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:

\[ \alpha(s) \equiv (p_1(s), p_2(s), \ldots, p_n(s)) \]
Predicate Abstraction

Concrete States:

\[
\begin{align*}
&\text{Concrete States:} \\
&\begin{array}{c}
\text{x=2} \\
\text{y=0}
\end{array}, \quad
\begin{array}{c}
\text{x=2} \\
\text{y=1}
\end{array}, \\
&\begin{array}{c}
\text{x=0} \\
\text{y=0}
\end{array}, \\
&\begin{array}{c}
\text{x=1} \\
\text{y=0}
\end{array}, \\
&\begin{array}{c}
\text{x=1} \\
\text{y=1}
\end{array}, \\
&\begin{array}{c}
\text{x=1} \\
\text{y=2}
\end{array},
\end{array}
\]

Predicates:

\[
\begin{align*}
p_1(s) &= (s.x > s.y) \\
p_2(s) &= (s.y = 0)
\end{align*}
\]

Abstract transitions?
Under- vs. Overapproximation

- How to abstract the transitions?
  - Depends on the property we want to show
  - Typically done in a **conservative** manner

- **Existential abstraction:**

\[ \hat{I}(\hat{s}) : \iff \exists s : I(s) \land \alpha(s) = \hat{s} \]

\[ \hat{R}(\hat{s}, \hat{s}') : \iff \exists s, s' : R(s, s') \land \alpha(s) = \hat{s} \land \alpha(s') = \hat{s}' \]

) **Preserves safety properties**
Predicate Abstraction

Abstract Transitions:

Property:

\[ p_1 \lor \neg p_2 \iff (s.x > s.y) \lor (s.y \neq 0) \]

Property holds. Ok.
Predicate Abstraction

Abstract Transitions:

Property:

\[ p_1 \iff (s.x > s.y) \]

This trace is spurious!
Predicate Abstraction

Abstract Transitions:

Property:

\[ p_1 \iff (s.x > s.y) \]

New Predicates:

\[ p_1(s) = (s.x > s.y) \]
\[ p_2(s) = (s.x = 2) \]
Let’s take existential abstraction seriously

Basic idea: with $n$ predicates, there are $2^n \leq 2^n$ 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**

\( i' = i + 1 \)

**Query**

\[ i \neq 1 \land i \neq 2 \land \overline{\text{even}(i)} \land \]
\[ i' = i + 1 \land \]
\[ i' \neq 1 \land i' \neq 2 \land \overline{\text{even}(i')} \]

**Current Abstract State**

\[
\begin{array}{ccc}
p_1 & p_2 & p_3 \\
0 & 0 & 0 \\
0 & 0 & 1 \\
0 & 1 & 0 \\
0 & 1 & 1 \\
1 & 0 & 0 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
1 & 1 & 1 \\
\end{array}
\]

**Next Abstract State**

\[
\begin{array}{ccc}
p'_1 & p'_2 & p'_3 \\
0 & 0 & 0 \\
0 & 0 & 1 \\
0 & 1 & 0 \\
0 & 1 & 1 \\
1 & 0 & 0 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
1 & 1 & 1 \\
\end{array}
\]
Existential Abstraction

Predicates

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

Basic Block

```
++;
```

Formula

$i' = i + 1$

Query

\[
i \neq 1 \land i \neq 2 \land \text{even}(i) \land \\
\quad i' = i + 1 \land \\
\quad i' \neq 1 \land i' \neq 2 \land \text{even}(i')
\]

... and so on ...

Current Abstract State

<table>
<thead>
<tr>
<th>$p_1$</th>
<th>$p_2$</th>
<th>$p_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Next Abstract State

<table>
<thead>
<tr>
<th>$p'_1$</th>
<th>$p'_2$</th>
<th>$p'_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Predicate Abstraction for Software

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

C program

```c
int main() {
    int i;
    i = 0;
    while (even(i))
        i++;
}
```

Predicates

```
p₁ ⇔ i = 0
p₂ ⇔ even(i)
```

Boolean program

```c
void main() {
    bool p₁, p₂;
    p₁ = TRUE;
    p₂ = TRUE;
    while (p₂)
        {
            p₁ = p₁ ? FALSE : *;
            p₂ = !p₂;
        }
}
```

[Example references:

-Ball, Rajamani '00
-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

Actual Program

Initial Abstraction

Concurrent Boolean Program

Verification

Model Checker

Verification

Property holds

No error or bug found

Counterexample

Simulation successful

Bug found

Simulation successful

Spurious counterexample

Abstraction refinement

[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

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

```plaintext
y = 8;
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!
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

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

TRANS (PC=0) \rightarrow next(b0_argc_ge_1)=b0_argc_ge_1
\& next(b1_argc_le_213646)=b1_argc_le_21646 & next(b2)=b2
\& (!b30 \mid b36)
\& (!b17 \mid !b30 \mid b42)
\& (!b30 \mid !b42 \mid b48)
\& (!b17 \mid !b30 \mid !b42 \mid b54)
\& (!b54 \mid b60)

TRANS (PC=1) \rightarrow 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
\ldots
SMV for the Boolean Program

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

Too hard for programmers, and therefore:

Locking Rule in SLIC

```plaintext
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

```
do {
    KeAcquireSpinLock();

    nPacketsOld = nPackets;

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

KeReleaseSpinLock();
```
Example

```c
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

```
Example

Add new predicate to boolean program (c2bp)

b : (nPacketsOld == nPackets)

```

```
do {
    KeAcquireSpinLock();

    nPacketsOld = nPackets;  
```

```
    if(request){
        request = request->Next;
        KeReleaseSpinLock();
        nPackets++;  
```

```
    } while (nPackets != nPacketsOld);  
```

```
} while (nPackets != nPacketsOld);
```

```
KeReleaseSpinLock();
```

U

L

L

L

L

L

L

L

L

L

L

L

L

L

L

L

U

U

U

U

U

U

E

Example

Add new predicate to boolean program (c2bp)

b : (nPacketsOld == nPackets)

```
do {
    KeAcquireSpinLock();

    nPacketsOld = nPackets;  
```

```
    if(request){
        request = request->Next;
        KeReleaseSpinLock();
        nPackets++;  
```

```
    } while (nPackets != nPacketsOld);  
```

```
} while (nPackets != nPacketsOld);
```

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

KeReleaseSpinLock();

b : (nPacketsOld == nPackets)
Example

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

KeReleaseSpinLock();
Abstraction Refinement Loop

- **Actual Program**
  - Initial Abstraction
  - Concurrent Boolean Program
  - Refinement
- **Model Checker**
  - Verification
  - Property holds
  - No error or bug found
- **Simulator**
  - Spurious counterexample
  - Simulation successful
  - Bug found
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

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

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

Predicate:

\( y > x \)
Example: Simulation

```c
int main() {
    int x, y;
    y = 1;
    x = 1;
    if (y > x) {
        y--; // Spurious trace!
    } else {
        y++;
    }
    assert(y > x);
}
```

```c
int main() {
    int x, y;
    y_1 = 1;
    x_1 = 1;
    if (y_1 > x_1) {
        y_2 = y_1 - 1;
    } else {
        y = y + 1;
    }
    assert(y_2 > x_1);
    assert(y_2 > x_0);
}
```

Spurious trace!
Abstraction Refinement Loop

- Actual Program
- Initial Abstraction
- Concurrent Boolean Program
- Verification
- Model Checker
- Refinement
- Simulator
- Abstraction refinement
- Spurious counterexample
- No error or bug found
- Property holds
- Simulation successful
- Bug found

Property holds
Simulation successful
Spurious counterexample
Abstraction refinement
Initial Abstraction
Manual Proof!

```c
int main() {
    int x, y;
    y=1;
    {y = 1}
    x=1;
    {x = 1 ∧ y = 1}
    if(y>x)
        y--;  
    else
        {x = 1 ∧ y = 1 ∧ ¬y > x}
        y++;
    {x = 1 ∧ y = 2 ∧ y > x}
    assert(y>x);
}
```

We are using strongest post-conditions here
Another Manual Proof

```c
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++;
}
```

We are using **weakest pre-conditions** here

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

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)
The abstraction by FASTABS is often too coarse

\[
p_1, p_2 := \text{schoose}[F, F], \text{schoose}[F, F];
\]

If no new predicates are found, the transitions in the abstract counterexample are checked

The spurious transition is eliminated by adding a constraint

\[
p_1, p_2 := \text{schoose}[F, F], \text{schoose}[F, F]
\]

\[
\text{constrain (p0 & 'p1) | (!p0 & !'p1)};
\]
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, unwinding assertions 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

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

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

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

- **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

```c
while.c

int table0[] = { 0xf324, 0 };
int table1[] = { 0xec26, 0x626e, 0 };
int table2[] = { 0 };
int *tables[] = { table0, table1, table2 };

int main()
{
    unsigned index;
    int *p;
    if(index>2) index=0;
    p=tables[index];
    while(*p!=0)
    {
        p++;
    }
}
```

Output:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>1 (0000000000000000000000000000000000000001)</td>
</tr>
<tr>
<td>c::table0</td>
<td>{ c::table0, c::table1, c::table2 }</td>
</tr>
<tr>
<td>c::table1</td>
<td>{ 60454, 25198, 0 }</td>
</tr>
<tr>
<td>c::table0</td>
<td>{ 62244, 0 }</td>
</tr>
<tr>
<td>p</td>
<td>c::table1+1</td>
</tr>
</tbody>
</table>
Implementation

3. Transformation into Equation

- After unwinding: Transform into SSA

\[
\begin{align*}
    x &= x + y; &\quad x_1 &= x_0 + y_0; \\
    x &= x \times 2; &\quad x_2 &= x_1 \times 2; \\
    a[i] &= 100; &\quad a_1[i_0] &= 100;
\end{align*}
\]

- 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

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

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

```c
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); 
}
```

\[
\begin{align*}
& ( y_1 = 8 \\
& \land y_2 = y_1 - 1 \\
& \land y_3 = y_1 + 1 \\
& \land y_4 = x_0 ? y_2 : y_3 ) \\
\implies & ( y_4 = 7 \lor y_4 = 9 )
\end{align*}
\]
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 instance 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

```c
void *p;
int i;
char c;

int main()
{
    int input1, input2, z;
    p=input1? (void *)&i : (void *)&c;
    if(input2)
        z=(int *)p;
    else
        z=(char *)p;
}
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>z</td>
<td>-1 (11111111111111111111111111111111)</td>
</tr>
<tr>
<td>input2</td>
<td>-2147483648 (10000000000000000000000000000000)</td>
</tr>
<tr>
<td>input1</td>
<td>0 (0000000000000000000000000000000000000000)</td>
</tr>
<tr>
<td>c:p</td>
<td></td>
</tr>
<tr>
<td>c:i</td>
<td>0 (0000000000000000000000000000000000000000)</td>
</tr>
<tr>
<td>c:c</td>
<td>0 (0000000000000000000000000000000000000000)</td>
</tr>
</tbody>
</table>
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

```
dynamic.c
int *p;
int global;

void f()
{
    int local;
    int input;
    p=input?&local:&global;
}

int main()
{
    int z;
    f();
    z=*p;
}
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>c::global</td>
<td>0 (0000000000000000000000000000000000000000)</td>
</tr>
<tr>
<td>input</td>
<td>-2147483648 (10000000000000000000000000000000)</td>
</tr>
<tr>
<td>c::p</td>
<td>&amp;c::f::0::local_1</td>
</tr>
</tbody>
</table>
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: **Contents** of array are interpreted

- **Problem:** SAT solver happy with first satisfying assignment that is found. ** Might not look nice.**
Example

```c
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);
}
```

State 1-3
- `a` = -8193 (111111111111111111011111111111111)
- `b` = -402 (11111111111111111111111001101110)
- `c` = -2080380800 (10000011111111111110100010…)
- `temp` = 0 (00000000000000000000000000000000)

CBMC

State 4 file sort.c line 10
- `temp` = -402 (11111111111111111111111101101110)

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

State 6 file sort.c line 12
- `c` = -402 (11111111111111111111111101101110)

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
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-guard)
Counterexample trace only contains assignments that are actually executed
The SAT solver picks some...
void f (int a, int b, 
   int c)
{
    if(a)
    {
      a=0;
      b=1;
    }
    assert(c);
}
Example

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

    assert(c);
}

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

```c
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);
}
```

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?