CS615: Formal Specification and Verification of Programs 2019

Lecture 17: Model Checking

Instructor: Ashutosh Gupta

IITB, India

Compile date: 2019-10-17

Key issues with abstract interpretation

- Predefined precision – may not be sufficient for the program at hand
- Verification data structure never blows up (may be a good thing?)
- Bug finding is not naturally integrated
  - If verification fails, no counterexample!!!!!
  - **A very big problem**
- Precision control is not flexible

Model checking - a different approach

- explore states — concrete/symbolic/abstract states
- Blows up the memory usage — we have loads of it
- Needs only two operations depending on program semantics and abstraction domain
  1. post ($sp/sp^{\#}$)
  2. comparison ($\sqsubseteq$)
- No need of sophisticated join $\sqcup$ and widening ∇ operators.

Let us explore model checking!

Topic 17.1

Concrete model checking - enumerate reachable states

Isn't enumeration impossible?

- Explore the transition graph explicitly
- If edge labels are guarded commands then finding next values are trivial
  - light weight machinery
- After resolving non-determinisms, concrete model checking reduces to program execution
- May be only finitely many states are reachable
- May be impossible to cover all states explicitly, but it may cover a portion of interest

Concrete model checking

Algorithm 17.1: Concrete model checking

Input: $P = (V, L, \ell_0, \ell_e, E)$
Output: SAFE if P is safe, UNSAFE otherwise

```
reach := ∅;
worklist := {(ℓ0, v) | v ∈ ℤ^|V|};
while worklist ≠ ∅ do
    choose (ℓ, v) ∈ worklist;
    worklist := worklist \ {(ℓ, v)};
    if (ℓ, v) ∉ reach then
        reach := reach ∪ {(ℓ, v)};
        foreach (ℓ, F(V, V'), ℓ') ∈ E do
            worklist := worklist ∪ {(ℓ', v') | F(v, v')};
if (ℓe, _) ∈ reach then
    return UNSAFE
else
    return SAFE
```

infinite set. what?

the choice defines the nature of exploration

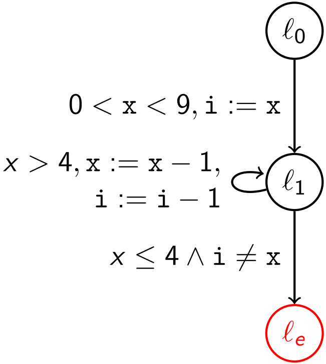
Exercise 17.1

Suggest improvements in the algorithm

Example: concrete model checking

Example 17.1

Consider the following



Let $V = [\mathtt{x}, \mathtt{i}]$

Initialization:
$reach = \emptyset$, $worklist = \{(\ell_0, v) | v \in \mathbb{Z}^2\}$

Choose a state:
Lets choose $(\ell_0, [8, 0])$

Update worklist:
$worklist := worklist \setminus \{(\ell_0, [8, 8])\}$

Add successors in worklist if state not visited:
$worklist := worklist \cup \{(\ell_1, [8, 8])\}$

Add to reach, since there are no more successors:
$reach := reach \cup \{(\ell_0, [8, 0])\}$

... go back to choosing a new state from worklist

Search strategy : depth first search (DFS)

- search deeper states first
- worklist is a stack
- Often, the depth is bounded by threshold in tools.
  - If the search visits a state at the threshold depth, the state is moved from worklist to reach set without considering the successors of the state.
  - If the state is visited again via a shorter depth, the state needs to be explored again.

Exercise 17.2

a. When would you like to use DFS?

b. Modify the previous algorithm to write down the DFS version.

Search strategy: breadth first search(BFS)

- search shallow states first
- If finite successors, no need to put any artificial bounds on breadth
- On time out, we may claim some guarantees of partial completeness

Exercise 17.3

a. Suggest a data structure for worklist
b. When do we have infinite successors?
c. When would you like to use BFS?

Directed search strategies

- The search is guided by the position of error states in the state space
- Assign an estimate of reaching error from each state
- Explore the state from wroklist that has least estimate
- The estimation function should have a low cost to compute.
- The estimation function should always underestimate the distance to error.(why?)
- In the area of artificial intelligence, there has been a few proposals to do directed search
  - A^*
  - IDA^*

Directed search strategies: A^*

- Worklist is a priority queue,
- Priority weight is assigned to a state in worklist based on sum of
  - the cost of reaching the state and
  - the estimate on cost of reaching error from the state
- Each time a new state is explored, we update estimates of the neighbours.

Exercise 17.4

Suggest an estimate function in the previous example

Optimizations: exploiting structure

- Symmetry reduction
  - e.g. MAC address of client is irrelevant in a banking software.
  - test one; test all.
- Assume guarantee — for modular software
  - Let us suppose a software consist of two components C_1 and C_2
  - We define specification for each component (A_i, G_i)
  - The specification implies that we assume A_i on inputs of C_i and the component guarantees G_i on outputs.
  - For each component i, we assume A_i and G_{1-i} and model check if G_i holds.

This approach simplifies each verification task.

Optimizations: exploiting structure II

Partial order reduction — for concurrent systems

- If order of two operations is irrelevant, then explore only one of the order.

Example 17.2

Let us suppose one thread opens a file and another sends a message on network.

Since the activities have no dependence between them, we need not consider both the orders between them.

Optimizations: reducing space

- hashed states - reach set contains hash of states (not sound)
- Stateless exploration - no reach set (redundant)

Trade-off among time, space, and soundness

Exercise 17.5

a. Write concrete model checking using hash tables
b. What is the standard hash function used in standard hash tables in C++ and Java?
c. Why stateless model checking is useful?

Blackbox model checking

We may have binary of the program and access to internal state.

We drive the program via various inputs and keep the record of the input choices such as

- explicit input
- scheduling of threads

Proof and counterexample

Definition 17.1

A proof of a program is an object that allows one to check safety of the program using a low complexity (preferably linear) algorithm in the size of the object.

Example 17.3

In our concrete model checking algorithm, reach set is the proof. The checker needs to find that no more states can be reached from reach.

Definition 17.2

A counterexample of a program is an execution that ends at ℓ_e.

A verification method may produce three possible outcomes for a program

- proof
- counterexample
- unknown or non-termination

Enabling counterexample generation

Algorithm 17.2: Concrete model checking

Input: $P = (V, L, \ell_0, \ell_e, E)$

Output: $\textsc{Safe}$ if P is safe, $\textsc{Unsafe}$ otherwise

$\mathit{reach} := \emptyset$; $\mathit{parents} := \lambda x.\textsc{NaN}$;

$\mathit{worklist} := \{(\ell_0, v) | v \in \mathbb{Z}^{|V|}\}$;

while $\mathit{worklist} \neq \emptyset$ **do**

  choose $(\ell, v) \in \mathit{worklist}$;

  $\mathit{worklist} := \mathit{worklist} \setminus \{(\ell, v)\}$;

  if $(\ell, v) \notin \mathit{reach}$ **then**

    $\mathit{reach} := \mathit{reach} \cup \{(\ell, v)\}$;

    foreach v' *s.t.* $F(v, v')$ *is sat and* $(\ell, F(V, V'), \ell') \in E$ **do**

      $\mathit{worklist} := \mathit{worklist} \cup \{(\ell', v')\}$; $\mathit{parents}((\ell', v')) := (\ell, v)$;

if $(\ell_e, v) \in \mathit{reach}$ **then**

  return $\textsc{Unsafe}(\mathit{traverseToInit}(\mathit{parents}, (\ell_e, v)))$

else

  return $\textsc{Safe}$

Exercise 17.6

add data structure to report counterexample

Topic 17.2

Symbolic methods

Why symbolic?

To avoid, state explosion problem

Symbolic methods

Now, we cover some methods that try/avoid to compute lfp

- Symbolic model checking
- Constraint based invariant generation

Symbolic state

Definition 17.3

A symbolic state s of $P = (V, L, \ell_0, \ell_e, E)$ *is a pair* (ℓ, F)*, where*

- $\ell \in L$
- F *is a formula over variables* V *in a given theory*

Symbolic model checking

Algorithm 17.3: Symbolic model checking

Input: $P = (V, L, \ell_0, \ell_e, E)$

Output: SAFE if P is safe, UNSAFE otherwise

$reach : L \rightarrow \Sigma(V) \ := \ \lambda x.\bot;$

$worklist \ := \ \{(\ell_0, \top)\};$

while $worklist \neq \emptyset$ **do**

    choose $(\ell, F) \in worklist$;

    $worklist \ := \ worklist \setminus \{(\ell, F)\};$

    if $\neg(F \Rightarrow reach(\ell))$ *is sat* **then**

        $reach \ := \ reach[\ell \mapsto reach(\ell) \vee F];$

        foreach $(\ell, \rho(V, V'), \ell') \in E$ **do**

            $worklist \ := \ worklist \cup \{(\ell', sp(F, \rho))\};$

if $reach(\ell_e) \neq \bot$ **then**

    return UNSAFE

else

    return SAFE

We need efficient implementations of various logical operators!

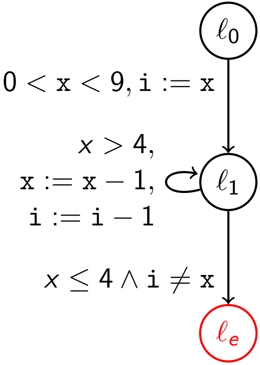
Exercise 17.7

Give a condition for definite termination?

Example: symbolic model checking

Example 17.4

Consider the following example



Let $V = [\mathtt{x}, \mathtt{i}]$

Init: $reach = \lambda x.\bot$, $worklist = \{(\ell_0, \top)\}$

Choose a state: $(\ell_0, \top)$ *(only choice)*

Update worklist: $worklist := \emptyset$

Add successors in worklist:

Since $\neg(\top \Rightarrow reach(\ell_0))$ *is sat,*

$\quad worklist := worklist \cup \{(\ell_1, 0 < x = i < 9)\}$

$\quad reach(\ell_0) := reach(\ell_0) \lor \top := \top$

Again choose a state: $\{(\ell_1, 0 < x = i < 9)\}$

Update worklist: $worklist := \emptyset$

Add successors in worklist:

Since $\neg(0 < x = i < 9 \Rightarrow reach(\ell_1))$ *is sat,*

$\quad worklist := worklist \cup \{(\ell_1, 3 < x = i < 9), (\ell_e, \bot)\}$

$\quad reach(\ell_1) := reach(\ell_1) \lor 0 < x = i < 9$

$\quad reach(\ell_e) := reach(\ell_e) \lor \bot$

Exercise 17.8

complete the run of the algorithm

Proof generation

If the symbolic model checker terminates with the answer SAFE, then it must also report a proof of the safety, which is the *reach* map.

It has implicitly computed a Hoare style proof of $P = (V, L, \ell_0, \ell_e, E)$.

$$(\ell, \rho(V, V'), \ell') \in E \quad \{reach(\ell)\}\rho(V, V')\{reach(\ell')\}$$

If an LTS program has been obtained from a simple language program then one may generate a Hoare style proof system.

Exercise 17.9

Describe the construction for the above translation

End of Lecture 17