Software Model Checking via Abstraction Refinement

Sriram K. Rajamani

Microsoft Research India

http://research.microsoft.com/slam/
Philosophy

• Model checking = exhaustive exploration of state space

• Challenge: realistic software has a huge state space?

• Approach: Abstraction-refinement
  – Construct an abstraction
    • a “simpler model” of the software that only contains the variables and relationships that are important to the property being checked
  – Model check the abstraction
    • easier because state space of the abstraction is smaller
  – Refine the abstraction
    • to reduce false errors
SLAM – Software Model Checking

SLAM models
- boolean programs: a new model for software

SLAM components
- model creation (c2bp)
- model checking (bebop)
- model refinement (newton)
SLIC

• Finite state language for stating rules
  – monitors behavior of C code
  – temporal safety properties (security automata)
  – familiar C syntax

• Suitable for expressing control-dominated properties
  – e.g. proper sequence of events
  – can encode data values inside state
state {
    enum {Locked, Unlocked}
    s = Unlocked;
}

KeAcquireSpinLock.entry {
    if (s==Locked) abort;
    else s = Locked;
}

KeReleaseSpinLock.entry {
    if (s==Unlocked) abort;
    else s = Unlocked;
}
The SLAM Process

- prog. P
- SLIC rule
- slic
- prog. P'
- c2bp
- boolean program
- bebop
- predicates
- newton
- path

Diagram showing the process with blocks labeled 'prog. P', 'SLIC rule', 'slic', 'prog. P', 'c2bp', 'bebop', 'newton', and arrows indicating the flow of the process.
Example

```c
do {
    KeAcquireSpinLock();

    nPacketsOld = nPackets;

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

KeReleaseSpinLock();
```

Does this code obey the locking rule?
do {
    KeAcquireSpinLock();

    if(*){
        KeReleaseSpinLock();
    }

} while (*);

KeReleaseSpinLock();
do {
    KeAcquireSpinLock();

    nPacketsOld = nPackets;

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

KeReleaseSpinLock();
do

KeAcquireSpinLock();

nPacketsOld = nPackets; b = true;

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

while (nPackets != nPacketsOld); !b

KeReleaseSpinLock();
do {
    KeAcquireSpinLock();
    b = true;
    if(*){
        KeReleaseSpinLock();
        b = b ? false : *;
    }
} while ( !b );
KeReleaseSpinLock();

Example

Model checking refined boolean program (bebop)
do {
  KeAcquireSpinLock();

  b = true;

  if (*){
    KeReleaseSpinLock();
    b = b ? false : *;
  }
} while ( !b );
Observations about SLAM

• Automatic discovery of invariants
  – driven by property and a finite set of (false) execution paths
  – predicates are \textit{not} invariants, but \textit{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
SLAM internals (with some simplifications)
C-

Types \( \tau \) ::= void | bool | int | ref \( \tau \)

Expressions \( e \) ::= \( c \) | \( x \) | \( e_1 \) op \( e_2 \) | &\( x \) | *\( x \)

LExpression \( l \) ::= \( x \) | *\( x \)

Declaration \( d \) ::= \( \tau \) \( x_1, x_2, \ldots, x_n \)

Statements \( s \) ::= skip | goto \( L_1, L_2 \ldots L_n \) | \( L : s \)
| assume(\( e \))
| \( l = e \)
| \( l = f (e_1, e_2, \ldots, e_n) \)
| return \( x \)
| \( s_1; s_2; \ldots; s_n \)

Procedures \( p \) ::= \( \tau \) \( f (x_1 : \tau_1, x_2 : \tau_2, \ldots, x_n : \tau_n) \) \( d \) \( s \)

Program \( g \) ::= \( d_1 \) \( d_2 \ldots d_n \) \( p_1 \) \( p_2 \) \( \ldots \) \( p_n \)
C--

Types $\tau ::= \text{void} | \text{bool} | \text{int}$

Expressions $e ::= c | x | e_1 \text{ op } e_2$

LExpression $l ::= x$

Declaration $d ::= \tau \ x_1, x_2, \ldots, x_n$

Statements $s ::= \text{skip} \mid \text{goto } L_1, L_2 \ldots L_n \mid L: s$
$\mid \text{assume}(e)$
$\mid l = e$
$\mid f(e_1, e_2, \ldots, e_n)$
$\mid \text{return}$
$\mid s_1; s_2; \ldots; s_n$

Procedures $p ::= f(x_1: \tau_1, x_2: \tau_2, \ldots, x_n: \tau_n) \ d \ s$

Program $g ::= d_1 \ d_2 \ldots \ d_n \ p_1 \ p_2 \ldots \ p_n$
BP

Types  \( \tau \) ::= void | bool

Expressions  \( e \) ::= c | x | \( e_1 \) op \( e_2 \)

LExpression  \( l \) ::= \( x \)

Declaration  \( d \) ::= \( \tau \ x_1,x_2,\ldots,x_n \)

Statements  \( s \) ::= skip | goto \( L_1,L_2 \ldots L_n \) | \( L: s \)
| \( \) | assume(\( e \))
| \( \) | \( l = e \)
| \( \) | \( f (e_1,e_2,\ldots,e_n) \)
| \( \) | return
| \( \) | \( s_1; s_2;\ldots; s_n \)

Procedures  \( p \) ::= \( f (x_1: \tau_1,x_2: \tau_2,\ldots,x_n: \tau_n) \) \( d \) \( s \)

Program  \( g \) ::= \( d_1 \) \( d_2 \) \ldots \( d_n \) \( p_1 \) \( p_2 \) \ldots \( p_n \)
Syntactic sugar

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

```
goto L1, L2;
```

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

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

```
L3: S3;
```
Example, in C

```c
int g;

main(int x, int y){
    cmp(x, y);
    if (!g) {
        if (x != y)
            assert(0);
    }
}

void cmp (int a , int b) {
    if (a == b)
        g = 0;
    else
        g = 1;
}
```
Example, in C--

```c
int g;

main(int x, int y){
    cmp(x, y);
    assume(!g);
    assume(x != y)
    assert(0);
}

void cmp(int a , int b) {
    goto L1, L2;
    L1: assume(a==b);
        g = 0;
        return;
    L2: assume(a!=b);
        g = 1;
        return;
}
```

Example, in C--
c2bp: Predicate Abstraction for C Programs

Given
- $P$: a C program
- $F = \{e_1, \ldots, e_n\}$
  - each $e_i$ a pure boolean expression
  - each $e_i$ represents set of states for which $e_i$ is true

Produce a boolean program $B(P,F)$
- same control-flow structure as $P$
- boolean vars $\{b_1, \ldots, b_n\}$ to match $\{e_1, \ldots, e_n\}$
- properties true of $B(P,F)$ are true of $P$
Assumptions

Given

- P: a C program
- \(F = \{e_1, ..., e_n\}\)
  - each \(e_i\) a pure boolean expression
  - each \(e_i\) represents set of states for which \(e_i\) is true

- Assume: each \(e_i\) uses either:
  - only globals (global predicate)
  - local variables from some procedure (local predicate for that procedure)

- Mixed predicates:
  - predicates using both local variables and global variables
  - complicate “return” processing
  - covered in Step 2
C2bp Algorithm

• Performs modular abstraction
  – abstracts each procedure in isolation

• Within each procedure, abstracts each statement in isolation
  – no control-flow analysis
  – no need for loop invariants
int g;

main(int x, int y) {
    cmp(x, y);
    assume(!g);
    assume(x != y)
    assert(0);
}

void cmp (int a, int b) {
    goto L1, L2
    L1: assume(a==b);
        g = 0;
        return;
    L2: assume(a!=b);
        g = 1;
        return;
}

Preds: {x==y} {g==0} {a==b}
int g;

main(int x, int y) {
    cmp(x, y);
    assume(!g);
    assume(x != y)
    assert(0);
}

dcl {g==0} ;

main( {x==y} ) {
    {g==0} 
    {a==b}
}

void cmp (int a, int b) {
    goto L1, L2

    L1: assume(a==b);
        g = 0;
        return;

    L2: assume(a!=b);
        g = 1;
        return;
}

void cmp ( {a==b} ) {
    ...
int g;

cmp(x, y);

assume(!g);
assume(x != y)
assert(0);
}

decl {g==0} ;

cmp( {x==y} );

main( {x==y} ) {
    cmp( {x==y} );
    assume( {g==0} );
    assume( !{x==y} );
    assert(0);
}

Preds: {x==y} 

{g==0} 

{a==b} 

void equal (int a , int b) {
  goto L1, L2

  L1: assume(a==b);
      g = 0;
      return;

  L2: assume(a!=b);
      g = 1;
      return;
}

void cmp ( {a==b} ) {
  goto L1, L2;

  L1: assume( {a==b} );
      {g==0} = T;
      return;

  L2: assume( !{a==b} );
      {g==0} = F;
      return;
}
C--

Types  \( \tau \)  ::=  void | bool | int

Expressions  \( e \)  ::=  c | x | \( e_1 \) op \( e_2 \)

LExpression  \( l \)  ::=  x

Declaration  \( d \)  ::=  \( \tau \)  \( x_1, x_2, \ldots, x_n \)

Statements  \( s \)  ::=  skip  |  goto \( L_1, L_2 \ldots L_n \)  |  \( L: s \)
          |  assume(\( e \))
          |  \( l = e \)
          |  \( f(e_1, e_2, \ldots, e_n) \)
          |  return
          |  \( s_1; s_2; \ldots; s_n \)

Procedures  \( p \)  ::=  \( f(x_1: \tau_1, x_2: \tau_2, \ldots, x_n: \tau_n) \)

Program  \( g \)  ::=  \( d_1 d_2 \ldots d_n \)  \( p_1 p_2 \ldots p_n \)
Abstracting Assignments

Suppose you are given an assignment $s$

• if $\text{Implies}_F(WP(s, e_i))$ is true before $s$ then
  – $e_i$ is true after $s$

• if $\text{Implies}_F(WP(s, \neg e_i))$ is true before $s$ then
  – $e_i$ is false after $s$

$\{e_i\} = \text{Implies}_F(WP(s, e_i)) \ ? \text{true} : \text{Implies}_F(WP(s, \neg e_i)) \ ? \text{false} : *;$
Abstracting Expressions via F

- $F = \{ e_1, \ldots, e_n \}$

- $\text{Implies}_F(e)$
  - *weakest* boolean function over $F$ that implies $e$

- $\text{ImpliedBy}_F(e)$
  - *strongest* boolean function over $F$ implied by $e$
  - $\text{ImpliedBy}_F(e) = !\text{Implies}_F(!e)$
Implies\textsubscript{F}(e) and ImpliedBy\textsubscript{F}(e)
Computing $\text{Implies}_F(e)$

- $F = \{ e_1, ..., e_n \}$

- minterm $m = d_1 \& \& ... \& \& d_n$
  - where $d_i = e_i$ or $d_i = \neg e_i$

- $\text{Implies}_F(e)$
  - disjunction of all minterms that imply $e$

- Naïve approach
  - generate all $2^n$ possible minterms
  - for each minterm $m$, use decision procedure to check validity of each implication $m \Rightarrow e$

- Many optimizations possible
  - Consider only k-ary cubes (for small values of $k$, such as $k = 2$)
  - Produces an approximate boolean program (which can be “fixed up” later on demand)
do {
    KeAcquireSpinLock();
    nPacketsOld = nPackets; b = true;
    if (request) {
        request = request->Next;
        KeReleaseSpinLock();
        nPackets++; b = b ? false : *;
    }
} while (nPackets != nPacketsOld); !b

KeReleaseSpinLock();
Assignment Example

Statement in P: \( y = y + 1; \)

Predicates in F: \( \{ x = y \} \)

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

\( \text{Implies}_F(x = y + 1) = \text{false} \)
\( \text{Implies}_F(x \neq y + 1) = x = y \)

Abstraction of assignment in B:
\( \{ x = y \} = \{ x = y \} ? \text{false} : *; \)
Absracting Assumes

- WP( assume(e) , Q ) = e⇒Q

- assume(e) is abstracted to:
  assume( ImpliedBy_F(e) )

- Example:
  F = {x==2, x<5}
  assume(x < 2) is abstracted to:
  assume( {x<5} && ![x==2] )
Abstracting Procedures

• Each predicate in $F$ is annotated as being either global or local to a particular procedure

• Procedures abstracted in two passes:
  – a $signature$ is produced for each procedure in isolation
  – procedure calls are abstracted given the callees’ signatures
Abstracting a procedure call

• Procedure call
  – a sequence of assignments from actuals to formals
  – see assignment abstraction

• Procedure return
  – NOP for C-- with assumption that all predicates mention either only globals or only locals
  – with pointers and with mixed predicates:
    • Most complicated part of c2bp
    • Covered in Step 2
int g;

main(int x, int y) {
    cmp(x, y);
    assume(!g);
    assume(x != y)
    assert(0);
}

decl {g==0} ;

main( {x==y} ) {
    cmp( {x==y} );
    assume( {g==0} );
    assume( !{x==y} );
    assert(0);
}

void cmp (int a , int b) {
    Goto L1, L2
    L1: assume(a==b);
        g = 0;
        return;
    L2: assume(a!=b);
        g = 1;
        return;
}

void cmp ( {a==b} ) {
    Goto L1, L2
    L1: assume( {a==b} );
        {g==0} = T;
        return;
    L2: assume( !{a==b} );
        {g==0} = F;
        return;
}
Precision loss during predicate abstraction

• For program P and F = \{e_1, ..., e_n\}, there exist two “ideal” abstractions:
  – Boolean(P,F) : most precise abstraction
  – Cartesian(P,F) : less precise abstraction, where each boolean variable is updated independently

• Theory:
  – with an “ideal” theorem prover, c2bp can compute Cartesian(P,F)

• Practice:
  – c2bp computes a less precise abstraction than Cartesian(P,F)
  – we use Das/Dill’s technique to incrementally improve precision
  – with an “ideal” theorem prover, the combination of c2bp + Das/Dill can compute Boolean(P,F)
Cartesian abstraction

\[ F = \{ (x!=5), (y<5), (y>5) \} \]

Statement: \( y = x \)

Abstract statement in the boolean program:
\[
\{ y < 5 \} = \{ x \neq 5 \} \ ? \ \ast : \text{false} \\
\{ y > 5 \} = \{ x \neq 5 \} \ ? \ \ast : \text{false}
\]

Suppose \( \{ x \neq 5 \} \) is true before the assignment
Then, only one of \( \{ y < 5 \} \) or \( \{ y > 5 \} \) can be true after the assignment

This correlation is lost in cartesian abstraction
Soundness of C2bp

Definition 1. Let $P$ be a program, $E$ be a set of predicates over constants and variables in $P$, and $B = \text{BP}(P, E)$ be the Boolean program abstraction computed by C2BP. Let $V$ be the $b$-variables corresponding to predicates in $E$ and $\xi$ be the mapping from $V$ to $E$. We say that a state $<L, \Omega>$, of $P$ is simulated by a state $<L', \Omega'>$, of $B$ if $L = L'$, and for all $b$-variables $b \in V$ in scope at $L$ in $B$ we have that:

$$(\Omega'(b) = \text{true} \Rightarrow (\Omega(\xi(b)) = \text{true}) \text{ and } (\Omega'(b) = \text{false} \Rightarrow (\Omega(\xi(b)) = \text{false}).$$

Theorem: The relation defined above is a simulation under the transition relations of the C program and Boolean program

[See: “Polymorphic Predicate Abstraction”, Ball, Millstein and Rajamani, ACM TOPLAS 2005]
The SLAM Process

- prog. P
- SLIC rule
- slic
- prog. P'
- c2bp
- boolean program
- bebop
- predicates
- path
- newton
- thumb-up
Bebop

• Model checker for boolean programs
• Based on CFL reachability
• Explicit representation of CFG
• Implicit representation of path edges and summary edges
• Generation of hierarchical error traces
Domains

g ∈ Global
l ∈ Local
f ∈ Frame
s ∈ Stack = Frame*
State = Global X Local X Stack

Transitions:
T ⊆ (Global X Local) X (Global X Local)
T^+ ⊆ Local X (Local X Frame)
T^- ⊆ (Local X Frame) X Local
Transition Relation

g ∈ Global
l ∈ Local
f ∈ Frame
s ∈ Stack = Frame*
State = Global X Local X Stack

Transitions:
T ⊆ (Global X Local) X (Global X Local)
T⁺ ⊆ Local X (Local X Frame)
T⁻ ⊆ (Local X Frame) X Local

STEP
T (g, l, g', l')
____________________
(g,l,s) → (g',l', s)

PUSH
T⁺(l, l', f)
____________________
(g,l,s) → (g',l', s.f)

POP
T⁻(l, f, l'')
____________________
(g,l,s.f) → (g',l', s)
Naïve model checking

Start with initial state: \((g_0, l_0, \varepsilon)\)

Find all the states \(s\) such that
\((g_0, l_0, \varepsilon) \rightarrow^* s\)

Even if Globals and Locals are finite, the set of reachable states could be infinite since the stack can grow infinite

No guarantee for termination

Still, assertion checking is decidable

Need to use a different algorithm (CFL reachability)

**STEP**
\[
T(g, l, g', l')
\]
\[
(g, l, s) \rightarrow (g', l', s)
\]

**PUSH**
\[
T^+(l, l', f)
\]
\[
(g, l, s) \rightarrow (g', l', s.f)
\]

**POP**
\[
T^-(l, f, l'')
\]
\[
(g, l, s.f) \rightarrow (g', l', s)
\]
CFL reachability

\[ P \subseteq (\text{Global} \times \text{Local}) \times (\text{Global} \times \text{Local}) \]
\[ \text{Sum} \subseteq (\text{Global} \times \text{Local}) \times \text{Frame} \times (\text{Global} \times \text{Local}) \]

\[
\text{CFL-INIT}
\]
\[
P(g_0, l_0, g_0, l_0)
\]

\[
\text{CFL-STEP}
\]
\[
P(g_1, l_1, g_2, l_2) \quad T(g_2, l_2, g_3, l_3)
\]
\[
P(g_1, l_1, g_3, l_3)
\]

\[
\text{CFL-PUSH}
\]
\[
P(g_1, l_1, g_2, l_2) \quad T^+(l_2, l_3, f,)
\]
\[
P(g_2, l_3, g_2, l_3)
\]

\[
\text{CFL-SUM}
\]
\[
P(g_1, l_1, g_2, l_2) \quad T^-(l_2, f, l_3,)
\]
\[
\text{Sum}(g_1, l_1, f, g_2, l_3)
\]

\[
\text{CFL-POP}
\]
\[
P(g_1, l_1, g_2, l_2) \quad T^+(l_2, l_3, f,) \quad \text{Sum}(g_2, l_3, f, g_3, l_4)
\]
\[
P(g_1, l_1, g_3, l_4)
\]

[Sharir-Pnueli 81] [Reps-Sagiv-Horwitz 95]
```c
decl g;
void main()
decl u, v;

[1] u := !v;

[2] equal(u, v);

[3] if (g) then
   R: skip;
   fi
[4] return;

void equal(a, b)

[5] if (a = b) then
[6]   g := 1;
   else
[7]   g := 0;
   fi
[8] return;
```
Symbolic CFL reachability

- Partition path edges by their “target”
  - \( \text{PE}(v) = \{ <d1,d2> | <\text{entry},d1> \rightarrow <v,d2> \} \)

- What is \(<d1,d2>\) for boolean programs?
  - A bit-vector!

- What is \(\text{PE}(v)\)?
  - A set of bit-vectors

- Use a BDD (attached to \(v\)) to represent \(\text{PE}(v)\)
```c
void main()

    decl g;
    decl u, v;

    u := !v;

    equal(u, v);

    if (g) then
      R: skip;
    fi

    return;

void equal(a, b)

    if (a = b) then
      g := 1;
    else
      g := 0;
    fi

    return;
```
Bebop: summary

- Explicit representation of CFG
- Implicit representation of path edges and summary edges
- Generation of hierarchical error traces

- Complexity: $O(E \cdot 2^{O(N)})$
  - $E$ is the size of the CFG
  - $N$ is the max. number of variables in scope
The SLAM Process

- **prog. P**
- **SLIC rule**
- **slic**
- **prog. P’**
- **c2bp**
  - boolean program
- **bebop**
  - predicates
  - path
- **newton**
  - path

The diagram illustrates the process of transforming a program (prog. P) using the SLIC rule (slic) to generate prog. P'. The transformed program is then processed by c2bp to generate a boolean program. The predicates and path are processed by bebop to generate newton, which is then used to provide a thumbs-up gesture.
Type 1: False error trace due to approximate predicate abstraction
Type 2: False error trace due to insufficient predicates

\[ A_0 \rightarrow A_1 \rightarrow A_2 \rightarrow A_3 \]
The SLAM Process (more accurate)
Type 1: False error trace due to approximate predicate abstraction
Type 1: False error trace due to approximate predicate abstraction

With this technique we can be as precise as boolean abstraction (rather than cartesian)
Type 2: False error trace due to insufficient predicates

$$ \gamma(A_0) \rightarrow \gamma(A_1) \rightarrow \gamma(A_2) \rightarrow \gamma(A_3) $$
Type 2: False error trace due to insufficient predicates

\[ A_0 \rightarrow A_1 \rightarrow A_2 \rightarrow A_3 \]

Add predicate: \((b_0 \lor b_1) \land \neg (b_2 \lor b_3)\)
The SLAM Process (more accurate)
c2bp with simplifying assumptions removed:

1. Allow mixed predicates
2. Allow pointers
C--

Types \( \tau \) ::= void | bool | int
Expressions \( e \) ::= c | x | \( e_1 \) op \( e_2 \)
LExpression \( l \) ::= \( x \)
Declaration \( d \) ::= \( \tau \) \( x_1, x_2, \ldots, x_n \)
Statements \( s \) ::= skip | goto \( L_1, L_2 \ldots L_n \) | \( L: s \)
| assume\((e)\)
| \( l = e \)
| \( f (e_1, e_2, \ldots, e_n) \)
| return
| \( s_1; s_2; \ldots; s_n \)
Procedures \( p \) ::= \( f (x_1: \tau_1, x_2: \tau_2, \ldots, x_n: \tau_n) \) \( d \) \( s \)
Program \( g \) ::= \( d_1 \) \( d_2 \ldots d_n \) \( p_1 \) \( p_2 \ldots p_n \)
C-

Types \( \tau \) ::= void | bool | int | ref \( \tau \)

Expressions \( e \) ::= c | x | e_1 op e_2 | &x | *x

LExpression \( l \) ::= x | *x

Declaration \( d \) ::= \( \tau \) \( x_1, x_2, \ldots, x_n \)

Statements \( s \) ::= skip | goto L_1, L_2 \ldots, L_n | L: s

| \[ \text{assume}(e) \]
| \[ l = e \]
| \[ l = f(e_1, e_2, \ldots, e_n) \]
| \[ \text{return } x \]
| \[ s_1; s_2; \ldots; s_n \]

Procedures \( p \) ::= \( \tau \) \( f(x_1: \tau_1, x_2: \tau_2, \ldots, x_n: \tau_n) \) \( d \) \( s \)

Program \( g \) ::= \( d_1 \) \( d_2 \) \ldots \( d_n \) \( p_1 \) \( p_2 \) \ldots \( p_n \)
Abstracting Procedure Returns

• Let $a$ be an actual at call-site $P(...)$
  – $\text{pre}(a) = \text{the value of } a \text{ before transition to } P$

• Let $f$ be a formal of a procedure $P$
  – $\text{pre}(f) = \text{the value of } f \text{ upon entry to } P$
Q() {
  {x==1} int x = 1;
  x = R(x)
}  

int R (int f) {
  int r = f+1;  
  f = 0;  
  return r;
}

pre(f) == x
x = r

WP(f=x, f==pre(f) ) = x==pre(f)  

x==pre(f) is true at the call to R

WP(x=r, x==2) = r==2  

pre(f)==x and x==1 and r==pre(f)+1 implies r==2

Q() {
  {x==1},{x==2} = T,F;
  s = R(T);
  {x==2} = s & {x==1};
}

bool R ( {f==pre(f)} ) {
  {r==pre(f)+1} = {f==pre(f)};
  {f==pre(f)} = *;
  return {r==pre(f)+1};
}
int R (int f) {
    int r = f + 1;
    f = 0;
    return r;
}

int x = 1;

x = R(x);

Q() {
    x = 1;
    x = R(x);
}

WP(f=x, f==pre(f) ) = x==pre(f)
x==pre(f) is true at the call to R

WP(x=r, x==2) = r==2  pre(f)==x and x==1 and r==pre(f)+1 implies r==2

Q() {
    x = 1;
    x = R(x);
}

bool R ( {f==pre(f)} ) {
    {r==pre(f)+1} = {f==pre(f)};
    {f==pre(f)} = *
    return {r==pre(f)+1};
}
Pointers and SLAM

• With pointers, C supports call by reference
  – Strictly speaking, C supports only call by value
  – With pointers and the address-of operator, one can simulate call-by-reference

• Boolean programs support only call-by-value-result
  – SLAM mimics call-by-reference with call-by-value-result

• Extra complications:
  – address operator (&) in C
  – multiple levels of pointer dereference in C
What changes with pointers?

• C2bp
  – abstracting assignments
  – abstracting procedure returns

• Newton
  – simulation needs to handle pointer accesses
  – need to copy local heap across scopes to match Bebop’s semantics
  – need to refine imprecise alias analysis using predicates

• Bebop
  – remains unchanged!
Recall : Abstracting Assignments

Suppose you are given an assignment s

- if $\text{Implies}_F(WP(s, e_i))$ is true before s then
  - $e_i$ is true after s

- if $\text{Implies}_F(WP(s, \neg e_i))$ is true before s then
  - $e_i$ is false after s

\[
\{e_i\} = \begin{cases} 
\text{Implies}_F(WP(s, e_i)) & \text{true} \\
\text{Implies}_F(WP(s, \neg e_i)) & \text{false} 
\end{cases} ;
\]
Assignments + Pointers

Statement in P:
*p = 3

Predicates in E:
{x==5}

Weakest Precondition:
WP( *p=3 , x==5 ) = x==5

What if *p and x alias?

Correct Weakest Precondition:
(p==&x and 3==5) or (p!=&x and x==5)

We use Das’s pointer analysis [PLDI 2000] to prune disjuncts representing infeasible alias scenarios.
Abstracting Procedure Return

• Need to account for
  – lhs of procedure call
  – mixed predicates
  – side-effects of procedure

• Boolean programs support only call-by-value-result
  – C2bp models all side-effects using return processing
Extending Pre-states

• Suppose formal parameter is a pointer
  – eg. P(int *f)

• pre( *f )
  – value of *f upon entry to P
  – can’t change during P

• * pre( f )
  – value of dereference of pre( f )
  – can change during P
Q() {  
{x==1}  
int x = 1;  
a = &x  
{x==2}  
R(&x);  
}  
}

int R (int *a) {  
{a==pre(a)}  
{*pre(a)==pre(*a)}  
*pre(a)=&x  
{x==1}  
{x==2}  
{pre(a)==&x}  
*pre(a)==pre(*a)  
{pre(a)==&x}  
{x==2}  
}

pre(x)==1 and pre(*a)==pre(x) and *pre(a)==pre(*a)+1 and pre(a)==&x implies x==2

Q() {  
{x==1}, {x==2} = T,F;  
s = R(T,T);  
{x==2} = s & {x==1};  
}  

bool R ( {a==pre(a)}, {*pre(a)==pre(*a)} ) {  
{pre(a)==pre(*a)+1} = {*pre(a)==pre(*a)} ;  
return {*pre(a)==pre(*a)+1};  
}
Further Reading

See papers, slides from:
http://research.microsoft.com/slam

In particular, see the papers:

- **Polymorphic Predicate Abstraction**, Thomas Ball, Todd Millstein, Sriram Rajamani, ACM TOPLAS 2005
- **Bebop: A Symbolic Model Checker for Boolean Programs**, Thomas Ball, Sriram K. Rajamani, SPIN 2000 Workshop on Model Checking of Software.