Formal Methods and Specification (LS 2021)
Lecture 13:
Bounded Software Model Checking

Stefan Ratschan

Katedra číslicového návrhu
Fakulta informačních technologií
České vysoké učení technické v Praze

Evropský sociální fond Praha & EU: Investujeme do vaší budoucnosti

Today

Method for automatic program correctness proofs that

- is used by Amazon Web Services to check memory safety of boot code at data centers [Cook et al., 2020]
- found bugs in a basic software package necessary for the internet (Internet Systems Consortium) [Cho et al., 2013]
- ...

Recapitulation: Operational Semantics

1: $x \leftarrow 2x$
2: **goto** 1

$$\{pc \mapsto 1, x \mapsto 1\} \to_P$$
$$\{pc \mapsto 2, x \mapsto 2\} \to_P$$
$$\{pc \mapsto 1, x \mapsto 2\} \to_P$$
$$\{pc \mapsto 2, x \mapsto 4\}$$

A program state is a function that

- assigns to the special variable pc a line number, and
- to each program variable a value of corresponding type.

Set of all states S.

For a certain program P, for states $s, s' \in S$, $s \to_P s'$ iff
the program can do a step from state s to state s' (*transition relation*)

Corresponding first-order formula: Φ_P

$s \to_P^* s'$: program can do a sequence of steps from s to s'

$[\![P]\!](s, s')$: program can do a sequence of steps from s to s' that ends in s'

Program Correctness

A program execution is *regular* iff
it satisfies all **assume**-assertions.

A program is *(partially) correct* iff
every regular program execution
from the first program line to a program line with an @-assertion,
satisfies the final @-assertion.

Definitions not formal: "program execution" undefined

But in the meanwhile we introduced this definition!

Program Correctness vs. Program States

1: $\quad$ **assume** ϕ_I

$\quad\quad\ldots$

$\quad\quad$ **@** $i \neq 0$

4223: $\quad q \leftarrow 1000/i$

$\quad\quad\ldots$

$\quad\quad$ **@** $l < 100$

7423: $\quad m \leftarrow a[l]$

$\quad\quad\ldots$

$\quad\quad$ **@** ϕ_O

8231: $\quad$ **return**

Correctness:

For every state s, s', with

- $pc(s) = 1$,
- s satisfies the **assume**-assertion on line 1,
- $s \rightarrow_P^* s'$, and
- $pc(s')$ is a line with an @ assertion,

s' satisfies the @ assertion.

Requirements on s: $\quad s \models pc = 1 \wedge \phi_I$

Requirements on s':

$$s' \models \begin{array}{l} pc = 4223 \Rightarrow i \neq 0 \wedge \\ pc = 7423 \Rightarrow l < 100 \wedge \\ pc = 8231 \Rightarrow \phi_O \end{array}$$

Program Correctness Based on Operational Semantics

A program is *(partially) correct* iff
every regular program execution
from the first program line to a program line with an @-assertion,
satisfies the final @-assertion.

Assumption: program has one statement **assume** ϕ_I,
on the first program line.

A program P is *correct* iff
for all states s s.t. $s \models I$,
for all states s' s.t. $s \rightarrow_P^* s'$,
$s' \models O$

where $I \equiv pc = 1 \wedge \phi_I$, and $O \equiv \bigwedge_{l \in L} pc = l \Rightarrow \phi_l$, where L is the set of all program lines with @-assertions and ϕ_l the formula on the corresponding line.

Now everything relates to program states, but
$\rightarrow_P^*$ still no first-order formula.

Program Correctness as a First-Order Formula

A program P is correct iff
for all states s s.t. $s \models I$,
for all states s' s.t. $s \rightarrow_P^* s'$,
$s' \models O$

$$\forall x, x' \,.\, [I(x) \wedge R(x, x')] \Rightarrow O(x')$$

is equivalent to

$$\neg \exists x, x' \,.\, [I(x) \wedge R(x, x') \wedge \neg O(x')]$$

which holds iff

$$I(x) \wedge R(x, x') \wedge \neg O(x') \text{ is not satisfiable}$$

Intuition: no bug

Today's Example

- $I(pc, x) \dot{\Leftrightarrow} pc = 1 \wedge x \geq 0$
- $O(pc, x) \dot{\Leftrightarrow} x \geq 0$

1: $x \leftarrow 2x$
2: **goto** 1

The method works for much bigger examples

But: Resulting formulas much too big for human usage

... fine for machine processing

$s \rightarrow_P s'$ iff

$$s \sqcup \pi(s') \models \begin{array}{l} [pc = 1 \wedge pc' = pc + 1 \wedge x' = 2x] \vee \\ [pc = 2 \wedge pc' = 1 \wedge x' = x] \end{array}$$

(transition constraint Φ_P)

Program Correctness as a First-Order Formula

In general

$$\neg\exists x, x' \,.\, [I(x) \wedge R(x, x') \wedge \neg O(x')]$$

Example specification:

- $I(pc, x) \dot{\Leftrightarrow} pc = 1 \wedge x \geq 0$
- $O(pc, x) \dot{\Leftrightarrow} x \geq 0$

Correctness in one step:

$$\begin{aligned}\neg\exists pc, x, pc', x' \,.\, [pc = 1 \wedge x \geq 0 \wedge \\ & [pc = 1 \wedge pc' = pc + 1 \wedge x' = 2x] \vee \\ & [pc = 2 \wedge pc' = 1 \wedge x' = x] \\ & \qquad\qquad\qquad \wedge \neg x' \geq 0]\end{aligned}$$

Demo

Two Steps

$$
\begin{aligned}
\neg\exists pc_1, x_1, pc_2, x_2, pc_3, x_3 \,.\, [pc_1 = 1 \wedge x_1 \geq 0 \wedge \\
&[pc_1 = 1 \wedge pc_2 = pc_1 + 1 \wedge x_2 = 2x_1] \vee \\
&[pc_1 = 2 \wedge pc_2 = 1 \wedge x_2 = x_1] \\
&\qquad\qquad \wedge \\
&[pc_2 = 1 \wedge pc_3 = pc_2 + 1 \wedge x_3 = 2x_2] \vee \\
&[pc_2 = 2 \wedge pc_3 = 1 \wedge x_3 = x_2] \\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge \neg x_3 \geq 0]
\end{aligned}
$$

Demo

Bounded Program Correctness

$$
\begin{aligned}
\neg\exists pc_1, x_1, \dots, pc_n, x_n \; . \\
pc_1 = 1 \wedge x_1 \geq 0 \wedge \\
&[pc_1 = 1 \wedge pc_2 = 2 \wedge x_2 = 2x_1] \vee \\
&[pc_1 = 2 \wedge pc_2 = 1 \wedge x_2 = x_1] \\
&\qquad \wedge \cdots \wedge \\
&[pc_{n-1} = 1 \wedge pc_n = 2 \wedge x_n = 2x_{n-1}] \vee \\
&[pc_{n-1} = 2 \wedge pc_n = 1 \wedge x_n = x_{n-1}] \\
&\qquad\qquad\qquad\qquad \wedge \neg x_n \geq 0
\end{aligned}
$$

For an arbitrary, but fixed n

We can use corresponding solvers.

Solvers often are not able to handle quantifiers:
verify unsatisfiability of the formula below the quantifier.

Bounded Program Correctness

In general: $BMC_{I,O,P}(n)$ ("bounded model checking") :=

$$\neg\exists v_1, \dots, v_n \,.\, I[v \leftarrow v_1] \wedge \bigwedge_{i=1,\dots,n-1} \Phi_P[v \leftarrow v_i, v' \leftarrow v_{i+1}] \wedge \neg O[v \leftarrow v_n]$$

where v is a placeholder for all program variables (including pc),
with respective indices, primes.

If we only use data types from decidable theories
we can check this formula automatically.

A sequence of states $s_1, \dots, s_n$ s.t. $\pi^1(s_1) \sqcup \dots \sqcup \pi^n(s_n) \models$

$$I[v \leftarrow v_1] \wedge \bigwedge_{i=1,\dots,n-1} \Phi_P[v \leftarrow v_i, v' \leftarrow v_{i+1}] \wedge \neg O[v \leftarrow v_n]$$

where for a state s, $\pi^i(s)$ is a function that
assigns the same values to variables with index i,

is called *counter-example*, *error trace*

Example:

Sequence of states $s_1, \dots, s_n$ s.t. $\pi^1(s_1) \sqcup \dots \sqcup \pi^n(s_n) \models$

$$I[v \leftarrow v_1] \wedge \bigwedge_{i=1,\dots,n-1} \Phi_P[v \leftarrow v_i, v' \leftarrow v_{i+1}] \wedge \neg O[v \leftarrow v_n]$$

1: $x \leftarrow -1$
2: **input** x
3: **while** $x \geq 0$ **do**
4: $\quad x \leftarrow 2x$

$I(pc, x) :\Leftrightarrow pc = 1$
$O(pc, x) :\Leftrightarrow pc = 4 \Rightarrow x \leq 10$

Counter-example to $BMC_{I,O,P}(6)$:

$\{pc \mapsto 1; x \mapsto 10;\}$
$\{pc \mapsto 2; x \mapsto -1;\}$
$\{pc \mapsto 3; x \mapsto 723;\}$
$\{pc \mapsto 4; x \mapsto 723;\}$
$\{pc \mapsto 3; x \mapsto 1446;\}$
$\{pc \mapsto 4; x \mapsto 1446;\}$

$$\pi^1(s_1) \sqcup \dots \sqcup \pi^6(s_6) =$$
$$\{pc_1 \mapsto 1; x_1 \mapsto 10; pc_2 \mapsto 2; x_2 \mapsto -1; pc_3 \mapsto 3; x_3 \mapsto 723; \dots\}$$

Bounded Verification

Correctness within $1, 2, \ldots$ steps:

$$\begin{array}{c} \models BMC(1) \\ \models BMC(2) \\ \models BMC(3) \\ \ldots \end{array}$$

Attention: $BMC(i+1)$ does not necessarily imply $BMC(i)$, $i = 1, \ldots$

For simplicity, slightly different definition than in MI-TES

CBMC Demo

`http://www.cprover.org/cbmc/`

`cbmc demo.c --bounds-check`

`cbmc no_bound.c --bounds-check`

`cbmc bubble_sort.c --bounds-check`

there is k s.t.

$$\bigcup_{i \in \{0,\ldots,k\}} \rightarrow_P^i \; = \; \rightarrow_P^*$$

`cbmc --help`

Does not explicitely generate whole BMC formula

Original article [Clarke et al., 2004]

Today: more and more usage of BMC in industry:
`http://www.btc-es.de`

Bounded vs. Unbounded Program Correctness

A program P is correct iff
for all states s s.t. $s \models I$,
for all states s' s.t. $s \rightarrow_P^* s'$,
$s' \models O$

$BMC(n+1)$ checks instead:
for all states s s.t. $s \models I$,
for all states s' s.t. $s \rightarrow_P^n s'$,
$s' \models O$

Difference? fixed n (we have to state it before)

Attention: $\rightarrow_P^n \neq \rightarrow_P^*$! But: $\rightarrow_P^n \subseteq \rightarrow_P^*$

Hence:
$\models BMC_{I,O,P}(n)$ does not imply that program P is correct wrt. I/O.

But: $\not\models BMC_{I,O,P}(n)$ implies that P has a bug wrt. I/O.

Consequences

Counter-examples always have a certain length

So we can always find them, if we have enough time:
for every program P not fulfilling specification I, O
there is a k s.t. $\neg BMC_{P,I,O}(k)$

Hence: Finding errors
in programs with data types in decidable theories
is semi-decidable.

In practice: Programs can do a huge number of steps,
and so we have to check $BMC_{I,O,P}(n)$ for huge n.

But: in certain applications that usually does not happen

For examples: embedded systems:
Reaction to a certain event may take only a short amount of time

Application: Equivalence Checking

function foo(x)
function foo_optimized(x)

input x
assert foo(x)=foo_optimized(x)

Demo: `cbmc equiv.c`

Further Application: Combination with Testing

For software in safety critical applications,
there are standards that require completeness of tests
according to a certain criteria.

Usually those criteria require that tests the cover program code
in a certain sense (*coverage criteria*).

For example: Tests have to execute each program line at least once.

Problem: How to find a test that executes line l?

Check $BMC_{I,O,P}(1)$, $BMC_{I,O,P}(2)$, $\ldots$ for $O :\Leftrightarrow pc \neq l$.

European Train Control System (ETCS) [Angeletti et al., 2010]

Further Application: Error Removal

Often we know the bug, but we do not know the reason for it.

For example: Can $x \leq 12$ in line 2643 imply division by zero in line 752?

Check $BMC_{I,O,P}(1)$, $BMC_{I,O,P}(2)$, $\ldots$ for

- $I :\Leftrightarrow pc = 2643 \wedge x \leq 12$,
- $O :\Leftrightarrow pc = 752 \Rightarrow y \neq 0$.

Still: $x \leq 12$ in line 2643 not necessarily reachable from an initial state.

Conclusion

BMC can prove program correctness
within a bounded number of steps.

Under the condition that all data structures are in decidable theories
(e.g., linear integer arithmetic)

Industrial tool: BTC EmbeddedTester:
`https://www.btc-es.de/en/` [Schrammel et al., 2017]

Literature I

Damiano Angeletti, Enrico Giunchiglia, Massimo Narizzano, Alessandra Puddu, and Salvatore Sabina. Using bounded model checking for coverage analysis of safety-critical software in an industrial setting. *Journal of Automated Reasoning*, 45:397–414, 2010. ISSN 0168-7433.

Chia Yuan Cho, Vijay D'Silva, and Dawn Song. Blitz: Compositional bounded model checking for real-world programs. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 136–146. IEEE, 2013.

Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004. ISBN 3-540-21299-X.

Literature II

Byron Cook, Kareem Khazem, Daniel Kroening, Serdar Tasiran, Michael Tautschnig, and Mark R Tuttle. Model checking boot code from aws data centers. *Formal Methods in System Design*, pages 1–19, 2020.

Peter Schrammel, Daniel Kroening, Martin Brain, Ruben Martins, Tino Teige, and Tom Bienmüller. Incremental bounded model checking for embedded software. *Formal Aspects of Computing*, 29(5):911–931, 2017.