CSC2/455 Software Analysis and Improvement
Model Checking

Sreepathi Pai
April 13, 2022
URCS
Outline

A Tour of CBMC

Some (unneeded?) History

Model Checking

Liveness Properties

Postscript
A Tour of CBMC

Some (unneeded?) History

Model Checking

Liveness Properties

Postscript
#include <assert.h>

int nondet_int();

int min(int a, int b) {
    if(a < b)
        return a;
    else
        return a;
}

void check_min() {
    int x, y, r;

    x = nondet_int();
    y = nondet_int();

    r = min(x, y);

    assert(r == x || r == y);
    assert(r <= x && r <= y);
}
$ cbmc --function check_min wrong_min.c
CBMC version 5.10 (cbmc-5.10) 64-bit x86_64 linux
Parsing wrong_min.c
Converting
Type-checking wrong_min
Generating GOTO Program
Adding CPROVER library (x86_64)
Removal of function pointers and virtual functions
Generic Property Instrumentation
Running with 8 object bits, 56 offset bits (default)
Starting Bounded Model Checking
size of program expression: 48 steps
simple slicing removed 2 assignments
Generated 2 VCC(s), 2 remaining after simplification
Passing problem to propositional reduction
converting SSA
Running propositional reduction
Post-processing
Solving with MiniSAT 2.2.1 with simplifier
403 variables, 1026 clauses
SAT checker: instance is SATISFIABLE
Solving with MiniSAT 2.2.1 with simplifier
403 variables, 0 clauses
SAT checker inconsistent: instance is UNSATISFIABLE
Runtime decision procedure: 0.00148068s

** Results:
[check_min.assertion.1] assertion r == x || r == y: SUCCESS
[check_min.assertion.2] assertion r <= x && r <= y: FAILURE

** 1 of 2 failed (2 iterations)
VERIFICATION FAILED
$ cbmc --trace --function check_min wrong_min.c
...
[check_min.assertion.2] assertion r <= x && r <= y: FAILURE
Trace for check_min.assertion.2:
...
State 20 file wrong_min.c line 15 function check_min thread 0
-----------------------------------------------
x=1073741824 (01000000 00000000 00000000 00000000)
State 21 file wrong_min.c line 16 function check_min thread 0
-----------------------------------------------
y=1 (00000000 00000000 00000000 00000001)
State 24 file wrong_min.c line 18 function check_min thread 0
-----------------------------------------------
a=1073741824 (01000000 00000000 00000000 00000000)
State 25 file wrong_min.c line 18 function check_min thread 0
-----------------------------------------------
b=1 (00000000 00000000 00000000 00000001)
State 30 file wrong_min.c line 18 function check_min thread 0
-----------------------------------------------
r=1073741824 (01000000 00000000 00000000 00000000)

Violated property:
file wrong_min.c line 21 function check_min
assertion r <= x && r <= y
r <= x && r <= y
Generating test cases

$ cbmc --cover branch --function min wrong_min_2.c
CBMC version 5.10 (cbmc-5.10) 64-bit x86_64 linux
...
converting SSA
Aiming to cover 3 goal(s)
Running propositional reduction
Post-processing
Solving with MiniSAT 2.2.1 with simplifier
334 variables, 569 clauses
SAT checker: instance is SATISFIABLE
Covered function min entry point
Covered function min block 1 branch false
Solving with MiniSAT 2.2.1 with simplifier
334 variables, 0 clauses
SAT checker: instance is SATISFIABLE
Covered function min block 1 branch true
Runtime decision procedure: 0.00146064s

** coverage results:
[min.coverage.1] file wrong_min_2.c line 6 function min entry point: SATISFIED
[min.coverage.2] file wrong_min_2.c line 6 function min block 1 branch false: SATISFIED
[min.coverage.3] file wrong_min_2.c line 6 function min block 1 branch true: SATISFIED

** 3 of 3 covered (100.0%)
** Used 2 iterations
Test suite:
  a=0, b=1
  a=1, b=0
Outline

A Tour of CBMC

Some (unneeded?) History

Model Checking

Liveness Properties

Postscript
The Plan

- Compiler Assignment
- Automatically Grade Assignments
- No two assignments produce identical code
- No two assignments produce identical code
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
• 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
Computing the minimum of three numbers

```c
int min_of_3_before(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;
}
```
Another implementation

```c
int min_of_3_after(int x, int y, int z) {
    int min3;

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

    return min3;
}
```
void check_eqv() {
    int a, b, c;

    a = nondet_int();
    b = nondet_int();
    c = nondet_int();

    assert(min_of_3_before(a, b, c) == min_of_3_after(a, b, c));
}
$ cbmc --trace --function check_eqv min3.c
CBMC version 5.10 (cbmc-5.10) 64-bit x86_64 linux
...
[check_eqv.assertion.1] assertion return_value_min_of_3_before == return_value_min_of_3_after: FAILURE
State 20 file min3.c line 45 function check_eqv thread 0
-------------------------------------------
a=1 (00000000 00000000 00000000 00000001)
State 21 file min3.c line 46 function check_eqv thread 0
-------------------------------------------
b=1073741824 (01000000 00000000 00000000 00000000)
State 22 file min3.c line 47 function check_eqv thread 0
-------------------------------------------
c=0 (00000000 00000000 00000000 00000000)
...
State 32 file min3.c line 13 function min_of_3_before thread 0
-------------------------------------------
min3=0 (00000000 00000000 00000000 00000000)
...
State 46 file min3.c line 33 function min_of_3_after thread 0
-------------------------------------------
min3=1 (00000000 00000000 00000000 00000001)

Violated property:
   file min3.c line 49 function check_eqv
   assertion return_value_min_of_3_before == return_value_min_of_3_after
   return_value_min_of_3_before == return_value_min_of_3_after

** 1 of 1 failed (1 iteration)
VERIFICATION FAILED
CBMC Architecture

- CBMC translates entire program into a “GOTO” program in SSA form
- It then “executes” every statement in the program
  - Values it does not know about are turned into “symbols” (Symbolic Execution)
- Program is then converted into boolean formulae (CNF)
- The formula is handed off to a SAT solver

Source: Kroening and Tautschnig, CBMC – C Bounded Model Checker (Competition Contribution), TACAS 2014
Loops: Definite Bounds

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

CBMC will unroll loop.
Loops: Symbolic Bounds

```c
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++;
            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

A Tour of CBMC

Some (unneeded?) History

Model Checking

Liveness Properties

Postscript
Basic Ideas

- Formula $\varphi$
  - Correctness (Safety) property
  - Propositional logic
  - Example: the argument to the assert statements
- Interpretation $\mathcal{K}$
  - More on this later
- We ask: $\mathcal{K} \models \varphi$?
  - Is $\varphi$ true in $\mathcal{K}$?
Transition System

\[ T = (Q, I, E, \delta) \]

- set of states \( Q \) (e.g. values of all variables)
- initial states \( I \subseteq Q \)
- action labels \( E \) (e.g. program statements)
- (total) transition relation \( \delta \subseteq Q \times E \times Q \)

- A run of \( T \) is the same as a trace of states
  - \( s_0 e_0 s_1 \ldots \) where \( (s_0, e_0, s_1) \in \delta \), and \( s_0 \in I \)

- A reachable state is a state that exists in some run.
min3 = min_of_3(x, y, z);
assert(min3 == x || min3 == y || min3 == z);
assert(min3 <= x);
assert(min3 <= y)
assert(min3 <= z)

- Let $\mathcal{V}$ be a set of propositions
  - e.g. $\text{min3} \leq x$
  - e.g. $\text{min3} \leq y$
- A Kripke structure $\mathcal{K} = (Q, I, E, \delta, \lambda)$ is a transition system where:
  - $\lambda : Q \rightarrow 2^{\mathcal{V}}$
- $\lambda$ 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 $P \in \mathcal{V}$
Let $p$ be the "must be one of inputs" proposition

Let $q, r, s$ be the $\leq x, \leq y, \leq z$ proposition

(Note: True propositions in internal states not shown)
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
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

A Tour of CBMC

Some (unneeded?) History

Model Checking

Liveness Properties

Postscript
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 $\sigma = q_0q_1\ldots$ be a sequence of states
  - $\sigma_i$ is the state $i$
  - $\sigma|_i$ is the suffix $q_iq_{i+1}\ldots$ of $\sigma$

- Let $\varphi$ be a formula

- $\sigma \models \varphi$ if $\varphi \in \lambda(\sigma_0)$

- $X\varphi$ (also a formula), read as “next $\varphi$”,
  - $\sigma \models X\varphi$ if $\sigma|_1 \models \varphi$

- $\varphi U \psi$ (also a formula), read as “$\varphi$ until $\psi$”
  - $\sigma \models \varphi U \psi$ if and only if there exists $k \in \mathbb{N}$
    - $\sigma|_k \models \psi$
    - for all $1 \leq i < k$, $\sigma|_i \models \varphi$
  - Note: $\varphi$ can continue to hold after $k$
More temporal properties

- $F \varphi$, “eventually $\varphi$”
  - $true U \varphi$
- $G \varphi$, “always $\varphi$”
  - $\neg F \neg \varphi$
- $\varphi W \psi$, “$\varphi$ unless $\psi$”
  - $(\varphi U \psi) \lor G \psi$
- $GF \varphi$
- $FG \varphi$
Some examples of invariants

- $G \neg (own_1 \land own_2)$
  - 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.
● Branching time logic for properties of systems
  ● Computation Tree Logic (CTL)
● $\text{EX} \varphi$, there exists a transition where $\varphi$ holds from current state
● $\text{EG} \varphi$, exists a path from current state where $\varphi$ holds on all states
● $\text{EU}$, exists a path until...
● Also $\text{Ax}$ 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

- \(\omega\)-automaton
  - run on infinite strings
- strings represent state sequences (actually \(\lambda(q_0)\lambda(q_1)\ldots\))
- non-deterministic as well as deterministic
  - but non-deterministic Büchi automata more powerful
Büchi-automata have a very close relation to logic.
Outline

A Tour of CBMC

Some (unneeded?) History

Model Checking

Liveness Properties

Postscript
Further Reading and Links

- Stephan Merz, An Introduction to Model Checking
  - Accessible and good introduction, with links to other material
- Javier Esparza, Automata Theory: An algorithmic approach
  - See Chapters 8, 9 and 14
- 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