Decision Procedures

chapters 12.1-12.3

Daniel Kroening and Ofer Strichman
With help of Nikolaj Bjorner 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

```python
def sum_inverse(arr):
    res = 0
    for nb in arr:
        res+=1/nb
    return res
```

Can we find a bug in this program?

Introduction to software verification

Yes.

If one of the numbers is equal to 0.

```
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
      6

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 differents executions trace)
- 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

```
1  void ReadBlocks(int data[], int cookie)
2  {
3    int i = 0;
4    while (true)
5    {
6      int next;
7      next = data[i];
8      if (!(i < next && next < N)) return;
9      i = i + 1;
10     for (; i < next; i = i + 1) {
11         if (data[i] == cookie)
12            i = i + 1;
13         else
14            Process(data[i]);
15     }
16   }
17 }
```

- 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

```
1  void ReadBlocks(int data[], int cookie)
2  {
3    int i = 0;
4    while (true)
5    {
6      int next;
7      next = data[i];
8      if (!(i < next && next < N)) return;
9      i = i + 1;
10     for (; i < next; i = i + 1) {
11         if (data[i] == cookie)
12            i = i + 1;
13         else
14            Process(data[i]);
15     }
16   }
17 }
```

- 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

Bounded Program Analysis

Line	Kind	Instruction or condition
3	Assignment	`i = 0;`
7	Assignment	`next = data[i];`
8	Branch	`i < next && next < N`
9	Assignment	`i = i + 1;`
10	Branch	`i < next`
11	Branch	`data[i] != cookie`
14	Function call	`Process(data[i]);`
10	Assignment	`i = i + 1;`
10	Branch	`!(i < next)`
7	Assignment	`next = data[i];`
8	Branch	`!(i < next && next < N)`

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

- 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

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.

Line	Kind	Instruction or condition
3	Assignment	`i = 0;`
7	Assignment	`next = data[i];`
8	Branch	`i < next && next < N`
9	Assignment	`i = i + 1;`
10	Branch	`i < next`
11	Branch	`data[i] != cookie`
14	Function call	`Process(data[i]);`
10	Assignment	`i = i + 1;`
10	Branch	`!(i < next)`
7	Assignment	`next = data[i];`
8	Branch	`!(i < next && next < N)`

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

Instruction or condition
$i_1 = 0;$
$next_1 = data_0[i_1];$
$i_1 < next_1 \&\& next_1 < N_0$
$i_2 = i_1 + 1;$
$i_2 < next_1$
$data_0[i_2] != cookie_0$
$Process(data_0[i_2]);$
$i_3 = i_2 + 1;$
$!(i_3 < next_1)$
$next_2 = data_0[i_3];$
$!(i_3 < next_2 \&\& next_2 < N_0)$

SSA form

Bounded Program Analysis

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

```
i₁ = 0;
next₁ = data₀[i₁];
i₁ < next₁ && next₁ < N₀
i₂ = i₁ + 1;
i₂ < next₁
data₀ [i₂ ] != cookie₀
Process(data₀ [i₂ ]);
i₃ = i₂ + 1;
!(i₃ < next₁ )
next₂ = data₀ [i₃ ];
!(i₃ < next₂ && next₂ < N₀)
```

$\Longleftrightarrow$

$$
\begin{array}{ll}
i_1 = 0 & \wedge \\
next_1 = data_0[i_1] & \wedge \\
(i_1 < next_1 \ \wedge \ next_1 < N_0) & \wedge \\
i_2 = i_1 + 1 & \wedge \\
i_2 < next_1 & \wedge \\
data_0[i_2] \neq cookie_0 & \wedge \\
i_3 = i_2 + 1 & \wedge \\
\neg(i_3 < next_1) & \wedge \\
next_2 = data_0[i_3] & \wedge \\
\neg(i_3 < next_2 \ \wedge \ next_2 < N_0) \ . &
\end{array}
$$

- 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** !

Assertion Checking: Example

Program 12.2.1 Reading blocks from an array

```
1  void ReadBlocks(int data[], int cookie)
2  {
3    int i = 0;
4    while (true)
5    {
6      int next;
7      next = data[i];
8      if (!(i < next && next < N)) return;
9      i = i + 1;
10     for (; i < next; i = i + 1) {
11         if (data[i] == cookie)
12             i = i + 1;
13         else
14             Process(data[i]);
15     }
16   }
17 }
```

- 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 :

$$i_1 = 0 \quad \land \quad \neg(0 \leq i_1 \land i_1 < N_0)$$

Assertion Checking: Example

Program 12.2.1 Reading blocks from an array

```
1  void ReadBlocks(int data[], int cookie)
2  {
3     int i = 0;
4     while (true)
5     {
6        int next;
7        next = data[i];
8        if (!(i < next && next < N)) return;
9        i = i + 1;
10       for (; i < next; i = i + 1) {
11           if (data[i] == cookie)
12              i = i + 1;
13           else
14              Process(data[i]);
15       }
16    }
17 }
```

$$i_1 = 0 \quad \wedge \quad \neg(0 \le i_1 \wedge i_1 < N_0)$$

- 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

```
1  void ReadBlocks(int data[], int cookie)
2  {
3    int i = 0;
4    while (true)
5    {
6      int next;
7      next = data[i];
8      if (!(i < next && next < N)) return;
9      i = i + 1;
10     for (; i < next; i = i + 1) {
11         if (data[i] == cookie)
12             i = i + 1;
13         else
14             Process(data[i]);
15     }
16   }
17 }
```

ERROR

- 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

```
1  void ReadBlocks(int data[], int cookie)
2  {
3    int i = 0;
4    while (true)
5    {
6      int next;
7      next = data[i];
8      if (!(i < next && next < N)) return;
9      i = i + 1;
10     for (; i < next; i = i + 1) {
11         if (data[i] == cookie)
12             i = i + 1;
13         else
14             Process(data[i]);
15     }
16   }
17 }
```

Line	Kind	Instruction or condition
3	Assignment	`i = 0;`
7	Assignment	`next = data[i];`
8	Branch	`i < next && next < N`
9	Assignment	`i = i + 1;`
10	Branch	`i < next`
11	Branch	`data[i] = cookie`
12	Assignment	`i = i + 1;`
10	Assignment	`i = i + 1;`
10	Branch	`!(i < next)`
7	Assertion	`0 <= i && i < N`

Table 12.3. A second path in Program 12.2.1

Assertion Checking: An Other Example

$$
\begin{aligned}
& i_1 = 0 && \wedge \\
& next_1 = data_0[i_1] && \wedge \\
& (i_1 < next_1 \;\wedge\; next_1 < N_0) && \wedge \\
& i_2 = i_1 + 1 && \wedge \\
& i_2 < next_1 && \wedge \\
& data_0[i_2] = cookie_0 && \wedge \\
& i_3 = i_2 + 1 && \wedge \\
& i_4 = i_3 + 1 && \wedge \\
& \neg(i_4 < next_1) && \wedge \\
& \neg(0 \leq i_4 \;\wedge\; i_4 < N_0)
\end{aligned}
$$

- This constraint path is **satisfiable** ?

$$
\{i_1 \mapsto 0,\; N_0 \mapsto 3,\; next_1 \mapsto 2,\; data_0 \mapsto \langle 2, 6, 5\rangle,\; i_2 \mapsto 1,\; cookie_0 \mapsto 6,\; i_3 \mapsto 2,\; i_4 \mapsto 3\}\ ,
$$

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: γ**

  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

```
1  if (i < next) {
2     if (data[i] == cookie)
3        i = i + 1;
4     else
5        Process(data[i]);
6
7     i = i + 1;
8
9     if (i < next) {
10       if (data[i] == cookie)
11          i = i + 1;
12       else
13          Process(data[i]);
14
15       i = i + 1;
16    }
17 }
```

```
1  γ1 = (i0 < next0);
2  γ2 = (data0[i1] == cookie0);
3  i1 = i0 + 1;
4
5
6  i2 = γ2 ? i1 : i0;
7  i3 = i2 + 1;
8
9  γ3 = (i3 < next0);
10 γ4 = (data0[i3] == cookie0);
11 i4 = i3 + 1;
12
13
14 i5 = γ4 ? i4 : i3;
15 i6 = i5 + 1;
16 i7 = γ3 ? i6 : i3;
17 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

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 int i=0, j=0;
2
3 while(data[i] != '\n')
4 {
5   i++;
6   j=i;
7 }
8
9 assert(i == j);
```

→

```
1 int i=0, j=0;
2
3 if(data[i] != '\n')
4 {
5   i=*;
6   j=*;
7   i++;
8   j=i;
9 }
10
11 assume(data[i] == '\n');
12
13 assert(i == j);
```

1. nondeterministic values to modified variables

2. assume negation of condition

3. replace “while” with “if”

Translate to Logical Formula

```
1  int i=0, j=0;
2
3  if(data[i] != '\n')
4  {
5     i=*;
6     j=*;
7     i++;
8     j=i;
9  }
10
11 assume(data[i] == '\n');
12
13 assert(i == j);
```

$$
\begin{aligned}
& i_1 = 0 && \wedge \\
& j_1 = 0 && \wedge \\
& \gamma_1 = (data_0[i_1] \neq \texttt{'\textbackslash n'}) && \wedge \\
& i_3 = i_2 + 1 && \wedge \\
& j_3 = i_3 && \wedge \\
& i_4 = \gamma_1 \, ? \, i_3 : i_1 && \wedge \\
& j_4 = \gamma_1 \, ? \, j_3 : j_1 && \wedge \\
& data_0[i_4] = \texttt{'\textbackslash n'} && \wedge \\
& i_4 \neq j_4
\end{aligned}
$$

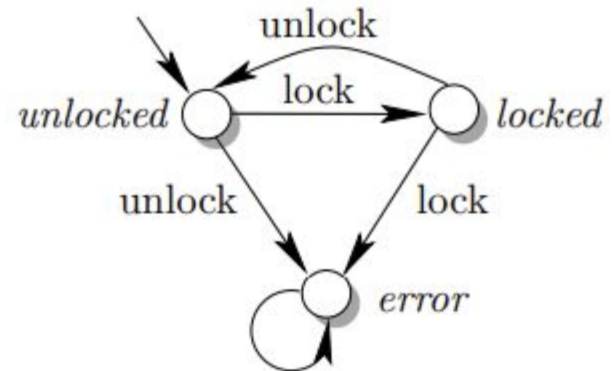
- This logical formula is not satisfiable
- The Assertion is true for every number of iterations!
- Ignores information from previous loop iterations

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  }
12  while(old_count != count);
13  unlock();
```

- This code cannot reach the error state.



Abstracting the locking mechanism

```
1   state_of_lock = unlocked;
2   do {
3       assert(state_of_lock == unlocked);
4       state_of_lock = locked;
5       old_count = count;
6       request = GetNextRequest();
7       if (request != NULL) {
8           ReleaseRequest(request);
9           assert(state_of_lock == locked);
10          state_of_lock = unlocked;
11          ProcessRequest(request);
12          count = count + 1;
13      }
14  }
15  while (old_count != count);
16  assert(state_of_lock == locked);
17  state_of_lock = unlocked;
```

1. replace the lock() and unlock() functions with a variable *state_of_lock*
2. add assertions that avoid the error state.

```
1    state_of_lock = unlocked;
2
3    state_of_lock = *;
4    old_count = *;
5    count = *;
6    request = *;
7
8    assert(state_of_lock == unlocked);
9    state_of_lock = locked;
10   old_count = count;
11   request = GetNextRequest();
12   if (request != NULL) {
13     ReleaseRequest(request);
14     assert(state_of_lock == locked);
15     state_of_lock = unlocked;
16     ProcessRequest(request);
17     count = count + 1;
18   }
19
20   assume(old_count == count);
21
22   assert(state_of_lock == locked);
23   state_of_lock = unlocked;
```

Application of the overapproximating transformation

- The obtained formula is satisfiable.
- first assert fails if *state_of_lock = locked*

  which is impossible in the original program.

- We need a better abstraction

Loop Invariants

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

- For example:

```
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:

```
1 A;
2 while(C) {
3       assert(I);
4       B;
5 }
```

- 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:

```
1 A;
2 assert(C ⟹ I);
```

Step case:

```
1 assume(C ∧ I);
2 B;
3 assert(C ⟹ I);
```

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

Example

Using earlier code: and invariant $0 \leq i < 10$.

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

Base case:

```
1 int i=0;
2 assert(i != 10  ⟹  i >= 0 && i < 10);
```

Step case:

```
1 assume(i != 10 && i >= 0 && i < 10);
2 i++;
3 assert(i != 10  ⟹  i >= 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 :)

```
1   state_of_lock = unlocked;
2
3   assert(state_of_lock == unlocked); // induction base case
4
5   state_of_lock = *;
6   old_count = *;
7   count = *;
8   request = *;
9
10  assume(state_of_lock == unlocked); // induction hypothesis
11
12  assert(state_of_lock == unlocked); // property
13  state_of_lock = locked;
14  old_count = count;
15  request = GetNextRequest();
16  if (request != NULL) {
17    ReleaseRequest(request);
18    assert(state_of_lock == locked); // property
19    state_of_lock = unlocked;
20    ProcessRequest(request);
21    count = count + 1;
22  }
23
24   // induction step case
25  assert(old_count != count  ⟹  state_of_lock == unlocked);
26
27  assume(old_count == count);
28
29  assert(state_of_lock == locked); // property
30  state_of_lock = unlocked;
```

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?