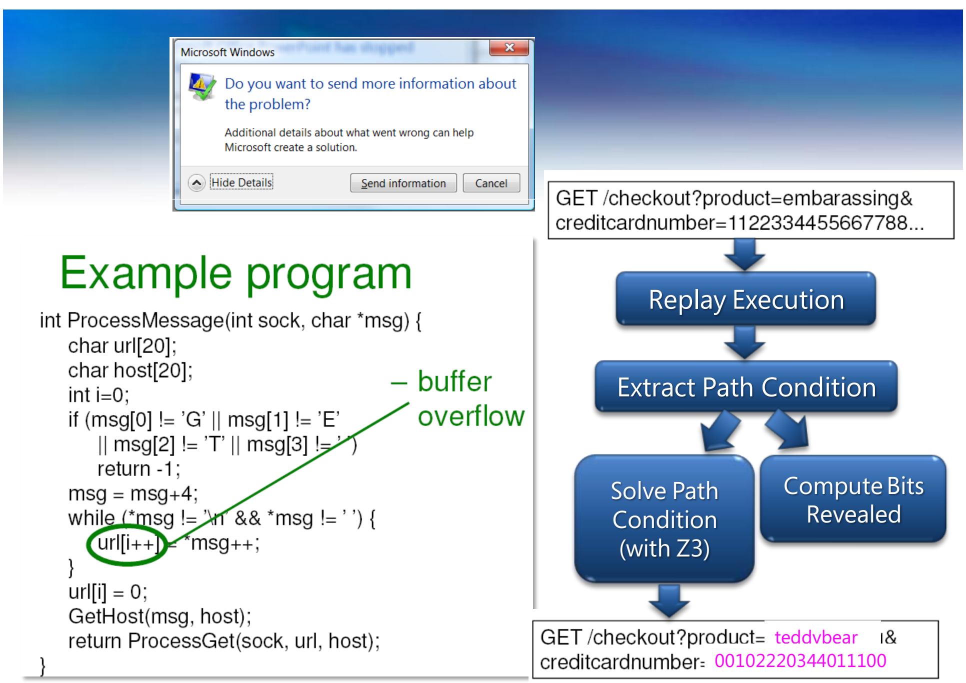
- Facts about pointers:
  - *&x = x
- Subsumption checks:
  - $\varphi \Rightarrow WP(l, \psi)$
  - $\varphi \Rightarrow \neg WP(l, \psi)$
- Structure sharing
  - Similar formulas in different states
- Simplification
  - Collapse/Reduce formulas

Better Bug Reporting with Better Privacy

Miguel Castro, Manuel Costa, Jean-Philippe Martin
ASPLOS 08



See also: Vigilante – Internet Worm Containment Miguel Castro, Manuel Costa, Lintao Zhang



Finding the buffer overflow

```
int ProcessMessage(int sock, char *msg) {
    char url[20];
    char host[20];
    int i=0;
    if (msg[0] != 'G' || msg[1] != 'E'
        || msg[2] != 'T' || msg[3] != ' ')
        return -1;
    msg = msg+4;
    while (*msg != '\n' && *msg != ' ') {
        url[i++] = *msg++;
    }
    url[i] = 0;
    GetHost(msg, host);
    return ProcessGet(sock, url, host);
}
```

– buffer overflow

R	A	N	D	O	M	\n	\n	\n

:assumption (= b0 bv71[8])
:assumption (= b1 bv69[8])
:assumption (= b2 bv84[8])
:assumption (= b3 bv32[8]))

G	E	T	' '	O	M	\n	\n	\n

:assumption (distinct b6 bv10[8] bv32[8])

G	E	T	' '	O	M	.	.	.

Privacy: *measure distance between original crash input and new input*

Microsoft Research

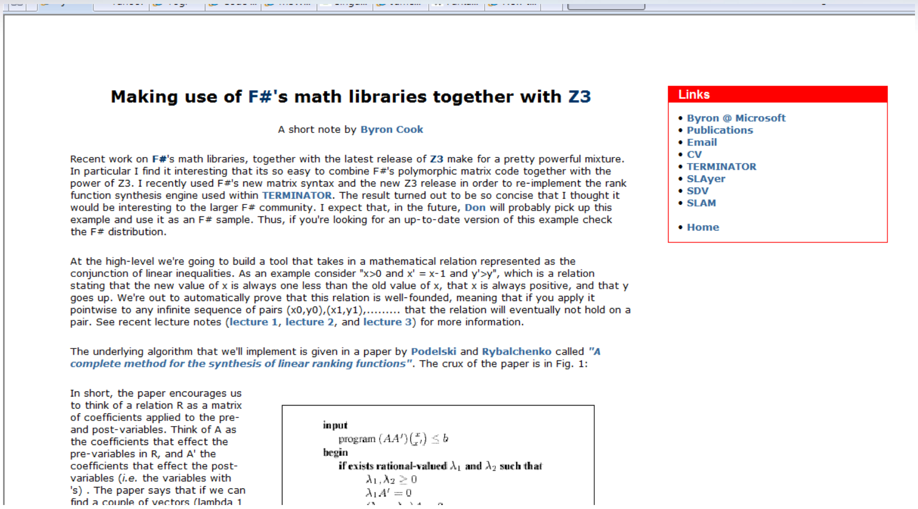
Program Termination

Byron Cook

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

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

Form Byron Cook's blog



Making use of F#'s math libraries together with Z3

A short note by Byron Cook

Recent work on F#'s math libraries, together with the latest release of Z3 make for a pretty powerful mixture. In particular I find it interesting that its so easy to combine F#'s polymorphic matrix code together with the power of Z3. I recently used F#'s new matrix syntax and the new Z3 release in order to re-implement the rank function synthesis engine used within TERMINATOR. The result turned out to be so concise that I thought it would be interesting to the larger F# community. I expect that, in the future, Don will probably pick up this example and use it as an F# sample. Thus, if you're looking for an up-to-date version of this example check the F# distribution.

At the high-level we're going to build a tool that takes in a mathematical relation represented as the conjunction of linear inequalities. As an example consider "x>0 and x' = x-1 and y'>y", which is a relation stating that the new value of x is always one less than the old value of x, that x is always positive, and that y goes up. We're out to automatically prove that this relation is well-founded, meaning that if you apply it pointwise to any infinite sequence of pairs (x0,y0),(x1,y1),......... that the relation will eventually not hold on a pair. See recent lecture notes (lecture 1, lecture 2, and lecture 3) for more information.

The underlying algorithm that we'll implement is given in a paper by Podelski and Rybalchenko called *"A complete method for the synthesis of linear ranking functions"*. The crux of the paper is in Fig. 1:

In short, the paper encourages us to think of a relation R as a matrix of coefficients applied to the pre- and post-variables. Think of A as the coefficients that effect the pre-variables in R, and A' the coefficients that effect the post-variables (*i.e.* the variables with 's) . The paper says that if we can find a couple of vectors (lambda 1

input
program $(AA')\binom{x}{x'} \leq b$
begin
if exists rational-valued λ_1 **and** λ_2 **such that**
$\lambda_1, \lambda_2 \geq 0$
$\lambda_1 A' = 0$

Links

- Byron @ Microsoft
- Publications
- Email
- CV
- TERMINATOR
- SLAyer
- SDV
- SLAM
- Home

Does this program Terminate?

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

$$x > 0 \wedge y > 0 \wedge$$
$$x' = x - 1 \wedge y' > y$$

$$\begin{array}{rcl} x & > & 0 \\ x' & \geq & x-1 \\ x' & \leq & x-1 \\ y & > & 0 \\ y' & > & y \end{array}$$

$$\begin{array}{rcrcrcrcrcl} 0x' & + & 0y' & + & -1x & + & 0y & + & 1 & \leq & 0 \\ 1x' & + & 0y' & + & -1x & + & 0y & + & 1 & \leq & 0 \\ -1x' & + & 0y' & + & 1x & + & 0y & + & -1 & \leq & 0 \\ 0x' & + & 0y' & + & 0x & + & -1y & + & 1 & \leq & 0 \\ 0x' & + & -1y' & + & 0x & + & 1y & + & 1 & \leq & 0 \end{array}$$

Microsoft Research

Rank function synthesis

$$\begin{array}{ccccccccccc} 0x' & + & 0y' & + & -1x & + & 0y & + & 1 & \leq & 0 \\ 1x' & + & 0y' & + & -1x & + & 0y & + & 1 & \leq & 0 \\ -1x' & + & 0y' & + & 1x & + & 0y & + & -1 & \leq & 0 \\ 0x' & + & 0y' & + & 0x & + & -1y & + & 1 & \leq & 0 \\ 0x' & + & -1y' & + & 0x & + & 1y & + & 1 & \leq & 0 \end{array}$$

Can we find *f, b,* such that the inclusion holds?

$$\subseteq \quad \begin{array}{ccc} f(x,y) & > & f(x',y') \\ f(x',y') & \geq & b \end{array}$$

That is:

$$\begin{array}{ccccccc} f(x',y') & + & -f(x,y) & + & 1 & \leq & 0 \\ & & -f(x',y') & + & b & \leq & 0 \end{array}$$

Rank function synthesis

$$\begin{array}{ccccccccccc} 0x' & + & 0y' & + & -1x & + & 0y & + & 1 & \leq & 0 \\ 1x' & + & 0y' & + & -1x & + & 0y & + & 1 & \leq & 0 \\ -1x' & + & 0y' & + & 1x & + & 0y & + & -1 & \leq & 0 \\ 0x' & + & 0y' & + & 0x & + & -1y & + & 1 & \leq & 0 \\ 0x' & + & -1y' & + & 0x & + & 1y & + & 1 & \leq & 0 \end{array} \subseteq \begin{array}{ccccccc} f(x',y') & + & -f(x,y) & + & 1 & \leq & 0 \\ & & -f(x',y') & + & b & \leq & 0 \end{array}$$

Search over linear templates:

$$\begin{array}{cccc} f(a,b) & \triangleq & c_1 a & + \; c_2 b \\ -f(a,b) & \triangleq & c_3 a & + \; c_4 b \\ c_1 & = & -1c_3 & \\ c_2 & = & -1c_4 & \end{array}$$

Rank function synthesis

Find c_1, c_2, c_3, c_4

$$\begin{array}{rcrcrcrcrcl}
0x' & + & 0y' & + & -1x & + & 0y & + & 1 & \le & 0 \\
1x' & + & 0y' & + & -1x & + & 0y & + & 1 & \le & 0 \\
-1x' & + & 0y' & + & 1x & + & 0y & + & -1 & \le & 0 \\
0x' & + & 0y' & + & 0x & + & -1y & + & 1 & \le & 0 \\
0x' & + & -1y' & + & 0x & + & 1y & + & 1 & \le & 0
\end{array}
\subseteq
\begin{array}{rcrcrcrcrcl}
c_1x' & + & c_2y' & + & c_3x & + & c_4y & + & 1 & \le & 0 \\
 & & & & c_3x' & + & c_4y' & + & b & \le & 0 \\
 & & & & 1c_1 & + & 1c_3 & + & 0 & \le & 0 \\
 & & & & -1c_1 & + & -1c_3 & + & 0 & \le & 0 \\
 & & & & 1c_2 & + & 1c_4 & + & 0 & \le & 0 \\
 & & & & -1c_2 & + & -1c_4 & + & 0 & \le & 0
\end{array}$$

Search over linear templates:

$$\begin{array}{rcrcl}
f(a,b) & \triangleq & c_1a & + & c_2b \\
-f(a,b) & \triangleq & c_3a & + & c_4b \\
c_1 & = & -1c_3 & & \\
c_2 & = & -1c_4 & &
\end{array}$$

Rank function synthesis

$\exists c_1, c_2, c_3, c_4, \forall x, y, x', y'$

$$\begin{array}{rcrcrcrcrcl}
0x' & + & 0y' & + & -1x & + & 0y & + & 1 & \le & 0 \\
1x' & + & 0y' & + & -1x & + & 0y & + & 1 & \le & 0 \\
-1x' & + & 0y' & + & 1x & + & 0y & + & -1 & \le & 0 \\
0x' & + & 0y' & + & 0x & + & -1y & + & 1 & \le & 0 \\
0x' & + & -1y' & + & 0x & + & 1y & + & 1 & \le & 0
\end{array}
\Rightarrow
\begin{array}{rcrcrcrcrcl}
c_1x' & + & c_2y' & + & c_3x & + & c_4y & + & 1 & \le & 0 \\
 & & & & c_3x' & + & c_4y' & + & b & \le & 0 \\
 & & & & 1c_1 & + & 1c_3 & + & 0 & \le & 0 \\
 & & & & -1c_1 & + & -1c_3 & + & 0 & \le & 0 \\
 & & & & 1c_2 & + & 1c_4 & + & 0 & \le & 0 \\
 & & & & -1c_2 & + & -1c_4 & + & 0 & \le & 0
\end{array}$$

Search over linear templates:

$$\begin{array}{rcrcl}
f(a,b) & \triangleq & c_1a & + & c_2b \\
-f(a,b) & \triangleq & c_3a & + & c_4b \\
c_1 & = & -1c_3 & & \\
c_2 & = & -1c_4 & &
\end{array}$$

Rank function synthesis – simplified version

$\exists c_1, c_2, c_3, c_4, \forall x, y, x', y'$

$$R \triangleq \begin{array}{ccccccccccc} 0x' & + & 0y' & + & -1x & + & 0y & + & 1 & \leq & 0 \\ 1x' & + & 0y' & + & -1x & + & 0y & + & 1 & \leq & 0 \\ -1x' & + & 0y' & + & 1x & + & 0y & + & -1 & \leq & 0 \\ 0x' & + & 0y' & + & 0x & + & -1y & + & 1 & \leq & 0 \\ 0x' & + & -1y' & + & 0x & + & 1y & + & 1 & \leq & 0 \end{array} \Rightarrow$$

$$\psi \triangleq c_1x' \times c_2y' \times c_3x \times c_4y \times 1 \leq 0$$

Search over linear templates:

$$\begin{array}{rcccc} f(a,b) & \triangleq & c_1 a & + & c_2 b \\ -f(a,b) & \triangleq & c_3 a & + & c_4 b \\ c_1 & = & -1c_3 & & \\ c_2 & = & -1c_4 & & \end{array}$$

Rank function synthesis

$\exists c_1, c_2, c_3, c_4, \forall x, y, x', y'$

$$R \triangleq \begin{array}{ccccccccccc} 0x' & + & 0y' & + & -1x & + & 0y & + & 1 & \leq & 0 \\ 1x' & + & 0y' & + & -1x & + & 0y & + & 1 & \leq & 0 \\ -1x' & + & 0y' & + & 1x & + & 0y & + & -1 & \leq & 0 \\ 0x' & + & 0y' & + & 0x & + & -1y & + & 1 & \leq & 0 \\ 0x' & + & -1y' & + & 0x & + & 1y & + & 1 & \leq & 0 \end{array} \Rightarrow$$

$$\psi \triangleq c_1x' + c_2y' + c_3x + c_4y + 1 \leq 0$$

Farkas' lemma. $R \Rightarrow \psi$ *iff* there exist real multipliers $\lambda_1, \ldots, \lambda_5 \geq 0$ such that

$$c_1 = \sum_{i=1}^{5} \lambda_i a_{i,1} \quad \wedge \cdots \wedge \quad c_4 = \sum_{i=1}^{5} \lambda_i a_{i,4} \quad \wedge \quad 1 \leq (\sum_{i=0}^{5} \lambda_i b_i)$$

Rank function synthesis

Instead solve: $\exists c_1, c_2, c_3, c_4, \lambda_1, \lambda_2, \lambda_3, \lambda_4, \lambda_5$

$$\begin{array}{rcrcrcrcrcr}
c_1 & = & 0\lambda_1 & + & 1\lambda_2 & + & -1\lambda_3 & + & 0\lambda_4 & + & 0\lambda_5 \\
c_2 & = & 0\lambda_1 & + & 0\lambda_2 & + & 0\lambda_3 & + & 0\lambda_4 & + & -1\lambda_5 \\
c_3 & = & -1\lambda_1 & + & -1\lambda_2 & + & 1\lambda_3 & + & 0\lambda_4 & + & 0\lambda_5 \\
c_4 & = & 0\lambda_1 & + & 0\lambda_2 & + & 0\lambda_3 & + & -1\lambda_4 & + & 1\lambda_5 \\
1 & \leq & 1\lambda_1 & + & 1\lambda_2 & + & -1\lambda_3 & + & 1\lambda_4 & + & 1\lambda_5 \\
c_1 & = & -1c_3 & \wedge & \lambda_1 \geq 0 & \wedge & \lambda_2 \geq 0 & \wedge & \lambda_3 \geq 0 & & \\
c_2 & = & -1c_4 & \wedge & \lambda_4 \geq 0 & \wedge & \lambda_5 \geq 0 & & & &
\end{array}$$

Farkas' lemma. $R \Rightarrow \psi$ *iff* there exist real multipliers $\lambda_1, \ldots, \lambda_5 \geq 0$ such that

$$c_1 = \sum_{i=1}^{5} \lambda_i a_{i,1} \quad \wedge \cdots \wedge \quad c_4 = \sum_{i=1}^{5} \lambda_i a_{i,4} \quad \wedge \quad 1 \leq (\sum_{i=0}^{5} \lambda_i b_i)$$

Rank function synthesis

Instead solve: $\exists c_1, c_2, c_3, c_4, \lambda_1, \lambda_2, \lambda_3, \lambda_4, \lambda_5$

$$\begin{array}{rcrcrcrcrcr}
c_1 & = & 0\lambda_1 & + & 1\lambda_2 & + & -1\lambda_3 & + & 0\lambda_4 & + & 0\lambda_5 \\
c_2 & = & 0\lambda_1 & + & 0\lambda_2 & + & 0\lambda_3 & + & 0\lambda_4 & + & -1\lambda_5 \\
c_3 & = & -1\lambda_1 & + & -1\lambda_2 & + & 1\lambda_3 & + & 0\lambda_4 & + & 0\lambda_5 \\
c_4 & = & 0\lambda_1 & + & 0\lambda_2 & + & 0\lambda_3 & + & -1\lambda_4 & + & 1\lambda_5 \\
1 & \leq & 1\lambda_1 & + & 1\lambda_2 & + & -1\lambda_3 & + & 1\lambda_4 & + & 1\lambda_5 \\
c_1 & = & -1c_3 & \wedge & \lambda_1 \geq 0 & \wedge & \lambda_2 \geq 0 & \wedge & \lambda_3 \geq 0 & & \\
c_2 & = & -1c_4 & \wedge & \lambda_4 \geq 0 & \wedge & \lambda_5 \geq 0 & & & &
\end{array}$$

Solver: Dual Simplex for Th(LRA).

See Byron Cook's blog for an F# program that produces input to Z3

Program Analysis as Constraint Solving

Sumit Gulwani, Saurabh Srivastava, Ramarathnam Venkatesan,
PLDI 2008

Loop invariants

Θ

while (c) {
    S
}
Post

$$\exists I \forall x \underbrace{\begin{bmatrix} \Theta(x) \Rightarrow I(x) \\ I(x) \wedge c(x) \wedge S(x,x') \Rightarrow I(x') \\ \neg c(x) \wedge I(x) \Rightarrow Post(x) \end{bmatrix}}_{\varphi_1(I,x)}$$

How to find loop invariant *I* ?

Loop invariants

$$\exists I \forall x \underbrace{\begin{bmatrix} \Theta(x) \Rightarrow I(x) \\ I(x) \wedge c(x) \wedge S(x,x') \Rightarrow I(x') \\ \neg c(x) \wedge I(x) \Rightarrow Post(x) \end{bmatrix}}_{\varphi_1(I,x)}$$

- Assume *I* is of the form $\sum_j a_j x_j \leq b$
- Simplified problem: $\exists A,b \forall x \varphi_1(\lambda x.Ax \leq b, x)$

Loop invariants ⇒ Existential

- Original: $\exists I \forall x \varphi_1(I,x)$
- Relaxed: $\exists A,b \forall x \varphi_1(\lambda x.Ax \leq b, x)$
- Farkas': $\forall x(Ax \leq 0 \Rightarrow bx \leq 0)$
  $\Leftrightarrow \exists \lambda, \lambda_1, .., \lambda_m (b = \lambda + \sum \lambda_k a_k)$
- Existential: $\exists A,b,\lambda \varphi_2(A,b,\lambda)$
  Problem: contains multiplication

Loop invariants ⇒ SMT solving

- Original: $\exists I \forall x \varphi_1(I, x)$
- Existential: $\exists A, b \exists \lambda \varphi_2(A, b, \lambda)$
- Bounded: $\exists A, b, p_1, p_2, p_3 \varphi_2\left(A, b, \begin{bmatrix} ite(p_1, 4, 0) + \\ ite(p_2, 2, 0) + \\ ite(p_3, 1, 0) \end{bmatrix}\right)$
- Or: Bit-vectors: $\exists A, b, \lambda : BitVec[8].\varphi_2(A, b, \lambda)$

Program Verification: Example

{n=1 ∧ m=1}

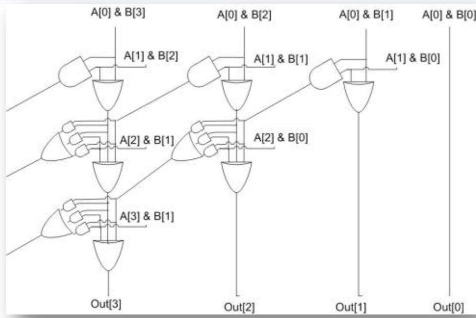
```
x := 0; y := 0;
while (x < 100)
   x := x+n;
   y := y+m;
```

{y ≥ 100}

Invariant Template	Satisfying Solution	Loop Invariant
$a_0 + a_1x + a_2y + a_3n + a_4m \geq 0$ $b_0 + b_1x + b_2y + b_3n + b_4m \geq 0$ $c_0 + c_1x + c_2y + c_3n + c_4m \geq 0$	$a_2=b_0=c_4=1,\ a_1=b_3=c_0=-1$	$y \geq x$ $m \geq 1$ $n \geq 1$
$a_0 + a_1x + a_2y + a_3n + a_4m \geq 0$ $b_0 + b_1x + b_2y + b_3n + b_4m \geq 0$	$a_2=b_2=1,\ a_1=b_1=-1$	$y \geq x$ $m \geq n$
$a_0 + a_1x + a_2y + a_3n + a_4m \geq 0$	UNSAT	Invalid triple or Imprecise Template

Digression: Bit-vectors and Z3

- Bit-vector multiplication
- For each sub-term A*B
  - Replace by fresh vector OUT
  - Create circuit for:
    OUT = A*B
  - Convert circuit into clauses:
    For each internal gate
    - Create fresh propositional variable
    - Represent gate as clause



{Out[0], ~A[0],~B[0]}, {A[0],~Out[0]}, {B[0],~Out[0]}, ...

Digression: Bit-vectors and Z3

Tableau + Relevancy Propagation DPLL =

$$\frac{\bigvee_{i=1}^{k} \varphi_i}{\varphi_1 \mid \cdots \mid \varphi_k}\ \vee \qquad \frac{\neg \bigvee_{i=1}^{k} \varphi_i}{\neg\varphi_1, \ldots, \neg\varphi_k}\ \neg\vee$$

$$\frac{\neg\neg\varphi}{\varphi}\ \neg\neg$$

$$\frac{\varphi \leftrightarrow \psi}{\varphi, \psi \mid \neg\varphi, \neg\psi}\ \leftrightarrow \qquad \frac{\neg(\varphi \leftrightarrow \psi)}{\varphi, \neg\psi \mid \neg\varphi, \psi}\ \neg\leftrightarrow$$

$$\frac{ite(\varphi_1, \varphi_2, \varphi_3)}{\varphi_1, \varphi_2 \mid \neg\varphi_1, \varphi_3}\ ite \qquad \frac{\neg ite(\varphi_1, \varphi_2, \varphi_3)}{\varphi_1, \neg\varphi_2 \mid \neg\varphi_1, \neg\varphi_3}\ \neg ite$$

$M \parallel F$	$\Longrightarrow M\, l \parallel F$	if $\begin{cases} l \text{ or } \bar{l} \text{ occurs in } F, \\ l \text{ is undefined in } M \end{cases}$	(Decide)
$M \parallel F, C \vee l$	$\Longrightarrow M\, l_{C\vee l} \mid F, C \vee l$	if $\begin{cases} M \models \neg C, \\ l \text{ is undefined in } M \end{cases}$	(UnitPropagate)
$M \parallel F, C$	$\Longrightarrow M \parallel F, C \mid C$	if $M \models \neg C$	(Conflict)
$M \parallel F \mid C \vee \bar{l}$	$\Longrightarrow M \parallel F \mid D \vee C$	if $l_{D\vee l} \in M$,	(Resolve)
$M \parallel F \mid C$	$\Longrightarrow M \parallel F, C \mid C$	if $C \notin F$	(Learn)
$M\, l'\, M' \parallel F \parallel C \vee l$	$\Longrightarrow M\, l_{C\vee l} \mid F$	if $\begin{cases} M \models \neg C, \\ l \text{ is undefined in } M \end{cases}$	(Backjump)
$M \parallel F \mid \square$	$\Longrightarrow$ *unsat*		(Unsat)

- Tableau goes outside in, DPLL inside out
- Relevancy propagation: If DPLL sets θ:ψ∨φ to **true**, θ is marked as *relevant*, then first of ψ, φ to be set to **true** gets marked as *relevant*.
- Used for circuit gates and for quantifier matching

Microsoft Research

Abstract Interpretation
and modular arithmetic

See Blog by Ruzica Piskac,
http://icwww.epfl.ch/~piskac/fsharp/

Material based on:
King & Søndergård, CAV 08
Muller-Olm & Seidl, ESOP 2005

Programs as transition systems

- Transition system:

$\langle$

L	locations,
V	variables,
$S = [V \rightarrow Val]$	states,
$R \subseteq L \times S \times S \times L$	transitions,
$\Theta \subseteq S$	initial states
$\ell_{init} \in L$	initial location

$\rangle$

Abstract abstraction

- Concrete reachable states: CR: $L \to \wp(S)$
- Abstract reachable states: AR: $L \to A$
- Connections:

$$\sqcup : A \times A \to A$$
$$\gamma : A \to \wp(S)$$
$$\alpha : S \to A$$
$$\alpha : \wp(S) \to A \quad \textit{where } \alpha(S) = \sqcup \{\alpha(s) \mid s \in S\}$$

Abstract abstraction

- Concrete reachable states:

$$\mathrm{CR}\ \ell\ x \leftarrow \Theta\ x \wedge \ell = \ell_{init}$$
$$\mathrm{CR}\ \ell\ x \leftarrow \mathrm{CR}\ \ell_0\ x_0 \wedge \mathrm{R}\ \ell_0\ x_0\ x\ \ell$$

- Abstract reachable states:

$$\mathrm{AR}\ \ell\ x \leftarrow \alpha(\Theta(x)) \wedge \ell = \ell_{init}$$
$$\mathrm{AR}\ \ell\ x \leftarrow \alpha(\gamma(\mathrm{AR}\ \ell_0 x_0) \wedge \mathrm{R}\ \ell_0\ x_0\ x\ \ell)$$

Why? fewer (finite) abstract states

Abstraction using SMT

Abstract reachable states:

$AR\ \ell_{init} \quad \leftarrow \alpha(\Theta)$

Find interpretation *M:*

$M \vDash \gamma(AR\ \ell_0\, x_0) \wedge R\ \ell_0\, x_0\, x\, \ell \wedge \neg\gamma(AR\ \ell\, x)$

Then:

$AR\ \ell \quad \leftarrow AR\ \ell \sqcup \alpha(x^M)$

Abstraction: Linear congruences

- States are linear congruences:

  $\mathbf{A}\, V = \mathbf{b} \bmod 2^m$

  - *V* is set of program variables.
  - **A** matrix, **b** vector of coefficients $[0..\ 2^m\text{-}1]$

Example

```
ℓ0: y ←x; c ←0;
ℓ1: while y != 0 do [ y ←y&(y-1); c ← c+1 ]
ℓ2:
```

- When at ℓ_2 :
  - y is 0.
  - c contains number of bits in x.

Abstraction: Linear congruences

- States are linear congruences:

$$\gamma\left(\begin{bmatrix} 2 & 3 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix} \bmod 2^3\right) \Leftrightarrow$$

$$2x_0 + 3x_1 = 1 \bmod 2^3 \wedge x_0 + x_1 = 3 \bmod 2^3 \Leftrightarrow$$

As Bit-vector constraints (SMTish syntax):

(and
 (= (bvadd (bvmul 010 x_0) (bvmul 011 x_1)) 001)
 (= (bvadd x_0 x_1) 011)
)

Abstraction: Linear congruences

- $\alpha(x=1, y=2) \triangleq \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$
- **(A** *V* = **b** mod 2^m**)** ⊔ **(A'** *V* = **b'** mod 2^m**)**
  - Combine: $\begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ -b & 0 & A & 0 & 0 \\ 0 & -b' & 0 & A' & 0 \\ 0 & 0 & -I & -I & I \end{bmatrix} \begin{bmatrix} s_1 \\ s_2 \\ x_1 \\ x_2 \\ x \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$
  - Triangulate (Muller-Olm & Seidl)
  - Project on *x*

Microsoft Research

Bounded Model Checking of Model Programs

Margus Veanes

FORTE 08

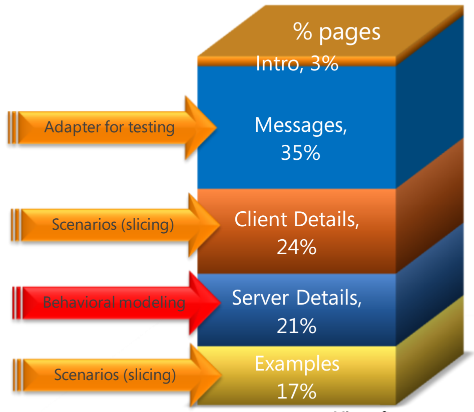
Goal:Model Based Development

Integration with symbolic analysis techniques at design time – *smart model debugging*
- *Theorem proving*
- Model checking
- Compositional reasoning

- Domain specific front ends
  - Different subareas require different adaptations
  - Model programs provide the common framework

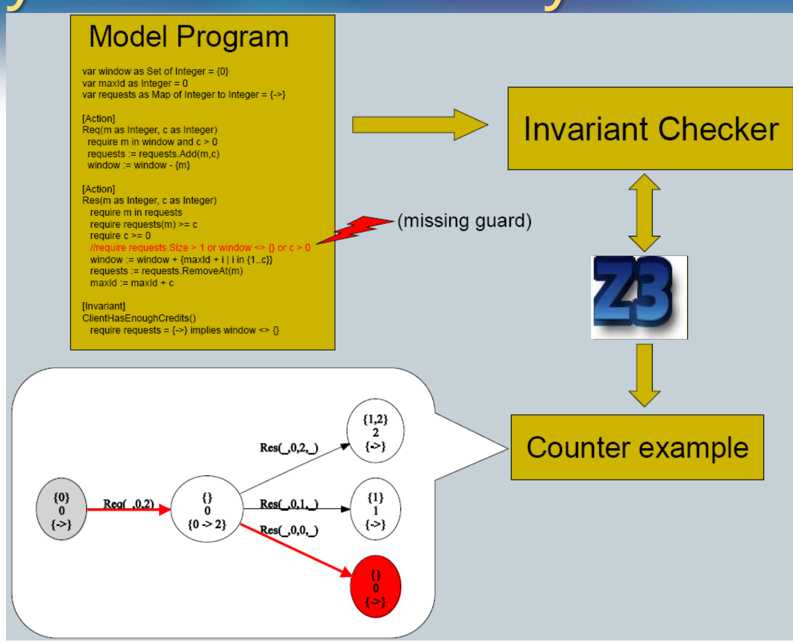
Motivating example
- SMB2 Protocol Specification
- Sweet spot for model-based testing and verification.

Sample protocol document for SMB2 (a network file protocol)



Microsoft Research

Symbolic Reachability



Bounded-reachability formula

- Given a model program *P* step bound *k* and reachability condition *φ*

$$Reach(P, \varphi, k) \stackrel{\text{def}}{=} I_P \wedge (\bigwedge_{0 \leq i < k} P[i]) \wedge (\bigvee_{0 \leq i \leq k} \varphi[i])$$

$$P[i] \stackrel{\text{def}}{=} \bigvee_{f \in A_P} \Big(action[i] = f(f_1[i], \ldots, f_n[i]) \wedge G_P^f[i]$$

$$\bigwedge_{v \in V_P^f} v[i+1] = t_v^f[i] \bigwedge_{v \in V_P \setminus V_P^f} v[i+1] = v[i]\Big)$$

Array model programs and quantifier elimination

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

Implementation using the SMT solver Z3

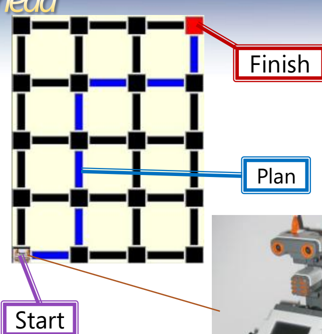
- Set comprehensions are introduced through skolem constant definitions using support for quantifiers in Z3
- Elimination of quantifiers is partial.
- Model is refined if a spurious model is found by Z3.
  - A spurious model may be generated by Z3 if an incomplete heuristic is used during quantifier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

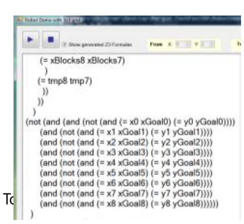
A different example:

Adaptive Planning with Finite Horizon Lookahead

Model program:

```
// Model program of walking in a grid until reaching goal
var x as Integer
var y as Integer
var xGoal as Integer
var yGoal as Integer
var xMax as Integer
var yMax as Integer
var yBlocks as Map of Integer to Set of Integer
var xBlocks as Map of Integer to Set of Integer

[Action]
Up()
    require y < yMax and not (y in yBlocks(x))
             and not (x = xGoal and y = yGoal)
    y := y + 1
[Action]
Down()
    require y > 0 and not (y-1 in yBlocks(x))
             and not (x = xGoal and y = yGoal)
    y := y - 1
[Action]
Right()
    require x < xMax and not (x in xBlocks(y))
             and not (x = xGoal and y = yGoal)
    x := x + 1
[Action]
Left()
    require x > 0 and not (x-1 in xBlocks(y))
             and not (x = xGoal and y = yGoal)
    x := x - 1
```





Microsoft Research

Verifying Garbage Collectors
- *Automatically and fast*

Chris Hawblitzel

http://www.codeplex.com/singularity/SourceControl/DirectoryView.aspx?SourcePath=%24%2fsingularity%2fbase%2fKernel%2fBartok%2fVerifiedGCs&changeSetId=14518

Context

Singularity

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

Bartok

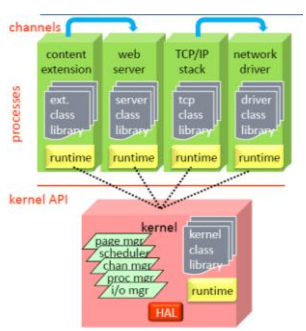
- MSIL → X86 Compiler

BoogiePL

- Procedural low-level language
- Contracts
- Verification condition generator

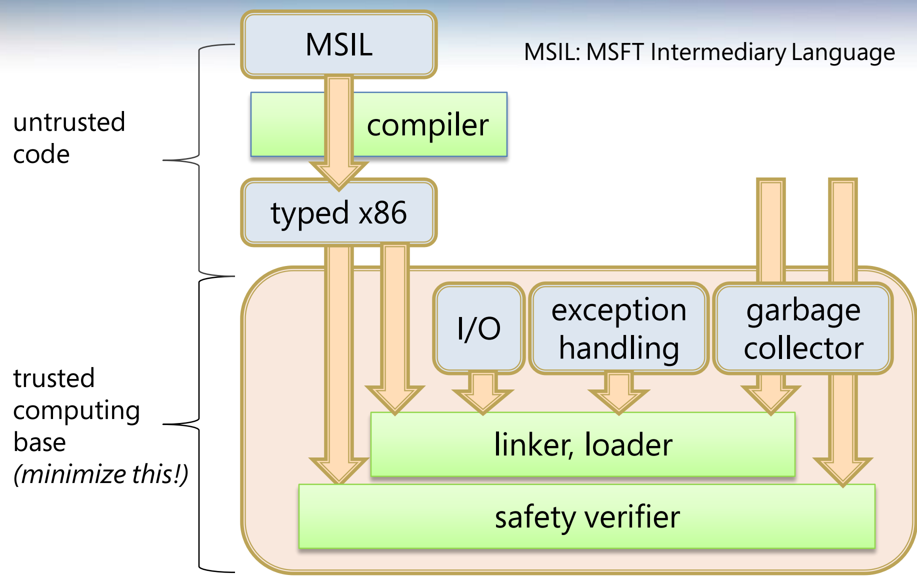
Garbage Collectors

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

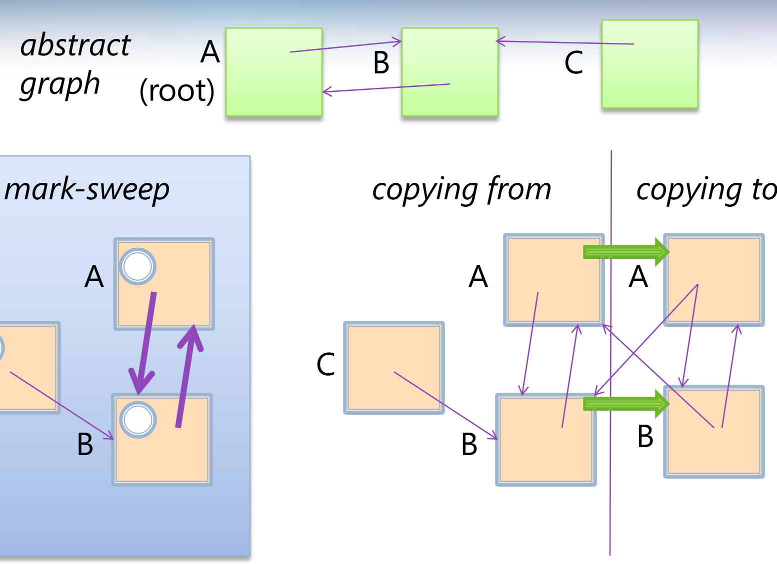


Microsoft Research

Goal: safely run untrusted code



Mark-sweep and copying collectors

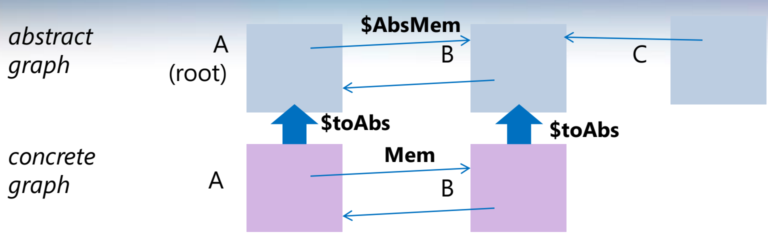


Garbage collector properties

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

(safety and effectiveness: verified; termination and efficiency: not verified)

Proving safety




```
procedure GarbageCollectMs()
  requires MsMutatorInv(root, Color, $toAbs, $AbsMem, Mem);
  modifies Mem, Color, $toAbs;
  ensure
{
  call M
  call Sv
}
```

```
function MsMutatorInv(...) returns (bool) {
      WellFormed($toAbs) && memAddr(root) && $toAbs[root] != NO_ABS
  && (forall i:int::{memAddr(i)} memAddr(i) ==> ObjInv(i, $toAbs, $AbsMem, Mem))
  && (forall i:int::{memAddr(i)} memAddr(i) ==> White(Color[i]))
  && (forall i:int::{memAddr(i)} memAddr(i) ==> ($toAbs[i]==NO_ABS <==>
Unalloc(Color[i])))}

function ObjInv(...) returns (bool) {  memAddr(i) && $toAbs[i] != NO_ABS ==>
    ... $toAbs[Mem[i, field1]] != NO_ABS ...
    ... $toAbs[Mem[i, field1]] == $AbsMem[$toAbs[i], field1] ... }
```

Controlling quantifier instantiation

- Idea: use marker

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

- Relativize quantifiers using marker

```
function GcInv(Color:[int]int, $toAbs:[int]int, $AbsMem:[int,int]int,
Mem:[int,int]int) returns (bool) {
  WellFormed($toAbs)
 && (forall i:int::{T(i)} T(i) ==> memAddr(i) ==>
     ObjInv(i, $toAbs, $AbsMem, Mem)
   && 0 <= Color[i] && Color[i] < 4
   && (Black(Color[i]) ==> !White(Color[Mem[i,0]]) && !White(Color[Mem[i,1]]))
   && ($toAbs[i] == NO_ABS <==> Unalloc(Color[i])))
}
```

Controlling quantifier instantiation

- Insert markers to enable triggers

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

Refinement Types for Secure Implementations

http://research.microsoft.com/F_7

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

Verifying protocol *reference* implementations

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

Example: access control for files

- **Un-trusted code** may call a **trusted library**
- **Trusted code** expresses security policy with assumes and asserts
- Each policy violation causes an assertion failure
- F_7 statically prevents any assertion failures by typing

```
type facts = CanRead of string
           | CanWrite of string

let read file = assert(CanRead(file)); …
let delete file = assert(CanWrite(file); …

let pwd = "C:/etc/passwd"
let tmp = "C:/temp/temp"

assume CanWrite(tmp)
assume ∀x . CanWrite(x) →CanRead(x)
```

```
let untrusted() =
  let v1 = read tmp in // ok
  let v2 = read pwd in //CanRead(pwd)
                       // assertion fails
```

Access control with refinement types

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

- Pre-conditions express access control requirements
- Post-conditions express results of validation
- F_7 type checks partially trusted code to guarantee that all preconditions (and hence all asserts) hold at runtime

Microsoft Research

Models for Domain Specific Languages with FORMULA & BAM

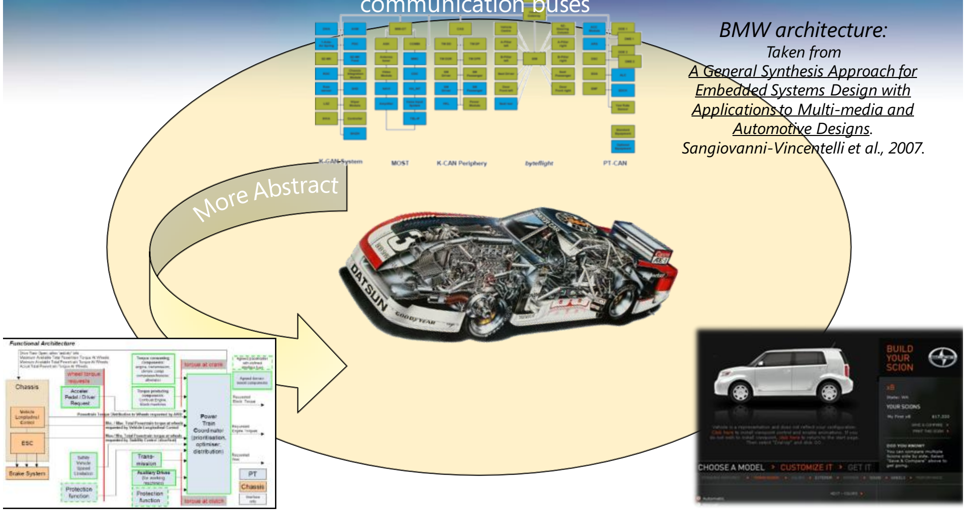
Ethan Jackson

FORTE 08

Designing Complex Systems Requires Multiple Abstractions

Automotive system is just processors and their communication buses

BMW architecture:
Taken from [A General Synthesis Approach for Embedded Systems Design with Applications to Multi-media and Automotive Designs](). Sangiovanni-Vincentelli et al., 2007.



Forget about the network; think about the software components

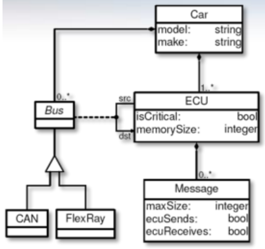
Functional architecture taken from AUTOSAR:
http://www.autosar.org

Product lines abstract across families of implementations

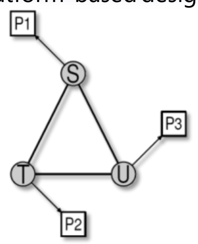
Screenshot of "Build Your Scion":
http://www.scion.org

Many Modeling Styles are Used to Build Abstractions

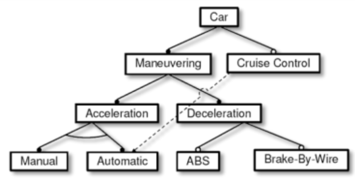
Abstraction: ECU/Bus
Style: Domain-specific Language



Abstraction: Scheduling Problem
Style: Platform-based design

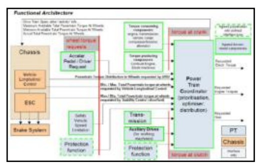


Abstraction: Automotive Product-line
Style: Feature Diagram



Instance

Instance



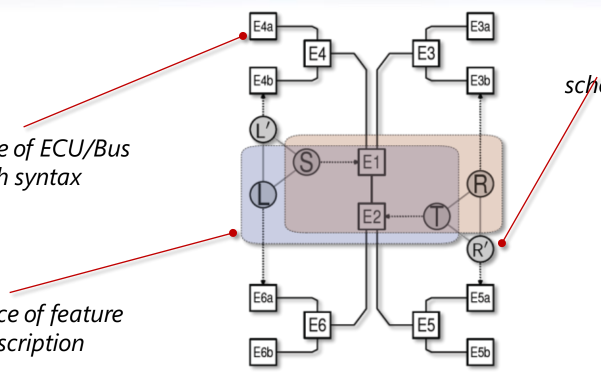
Instance

A Notorious Problem: How Do We Compose Abstractions?

We view each abstraction as providing (among other things) a constraint system representing the legal "models" of the abstraction. Composition occurs via these constraint systems:

Instance of ECU/Bus rich syntax

Instance of feature description



Instance of scheduling problem

Integration of Multiple Abstractions

For example, this instance must satisfy the constraints of each abstraction used in its construction.

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

```
domain TaskMap {
  /// Primitives of the abstraction
  Task        : (ID).
  Processor   : (ID).
  Taskmap     : (ID,ID).
  Constraint  : (ID,ID).

  /// Rules for detecting bad schedules
  no_map(Task(x))           :- Task(x), !Taskmap(x,_).
  bad_map(Task(x),Task(y))  :- Taskmap(x,z), Taskmap(y,z),
                               Constraint(x,y).

  /// Rules for declaring bad models
  malform(no_map(x))      :- no_map(x).
  malform(bad_map(x,y))   :- bad_map(x,y).

  /// Endpoints of relations are defined
  Task(x), Processor(y)   :- TaskMap(x,y).
  Task(x), Task(y)        :- Constraint(x,y).

  /// Ask if there exists a well-formed schedule.
  :? Constraint(x,y), Constraint(y,z), !malform(m).
}
```

A *domain* encapsulates a reusable, composable constraint system

Special function symbols (*malform*, *wellform*) capture legal instances in a domain-independent way.

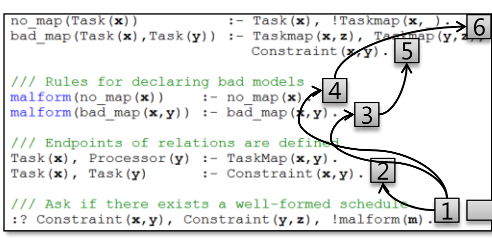
FORMULA can construct satisfying instances to logic program queries using Z3.

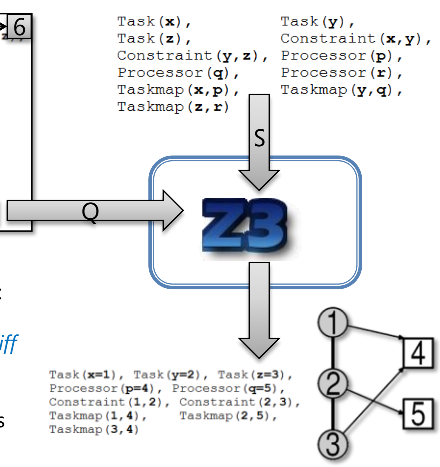
Search for satisfying instances are Reduced to Z3

This model finding procedure allows us to:

1. Determine if a composition of abstractions contains inconsistencies
2. Construct (partial) architectures that satisfy many domain constraints.
3. Generate design spaces of architectural invariants.

Reduction to Z3 works as follows:





Symbolic backwards chaining yields a set of candidate terms ***S*** with the following property:

*A finite instance exists that satisfies the query **Q** iff some subset of **S** satisfies the query **Q**.*

Once the finite set S is calculated, then S + Q is reduced to SMT and evaluated by Z3.

Selected Background on SMT

Basics

Pre-requisites and notation

Language of logic - summary

- Functions , Variables, Predicates
  - *f, g,* *x, y, z,* P, Q, =
- Atomic formulas, Literals
  - P(x,f(y)), ¬Q(y,z)
- Quantifier free formulas
  - P(f(a), b) ∧ c = g(d)
- Formulas, sentences
  - ∀x . ∀y . [P(x, f(x)) ∨ g(y,x) = h(y)]

Language: Signatures

- A *signature* Σ is a finite set of:
  - Function symbols:

    Σ_F = { *f, g,* … }
  - Predicate symbols:

    Σ_P = { *P, Q,* =, true, false, … }
  - And an *arity* function:

    Σ → N
- Function symbols with arity 0 are *constants*
- A countable set *V* of *variables*
  - disjoint from *Σ*

Language: Terms

- The set of *terms* $T(\Sigma_F, V)$ is the smallest set formed by the syntax rules:
  - $t \in T \quad ::= v \qquad\qquad v \in V$
    $\qquad\qquad | \quad f(t_1, \ldots, t_n) \qquad f \in \Sigma_F \; t_1, \ldots, t_n \in T$
- *Ground terms* are given by $T(\Sigma_F, \emptyset)$

Language: Atomic Formulas

- $a \in Atoms \quad ::= P(t_1, \ldots, t_n)$
  $\qquad\qquad P \in \Sigma_P \; t_1, \ldots, t_n \in T$

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

  Literals are (negated) atoms:
- $l \in Literals \quad ::= a \mid \neg a \qquad a \in Atoms$

Language: Quantifier free formulas

- The set QFF(Σ,V) of *quantifier free formulas* is the smallest set such that:

$\varphi \in$ QFF	$::= a \in Atoms$	*atoms*
	$\mid \neg \varphi$	*negations*
	$\mid \varphi \leftrightarrow \varphi'$	*bi-implications*
	$\mid \varphi \wedge \varphi'$	*conjunction*
	$\mid \varphi \vee \varphi'$	*disjunction*
	$\mid \varphi \rightarrow \varphi'$	*implication*

Language: Formulas

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

$\varphi ::= \ldots$

$\mid$	$\forall x . \varphi$	*universal quant.*
$\mid$	$\exists x . \varphi$	*existential quant.*

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

Theories

- A (first-order) *theory T (over signature Σ) is a set of (deductively closed) sentenes (over Σ and V)*
- Let *DC*(Γ) be the deductive closure of a set of sentences Γ.
  - For every theory T, *DC*(T) = T
- A theory T is *constistent* if *false* $\not\in T$
- We can view a (first-order) theory *T* as the class of all *models* of *T* (due to completeness of first-order logic).

Models (Semantics)

- A model *M* is defined as:
  - Domain *S;* set of elements.
  - Interpretation, $f^M : S^n \rightarrow S$ for each $f \in \Sigma_F$ with arity(f) = n
  - Interpretation $P^M \subseteq S^n$ for each $P \in \Sigma_P$ with arity(P) = n
  - Assignment $x^M \in S$ for every variable $x \in V$
- *A formula* φ is true in a model *M* if it evaluates to true under the given interpretations over the domain *S*.
- *M* is a *model for the theory T* if all sentences of *T* are true in *M*.

T-Satisfiability

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

$$M \models_T \varphi(x)$$

T-Validity

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

$$\models_T \varphi(x)$$

Checking validity

- Checking the validity of φ in a theory *T*:

φ is *T-valid*

$\equiv$ *T-unsat*:	$\neg\varphi$	
$\equiv$ *T-unsat*:	$\forall x \exists y \forall z \exists u \,.\, \phi$	(prenex of $\neg\varphi$)
$\equiv$ *T-unsat*:	$\forall x \forall z \,.\, \phi[f(x),g(x,z)]$	(skolemize)
$\Leftarrow$ *T-unsat*:	$\phi[f(a_1),g(a_1,b_1)] \wedge \ldots$ $\wedge\ \phi[f(a_n),g(a_n,b_n)]$	(instantiate) ($\Rightarrow$ if compactness)
$\equiv$ *T-unsat*:	$\phi_1 \vee \ldots \vee \phi_m$	(DNF)

where each ϕ_i is a conjunction.

Checking Validity – the morale

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

Clauses – CNF conversion

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

$$\varphi : x = 5 \Leftrightarrow (y < 3 \vee z = x)$$ *is not in CNF.*

Clauses – CNF conversion

$$\varphi : x = 5 \Leftrightarrow (y < 3 \vee z = x)$$

Equi-satisfiable CNF formula

$$\varphi' : (\neg p \vee x = 5) \wedge (p \vee \neg x = 5) \wedge$$
$$(\neg p \vee y < 3 \vee z = x) \wedge$$
$$(p \vee \neg y < 3) \wedge (p \vee \neg z = x)$$

Clauses – CNF conversion

cnf(φ) = **let** (q,F) = cnf′(φ) **in** $q \wedge F$

cnf′(a) = (a, *true*)

cnf′($\varphi \wedge \varphi'$) = **let** (q,F_1) = cnf′(φ)
(r, F_2) = cnf′(φ')
p = fresh Boolean variable
in
$(p, F_1 \wedge F_2 \wedge (\neg p \vee q) \wedge$
$(\neg p \vee r) \wedge$
$(\neg p \vee \neg q \vee \neg r))$

Exercise: cnf′($\varphi \vee \varphi'$), cnf′($\varphi \leftrightarrow \varphi'$), cnf′($\neg \varphi$)

Clauses - CNF

- Main properties of basic CNF
  - Result F is a set of *clauses.*
  - φ is T-satisfiable iff cnf(φ) is.
  - size(cnf(φ)) $\leq$ 4(size(φ))
  - $\varphi \Leftrightarrow \exists\, p_{aux}$ cnf(φ)

DPLL(∅)

DPLL - *classique*

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

Modern DPLL – as transitions

- *Maintain states of the form:*
  - *M || F - during search*
  - *M || F || C – for backjumping*
  - *M a partial model, F are clauses, C is a clause.*
- **Decide** M || F ⇒ Ml^d || *F* **if** *l* ∈ *F \ M*
  d is a *decision* marker
- **Propagate** M || F ⇒ Ml^C || *F*
  if *l* ∈ *C* ∈ *F*, *C* = (*C′* ∨ *l*), *M* $\models_T$ ¬*C′*

Modern DPLL – as transitions

- **Conflict** M || F ⇒ *M* || *F* || C **if** *C* ∈ *F*, *M* $\models_T$ ¬*C*
- **Learn** M || F || C ⇒ *M* || *F*, C || C ***i.e,*** *add C* to *F*
- **Resolve** M$p^{(C' \vee p)}$ || F || C ∨ ¬*p* ⇒ *M* || *F* || C ∨ C′
- **Skip** M*p* || F || C ⇒ *M* || *F* || C **if** ¬*l* ∉*C*
- **Backjump** MM′l^d|| F || C ⇒ *M*¬l^C || *F*
  if ¬*l* ∈*C* and M′ does not intersect with ¬*C*

Microsoft Research

DPLL(E)

DPLL(*E*)

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

E - conflicts

Recall **Conflict**:

- **Conflict** M || F $\Rightarrow$ *M* || *F* || *C* **if** $C \in F$, $M \models_T \neg C$

A version more useful for theories:

- **Conflict** M || F $\Rightarrow$ *M* || *F* || *C* **if** $C \subseteq \neg M$, $\models_T C$

E - conflicts

Example

- M = $fff(a) = a$, $g(b) = c$, $fffff(a) = a$, $a \neq f(a)$
- $\neg C$ = $fff(a) = a$, $fffff(a) = a$, $a \neq f(a)$
- $\models_E fff(a) \neq a \vee fffff(a) \neq a \vee a = f(a)$

- Use *C* as a conflict clause.

Microsoft Research

Linear Arithmetic

Approaches to linear arithmetic

- Fourier-Motzkin:
  - Quantifier elimination procedure

    $\exists x\, (t \leq ax \;\wedge t' \leq bx \wedge cx \leq t'') \Leftrightarrow ct \leq at' \wedge ct' \leq bt''$
  - Polynomial for difference logic.
  - Generally: exponential space, doubly exponential time.
- Simplex:
  - Worst-case exponential, but
  - Time-tried practical efficiency.
  - Linear space

Combining Theory Solvers

Nelson-Oppen procedure

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

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

Interaction:
Guess a partition of V_{shared}
Express the partition as a conjunction of equalities.
Example, $\{ x_1 \}, \{ x_2 , x_3 \}, \{ x_4 \}$ is represented as:
$\psi: x_1 \neq x_2 \wedge x_1 \neq x_4 \wedge x_2 \neq x_4 \wedge x_2 = x_3$

Component Procedures:
Use solver 1 to check satisfiability of $L_1 \wedge \psi$
Use solver 2 to check satisfiability of $L_2 \wedge \psi$

NO – reduced guessing

- Instead of guessing, we can often *deduce* the equalities to be shared.
- **Interaction:** $T_1 \wedge L_1 \models x = y$ *then add equality to* ψ.
- If theories are *convex,* then we can:
  - Deduce all equalities.
  - Assume every thing not deduced is distinct.
  - Complexity: $O(n^4 \times T_1(n) \times T_2(n))$.

Model-based combination

- Reduced guessing is only complete for convex theories.
- Deducing all implied equalities may be expensive.
  - Example: Simplex – no direct way to extract from just bounds and β
- *But:* backtracking is pretty cheap nowadays:
  - If $\beta(x) = \beta(y)$, then x, y are equal in arithmetical component.

Model-based combination

- Backjumping is cheap with modern DPLL:
  - If $\beta(x) = \beta(y)$, then x, y are equal in arithmetical model.
  - So let's add x = y to ψ, but allow to backtrack from guess.
- In general: if M_1 is the current model
  - $M_1 \models x = y$ then add literal $(x = y)^d$

Microsoft Research

Arrays

Theory of arrays

- Functions: $\Sigma_F = \{ \textit{read}, \textit{write} \}$
- Predicates: $\Sigma_P = \{ = \}$
- Convention *a[i]* means: *read(a,i)*

- Non-extensional arrays T_A:
  - $\forall a, i, v \,.\, write(a,i,v)[i] = v$
  - $\forall a, i, j, v \,.\, i \neq j \Rightarrow write(a,i,v)[j] = a[j]$

- Extensional arrays: $T_{EA} = T_A +$
  - $\forall a, b. ((\forall i.\ a[i] = b[i]) \Rightarrow a = b)$

Decision procedures for arrays

- Let *L* be literals over $\Sigma_F = \{ \textit{read}, \textit{write} \}$
- Find *M* such that: $M \models_{T_A} L$

- Basic algorithm, reduce to *E:*
  - for every sub-term *read(a,i), write(b,j,v) in L*
    - $i \neq j \wedge a = b \Rightarrow read(write(b,j,v),i) = read(a,i)$
    - $read(write(b,j,v),j) = v$
  - Find M_E, *such that*
    $M_E \models_E L \wedge \text{AssertedAxioms}$

Microsoft Research

Quantifiers and E-graph matching

DPLL(QT) – *cute quantifiers*

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

$$\varphi[\forall x.\psi(x) \leftarrow \boldsymbol{p}]$$

DPLL(QT)

- Suppose DPLL(T) sets $\boldsymbol{p}$ to **false**
  - $\Rightarrow$ any model M for φ must satisfy:

    $$M \vDash \neg \forall x.\psi(x)$$

  - $\Rightarrow$ for some sk_x: $M \vDash \neg \psi(sk_x)$
  - In general: $\vDash \neg \boldsymbol{p} \rightarrow \neg \psi(sk_x)$

DPLL(QT)

- Suppose DPLL(T) sets $\boldsymbol{p}$ to **true**
  - $\Rightarrow$ any model M for φ must satisfy:

    $$M \vDash \forall x.\psi(x)$$

  - $\Rightarrow$ for every term t: $M \vDash \psi(t)$
  - In general: $\vDash \boldsymbol{p} \rightarrow \psi(t)$
    For every term t.

DPLL(QT)

- Summary of auxiliary axioms:
  - $\models \neg$ ***p*** $\rightarrow \neg\, \psi(sk_x)$ For fixed, fresh sk_x
  - $\models$ ***p*** $\rightarrow \psi(t)$ For every term *t.*
- Which terms *t* to use for auxiliary axioms of the second kind?

DPLL(QT) with E-matching

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

  ∀a,i,v **{ write(a,i,v) }** . read(write(a,i,v),i) = v

DPLL(QT) with E-matching

- ⊨ ***p*** → ψ(*t*) For every term *t*.

- Approach:
  - Add patterns to quantifiers
  - Search for pattern matches in *E*-graph.

    ∀a,i,v **{ write(a,i,v) }** . read(write(a,i,v),i) = v

    - Add equality every time there is a write(b,j,w) term in *E*.

Microsoft Research

Z3 -An Efficient SMT Solver

Main features

- Linear real and integer arithmetic.
- Fixed-size bit-vectors
- Uninterpreted functions
- Extensional arrays
- Quantifiers
- Model generation
- Several input formats (Simplify, SMT-LIB, Z3, Dimacs)
- Extensive API (C/C++, .Net, OCaml)

Microsoft Research

Web

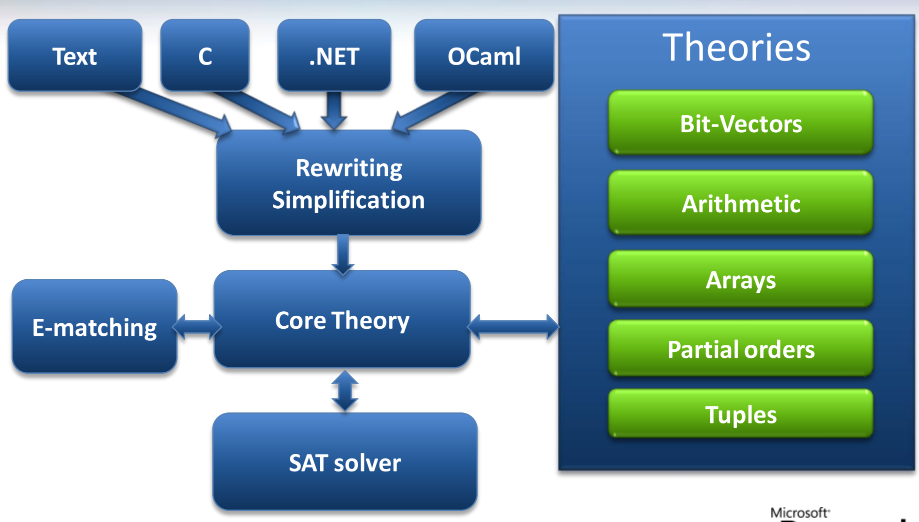
Microsoft Research

Supporting material

- http://research.microsoft.com/projects/z3/documentation.html

Microsoft Research

Z3: Core System Components



Microsoft Research

Example: C API

```
for (n = 2; n <= 5; n++) {
    printf("n = %d\n", n);
    ctx = Z3_mk_context(cfg);

    bool_type   = Z3_mk_bool_type(ctx);
    array_type  = Z3_mk_array_type(ctx, bool_type, bool_type);

    /* create arrays */
    for (i = 0; i < n; i++) {
        Z3_symbol s = Z3_mk_int_symbol(ctx, i);
        a[i]        = Z3_mk_const(ctx, s, array_type);
    }

    /* assert distinct(a[0], ..., a[n]) */
    d = Z3_mk_distinct(ctx, n, a);
    printf("%s\n", Z3_ast_to_string(ctx, d));
    Z3_assert_cnstr(ctx, d);

    /* context is satisfiable if n < 5 */
    if (Z3_check(ctx) == l_false)
        printf("unsatisfiable, n: %d\n", n);

    Z3_del_context(ctx);
}
```

Given arrays:

bool a1[bool];
bool a2[bool];
bool a3[bool];
bool a4[bool];

All can be distinct.

Add:

bool a5[bool];

Two of a1,..,a5 must be equal.

Microsoft Research

Example: SMT-LIB

```
(benchmark integer-linear-arithmetic
:status sat
:logic QF_LIA
:extrafuns ((x1 Int) (x2 Int) (x3 Int)
            (x4 Int) (x5 Int))
:formula (and (>= (- x1 x2) 1)
          (<= (- x1 x2) 3)
          (= x1 (+ (* 2 x3) x5))
          (= x3 x5)
          (= x2 (* 6 x4)))
)
```

```
(benchmark array
 :logic QF_AUFLIA
 :status unsat
 :extrafuns ((a Array)  (b Array)  (c Array))
 :extrafuns ((i Int) (j Int))

 :formula (and
          (= (store a i v) b)
          (= (store a j w) c)
          (= (select b j) w)
          (= (select c i) v)
          (not (= b c))
)
```

SMT-LIB syntax – basics

- *benchmark* ::= (benchmark *name*
  [:status (sat | unsat | unknown)]
  :logic *logic-name*
  *declaration**)
- *declaration* ::= :extrafuns (*func-decl**)
  | :extrapreds (*pred-decl**)
  | :extrasorts (*sort-decl**)
  | :assumption *fmla*
  | :formula *fmla*
- *sort-decl* ::= *id* - identifier
- *func-decl* ::= *id sort-decl* sort-decl* - name of function, domain, range
- *pred-decl* ::= *id sort-decl** - name of predicate, domain
- *fmla* ::= (and *fmla**) | (or *fmla**) | (not *fmla*)
  | (if_then_else *fmla fmla fmla*) | (= *term term*)
  | (implies *fmla fmla*) (iff *fmla fmla*) | (*predicate term**)
- *Term* ::= (ite *fmla term term*)
  | (*id term**) - function application
  | *id* - constant

SMT-LIB syntax - basics

- Logics:
  - QF_UF – Un-interpreted functions. Built-in sort **U**
  - QF_AUFLIA – Arrays and Integer linear arithmetic.
    - Built-in Sorts:
      - **Int, Array** (of Int to Int)
    - Built-in Predicates:
      - **<=, >=, <, >,**
    - Built-in Functions:
      - +, *, -, select, store.
    - Constants: 0, 1, 2, …

SMT-LIB – encodings

- Q: There is no built-in function for *max* or *min*. How do I encode it?
  - (max x y) is the same as (ite (> x y) x y)
  - Also: replace (max x y) by fresh constant *max_x_y* add assumptions:
    :assumption (implies (> x y) (= max_x_y x))
    :assumption (implies (<= x y) (= max_x_y y))
- Q: Encode the predicate *(even n)*, that is true when *n* is even.

Quantifiers

Quantified formulas in SMT-LIB:

- *fmla* ::= ...
  | (forall *bound** *fmla*)
  | (exists *bound** *fmla*)
- *Bound* ::= (*id sort-id*)

- Q: I want *f* to be an injective function. Write an axiom that forces *f* to be injective.
- Patterns: guiding the instantiation of quantifiers (Lecture 5)
  - *fmla* ::= ...
    | (forall (?x A) (?y B) *fmla* :pat { *term* })
    | (exists (?x A) (?y B) *fmla* :pat { *term* })
- Q: what are the patterns for the injectivity axiom?

Using the Z3 (managed) API

Create a context *z3*:

```
open Microsoft.Z3
open System.Collections.Generic
open System

let par = new Config()
do  par.SetParamValue("MODEL", "true")
let z3 = new TypeSafeContext(par)
```

```
let check (fmla) =
  z3.Push();
  z3.AssertCnstr(fmla);
  (match z3.Check() with
  | LBool.False -> Printf.printf "unsat\n"
  | LBool.True  -> Printf.printf "sat\n"
  | LBool.Undef -> Printf.printf "unknown\n"
  | _ -> assert false);
  z3.Pop(1ul)
```

Check a formula

-**Push**
-**AssertCnstr**
-**Check**
-**Pop**

Using the Z3 (managed) API

```
let (===) x y   = z3.MkEq(x,y)
let (==>) x y   = z3.MkImplies(x,y)
let (&&) x y    = z3.MkAnd(x,y)
let neg x       = z3.MkNot(x)

let a      = z3.MkType("a")
let f_decl = z3.MkFuncDecl("f",a,a)
let x      = z3.MkConst("x",a)
let f x    = z3.MkApp(f_decl,x)
```

Declaring z3 shortcuts, constants and functions

Proving a theorem

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

compared to

```
(benchmark euf
 :logic QF_UF
 :extrafuns ((f U U) (x U))
:formula (not (implies (and (= x (f(f(f(f(f x)))))) (= x (f(f(f x))))) (= x (f x))))
```

Enumerating models

We want to find models for

$$2 < i_1 \le 5 \wedge 1 < i_2 \le 7 \wedge -1 < i_3 \le 17 \wedge$$
$$0 \le i_1 + i_2 + i_3 \wedge i_2 + i_3 = i_1$$

But we only care about different i_1

Enumerating models

- Representing the problem

$$2 < i_1 \le 5 \wedge$$
$$1 < i_2 \le 7 \wedge$$
$$-1 < i_3 \le 17 \wedge$$
$$0 \le i_1 + i_2 + i_3 \wedge$$
$$i_2 + i_3 = i_1$$

```
void Test() {
    Config par = new Config();
    par.SetParamValue("MODEL", "true");
    z3 = new TypeSafeContext(par);
    intT = z3.MkIntType();
    i1 = z3.MkConst("i1", intT);   i2 = z3.MkConst("i2", intT);
    i3 = z3.MkConst("i3", intT);

    z3.AssertCnstr(Num(2)  < i1 & i1 <= Num(5));
    z3.AssertCnstr(Num(1)  < i2 & i2 <= Num(7));
    z3.AssertCnstr(Num(-1)  < i3 & i3 <= Num(17));
    z3.AssertCnstr(Num(0)  <= i1 + i2 + i3 & Eq(i2 + i3, i1));
    Enumerate();
    par.Dispose();
    z3.Dispose();
  }
```

Enumerating models

Enumeration:

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

TermAst Eq(TermAst t1, TermAst t2) { return z3.MkEq(t1,t2); }

TermAst Num(int i) { return z3.MkNumeral(i, intT); }
```

```
partitions:
*2 {i2} -> 2:int
*3 {i3} -> 1:int
*4 {i1} -> 3:int
Block (= 3 i1)
partitions:
*2 {i2 i3} -> 2:int
*4 {i1} -> 4:int
Block (= 4 i1)
partitions:
*2 {i2} -> 2:int
*3 {i3} -> 3:int
*4 {i1} -> 5:int
Block (= 5 i1)
```

Push, Pop

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

Maximize(i3,-1,17):

```
lo: -1, hi: 17, mid: 8
lo: -1, hi: 8, mid: 3
lo: -1, hi: 3, mid: 1
lo: 2, hi: 3, mid: 2
Optimum: 3
```

Push, Pop – but reuse search

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