Embedded Software Verification Challenges and Solutions

Shuvendu Lahiri, Microsoft, Redmond

Chao Wang, NEC Labs, Princeton

Daniel Kroening, Oxford University

ICCAD Tutorial
November 11, 2008

Outline

- What programs?
- The Formal Basics of Program Verification
- Static Program Analysis
- **Predicate Abstraction**
- **Bounded Model Checking (BMC)**

Motivation

- **Software has too many state variables**

) State Space Explosion

- **Graf/Saïdi 97: Predicate Abstraction**
- **Idea: Only keep track of predicates on data**

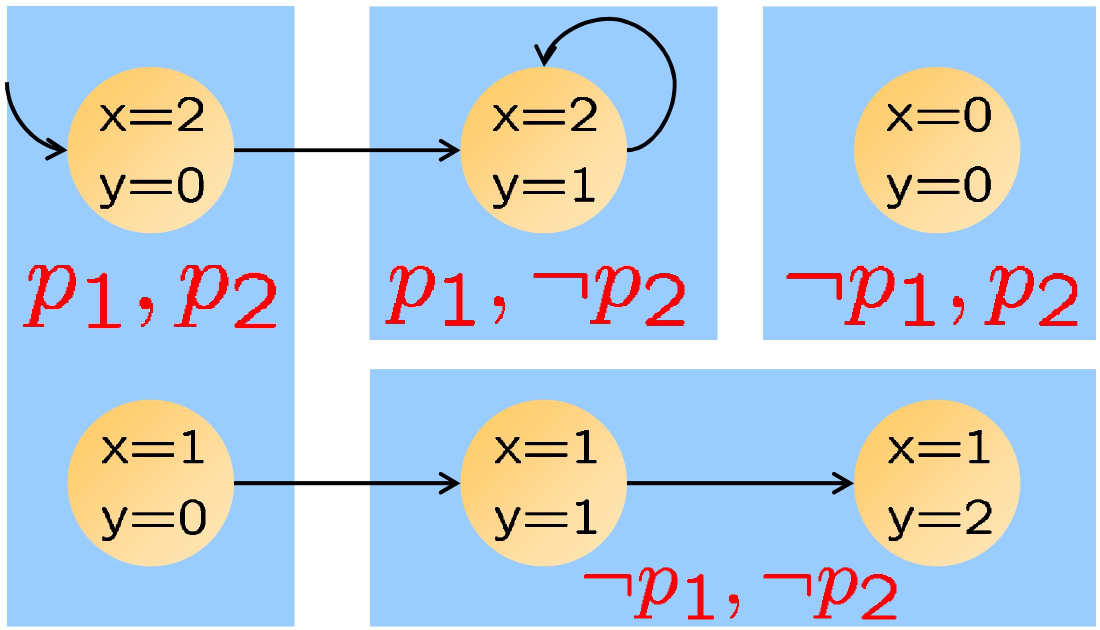
$$p_1(s), \ldots, p_n(s)$$

- **Abstraction function:**

$$\alpha(s) \;=\; (p_1(s), p_2(s), \ldots, p_n(s))$$

Predicate Abstraction

Concrete States:



Predicates:

$$p_1(s) = (s.x > s.y)$$
$$p_2(s) = (s.y = 0)$$

Abstract transitions?

Under- vs. Overapproximation

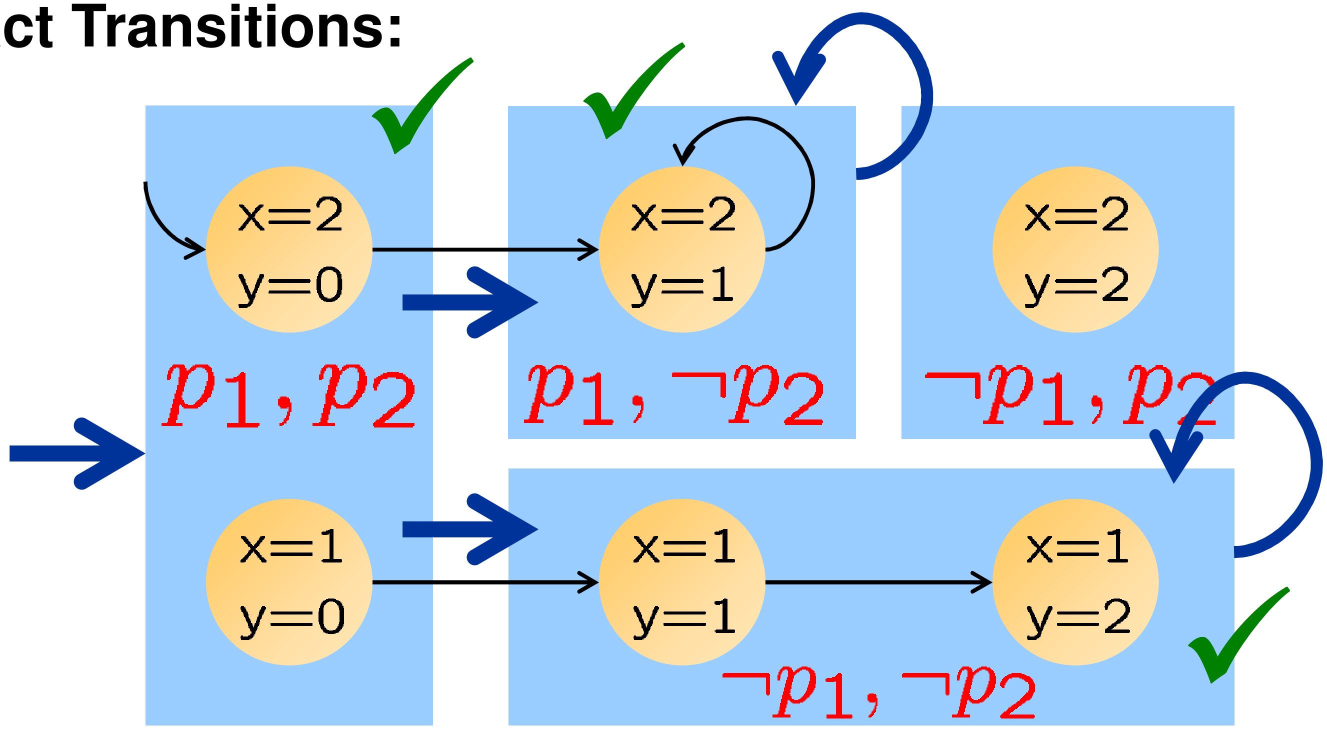
- **How to abstract the transitions?**
  - **Depends on the property we want to show**
  - **Typically done in a <u>conservative</u> manner**
- **<u>Existential abstraction</u>:**

$$\begin{aligned} \widehat{I}(\widehat{s}) \quad &:\Longleftrightarrow \quad \exists s : I(s) \wedge \alpha(s) = \widehat{s} \\ \widehat{R}(\widehat{s}, \widehat{s}') \quad &:\Longleftrightarrow \quad \exists s, s' : R(s, s') \\ & \qquad\qquad \wedge \alpha(s) = \widehat{s} \wedge \alpha(s') = \widehat{s}' \end{aligned}$$

) **<u>Preserves safety properties</u>**

Predicate Abstraction

Abstract Transitions:



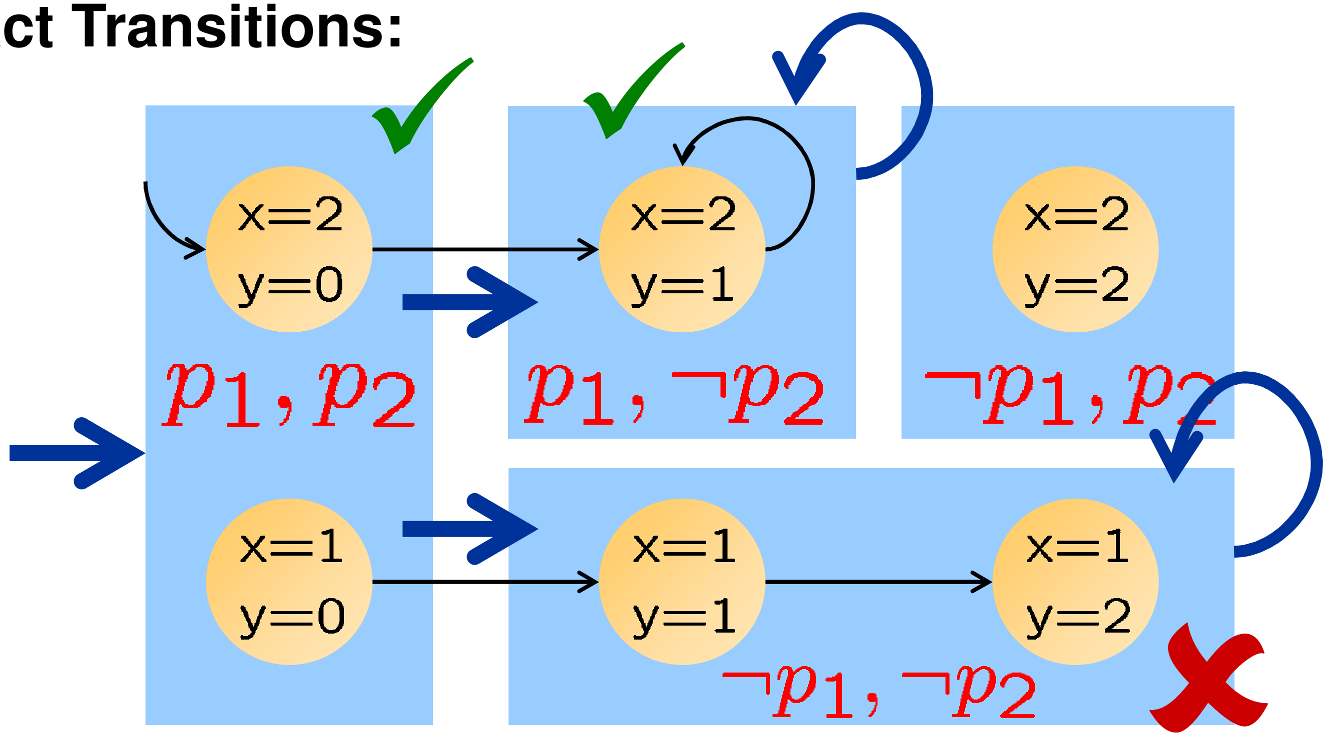
Property:

$$p_1 \vee \neg p_2 \quad \Longleftrightarrow \quad (s.x > s.y) \vee (s.y \neq 0)$$

Property holds. Ok.

Predicate Abstraction

Abstract Transitions:



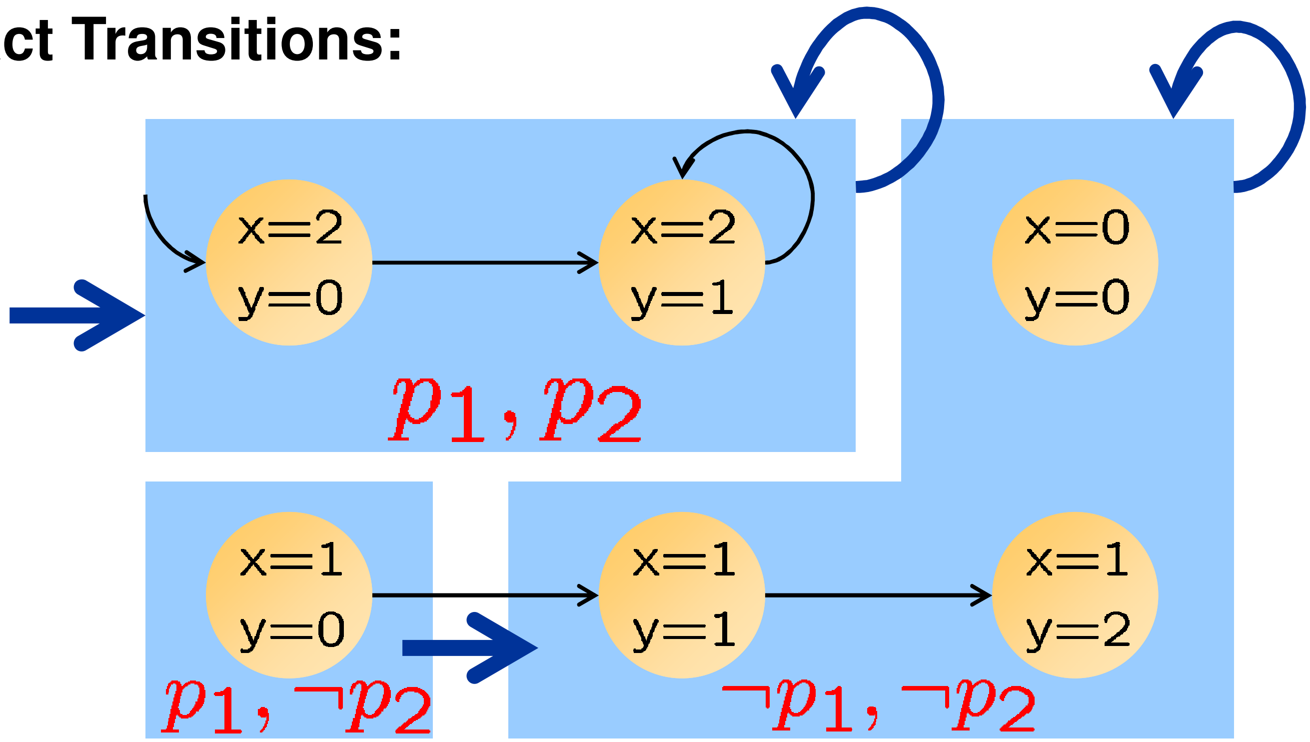
Property:

$$p_1 \iff (s.x > s.y)$$

This trace is <u>spurious</u>!

Predicate Abstraction

Abstract Transitions:



Property:

$$p_1 \iff (s.x > s.y)$$ ✓

New Predicates:

$$p_1(s) = (s.x > s.y)$$
$$p_2(s) = (s.x = 2)$$

Predicate Abstraction for Software

- Let's take existential abstraction seriously
- Basic idea: with n predicates, there are 2^n £ 2^n possible abstract transitions
- Let's just check them!

Existential Abstraction

Predicates

$$p_1 \iff i = 1$$
$$p_2 \iff i = 2$$
$$p_3 \iff \text{even}(i)$$

Basic Block

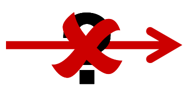
```
i++;
```

Formula

$$i' = i + 1$$

p_1	p_2	p_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Current Abstract State



p'_1	p'_2	p'_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Next Abstract State

Query

$$i \neq 1 \wedge i \neq 2 \wedge \overline{\text{even}(i)} \wedge$$
$$i' = i + 1 \wedge$$
$$i' \neq 1 \wedge i' \neq 2 \wedge \overline{\text{even}(i')}$$

Existential Abstraction

Predicates

$$p_1 \iff i = 1$$
$$p_2 \iff i = 2$$
$$p_3 \iff \text{even}(i)$$

Basic Block

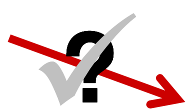
```
i++;
```

Formula

$$i' = i + 1$$

p_1	p_2	p_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Current Abstract State



p'_1	p'_2	p'_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Next Abstract State

Query

$$i \neq 1 \wedge i \neq 2 \wedge \overline{\text{even}(i)} \wedge$$
$$i' = i + 1 \wedge$$
$$i' \neq 1 \wedge i' \neq 2 \wedge \text{even}(i')$$

… and so on …

Predicate Abstraction for Software

- A precise existential abstraction can be way too slow
- Use an over-approximation instead
  - Fast to compute
  - But has <u>additional transitions</u>
- E.g.:
  - SLAM (FastAbs)
  - Predicate partitioning

Example for Predicate Abstraction

```
int main() {
  int i;

  i=0;

  while(even(i))
    i++;
}
```

+

$p_1 \Leftrightarrow i=0$
$p_2 \Leftrightarrow even(i)$

=

```
void main() {
  bool p1, p2;

  p1=TRUE;
  p2=TRUE;

  while(p2)
  {
    p1=p1?FALSE:*;
    p2=!p2;
  }
}
```

C program **Predicates** **Boolean program**

[Graf, Saidi '97]
[Ball, Rajamani '00]

Predicate Abstraction for Software

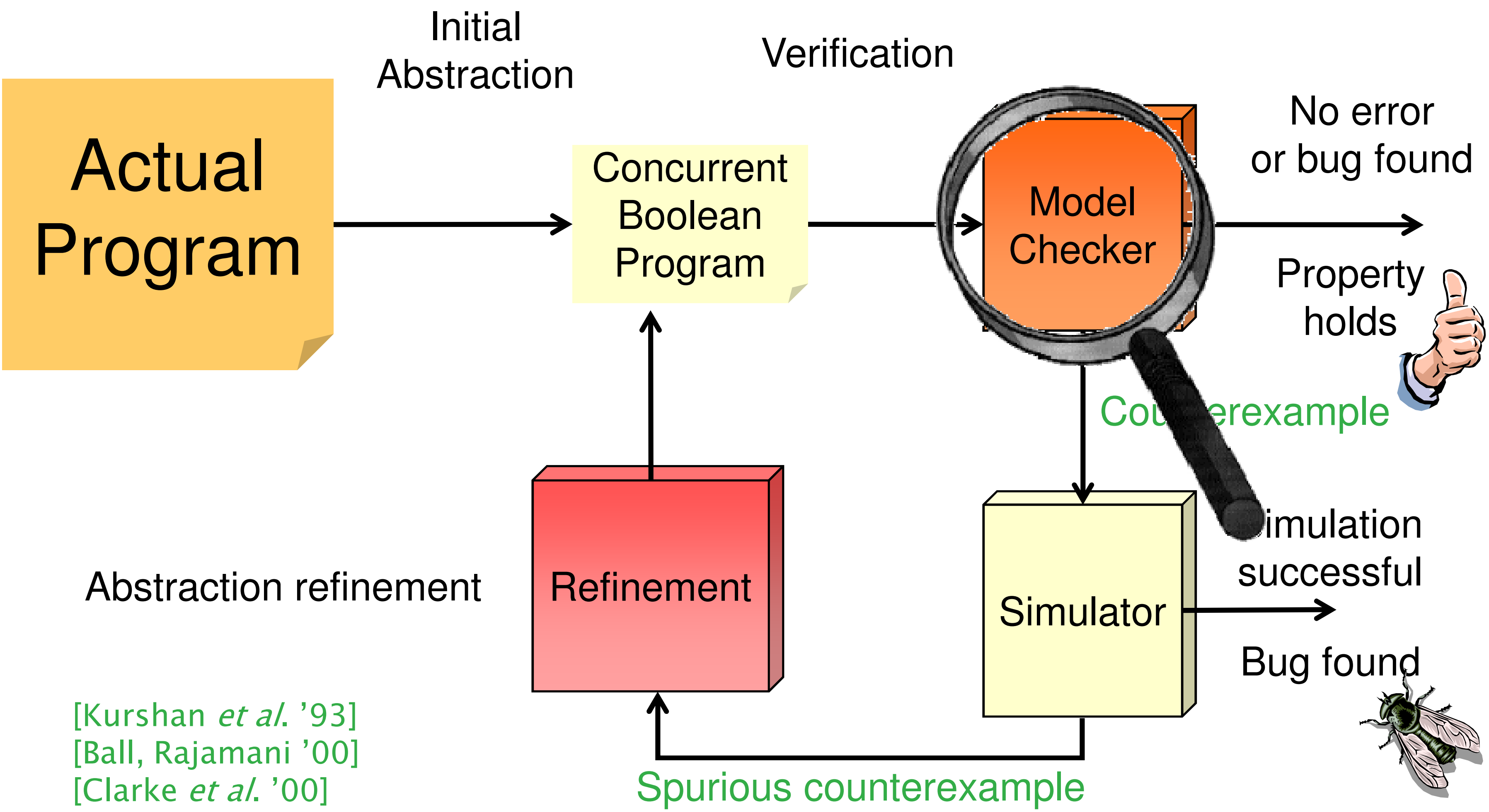
- **How do we get the predicates?**
- **Automatic abstraction refinement!**

  [Kurshan *et al.* '93]

  [Ball, Rajamani '00]

  [Clarke *et al.* '00]

Abstraction Refinement Loop



[Kurshan *et al.* '93]
[Ball, Rajamani '00]
[Clarke *et al.* '00]

Checking the Boolean Program

- **No more integers!**
- **But:**
  - **function calls**
  - **non-determinism**
  - **Concurrency if original program is concurrent**
- **BDD-based model checking now scales**
  - **For sequential programs**
  - **Bebop (MSR)**
  - **Even SMV!**

SMV for the Boolean Program

GOTO Program

```
int main() {
   int x, y;
   y=8;
   if(x>0)
     while(y>0) {
       y--;
       x++;
     }
}
```

```
    y = 8;
    IF !(x > 0) THEN GOTO 2
1:  IF !(y > 0) THEN GOTO 2
    y = y - 1;
    x = x + 1;
    GOTO 1
2:  SKIP
```

- **Function calls can be inlined**
- **Be careful with side-effects!**

SMV for the Boolean Program

① **Program Variables**

```
VAR b0_argc_ge_1: boolean;              -- argc >= 1
VAR b1_argc_le_2147483646: boolean;  -- argc <= 2147483646
VAR b2: boolean;                        -- argv[argc] == NULL
VAR b3_nmemb_ge_r: boolean;             -- nmemb >= r
VAR b4: boolean;                        -- p1 == &array[0]
VAR b5_i_ge_8: boolean;                 -- i >= 8
VAR b6_i_ge_s: boolean;                 -- i >= s
VAR b7: boolean;                        -- 1 + i >= 8
VAR b8: boolean;                        -- 1 + i >= s
VAR b9_s_gt_0: boolean;                 -- s > 0
VAR b10_s_gt_1: boolean;                -- s > 1
...
```

SMV for the Boolean Program

② **Control Flow**

```
-- program counter: 56 is the "terminating" PC
VAR PC: 0..56;
ASSIGN init(PC):=0; -- initial PC

ASSIGN next(PC):=case
    PC=0: 1; -- other
    PC=1: 2; -- other
    . . .
    PC=19: case  -- goto (with guard)
      guard19: 26;
      1: 20;
    esac;
    . . .
```

SMV for the Boolean Program

③ **Data**

```
TRANS (PC=0) -> next(b0_argc_ge_1)=b0_argc_ge_1
              & next(b1_argc_le_213646)=b1_argc_le_21646 & next(b2)=b2
              & (!b30 | b36)
              & (!b17 | !b30 | b42)
              & (!b30 | !b42 | b48)
              & (!b17 | !b30 | !b42 | b54)
              & (!b54 | b60)

TRANS (PC=1) -> next(b0_argc_ge_1)=b0_argc_ge_1
              & next(b1_argc_le_214646)=b1_argc_le_214746 & next(b2)=b2
              & next(b3_nmemb_ge_r)=b3_nmemb_ge_r & next(b4)=b4
              & next(b5_i_ge_8)=b5_i_ge_8 & next(b6_i_ge_s)=b6_i_ge_s
              . . .
```

SMV for the Boolean Program

④ **Property**

```
-- the specification

-- file main.c line 20 column 12 function c::very_buggy_function
SPEC AG ((PC=51) -> !b23)
```

SLAM

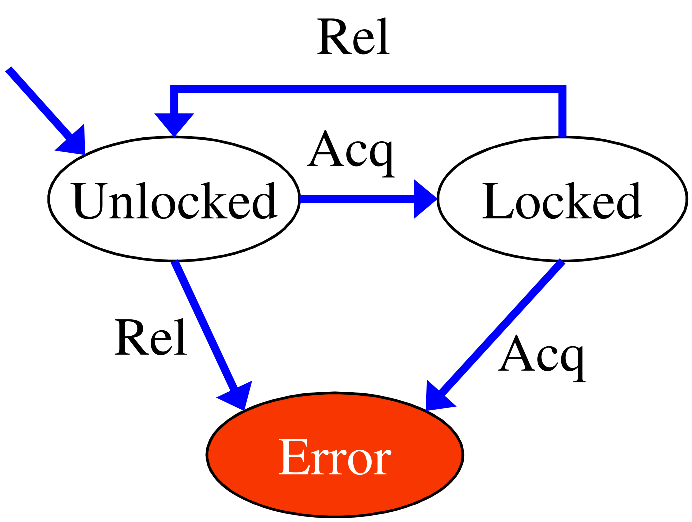
- Microsoft blames most Windows crashes on third party device drivers
- The Windows device driver API is quite complicated
- Low level C code
- SLAM: Tool to automatically check device drivers for certain errors
- To be shipped with Device Driver Development Kit
- Full detail (and all the slides) available at http://research.microsoft.com/slam/

SLIC

- Finite state language for stating rules
  - monitors behavior of C code
  - temporal safety properties (security automata) – similar to what SPIN does
  - familiar C syntax
- Suitable for expressing control-dominated properties
  - e.g., proper sequence of events
  - can encode data values inside state

State Machine for Locking

Locking Rule in SLIC



Too hard for programmers, and therefore:

```
state {
  enum {Locked,Unlocked}
     s = Unlocked;
}

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

KeReleaseSpinLock.entry {
  if (s==Unlocked) abort;
  else s = Unlocked;
}
```

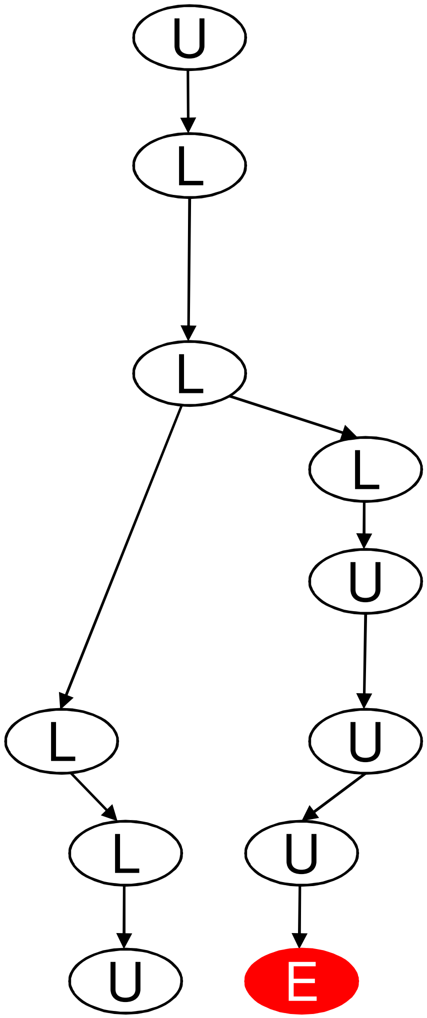
Example

Does this code obey the locking rule?

```
do {
   KeAcquireSpinLock();

   nPacketsOld = nPackets;

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

KeReleaseSpinLock();
```

Example

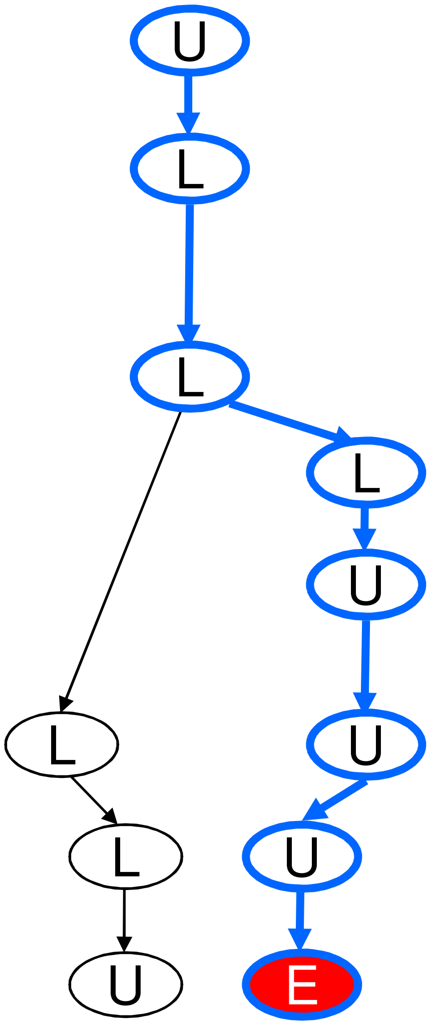
Model checking
boolean program
(bebop)




```
do {
   KeAcquireSpinLock();

    if(*){

        KeReleaseSpinLock();

    }
} while (*);

KeReleaseSpinLock();
```

Example

Is error path feasible in C program? (newton)

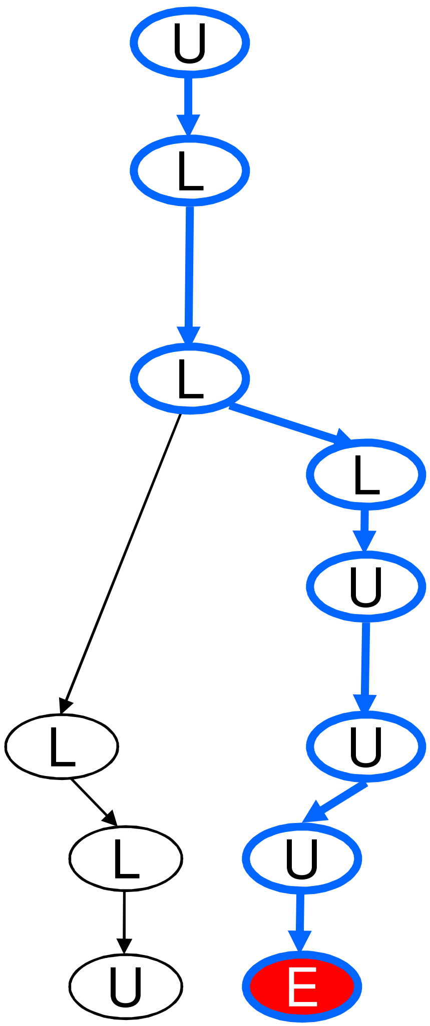



```
do {
   KeAcquireSpinLock();

   nPacketsOld = nPackets;

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

KeReleaseSpinLock();
```

Example

b : (nPacketsOld == nPackets)

Add new predicate to boolean program (c2bp)

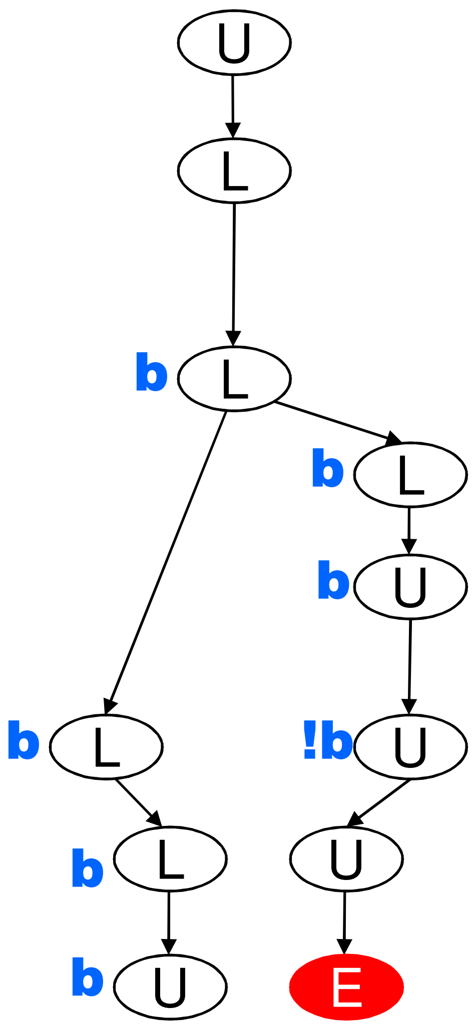



```
do {
   KeAcquireSpinLock();

   nPacketsOld = nPackets;   b = true;

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

KeReleaseSpinLock();
```

Example

b : (nPacketsOld == nPackets)

Model checking refined boolean program (bebop)

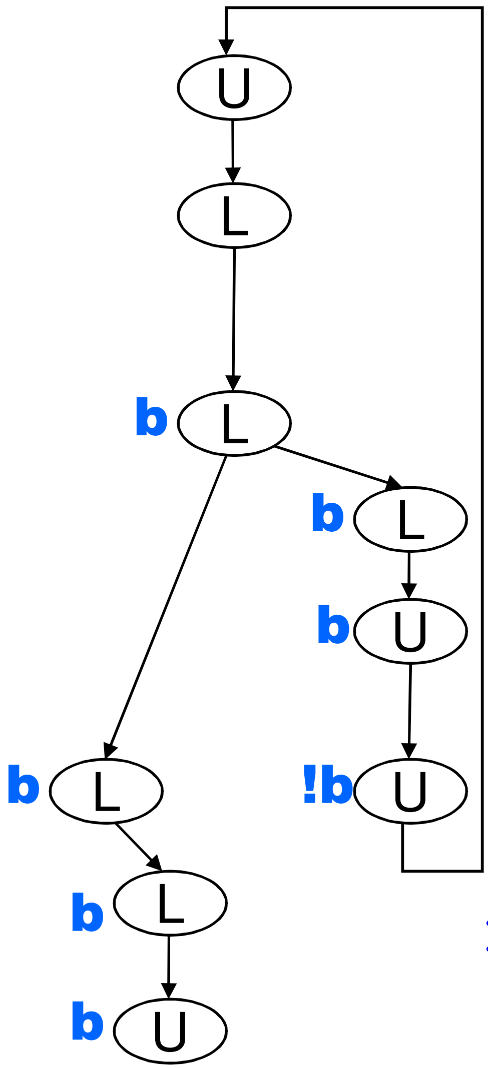



```
do {
   KeAcquireSpinLock();

   b = true;

   if(*){

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

KeReleaseSpinLock();
```

Example

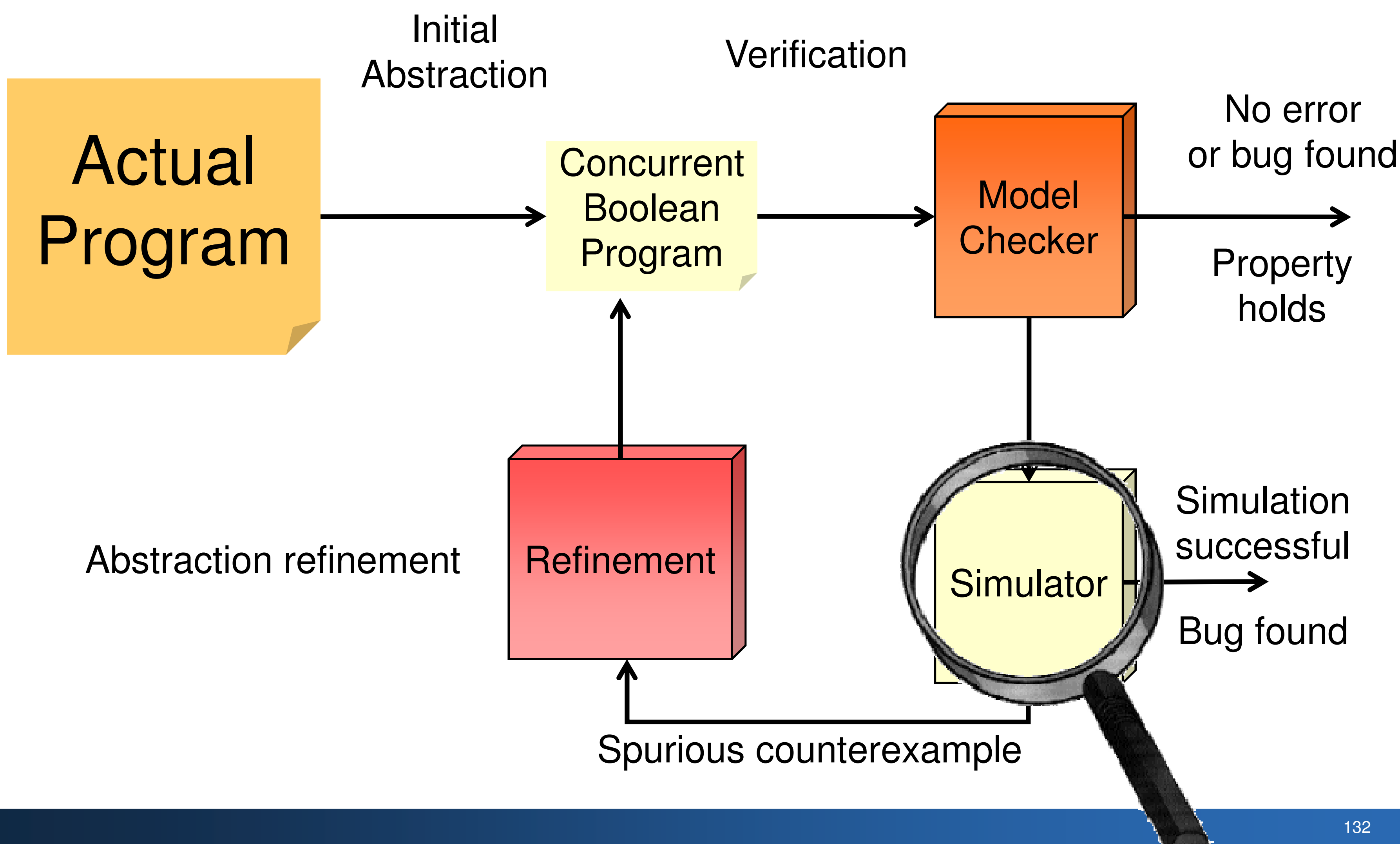
b : (nPacketsOld == nPackets)

Model checking
refined
boolean program
(bebop)




```
do {
   KeAcquireSpinLock();

   b = true;

   if(*){

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

KeReleaseSpinLock();
```

Abstraction Refinement Loop



Simulation

- Given an abstract counterexample, check if there exists corresponding concrete counterexample
- Transform path into SSA
- Add if/while guards as constraints
- Q: What about threads?

Example

```
int main() {
   int x, y;
   y=1;
   x=1;
   if(y>x)
     y--;
   else
     y++;

   assert(y>x);
}
```

α

```
main() {
   bool b0; // y>x
   b0=*;
   b0=*;
   if(b0)
     b0=*;
   else
     b0=*;

   assert(b0);
}
```

Predicate:
y>x

Example: Simulation

```
int main() {
  int x, y;
  y=1;
  x=1;
  if(y>x)
    y--;
  else
    y++;

  assert(y>x);
}
```

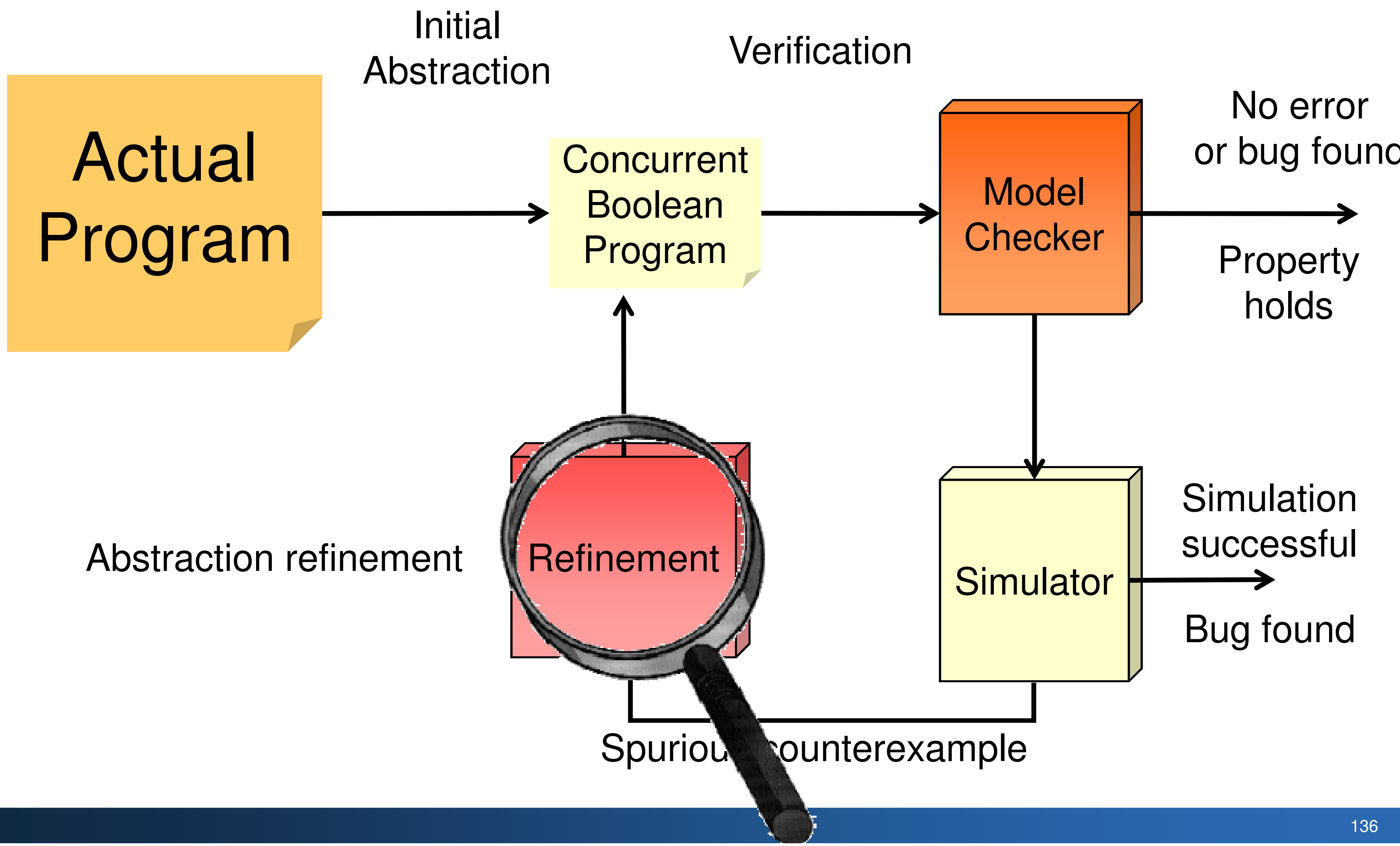
SSA

```
int main() {
  int x, y;
  y1=1;
  x1=1;
  if(y1>x1)
    y2=y1-1;
  else
    y=y+1;

  assert(y2>x1)
}
```

$$
\begin{aligned}
& y_1 = 1 \\
\wedge\; & x_1 = 1 \\
\wedge\; & y_1 > x_1 \\
\wedge\; & y_2 = y_1 - 1 \\
\wedge\; & \neg(y_2 > x_0)
\end{aligned}
$$

Spurious trace!

Abstraction Refinement Loop



Manual Proof!

```
int main() {
  int x, y;
  y=1;
```
$\{y = 1\}$
```
  x=1;
```
$\{x = 1 \wedge y = 1\}$
```
  if(y>x)
    y--;
  else
```
$\{x = 1 \wedge y = 1 \wedge \neg y > x\}$
```
    y++;
```
$\{x = 1 \wedge y = 2 \wedge y > x\}$
```
  assert(y>x);
}
```

We are using *strongest post-conditions* here

Another Manual Proof

```
int main() {
  int x, y;
  y=1;
```
$\{\neg y > 1 \rightarrow y + 1 > 1\}$
```
  x=1;
```
$\{\neg y > x \rightarrow y + 1 > x\}$
```
  if(y>x)
    y--;
  else
```
$\{y + 1 > x\}$
```
    y++;
```
$\{y > x\}$
```
  assert(y>x);
}
```

We are using *weakest pre-conditions* here

$$wp(x\texttt{:=}E, P) = P[x/E]$$
$$wp(S\texttt{;}T, Q) = wp(S, wp(T, Q))$$
$$wp(\texttt{if(}c\texttt{)}\ A\ \texttt{else}\ B, P) = (B \rightarrow wp(A, P)) \wedge (\neg B \rightarrow wp(B, P))$$

The proof for the “true” branch is missing

Refinement Algorithm

- Using WP:
  1. Start with failed guard P
  2. Compute WP(P) along the path
- Using SP:
  1. Start at beginning
  2. Compute SP(P) along the path
- Both methods eliminate the trace
- Advantages/Disadvantages?

Refinement

- Need to distinguish two sources of spurious behavior
  1. Too few predicates
  2. Laziness during abstraction
- SLAM:
  - First tries to find new predicates (NEWTON) using strongest post-conditions
  - If this fails, second case is assumed. Transitions are refined (CONSTRAIN)

SLAM: CONSTRAIN

- **The abstraction by FASTABS is often too coarse**

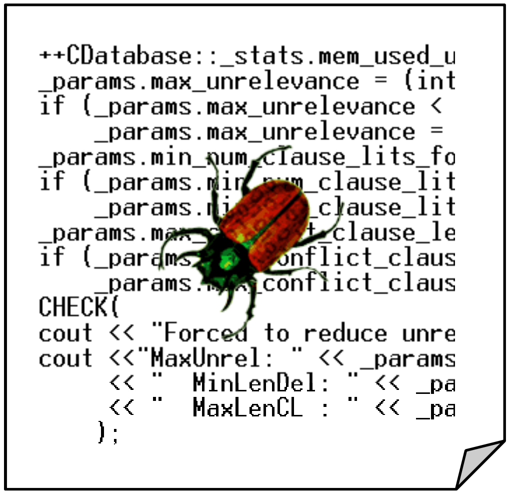
```
p1,p2 := schoose[F, F], schoose[F, F];
```

- **If no new predicates are found, the transitions in the abstract counterexample are checked**
- **The spurious transition is eliminated by adding a constraint**

```
p1,p2 := schoose[F, F], schoose[F, F]
    constrain (p0 & 'p1) | (!p0 & !'p1);
```

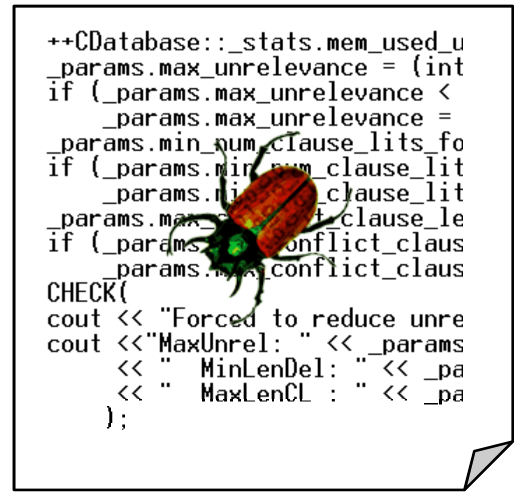
Bounded Model Checking

1. **Unwinding ANSI-C Programs**
2. **Supported Language Features**
3. **How to make it look nice**
4. **Case Studies**
5. **Recent Results**



BMC Overview

- **Problem: Fixpoint computation is too expensive for Software**
- **Idea:**
  - **Unwind program into equation**
  - **Check equation using SAT**
- **Advantages:**
  - **Completely automated**
  - **Allows full set of ANSI-C, including full treatment of pointers and dynamic memory**
- **Properties:**
  - **Simple assertions**
  - **Security (Pointers/Arrays)**
  - **Run time guarantees (WCET)**



ANSI-C Transformation

1. **Preparation**
   - **Side effect removal**
   - `continue`, `break` **replaced by** `goto`
   - `for`, `do while` **replaced by** `while`
2. **Unwinding**
   - **Loops are unwound: to guarantee that enough unwinding is done, <u>unwinding assertions</u> are added**
   - **Same for backward** `goto` **jumps and recursive functions**

Bounded Model-Checking

```
void f(...) {
  ...
  while(cond) {
    Body;
  }
  Remainder;
}
```

- **while() loops are unwound iteratively**
- **Break / continue replaced by goto**

Bounded Model-Checking

```
void f(...) {
  ...
  if(cond) {
    Body;
    while(cond) {
      Body;
    }
  }
  Remainder;
}
```

- **while() loops are unwound iteratively**
- **Break / continue replaced by goto**

Bounded Model-Checking

```
void f(...) {
  ...
  if(cond) {
    Body;
    if(cond) {
      Body;
      while(cond) {
        Body;
      }
    }
  }
  Remainder;
}
```

- **while() loops are unwound iteratively**
- **Break / continue replaced by goto**

Bounded Model-Checking

```
void f(...) {
  ...
  if(cond) {
    Body;
    if(cond) {
      Body;
      if(cond) {
        Body;
        while(cond) {
          Body;
        }
      }
    }
  }
  Remainder;
}
```

- **while() loops are unwound iteratively**
- **Break / continue replaced by goto**
- **Assertion inserted after last iteration: violated if program runs longer than bound permits**

Bounded Model-Checking

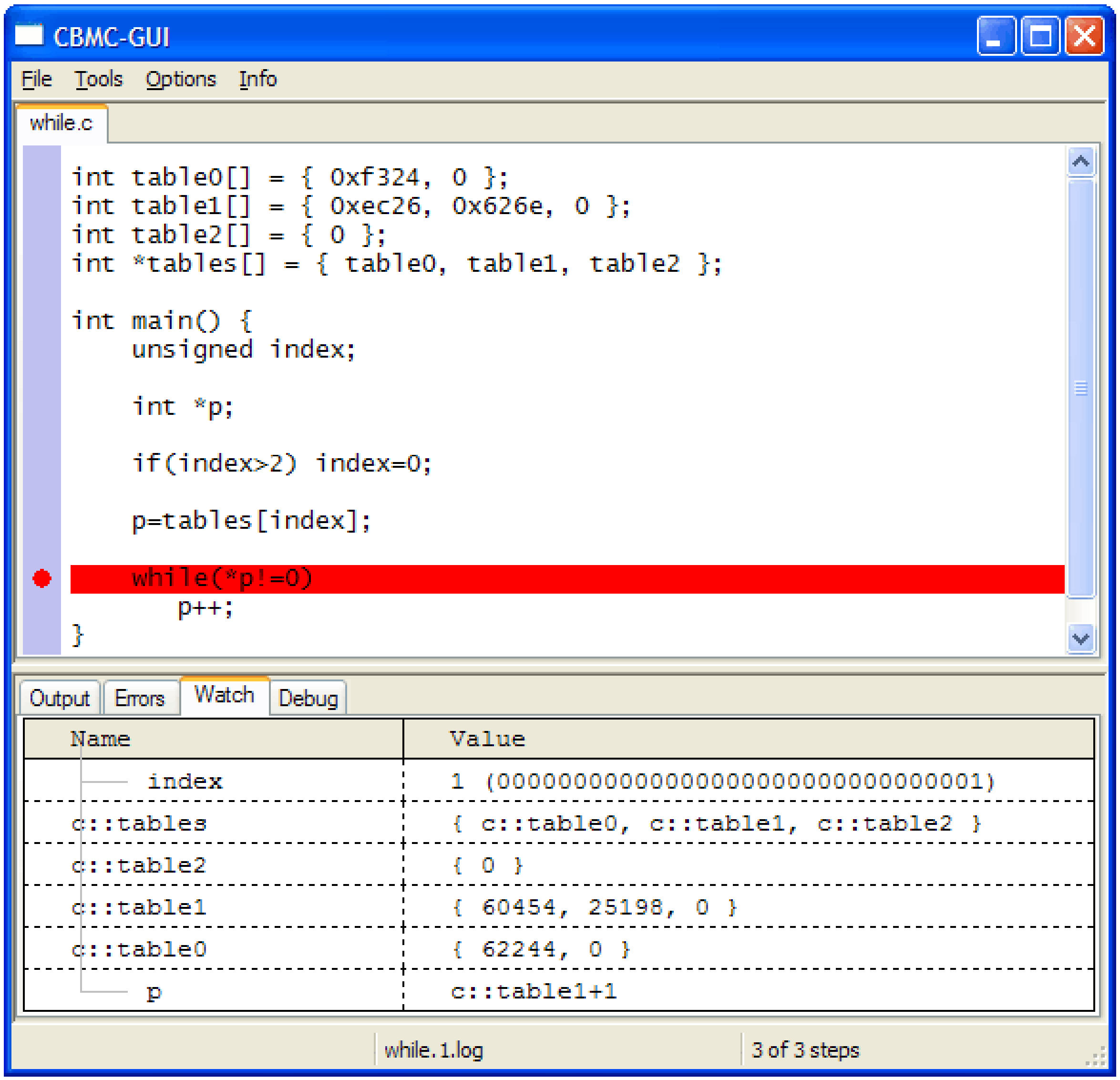
```
void f(...) {
  ...
  if(cond) {
    Body;
    if(cond) {
      Body;
      if(cond) {
        Body;
        assert(!cond);

      }
    }
  }
  Remainder;
}
```

Unwinding assertion

- **while() loops are unwound iteratively**
- **Break / continue replaced by goto**
- **Assertion inserted after last iteration: violated if program runs longer than bound permits**
- **Positive correctness result!**

Example Unwinding Assertion

With Bound 1



A

Implementation

3. Transformation into Equation

- **After unwinding: Transform into SSA**

Example:

```
x=x+y;        x1=x0+y0;
x=x*2;   ρ→   x2=x1*2;
a[i]=100;     a1[i0]=100;
```

- **Generate constraints by simply conjoining equations resulting from assignments**

- **For arrays, use simple lambda notation**

$$v_\alpha[a] = e \quad \Rightarrow \quad v_\alpha = \lambda i : \begin{cases} \rho(e) & : \; i = \rho(a) \\ v_{\alpha-1}[i] & : \; \text{otherwise} \end{cases}$$

Example

```
int main() {
   int x, y;
   y=8;
   if(x)
     y--;
   else
     y++;

   assert
       (y==7 ||
       y==9);
}
```

ρ

```
int main() {
   int x, y;
   y1=8;
   if(x0)
     y2=y1-1;
   else
     y3=y1+1;
   y4= x0 ?y2:y3;
   assert
       (y4==7 ||
       y4==9);
}
```

$$
\begin{aligned}
(\quad & y_1 = 8 \\
\wedge \quad & y_2 = y_1 - 1 \\
\wedge \quad & y_3 = y_1 + 1 \\
\wedge \quad & y_4 = x_0 \, ? \, y_2 : y_3 \;) \\
\Longrightarrow \quad & (y_4 = 7 \vee y_4 = 9)
\end{aligned}
$$

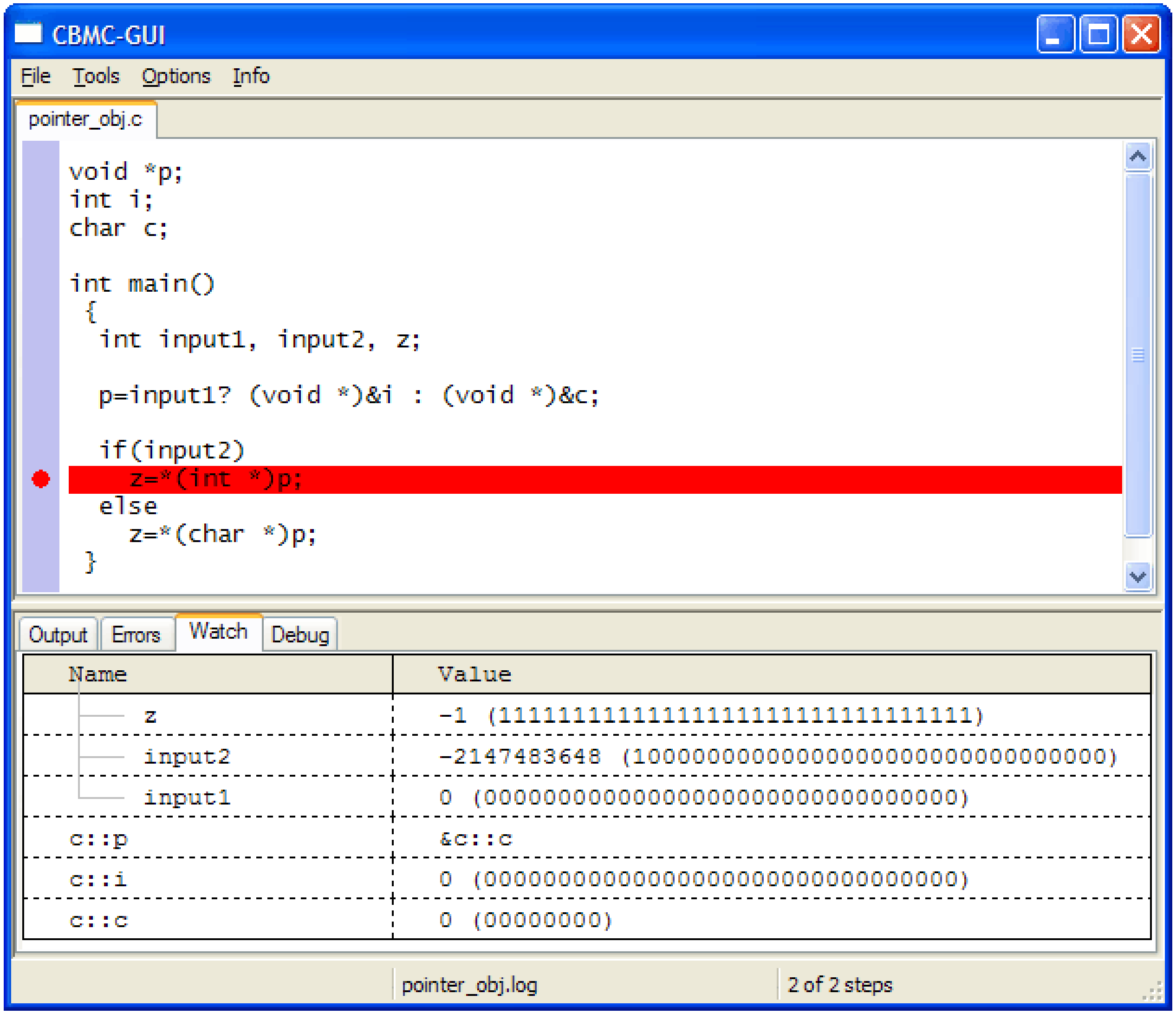
Supported Language Features

- **ANSI-C is a low level language, not meant for verification but for efficiency**
- **Complex language features, such as**
  - **Bit vector operators (shifting, and, or,…)**
  - **Pointers, pointer arithmetic**
  - **Dynamic memory allocation: malloc/free**
  - **Dynamic data types: `char s[n]`**
  - **Side effects**
  - `float` / `double`
  - **Non-determinism**
  - **Timing properties**

Pointers

- **While unwinding, record right hand side of assignments to pointers**
- **This results in very precise points-to information**
  - **Separate for each pointer**
  - **Separate for each <u>instance</u> of each program location**
- **Dereferencing operations are expanded into case-split on pointer object (not: offset)**
  - **Generate assertions on offset and on type**
- **Pointer data type assumed to be part of bit-vector logic**
  - **Consists of pair <object, offset>**

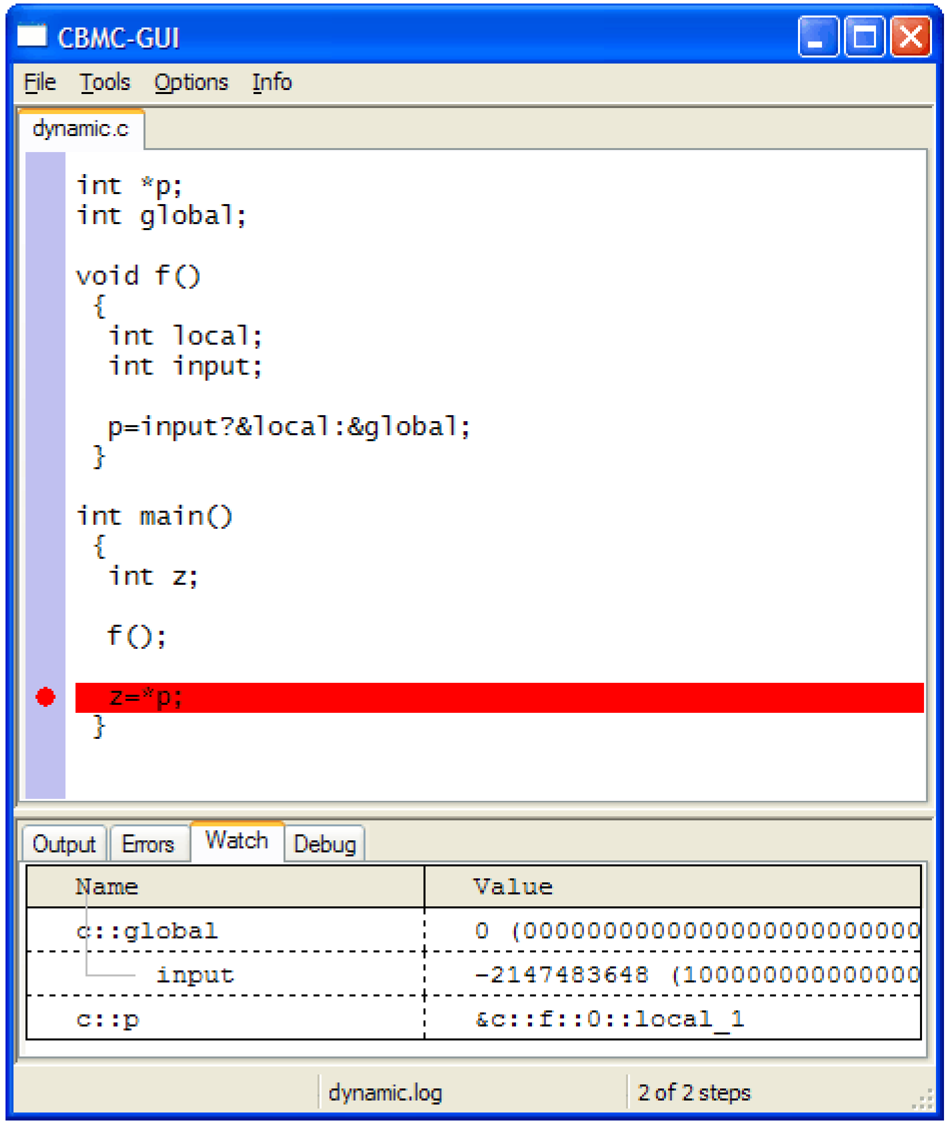
Pointer Typecast Example



Dynamic Objects

- **Dynamic Objects:**
  - `malloc` / `free`
  - Local variables of functions
- **Auxiliary variables for each dynamically allocated object:**
  - Size (number of elements)
  - Active bit
  - Type
- `malloc` sets size (from parameter) and sets active bit
- `free` asserts that active bit is set and clears bit
- Same for local variables: active bit is cleared upon leaving the function

Dynamic Objects



Deciding Bit-Vector Logic with SAT

- **Pro: all operators modeled with their precise semantics**
- **Arithmetic operators are flattened into circuits**
  - **Not efficient for multiplication, division**
  - **Fixed-point for `float`/`double`**
- **Unbounded arrays**
  - **Use uninterpreted functions to reduce to equality logic**
  - **Similar implementation in UCLID**
  - **But: <u>Contents</u> of array are interpreted**
- **Problem: SAT solver happy with first satisfying assignment that is found. <u>Might not look nice</u>.**

Example

```
void f (int a, int b, int c)
{
  int temp;
  if (a > b) {
    temp = a; a = b; b = temp;
  }
  if (b > c) {
    temp = b; b = c; c = temp;
  }
  if (a < b) {
    temp = a; a = b; b = temp;
  }
  assert (a<=b && b<=c);
}
```

CBMC

```
State 1-3
  a=-8193 (11111111111111111101111111111111)
  b=-402 (11111111111111111111111001101110)
  c=-2080380800 (10000011111111111110100010…)
  temp=0 (00000000000000000000000000000000)

State 4 file sort.c line 10
  temp=-402 (11111111111111111111111001101110)

State 5 file sort.c line 11
  b=-2080380800 (10000011111111111110100010…)

State 6 file sort.c line 12
  c=-402 (11111111111111111111111001101110)

Failed assertion: assertion file
sort.c line 19
```

Problem (I)

- Reason: SAT solver performs DPLL backtracking search
- Very first satisfying assignment that is found is reported
- Strange values artifact from bit-level encoding
- Hard to read
- Would like nicer values

Problem (II)

- **Might not get shortest counterexample!**
- **Not all statements that are in the formula actually get executed**
- **There is a variable for each statement that decides if it is executed or not (conjunction of `if`-guards)**
- **Counterexample trace only contains assignments that are actually executed**
- **The SAT solver picks some…**

Example

```
void f (int a, int b,
        int c)
{
  if(a)
  {
    a=0;
    b=1;
  }

  assert(c);
}
```

CBMC

```
{-1} b_1#2 == (a_1#0?b_1#1:b_1#0)
{-2} a_1#2 == (a_1#0?a_1#1:a_1#0)
{-3} b_1#1 == 1
{-4} a_1#1 == 0
{-5} \guard#1 == a_1#0
{-6} \guard#0 == TRUE
|-----------------------------
{1} c_1#0
```

from SSA

assign-ments

Example

```
void f (int a, int b,
        int c)
{
  if(a)
  {
    a=0;
    b=1;
  }

  assert(c);
}
```

CBMC

```
State 1-3
  a=1 (00000000000000000000000000000001)
  b=0 (00000000000000000000000000000000)
  c=0 (00000000000000000000000000000000)

State 4 file length.c line 5
  a=0 (00000000000000000000000000000000)

State 5 file length.c line 6
  b=1 (00000000000000000000000000000001)

Failed assertion: assertion
file length.c line 11
```

Basic Solution

- **Counterexample length typically considered to be most important**
  - **E.g., SPIN iteratively searches for shorter counterexamples**
- **Phase one: Minimize length**

$$min \sum_{g \in G} l_g \cdot l_w$$

- **l_g: Truth value (0/1) of guard,**
  l_w: Weight = number of assignments
- **Phase two: Minimize values**

Example

```
void f (int a, int b, int c)
{
  int temp;
  if (a > b) {
    temp = a; a = b; b = temp;
  }
  if (b > c) {
    temp = b; b = c; c = temp;
  }
  if (a < b) {
    temp = a; a = b; b = temp;
  }
  assert (a<=b && b<=c);
}
```

CBMC + Mini-mization

```
State 1-3
  a=0  (00000000000000000000000000000000)
  b=0  (00000000000000000000000000000000)
  c=-1  (11111111111111111111111111111111)
  temp=0  (00000000000000000000000000000000)

State 4 file sort.c line 10
  temp=0  (00000000000000000000000000000000)

State 5 file sort.c line 11
  b=-1  (11111111111111111111111111111111)

State 6 file sort.c line 12
  c=0  (00000000000000000000000000000000)

Failed assertion: assertion file
sort.c line 19
```

Experiment: Train Controller

- Actually runs on trains
- Part provided to us: braking profiles
- ANSI-C plus assumptions on arithmetic
- Size: about 30.000 lines
- Software computes all values twice (two channels) – the second time with inverted values or with offset
- Properties: WCET, Equivalence of channels, pointers/arrays

Current Status

- Added support for C++, IEEE Floating-Point
- Industrial users:
  - Automotive
  - Avionics
  - Embedded/medical
- Didn’t talk about: HW/SW co-verification

Future Work

- CBMC for concurrent programs
- Better decision procedures for complex bit-vector arithmetic
- Build counterexample quality measure (length, values) into SAT solver
  - Splitting heuristic

Questions?