Software Reliability

Lecture 4

Bounded Model Checking for Software

Alastair Donaldson

www.doc.ic.ac.uk/~afd

Hardware verification using SAT

A circuit can be described as a **transition system** in Boolean logic

Transition system can be **unwound** to consider all states of hardware reachable within *k* transitions

SAT solver can be used to check whether the unwound transition system can reach **bad** states

Known as **bounded model checking**

Look for bugs up to **bound** *k*

Model checking term used mainly for historical reasons

The most widely used technique for hardware verification

Why is BMC successful?

Almost **completely** automatic

Designer needs to express correctness property, that's all

Scales to fairly large designs, due to amazing advances in SAT solving

No abstraction => **no false positives** – great for finding bugs

In some cases can allow full verification

Transition systems

$\vec{\mathbf{x}}$ – vector of Boolean state variables

$\mathbf{I}(\vec{\mathbf{x}})$ – predicate describing initial state of system

$\mathbf{T}(\vec{\mathbf{x}}, \vec{\mathbf{x}}')$ – relation describing transitions between states

$\mathbf{P}(\vec{\mathbf{x}})$ – predicate describing correctness property

Simple example

Boolean state variables: **a**, **b**, **c**, **d**

Initial states:

I(**a**, **b**, **c**, **d**) = !(**b** || **c** || **d**)

What are the initial states?

Transition relation:

T((**a**, **b**, **c**, **d**), (**a'**, **b'**, **c'**, **d'**)) =

a' && (**b'** <=> **a**) && (**c'** <=> **b**) && (**d'** <=> **c**)

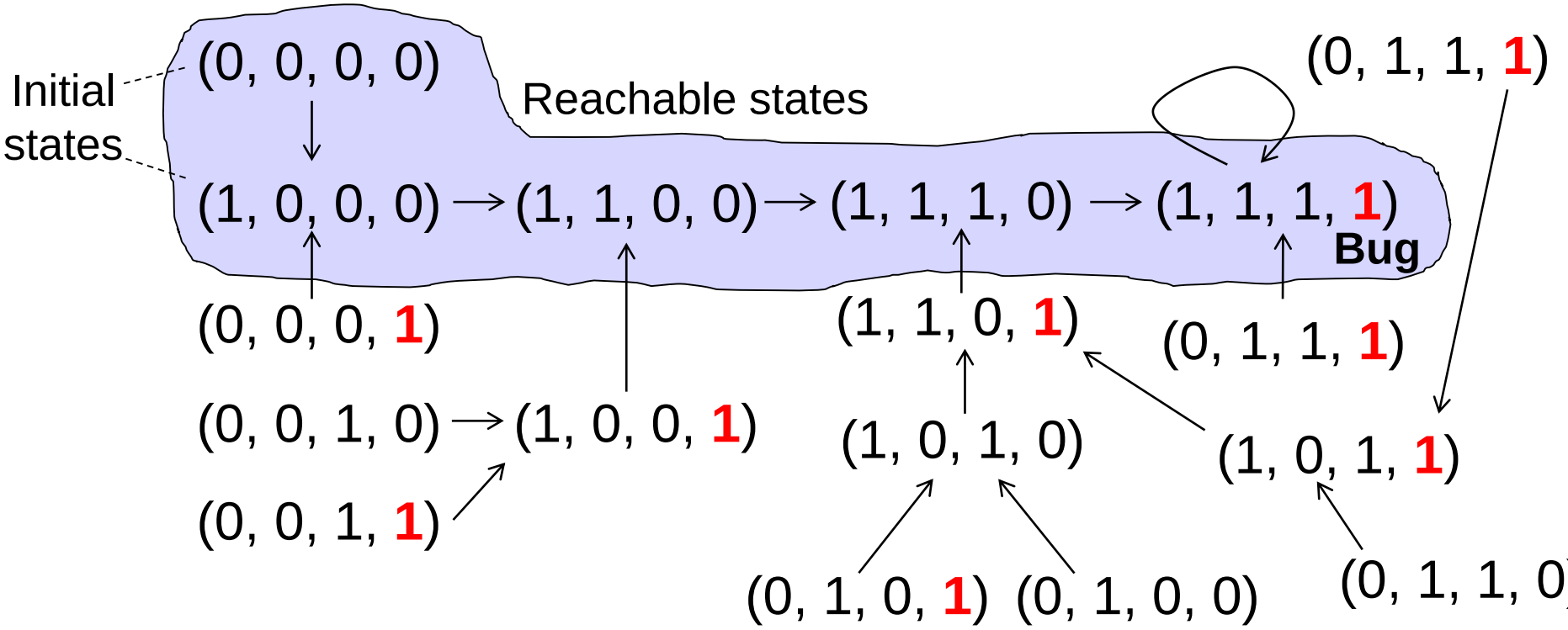
Correctness property:

P(**a**, **b**, **c**, **d**) = !**d**

Simple example

Is it correct? Let's use 1 and 0 for *true* and *false*

Explore state space:



BMC for transition systems

Given transition system:

$\mathbf{I}(\vec{\mathbf{x}})$ – initial states

$\mathbf{T}(\vec{\mathbf{x}}, \vec{\mathbf{x}}')$ – transition relation

$\mathbf{P}(\vec{\mathbf{x}})$ – correctness property

check formula: $\mathbf{I}(\vec{\mathbf{x}}_0)$ && $\mathbf{T}(\vec{\mathbf{x}}_0, \vec{\mathbf{x}}_1)$ && … && $\mathbf{T}(\vec{\mathbf{x}}_{k-1}, \vec{\mathbf{x}}_k)$

&& !($\mathbf{P}(\vec{\mathbf{x}}_0)$ && … && $\mathbf{P}(\vec{\mathbf{x}}_k)$)

Formula requests a sequence of states:

- starting in the initial state
- connected by transitions
- such that at least one state is **bad**

Formula SAT: **P does not hold**

UNSAT: **P** holds along all paths of length <= **k**

BMC for simple example

I(**a**, **b**, **c**, **d**) = !(**b** || **c** || **d**)

T((**a**, **b**, **c**, **d**), (**a'**, **b'**, **c'**, **d'**)) =

a' && (**b'** <=> **a**) && (**c'** <=> **b**) && (**d'** <=> **c**)

P(**a**, **b**, **c**, **d**) = !**d**

BMC with $k = 2$

$$!(\mathbf{b}_0 \,||\, \mathbf{c}_0 \,||\, \mathbf{d}_0) \;\&\&\; (\mathbf{a}_1 \;\&\&\; (\mathbf{b}_1 <=> \mathbf{a}_0) \;\&\&\; (\mathbf{c}_1 <=> \mathbf{b}_0) \;\&\&\; (\mathbf{d}_1 <=> \mathbf{c}_0))$$
$$\&\&\; (\mathbf{a}_2 \;\&\&\; (\mathbf{b}_2 <=> \mathbf{a}_1) \;\&\&\; (\mathbf{c}_2 <=> \mathbf{b}_1) \;\&\&\; (\mathbf{d}_2 <=> \mathbf{c}_1))$$
$$\&\&\; !(!\mathbf{d}_0 \;\&\&\; !\mathbf{d}_1 \;\&\&\; !\mathbf{d}_2)$$

UNSAT: P holds up to depth 2

Initial states: (0, 0, 0, 0) → (1, 0, 0, 0) → (1, 1, 0, 0) → (1, 1, 1, 0)

BMC for simple example

I(**a**, **b**, **c**, **d**) = !(**b** || **c** || **d**)

T((**a**, **b**, **c**, **d**), (**a'**, **b'**, **c'**, **d'**)) =

a' && (**b'** <=> **a**) && (**c'** <=> **b**) && (**d'** <=> **c**)

P(**a**, **b**, **c**, **d**) = !**d**

BMC with $k = 3$

$$!(\mathbf{b}_0 \,||\, \mathbf{c}_0 \,||\, \mathbf{d}_0) \;\&\&\; (\mathbf{a}_1 \;\&\&\; (\mathbf{b}_1 <=> \mathbf{a}_0) \;\&\&\; (\mathbf{c}_1 <=> \mathbf{b}_0) \;\&\&\; (\mathbf{d}_1 <=> \mathbf{c}_0))$$
$$\&\&\; (\mathbf{a}_2 \;\&\&\; (\mathbf{b}_2 <=> \mathbf{a}_1) \;\&\&\; (\mathbf{c}_2 <=> \mathbf{b}_1) \;\&\&\; (\mathbf{d}_2 <=> \mathbf{c}_1))$$
$$\&\&\; (\mathbf{a}_3 \;\&\&\; (\mathbf{b}_3 <=> \mathbf{a}_2) \;\&\&\; (\mathbf{c}_3 <=> \mathbf{b}_2) \;\&\&\; (\mathbf{d}_3 <=> \mathbf{c}_2))$$
$$\&\&\; !(!\mathbf{d}_0 \;\&\&\; !\mathbf{d}_1 \;\&\&\; !\mathbf{d}_2 \;\&\&\; !\mathbf{d}_3\ )$$

SAT: P **violated** at depth 3

Initial states: (0, 0, 0, 0) ↓ (1, 0, 0, 0) → (1, 1, 0, 0) → (1, 1, 1, 0) → (1, 1, 1, **1**)

Bug

BMC for software

- System-level software: bit-level accuracy is often important (if software does bit-manipulation)
- Idea: represent operations using their circuits
- Either:
  - Apply BMC for transition systems

  or
  - Directly unwind loops of program
- We shall study the latter approach
- BMC for software is implemented by the CBMC tool (a bounded model checker for C programs) – cprover.org

Loop-based unwinding: program level

```
while(c) {
    B;
}
```

```
if(c) {
  B;
  if(c) {
    assume(false);
  }
}
```

Unwind x 1

```
if(c) {
  B;
  if(c) {
    B;
    if(c) {
      assume(false);
    }
  }
}
```

Unwind x 2

k

```
if(c) {
  B;
  if(c) {
    B;
    …
      if(c) {
        B;
        if(c) {
          assume(false);
        }
      }
    …
  }
}
```

Unwind x **k**

Resulting loop-free programs can be analysed using:
predication + SSA renaming + formula generation

Loop-based unwinding and correctness:

P

```
while(c) {
    B;
}
```

P'

k

```
if(c) {
  B;
  if(c) {
    B;
    …
      if(c) {
        B;
        if(c) {
          assume(false);
        }
      }
    …
  }
}
```

Unwind x **k**

P' incorrect => P incorrect

P' correct does not imply P correct

P' correct tells us that **P** cannot go wrong within **k** loop iterations

- **there may be deeper bugs**

Loop-based unwinding: **under-approximation**

Exercises:

Write a simple program that is correct for a small loop unwinding depth but incorrect for a larger loop unwinding depth

We can unwind with k = 0. What does this look like?
Can this ever be useful?

Unwinding depth can be chosen loop-by-loop

```
while(c) {
    B;
}

while(d) {
    E;
}
```

```
if(c) {
  B;
  if(c) {
    assume(false);
  }
}

if(d) {
  E;
  if(d) {
    E;
    if(d) {
      assume(false);
    }
  }
}
```

Unwind x 1

Unwind x 2

Unwinding with nested loops

```
while(c) {
  while(d) {
    E;
  }
}
```

```
if(c) {
  while(d) {
    E;
  }
  if(c) {
    while(d) {
      E;
    }
    if(c) {
      assume(false);
    }
  }
}
```

Unwind x 2

Inner loop gets **duplicated**

Unwinding with nested loops

After unwinding an outer loop, we can unwind duplicated inner loops using **different depths** if we wish

```
while(c) {
  while(d)
    E;
  }
}
```

```
if(c) {
  while(d) {
    E;
  }
  if(c) {
    while(d) {
      E;
    }
    if(c) {
      assume(false);
    }
  }
}
```

Unwind x 2

```
if(c) {
  if(d) {
    E;
    if(d) {
      assume(false);
    }
  }
  if(c) {
    if(d) {
      E;
      if(d) {
        E;
        if(d) {
          assume(false);
        }
      }
    }
    if(c) {
      assume(false);
    }
  }
}
```

Unwind x 1

Unwind x 2

Unwinding with nested loops: complexity

Suppose **S** is a loop-free fragment of code

If we unwind **all** loops **k** times, how many copies of **S** will there be?

```
while(c1) {
  S;
}
while(c2) {
  S;
}
…
while(cd) {
  S;
}
```

$k \times d$

```
while(c1) {
  while(c2) {
    …
    while(cd) {
      S;
    }
    …
  }
}
```

k^d

Sound BMC: unwinding assertions

In some domains (especially embedded systems) all loops have fixed, relatively small bounds

Makes it possible to **completely** unwind loops to achieve **sound** verification

…but if bounds are not **explicit**, how do we know what it means to **completely** unwind?

Explicit bound

```
while(i < 10) {
  // Statements that do
  // not modify i
  i = i + 1;
}
```

Implicit bound

```
while(!finished) {
  // Statements that guarantee
  // setting finished to true
  // within 10 iterations
}
```

Unwinding assertions

```
while(c) {
    B;
}
```

```
if(c) {
  B;
  if(c) {
    assert(false);
    assume(false);
  }
}
```

Unwind x 1

```
if(c) {
  B;
  if(c) {
    B;
    if(c) {
      assert(false);
      assume(false);
    }
  }
}
```

Unwind x 2

Insert check to determine whether we have unwound completely

```
if(c) {
  B;
  if(c) {
    B;
    …
      if(c) {
        B;
        if(c) {
          assert(false);
          assume(false);
        }
      }
    …
  }
}
```

k

Unwind x **k**

Unwinding assertion

Correctness with unwinding assertion

P

```
while(c) {
    B;
}
```

P'

```
if(c) {
  B;
  if(c) {
    B;
    …
      if(c) {
        B;
        if(c) {
          assert(false);
          assume(false);
        }
      }
    …
  }
}
```

k

Unwind x **k** + unwinding assertion

Do we need both of these?

P' correct => P correct

P' incorrect does not imply P incorrect

P' incorrect tells us either

- an assertion of **P** can fail (**P** is **incorrect**) **or**
- the unwinding assertion can fail (**unwinding depth was insufficient**)

Note: we can tell which assertion failed; **may** deduce that **P** is incorrect

Is this an **under-** or **over-**approximation?

A formal definition for over- and under-approximation

Let **P** and **P'** be programs.

P over-approximates P'
(equivalently, **P' under-approximates P**)
if for every pre-condition **A** and post-condition **B** we have:

{ **A** } **P** { **B** } is valid

=>

{ **A** } **P'** { **B** } is valid

A way to think of this:

"P' can fail in the same or **fewer** ways than **P"**

Note: **P over- and under-approximates P**

We can have distinct programs **P** and **P'** such that **P** under-approximates **P'** and vice-versa. What does this mean?

Over-approximating by adding assertions

Let **P** be a program and **S** a statement appearing in **P**

Let **P'** be identical to **P**, except that **S** is replaced by:

assert(**e**);
S

Then **P'** is an over-approximation of **P** according to our definition

Justification: **P** can fail in fewer ways than **P'** because the addition of assert(**e**) gives **P'** an extra opportunity for failure

Over-approximating using assert(false)

Let **P** be a program and **S** a statement appearing in **P**

Let **P'** be identical to **P**, except that **S** is replaced by:

assert(**false**);
T

where **T** is any statement

Then **P'** is an over-approximation of **P** according to our definition

Justification: any input to **P** that causes **S** to be executed will cause **P'** to fail. On an input that does not cause **S** to be executed in **P** the programs behave identically

Unwinding assertion gives over-approximation

```
while(c) {
    B;
}
```

Unpeel first **k** loop iterations gives **precisely equivalent** program

k

```
if(c) {
  B;
  if(c) {
    B;
    …
      if(c) {
        B;
        if(c) {
          B;
          while(c) {
            B;
          }
        }
      }
    …
  }
}
```

over-approximate

k

```
if(c) {
  B;
  if(c) {
    B;
    …
      if(c) {
        B;
        if(c) {
          assert(false);
          assume(false);
        }
      }
    …
  }
}
```

Summary

Bounded model checking can find bugs in programs

Program is **under-approximated** by unwinding loops up to some user-specified depth

Unwinding assertions can be added to enable verification of programs for which all loops have relatively small upper bounds