Decision Procedures
chapters 12.1-12.3

Daniel Kroening and Ofer Strichman
With help of Nikolaj Bjørner and Leonardo de Moura
Program

• Introduction to software verification

• Bounded Program Analysis

• Unbounded Program Analysis
Introduction to software verification

- Example of algorithm:
  **Input**: array of integer
  **Output**: sum of the inverse of each element of the array

Can we find a bug in this program?
Introduction to software verification

Yes.

If one of the numbers is equal to 0.

```python
In [2]: a = [1, 0, 3]

In [3]: sum_inverse(a)
ZeroDivisionError
Traceback (most recent call last)
<ipython-input-3-d0c08a478f9d> in <module>
----> 1 sum_inverse(a)

<ipython-input-1-7b2b17cba832> in sum_inverse(arr)
    2   res = 0
    3   for nb in arr:
----> 4       res+=1/nb
    5   return res

ZeroDivisionError: division by zero
```
Introduction to software verification

How to detect errors/bugs?

• **Method 1 - Testing:** The traditional method to detect bugs:
  • Execute the program with a set of inputs and verify the outputs.
  • Can never guarantee the absence of errors.
  • Testing can only declare a program as incorrect if a test fails.
Introduction to software verification

How to detect errors/bugs?

- **Method 2 - Formal verification:** check if a given specification is satisfied for **all possible inputs**.
- Example: A given code will never divide by 0.
- Is it possible?
The Reachability Problem

• “The problem of checking whether a given state occurs in any execution of the program.”

• Like the halting problem, this problem is undecidable – no algorithm can give always the correct answer in a finite amount of time.
  
  • because of **unbounded allocation of memory.** (allocation in loops)
  
  • That mean a program can have an unbounded amount of states.
Side note about finite memory space

• The memory of a computer is finite therefore a theoretical solution to the reachability problem is possible.

• It does not help for practical solutions unless the memory bound is very small → small amount of possible program states.

• It also restricts the proof to a specific memory bound

• Also there is a problem with unbounded recursion (even without memory allocation)

• Therefore, we will treat the problem as undecidable.
Partial Solutions

- Solutions only for certain programs. (testing is an example)

- **SMT** (Satisfiability modulo theories) - generalizes SAT to more complex formulas involving real numbers, integers, and various data structures.

- Modern tools use solvers for Satisfiability Modulo Theories (SMT)
  These solvers find a simultaneous assignment to variables that satisfies a given logical formula.

- Logical solutions seems static and programs seems to be dynamic.
  - We will try to bridge this gap, By representing the program as a logical formula.
Bounded Program Analysis
Bounded Program Analysis

Some definitions:

- **Execution path** is a sequence of program instructions executed during a run of a program.
- **Execution trace** is a sequence of states that are observed along an execution path (different inputs have different execution traces).
- An **assertion** is a program instruction that takes a condition as argument, and if the condition evaluates to false, it reports an error and aborts.
Bounded Program Analysis

Program 12.2.1 Reading blocks from an array

```c
void ReadBlocks(int data[], int cookie)
{
    int i = 0;
    while (true)
    {
        int next;
        next = data[i];
        if (!((i < next && next < N)) return;
        i = i + 1;
        for (; i < next; i = i + 1) {
            if (data[i] == cookie)
                i = i + 1;
            else
                Process(data[i]);
        }
    }
}
```

- The first element of each block contains the index of the next block
- Skipping data that is equal to the input parameter cookie
Bounded Program Analysis

Program 12.2.1 Reading blocks from an array

```c
void ReadBlocks(int data[], int cookie)
{
    int i = 0;
    while (true)
    {
        int next;
        next = data[i];
        if (!(i < next && next < N)) return;
        i = i + 1;
        for (; i < next; i = i + 1) {
            if (data[i] == cookie)
                i = i + 1;
            else
                Process(data[i]);
        }
    }
}
```

- We assume that verification tools has some heuristic to choose execution path
- Suppose, that it choose the following path:
  - begin at line 3
  - Run for loop once, take the else branch and exit the for loop
  - exit the while loop during the second iteration in line 8
We assume that verification tools has some heuristic to choose execution path.

Suppose, that it choose the following path:

- begin at line 3
- Run for loop once, take the else branch and exit the for loop
- exit the while loop during the second iteration in line 8

Table 12.1. Sequence of statements along a path of Program 12.2.1

<table>
<thead>
<tr>
<th>Line</th>
<th>Kind</th>
<th>Instruction or condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Assignment</td>
<td>( i = 0; )</td>
</tr>
<tr>
<td>7</td>
<td>Assignment</td>
<td>( \text{next} = \text{data}[i]; )</td>
</tr>
<tr>
<td>8</td>
<td>Branch</td>
<td>( i &lt; \text{next} &amp;&amp; \text{next} &lt; N )</td>
</tr>
<tr>
<td>9</td>
<td>Assignment</td>
<td>( i = i + 1; )</td>
</tr>
<tr>
<td>10</td>
<td>Branch</td>
<td>( i &lt; \text{next} )</td>
</tr>
<tr>
<td>11</td>
<td>Branch</td>
<td>( \text{data}[i] \neq \text{cookie} )</td>
</tr>
<tr>
<td>14</td>
<td>Function call</td>
<td>( \text{Process} \left( \text{data}[i] \right) )</td>
</tr>
<tr>
<td>10</td>
<td>Assignment</td>
<td>( i = i + 1; )</td>
</tr>
<tr>
<td>10</td>
<td>Branch</td>
<td>( !(i &lt; \text{next}) )</td>
</tr>
<tr>
<td>7</td>
<td>Assignment</td>
<td>( \text{next} = \text{data}[i]; )</td>
</tr>
<tr>
<td>8</td>
<td>Branch</td>
<td>( !(i &lt; \text{next} &amp;&amp; \text{next} &lt; N) )</td>
</tr>
</tbody>
</table>
Bounded Program Analysis

Rewrite instructions and conditions into static single assignment representation (SSA),
A “time-stamped version” of the program variables (every time a variable is written, a new symbol is introduce for it.

<table>
<thead>
<tr>
<th>Line</th>
<th>Kind</th>
<th>Instruction or condition</th>
<th>Instruction or condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Assignment</td>
<td>( i = 0; )</td>
<td>( i_1 = 0; )</td>
</tr>
<tr>
<td>7</td>
<td>Assignment</td>
<td>( \text{next} = \text{data}[i]; )</td>
<td>( \text{next}_1 = \text{data}_0[i_1]; )</td>
</tr>
<tr>
<td>8</td>
<td>Branch</td>
<td>( i &lt; \text{next} &amp;&amp; \text{next} &lt; N )</td>
<td>( i_1 &lt; \text{next}_1 &amp;&amp; \text{next}_1 &lt; N_0 )</td>
</tr>
<tr>
<td>9</td>
<td>Assignment</td>
<td>( i = i + 1; )</td>
<td>( i_2 = i_1 + 1; )</td>
</tr>
<tr>
<td>10</td>
<td>Branch</td>
<td>( i &lt; \text{next} )</td>
<td>( i_2 &lt; \text{next}_1 )</td>
</tr>
<tr>
<td>11</td>
<td>Branch</td>
<td>( \text{data}[i] != \text{cookie} )</td>
<td>( \text{data}_0[i_2] != \text{cookie}_0 )</td>
</tr>
<tr>
<td>14</td>
<td>Function call</td>
<td>( \text{Process(data}[i]); )</td>
<td>( \text{Process(data}_0[i_2]); )</td>
</tr>
<tr>
<td>10</td>
<td>Assignment</td>
<td>( i = i + 1; )</td>
<td>( i_3 = i_2 + 1; )</td>
</tr>
<tr>
<td>10</td>
<td>Branch</td>
<td>( !(i &lt; \text{next}) )</td>
<td>( !(i_3 &lt; \text{next}_1) )</td>
</tr>
<tr>
<td>7</td>
<td>Assignment</td>
<td>( \text{next} = \text{data}[i]; )</td>
<td>( \text{next}_2 = \text{data}_0[i_3]; )</td>
</tr>
<tr>
<td>8</td>
<td>Branch</td>
<td>( !(i &lt; \text{next} &amp;&amp; \text{next} &lt; N) )</td>
<td>( !(i_3 &lt; \text{next}_2 &amp;&amp; \text{next}_2 &lt; N_0) )</td>
</tr>
</tbody>
</table>

Table 12.1. Sequence of statements along a path of Program 12.2.1

SSA form
Bounded Program Analysis

- Now, we can translate the SSA into **logical formula** called the **path constraint**.

```
i_1 = 0;
next_1 = data_0[i_1];
i_1 < next_1 && next_1 < N_0
i_2 = i_1 + 1;
i_2 < next_1
data_0[i_2] != cookie_0
Process(data_0[i_2]);
i_3 = i_2 + 1;
!(i_3 < next_1)
next_2 = data_0[i_3];
!(i_3 < next_2 && next_2 < N_0)
```

```
i_1 = 0
next_1 = data_0[i_1]
(i_1 < next_1 ∧ next_1 < N_0)
i_2 = i_1 + 1
(i_2 < next_1)
data_0[i_2] ≠ cookie_0
(i_3 = i_2 + 1)
¬(i_3 < next_1)
next_2 = data_0[i_3]
¬(i_3 < next_2 ∧ next_2 < N_0).
```

- Note that the equality symbol in the formula denotes mathematical equality, whereas it denotes an assignment.
Assertion Checking

- Assume we have path that lead to an assertion.
- How we can use path constraint to check if an assertion can be violated?

We add to the path constraint the negation of the assertion! If the new path constraint is satisfiable then the assertion can be violated!
Program 12.2.1 Reading blocks from an array

```
void ReadBlocks(int data[], int cookie)
{
  int i = 0;
  while (true)
  {
    int next;
    next = data[i];
    if (!((i < next && next < N)) return;
    i = i + 1;
    for (; i < next; i = i + 1) {
      if (data[i] == cookie)
        i = i + 1;
      else
        Process(data[i]);
    }
  }
}
```

- Consider the path that executes the assignment in Line 3 and then lead to an assertion that checks if the variable \(i\) is within the required range for the array access in Line 7.

  • The path constraint is: \(i_1 = 0\)

  The assertion is: \(0 \leq i_1 \land i_1 < N_0\)

- Then the new path constraint is:

  \(i_1 = 0 \land \neg(0 \leq i_1 \land i_1 < N_0)\)
Assertion Checking: Example

**Program 12.2.1** Reading blocks from an array

```c
void ReadBlocks(int data[], int cookie) {
    int i = 0;
    while (true) {
        int next;
        next = data[i];
        if (!(i < next && next < N)) return;
        i = i + 1;
        for (; i < next; i = i + 1) {
            if (data[i] == cookie)
                i = i + 1;
            else
                Process(data[i]);
        }
    }
}
```

\[ i_1 = 0 \land \neg(0 \leq i_1 \land i_1 < N_0) \]

- This path constraint is **satisfiable**?

\[ \{i_1 \mapsto 0, \ N_0 \mapsto 0\} \]
Bounded Program Analysis

Program 12.2.1 Reading blocks from an array

```c
void ReadBlocks(int data[], int cookie)
{
    int i = 0;
    while (true)
    {
        int next;
        next = data[i];
        if (!(i < next && next < N)) return;
        i = i + 1;
        for (; i < next; i = i + 1) {
            if (data[i] == cookie)
                i = i + 1;
            else
                Process(data[i]);
        }
    }
}
```

- There are at least two arrays out of bounds errors!
  Which ones?
- The simple one: empty data
  - data = []
Assertion Checking: An Other Example

Program 12.2.1 Reading blocks from an array

```c
void ReadBlocks(int data[], int cookie) {
    int i = 0;
    while (true) {
        int next;
        next = data[i];
        if (!(i < next && next < N)) return;
        i = i + 1;
        for (; i < next; i = i + 1) {
            if (data[i] == cookie) i = i + 1;
            else Process(data[i]);
        }
    }
}
```

<table>
<thead>
<tr>
<th>Line</th>
<th>Kind</th>
<th>Instruction or condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Assignment</td>
<td>i = 0;</td>
</tr>
<tr>
<td>7</td>
<td>Assignment</td>
<td>next = data[i];</td>
</tr>
<tr>
<td>8</td>
<td>Branch</td>
<td>i &lt; next &amp;&amp; next &lt; N</td>
</tr>
<tr>
<td>9</td>
<td>Assignment</td>
<td>i = i + 1;</td>
</tr>
<tr>
<td>10</td>
<td>Branch</td>
<td>i &lt; next</td>
</tr>
<tr>
<td>11</td>
<td>Branch</td>
<td>data[i] = cookie</td>
</tr>
<tr>
<td>12</td>
<td>Assignment</td>
<td>i = i + 1;</td>
</tr>
<tr>
<td>10</td>
<td>Assignment</td>
<td>i = i + 1;</td>
</tr>
<tr>
<td>10</td>
<td>Branch</td>
<td>!(i &lt; next)</td>
</tr>
<tr>
<td>7</td>
<td>Assertion</td>
<td>0 &lt;= i &amp;&amp; i &lt; N</td>
</tr>
</tbody>
</table>

Table 12.3. A second path in Program 12.2.1
Assertion Checking: An Other Example

\[ i_1 = 0 \quad \land \]
\[ \text{next}_1 = data_0[i_1] \quad \land \]
\[ (i_1 < \text{next}_1 \land \text{next}_1 < N_0) \quad \land \]
\[ i_2 = i_1 + 1 \quad \land \]
\[ i_2 < \text{next}_1 \quad \land \]
\[ data_0[i_2] = \text{cookie}_0 \quad \land \]
\[ i_3 = i_2 + 1 \quad \land \]
\[ i_4 = i_3 + 1 \quad \land \]
\[ \neg (i_4 < \text{next}_1) \quad \land \]
\[ \neg (0 \leq i_4 \land i_4 < N_0) \]

- This constraint path is **satisfiable**.

\{ i_1 \mapsto 0, \ N_0 \mapsto 3, \ \text{next}_1 \mapsto 2, \ data_0 \mapsto \langle 2, 6, 5 \rangle, \ i_2 \mapsto 1, \ \text{cookie}_0 \mapsto 6, \ i_3 \mapsto 2, \ i_4 \mapsto 3 \} \ .
Assertion Checking: Summary

• We reduced the problem of verifying the correctness of a path in a program to a problem of checking the satisfiability of a formula

• This formula is called the verification condition (VC).
Checking Feasibility of all paths in Bounded program

- Exponential number of path with the number of branches.

- Then we need to check exponential number of path: $\rightarrow$ infeasible.
Transform SSA for encoding all possible paths (with loop limits)

- **Loop will be unfolded k times.**
  - We will write k times the loop. And change `for/while` to an `if` statement

- **Condition of each if statement will be assign by a new variable:** $\gamma$
  It allows to have different branches in the same SSA

- **When the control flow reconverges,** A statement is added that assigns the correct value to all those variables that have been modified in any of the branches.
Program 12.2.2 An unfolding of the for loop in Program 12.2.1 and its SSA form with φ-instructions

if (i < next) {
  if (data[i] == cookie) {
    i = i + 1;
  } else {
    Process(data[i]);
  }
  i = i + 1;
}

γ1 = (i0 < next0);
γ2 = (data0[i1] == cookie0);
i1 = i0 + 1;
i2 = γ2 ? i1 : i0;
i3 = i2 + 1;
γ3 = (i3 < next0);
γ4 = (data0[i3] == cookie0);
i4 = i3 + 1;
i5 = γ4 ? i4 : i3;
i6 = i5 + 1;
i7 = γ3 ? i6 : i3;
i8 = γ1 ? i7 : i0;

UNFOLDING
k=2

flow reconverges
Bounded Program Analysis: Summary

- Given the SSA form of the (unfolded) program, we can construct a formula that captures exactly all the possible traces that it can execute.
- To check an assertion in the program, we need to add its negation to the formula.
- The resulting formula is then finally given to a suitable decision procedure. (SMT solver)
Unbounded Program Analysis
We have seen a technique that **under-approximates** the behavior of the program by limiting the depth of loops.

We now show a transformation that gets rid of loops but **over-approximates** the original behavior.

When Successful, it proves for every possible input. But it might find errors in correct programs.
Nondeterministic Assignments

We do so in three steps:

1. For each loop and each program variable that is modified by the loop, add an assignment at the beginning of the loop that assigns a nondeterministic value to the variable.

2. After each loop, add an assumption that the negation of the loop condition holds.
   
   (assume(C) - aborts any path that does not satisfy C)

3. Replace each while loop with an if statement.
Example

1. nondeterministic values to modified variables
2. assume negation of condition
3. replace “while” with “if”

```
int i=0, j=0;
while(data[i] != '\n') {
    i++;
    j=i;
}
assert(i == j);
```

```
int i=0, j=0;
if(data[i] != '\n') {
    i=\n;
    j=\n;
    i++;
    j=i;
}
assume(data[i] == '\n');
assert(i == j);
```
Translate to Logical Formula

This logical formula is not satisfiable

- The Assertion is true for every number of iterations!

- Ignores information from previous loop iterations

```c
int i=0, j=0;
if(data[i] != '\n') {
    i=*
    j=*
    i++
    j=i;
}
assume(data[i] == '\n');
assert(i == j);
i_1 = 0
j_1 = 0
\gamma_1 = (data_0[i_1] \neq 'n')
i_3 = i_2 + 1
j_3 = i_3
i_4 = \gamma_1 ? i_3 : i_1
j_4 = \gamma_1 ? j_3 : j_1
data_0[i_4] = 'n'
i_4 \neq j_4
```
Too coarse overapproximation example

Program 12.3.2 Processing requests using locks

```
1 do {
2     lock();
3     old_count = count;
4     request = GetNextRequest();
5     if (request ! = NULL) {
6         ReleaseRequest(request);
7         unlock();
8         ProcessRequest(request);
9         count = count + 1;
10     }
11 }
```

- This code cannot reach the error state.
Abstracting the locking mechanism

1. replace the lock() and unlock() functions with a variable `state_of_lock`

2. add assertions that avoid the error state.

```c
state_of_lock = unlocked;

do {
    assert(state_of_lock == unlocked);
    state_of_lock = locked;
    old_count = count;
    request = GetNextRequest();
    if (request != NULL) {
        ReleaseRequest(request);
        assert(state_of_lock == locked);
        state_of_lock = unlocked;
        ProcessRequest(request);
        count = count + 1;
    }
}

while (old_count != count);
assert(state_of_lock == locked);
state_of_lock = unlocked;
```
Application of the overapproximating transformation

- The obtained formula is satisfiable.
- first assert fails if \textit{state\_of\_lock} = \textit{locked} which is impossible in the original program.
- We need a better abstraction

```c
state_of_lock = unlocked;
state_of_lock = *;
old_count = *;
count = *;
request = *;

assert(state_of_lock == unlocked);
state_of_lock = locked;
old_count = count;
request = getNextRequest();
if (request != NULL) {
    ReleaseRequest(request);
    assert(state_of_lock == locked);
    state_of_lock = unlocked;
    ProcessRequest(request);
    count = count + 1;
}

assume(old_count == count);

assert(state_of_lock == locked);
state_of_lock = unlocked;
```
Loop Invariants

- Predicate that holds at the beginning of the loop body, every iteration.

- For example:

```plaintext
1 int i=0;
2
3 while(i != 10) {
4     ...
5     i++;
6 }
```

- The following predicate is a loop invariant: \( 0 \leq i < 10 \).
How to prove a predicate is a loop invariant?

- **Induction.**

- Suppose that our program match this template:

```cpp
A;
while (C) {
  assert (I);
  B;
}
```

- Proof in 2 steps:
  1. *Base case*: invariant satisfied when entering loop for the first time
  2. *Step case*: from a state that satisfied the invariant, executing the loop body once brings to a state that satisfies it as well.
Induction proof

Construct 2 loop-free programs that check the 2 steps.

*Base case:* 

```plaintext
1 A;
2 assert (C \implies I);
```

*Step case:* 

```plaintext
1 assume (C \land I);
2 B;
3 assert (C \implies I);
```

- Both programs are loop free, thus bounded, and can be verified with a simple logical formula.
Example

Using earlier code:

```java
int i=0;
while(i != 10) {
    ...
    i++;
}
```

Base case:

```java
int i=0;
assert (i != 10 \implies i \geq 0 && i < 10);
```

Step case:

```java
assume (i != 10 && i \geq 0 && i < 10);
i++;
assert (i != 10 \implies i \geq 0 && i < 10);
```
Better Transformation with loop invariants

In addition to the previous 3 steps of the transformation, we add:

For every loop, add the invariant $I$ with 3 steps:

1. **Base Case** - add an assertion that $I$ holds before the nondeterministic assignments

2. **Induction Hypothesis** - add an assumption that $I$ holds after the nondeterministic assignments.

3. **Induction Step** - Add an assertion that $C \Rightarrow I$ holds at the end of the loop body.
Back to the locks example

Now we claim the program is correct :)
Finding a good loop invariant

- Is it a challenge to find an invariant strong enough. (In this example we guessed it)

- Finding suitable invariants is not simple and is an area of active research.

- A heuristic can select candidates and then try to proof and confirm the invariants.
Summary:

- Formal software verification can guarantee a program is correct.
- We can do that by translating the program to a logical formula, by adding the negation of the assertion to the formula.
- If the formula is satisfiable, the assertion does not hold.
- We can deal with loops in programs using bounded or unbounded methods.
- There is no easy solution.
Questions?