CS 267: Automated Verification

Lecture 10: Explicit State Model Checking Hueristics

Instructor: Tevfik Bultan

Explicit Model Checking Hueristics

There are several heuristics for explicit state model checking (implemented in Spin)

- Nested depth First Search
- Counter-Example Generation
- Bit-State Hashing
- On-The-Fly Model Checking
- Partial Order Reduction

We will discuss some of them in this lecture

Buchi Automata Language Emptiness Check

Given a Buchi automaton $A = (\Sigma, Q, \Delta, Q_0, F)$, is $L(A) = \varnothing$?

$L(A) \neq \varnothing$ if and only if there exists a run $r = q_0, q_1, q_2, \ldots$ s.t.

- $q_0 \in Q_0$,
- for all $i \geq 0$, there exists an $a_i \in \Sigma$ such that $(q_i, a_i, q_{i+1}) \in \Delta$ and
- $\inf(r) \cap F \neq \varnothing$

Such a run exists if and only if there exists an accepting state $q \in F$ such that

- q is reachable from an initial state in Q_0 and
- q is reachable from itself (i.e., q is contained in a cycle)

Buchi Automata Language Emptiness Check

- Any run of a Buchi automaton has a suffix in which all the states on that suffix appear infinitely many times.
  - Each state on that suffix is reachable from any other state
  - Hence these states form a strongly connected component
  - If there is an accepting state among those states than the run is an accepting run
- So emptiness check involves finding a strongly connected component that contains an accepting state and is reachable from an initial state

Finding Reachable Cycles

- To find cycles in a graph one can use a depth-first search algorithm which constructs the strongly connected components in linear time by adding two integer numbers to every state reached [Tarjan, 72]

- If a strongly connected component reachable from an initial state contains an accepting state then the language accepted by the Buchi automaton is not empty.

- There is a more memory efficient algorithm for checking the same condition which is called nested depth first search.

Nested Depth First Search

[Corcoubetis, Vardi, Wolper, Yannakakis 92]

- Do a depth first search from the initial states
  - While doing this search build a postorder list (children appear before their parents) of reachable accepting states. Let this ordered list be $L = q_1, q_2, \dots, q_k$ where q_1 is the first postorder reachable state and q_k is the last.
- Do a second depth first search from the elements in L
  - Start the search from q_1
  - Once the search from q_i is finished (either q_i is reached, i.e., a cycle is found, or there are no more reachable states from q_i), restart the search from q_{i+1} but do not reconsider the states that have been visited during searches from q_j, for $j \leq i$.

Nested Depth First Search

- This algorithm visits each state in the graph once in each depth-first search, and it only needs to mark each visited state
  - Hence, there is no need to store two integer variables per state, this search can be implemented using one Boolean variable per state.

- We can also interleave the first and the second depth-first search.
  - In the interleaved nested depth first search two Boolean variables per state are used to mark if the stored state is visited during the first search or the second search.

Nested DFS

```
main()  {
  Stack = Q_0; //initial states
  Queue = {};
  StateSpace = {};
  search1();
  while Queue not empty  {
    s = head(Queue);
    remove s from Queue;
    push s to Stack;
    seed = s;
    search2();
  }
}
```

```
search1()  {
  if Stack is empty return();
  s = top(Stack);
  add (s,1) to StateSpace;
  for each successor t of s do {
      if (t,1) not in StateSpace  {
        push t to Stack;
        search1();
      }
  }
  if accepting(s) add s to Queue;
  remove s from Stack;
}
```

```
search2()  {
  if Stack is empty return();
  s := top(Stack);
  add (s,2) to StateSpace;
  for each successor t of s do
    if (t,2) not in StateSpace  {
      push t to Stack;
      search2();
    }
    else
      if (t == seed) report_cycle();
  remove s from Stack;
}
```

Nested DFS with Interleaving

```
main() {
  Stack1 = Q_0;
  Stack2 = {};
  StateSpace = {};
  search1();
}
```

```
search1() {
  if Stack1 is empty return();
  s = top(Stack1);
  add (s,1) to StateSpace;
  for each successor t of s do {
    if (t,1) not in StateSpace {
      push t to Stack1;
      search1();
    }
  }
  if accepting(s) {
    seed = s;
    push s to Stack2;
    search2();
  }
  remove s from Stack1;
}
```

```
search2() {
  if Stack2 is empty return();
  s = top(Stack2);
  add (s,2) to StateSpace;
  for each successor t of s do {
    if (t,2) not in StateSpace {
      push t to Stack2;
      search2();
    }
    else
      if (t == seed) report_cycle();
  }
  remove s from Stack2;
}
```

Why Does Nested DFS Work?

- Assume that q_i appears in the postorder list before q_j, then the following are possible:
  1. q_i is reachable from q_j and q_j is reachable from q_i
  2. q_i is reachable from q_j and q_j is not reachable from q_i
  3. q_i is not reachable from q_j and q_j is not reachable from q_i

- Note that the following is not possible:
  – q_i is not reachable from q_j and q_j is reachable from q_i

  since this would violate the postorder

Why Does Nested DFS work?

- If q_j is reachable from q_i, where q_i appears before q_j in the postorder list, then q_i is reachable from itself
  – Note that, this corresponds to the case 1 in the previous slide which means that q_i is reachable from itself

- So, for the ***first*** q_j that is reachable from itself the path from q_j to itself cannot contain any node reachable from a q_i if q_i appears before q_j in the postorder list
  – If it did, then q_i would be reachable from itself and q_j would not be the first node that is reachable from itself

The Explicit Stack

- Why do we keep an explicit stack during the depth first search (in addition to the control stack that is automatically handled via recursive procedure calls)?

- In the report_cycle() procedure we use the contents of Stack1 and Stack2 to print the counter-example path
  – Note that if we print the states in Stack1 and Stack2 in the order they are pushed to the stack then we end up printing a counter-example path.

Bit State Hashing

- We are storing visited states in the StateSpace
  – Each state can be inserted to the StateSpace twice (once for the first search and once for the second search)
- Assume that
  – we have M bytes of memory
  – we use K bytes of storage per state
  – the transition systems has R reachable states
- Then the portion of state space we can cover is

  $M / (2 \times K \times R)$
- The idea in bit-state hashing is to improve the coverage of the state space using an hash function
  – However this may cause us miss some bugs!

Bit State Hashing (aka Bloom Filters)

- The idea is to use two ***boolean*** arrays as hash tables and use a hash function to mark these arrays
- When we visit a state we will compute the hash value for that state and we will mark the entry that corresponds to the hash value in the hash table as visited
- If later on another state is mapped to the same hash value it will not be explored since that entry has been marked as visited
- Note that normally we would store the value (i.e., the state) in the hash table to resolve conflicts. In bit state hashing we are discarding the value to save memory.
  - When there is a hash collision some states are not explored since the entry corresponding to them are marked as visited earlier by another state.

Bit State Hashing

- Bit state hashing is better than partial depth first search for two reasons:
  - the states that are ignored during bit state hashing are randomly distributed
  - we can explore more states using bit state hashing since we are using less memory per state
- The portion of state space we can cover using bit state hashing is

  $(M \times 4) / R$

  - Remember that without bit state hashing the portion of the state space we can cover was

    $M / (2 \times K \times R)$

Nested DFS with Interleaving and Bit State Hashing

```
main() {
  boolean H1[MAX];
  boolean H2[MAX];
  Stack1 = Q_0;
  Stack2 = {};
  set H1 and H2 to false
  search1();
}
```

```
search1() {
  if Stack1 is empty return();
  s = top(Stack1);
  H1[hash(s)] := true;
  for each successor t of s do {
    if H1[hash(t)] = false {
      push t to Stack1;
      search1();
    }
  }
  if accepting(s) {
    seed = s;
    push s to Stack2;
    search2();
  }
  remove s from Stack1;
}
```

```
search2() {
  if Stack2 is empty return();
  s = top(Stack2);
  H2[hash(s)] := true;
  for each successor t of s do {
    if H2[hash(t)] = false {
      push t to Stack2;
      search2();
    }
    else
      if (t == seed) report_cycle();
  }
  remove s from Stack2;
}
```

Bit State Hashing

- Due to bit state hashing we may miss some bugs
- However, if we find a bug using bit state hashing, it is a real bug
  - Bit state hashing will not cause us to report spurious bugs since the states stored in the stacks denote a real execution path

On-The-Fly Model Checking

- In the automata-based model checking:
  - we are looking for accepting states
    - that are reachable from an initial state and
    - that are part of a cycle
  - in the automaton that corresponds to the product of
    - the transition system automaton and
    - the negated property automaton
- If we construct the product automaton first and then do the search for accepting cycles,
  - then we would traverse the whole state space of the transition system while computing the product

On-The-Fly Model Checking

- In on-the-fly model checking we do not construct the product automaton before the search
- Instead we construct the product automaton during the nested depth first search
- This is what happens:
  - During the depth first search we execute the transitions of the transition system automaton and the property automaton synchronously to find the successors of a given state
  - A state is accepting if the property automaton is at an accepting state (all states of the transition system automaton are accepting)

On-The-Fly Model Checking

- This type of on-demand exploration of the state space helps us avoid visiting all the states in case we find an accepting cycle (i.e., a counter-example behavior)

- Note that once we find an accepting cycle we can stop the search and report the counter-example
  - We do not have to traverse the rest of the state space