CSC2/455 Software Analysis and Improvement
Program Analysis II – Model Checking

Sreepathi Pai

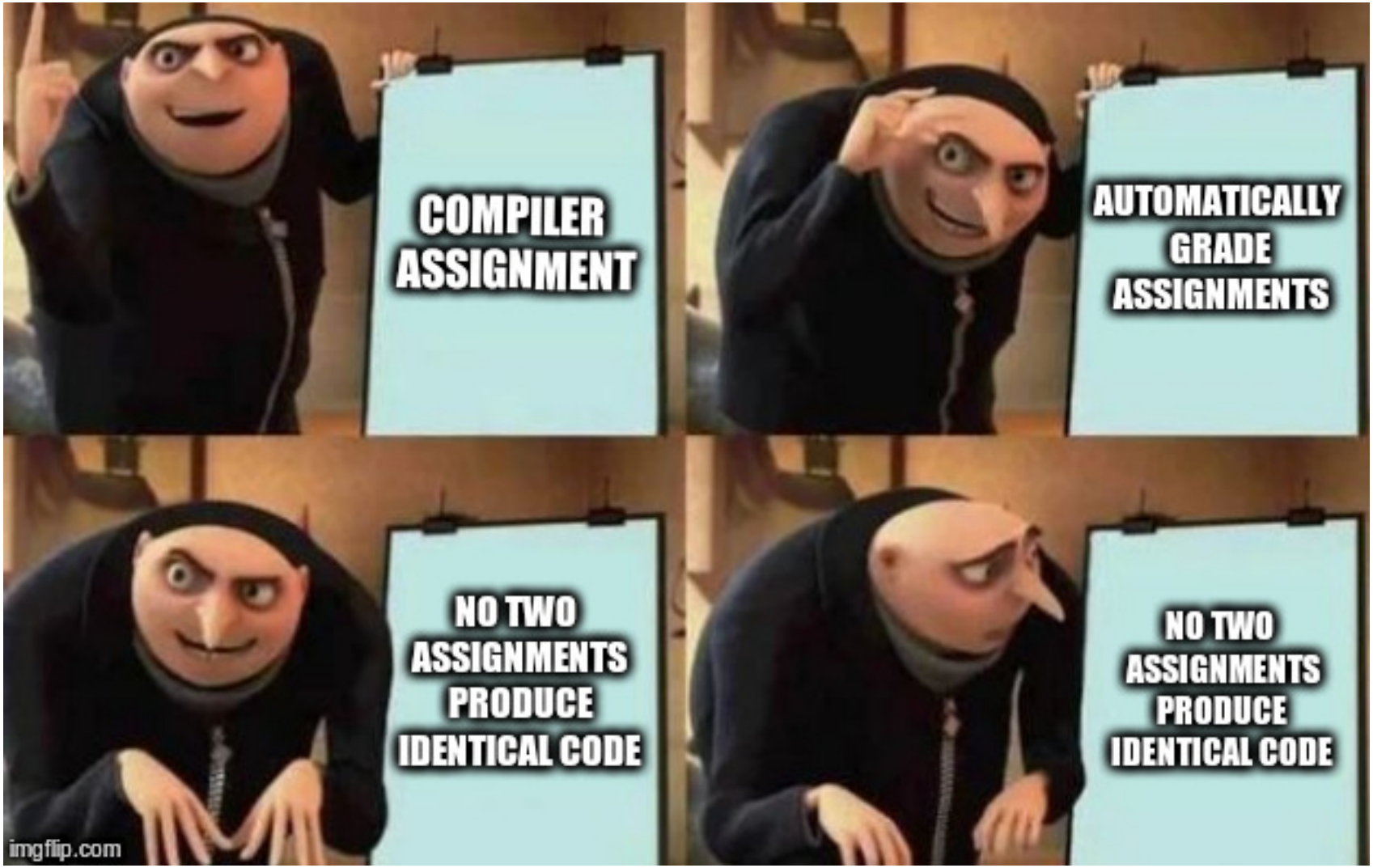
URCS

April 24, 2019

Outline

Outline

The Plan


COMPILER ASSIGNMENT
AUTOMATICALLY GRADE ASSIGNMENTS
NO TWO ASSIGNMENTS PRODUCE IDENTICAL CODE
NO TWO ASSIGNMENTS PRODUCE IDENTICAL CODE
imgflip.com

Check for Equivalence

- A: Original source program
- B: Compiler-generated program (e.g. your 3-address code)
- Is $A = B$?
  - Program equivalence problem
  - Undecidable in general

Test?

- Develop test cases
- Run B with these test cases
  - Works
  - Tests may miss bugs
- Also, many programs harder to test
  - Don't have `main`
  - Accept input interactively
  - Buggy compilers may introduce infinite loops

Solution

- Ended up using bounded model checking for C
  - CBMC
- Allows me to check that certain properties hold across all executions
- Can still require manual inspection
  - And I manually inspected all your results – successful or not

Computing the minimum of three numbers

```
int min_of_3(int x, int y, int z) {
   int min3;

   if(x > y) {
      if(y > z) {
         min3 = z;
      } else {
         min3 = y;
      }
   } else {
     if(x > z) {
         min3 = z;
     } else {
         min3 = x;
     }
   }

   return min3;
}
```

Adding Assertions

```
int min_of_3(int x, int y, int z) {
   ...

   __CPROVER_assert(min3 == x || min3 == y || min3 == z,
                    "must be one of inputs");

   __CPROVER_assert(min3 <= x, "<= x");
   __CPROVER_assert(min3 <= y, "<= y");
   __CPROVER_assert(min3 <= z, "<= z");

   return min3;
}
```

(Note: CBMC can also use existing `assert` statements)

Verifying

```
$ cbmc --function min_of_3 filename.c
CBMC version 5.6 64-bit x86_64 linux
 ...
Removal of function pointers and virtual functions
Partial Inlining
Generic Property Instrumentation
Starting Bounded Model Checking
size of program expression: 64 steps
simple slicing removed 5 assignments
Generated 4 VCC(s), 4 remaining after simplification
Passing problem to propositional reduction
converting SSA
Running propositional reduction
Post-processing
 ...

** Results:
[min_of_3.assertion.1] must be one of inputs: SUCCESS
[min_of_3.assertion.2] <= x: SUCCESS
[min_of_3.assertion.3] <= y: SUCCESS
[min_of_3.assertion.4] <= z: SUCCESS

** 0 of 4 failed (1 iteration)
VERIFICATION SUCCESSFUL
```

Another implementation

```
int min_of_3(int x, int y, int z) {
   int min3;

   if(x > y && y > z) {
     min3 = z;
   } else {
     if(x > y)
       min3 = y;
     else
       min3 = x;
   }

   __CPROVER_assert(min3 == x || min3 == y || min3 == z,
                    "must be one of inputs");

   __CPROVER_assert(min3 <= x, "<= x");
   __CPROVER_assert(min3 <= y, "<= y");
   __CPROVER_assert(min3 <= z, "<= z");

   return min3;
}
```

Verifying

```
CBMC version 5.6 64-bit x86_64 linux
 ...
Partial Inlining
Generic Property Instrumentation
Starting Bounded Model Checking
size of program expression: 58 steps
simple slicing removed 5 assignments
Generated 4 VCC(s), 4 remaining after simplification
Passing problem to propositional reduction
converting SSA
Running propositional reduction
Post-processing
Solving with MiniSAT 2.2.1 with simplifier
 ...
Runtime decision procedure: 0.018s

** Results:
[min_of_3.assertion.1] must be one of inputs: SUCCESS
[min_of_3.assertion.2] <= x: SUCCESS
[min_of_3.assertion.3] <= y: SUCCESS
[min_of_3.assertion.4] <= z: FAILURE

** 1 of 4 failed (2 iterations)
VERIFICATION FAILED
```

What! My code, wrong?

```
$ cbmc --trace --function min_of_3 file.c
  ...
State 17 file min3_2.c line 1 thread 0
------------------------------------------------
  INPUT x: -1412553063 (10101011110011100010011010011001)

State 19 file min3_2.c line 1 thread 0
------------------------------------------------
  INPUT y: -1151925590 (10111011010101110000001010101010)

State 21 file min3_2.c line 1 thread 0
------------------------------------------------
  INPUT z: -1949367656 (10001011110011110000001010011000)

 ...

State 30 file min3_2.c line 10 function min_of_3 thread 0
------------------------------------------------
  min3=-1412553063 (10101011110011100010011010011001)

Violated property:
  file min3_2.c line 17 function min_of_3
  <= z
  min3 <= z
```

Loops: Definite Bounds

```
for(i = 0; i < 10; i++) {
  ...
}
```

CBMC will unroll loop.

Loops: Symbolic Bounds

```
for(i = 0; i < N; i++)
   B;
```

gets unrolled by a fixed number (B is body), with unroll assert:

```
i = 0;
if(i < N) {
  B;
  i++;

  if(i < N) {
     B;
     i++;

     assert(N == 2);
  }
}
```

- If `assert` fails, unrolling was insufficient.
  - Not sound!
  - Otherwise, conclusion is sound

Other complications

- Pointers, arrays, dynamic memory allocation, etc.
- See CPROVER manual for more details

Outline

Basic Ideas

- Formula φ
  - Correctness (Safety) property
  - Propositional logic
  - Example: first argument of all the `__CPROVER_assert` statements
- Intepretation $\mathcal{K}$
  - More on this later
- We ask: $\mathcal{K} \models \varphi$?
  - Is φ true in $\mathcal{K}$?

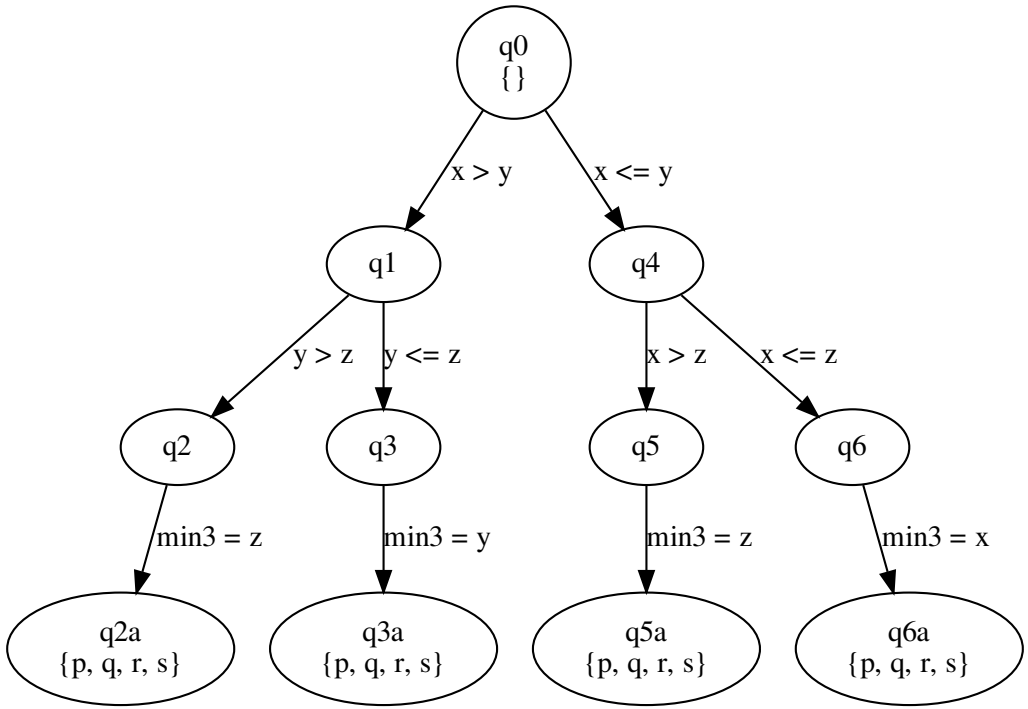
Transition System

- $\mathcal{T} = (Q, I, E, \delta)$
  - set of states Q (e.g. values of all variables)
  - initial states $I \in Q$
  - action labels E (e.g. program statements)
  - (total) transition relation $\delta \subset Q \times E \times Q$
- A run of $\mathcal{T}$ is the same as a trace of states
  - $s0e0s1...$ where $(s0, e0, s1) \in \delta$, and $s0 \in I$
- A reachable state is a state that exists in some run.

Kripke Structures

- Let $\mathcal{V}$ be a set of propositions
  - e.g. `min3 <= x`
  - e.g. `min3 <= y`
- A Kripke structure $\mathcal{K} = (Q, I, E, \delta, \lambda)$ is a transition system where:
  - $\lambda : Q \rightarrow 2^{\mathcal{V}}$
- λ is a function that maps a state q to the (subset) of propositions from $\mathcal{V}$ that are true in that state
  - $q \models P$ where $P \in \mathcal{V}$

Kripke structure for our min-of-3 example



- Let p be the "must be one of inputs" proposition
- Let q, r, s be the $<= x, <= y, <= z$ proposition
- (Note: True propositions in internal states not shown)

Invariants

- An invariant is a safety property for the system that holds in every reachable state
- An inductive invariant holds in the initial state, and is preserved by all transitions
  - including transitions from unreachable states
  - more on this when we discuss Hoare Logic

Invariant Checking Algorithm: High level details

- Assume finite Kripke structure
- Given an invariant to check,
  - Enumerate all reachable states
  - Check that invariant holds in all of them

Invariant Checking Algorithm: Pseudocode

```
def verify_inv(ks, inv):
    done = set()
    todo = set()

    for s in ks.initial_states():
        if s in done: continue

        todo.add(s)

        while len(todo) > 0:
            ss = todo.pop()
            done.add(ss)

            if not ss.satisfies(inv): return False

            for succ in ss.successors():
                if succ not in done: todo.add(succ)

    return True
```

based on Figure 3.3 in S. Merz, An introduction to Model Checking.

Outline

Progress

- Does something "good" eventually happen?
- Does the system ever deadlock?
- Does the system livelock?
  - An action e is no longer possible after a particular state q_i
- These require reasoning over *sequences* of states
  - These can be infinite even in a finite Kripke structure

These properties need a *temporal* logic, that incorporates notions of (logical) "time points" into formulae we want to check.

Specifying temporal properties in PTL

- Let $\sigma = q_0 q_1 \ldots$ be a sequence of states
  - σ_i is the state i
  - $\sigma|_i$ is the suffix $q_i q_{i+1} \ldots$ of σ
- Let φ be a formula
- $\sigma \models \varphi$ if $\varphi \in \lambda(\sigma_0)$
- $\mathrm{X}\varphi$ (also a formula), read as "next φ",
  - $\sigma \models \mathrm{X}\varphi$ if $\sigma|_1 \models \varphi$
- $\varphi \mathrm{U} \psi$ (also a formula), read as "φ until ψ"
  - $\sigma \models \varphi \mathrm{U} \psi$ if and only if there exists $k \in N$
  - $\sigma|_k \models \psi$
  - for all $1 \leq i < k$, $\sigma|_i \models \varphi$
  - Note: φ can continue to hold after k

More temporal properties

- $\mathrm{F}\varphi$, "eventually φ"
  - $\mathrm{true}\mathrm{U}\varphi$
- $\mathrm{G}\varphi$, "always φ"
  - $\neg\mathrm{F}\neg\varphi$
- $\varphi\mathrm{W}\psi$, "φ unless ψ"
  - $(\varphi\mathrm{U}\psi) \vee G\psi$
- $\mathrm{GF}\varphi$
- $\mathrm{FG}\varphi$

Some examples of invariants

- $\mathrm{G}\neg(own_1 \wedge own_2)$
  - where own_1 and own_2 are propositions representing states in which locks for resource are obtained by process 1 and 2
- Other properties (see the reading)
  - weak and strong fairness
  - precedence
  - etc.

Existential and Universal Properties: CTL

- Branching time logic for properties of systems
  - Computation Tree Logic (CTL)
- $\mathrm{EX}\varphi$, there exists a transition where φ holds from current state
- $EG\varphi$, exists a path from current state where φ holds on all states
- EU, exists a path until...
- Also Ax properties, properties that hold on all possible paths from current state

Verifying PTL and CTL invariants?

- State sequences of infinite length possible
- How do we check invariants?

Büchi Automata

- ω-automaton
  - run on infinite strings
- strings represent state sequences (actually $\lambda(q_0)\lambda(q_1)...$)
- non-deterministic as well as deterministic
  - but non-deterministic Büchi automata more powerful

Büchi Automata Example

Stephan Merz, An Introduction to Model Checking

Outline

Further Reading and Links

- Stephan Merz, An Introduction to Model Checking
  - Accessible and good introduction, with links to other material
- Spin Model Checker
- Selected industrial applications
  - CACM, "How Amazon Web Services Uses Formal Methods
  - CACM, "A Decade of Software Model Checking with SLAM
- A segue into compiler verification
  - Ken Thompson, Reflections on Trusting Trust, Turing Award Lecture 1984
  - The COMPCERT project