8. BOUNDED, INFINITE-STATE MC, COMPOSITIONAL REASONING

Jan Kofroň
TODAY

- Bounded model checking
- Infinite-state model checking
- Compositional reasoning
Part I: Bounded Model Checking
BOUNDED MODEL CHECKING

System model

AG (start → AF heat)

Property specification

Property satisfied

Property violated
BOUNDED MODEL CHECKING

System model

AG (start → AF heat)

Property specification

Property satisfied

Property violated
BOUNDDED MODEL CHECKING

Let $M = \{S, I, R, L\}$ be Kripke structure

Define predicate $\text{Reach}(s, s') \equiv R(s, s')$

$\llbracket M \rrbracket^k = \bigwedge_{i=0}^{k-1} \text{Reach}(s_i, s_{i+1})$

$\llbracket M \rrbracket^k$ contains states reachable in exactly $k$ steps

Then search for counterexamples formed by $k$ states
Input: $M$, $\neg \varphi$

1. $k = 0$
2. Is $\neg \varphi$ satisfiable in $[M]^k$?
   - YES: $M \models \neg \varphi$, terminate
3. Is $k < \text{threshold}$?
   - NO: $M \not\models_k \neg \varphi$, terminate
4. Increment $k$
5. Go to 2.
Realized by constructing formula capturing transitions in program

- trying to reach assertion violation, i.e., violation of formula $\text{AG}(p)$
- checking for its satisfiability using SAT/SMT solver
  
  - SAT/SMT solvers – tools for deciding satisfiability of logical formulae
  - satisfying assignment of formulae containing negated property corresponds to counter-example
  - NP-complete problem – the hard part of verification
BMC – Example

1: int i = 4;
2: int s = 0;
3: while (1) {
4:   s += i;
5: if (i > 0)
6:   i --;
7: assert(s < 10);
8: }

Jan Kofroň: Behaviour Models and Verification
First unwind loops up to bound (k).

1: int i = 4;
2: int s = 0;
3: while (1) {
4:     s += i;
5:     if (i > 0)
6:         i --;
7:     assert(s < 10);
8: }

1: int i = 4;
2: int s = 0;
3: 
4:     s += i;
5:     if (i > 0)
6:         i --;
7:     assert(s < 10);
8:     s += i;
9:     if (i > 0)
10:        i --;
11:    assert(s < 10);
...
1: int i = 4;
2: int s = 0;
3: 
4: s += i;
5: if (i > 0)
6:  i --;
7: assert(s < 10);
8: s += i;
9: if (i > 0)
10:  i --;
11: assert(s < 10);
Transform each line of code into (CNF) formula.

1: \( \text{int } i = 4; \)
\[ f_1: (pc_1 = 1) \land (i_2 = 4) \land (pc_2 = 2) \]

2: \( \text{int } s = 0; \)
\[ f_2: (pc_2 = 2) \land (i_3 = i_2) \land (s_3 = 0) \land (pc_3 = 3) \]

3: 
\[ f_3: (pc_3 = 3) \land (i_4 = i_3) \land (s_4 = s_3) \land (pc_4 = 4) \]

4: \( s += i; \)
\[ f_4: (pc_4 = 4) \land (i_5 = i_4) \land (s_5 = s_4 + i_4) \land (pc_5 = 5) \]

5: \( \text{if } (i > 0) \)
\[ f_5: (pc_5 = 5) \land (i_6 = i_5) \land (s_6 = s_5) \land (pc_6 = 6) \]

6: \( i --; \)
\[ f_6: (pc_6 = 6) \land (((i_6 > 0) \land (i_7 = i_6 - 1)) \lor ((i_6 \leq 0) \land (i_7 = i_6))) \land (s_7 = s_6) \land (pc_7 = 7) \]

7: \( \text{assert}(s < 10); \)
\[ f_7: (pc_7 = 7) \land (s_7 \geq 10) \land (pc_8 = 8) \]

8: \( s += i; \)
\[ f_8: (pc_8 = 8) \land (i_9 = i_8) \land (s_9 = s_8 + i_8) \land (pc_9 = 9) \]

9: \( \text{if } (i > 0) \)
\[ f_9: (pc_9 = 9) \land (i_10 = i_9) \land (s_10 = s_9) \land (pc_{10} = 10) \]

10: \( i --; \)
\[ f_{10}: (pc_{10} = 10) \land (((i_{10} > 0) \land (i_{11} = i_{10} - 1)) \lor ((i_{10} \leq 0) \land (i_{11} = i_{10}))) \land (s_{11} = s_{10}) \land (pc_{11} = 11) \]

11: \( \text{assert}(s < 10); \)
\[ f_{11}: (pc_{11} = 11) \land (s_{11} \geq 10) \land (pc_{12} = 12) \]
Transform each line of code into (CNF) formula.

1: `int i = 4;`  
   \[ f_1 : (pc_1 = 1) \land (i_2 = 4) \land (pc_2 = 2) \]

2: `int s = 0;`  
   \[ f_2 : (pc_2 = 2) \land (i_3 = i_2) \land (s_3 = 0) \land (pc_3 = 3) \]

3:  
   \[ f_3 : (pc_3 = 3) \land (i_4 = i_3) \land (s_4 = s_3) \land (pc_4 = 4) \]

4: `s += i;`  
   \[ f_4 : (pc_4 = 4) \land (i_5 = i_4) \land (s_5 = s_4 + i_4) \land (pc_5 = 5) \]

5: `if (i > 0)`  
   \[ f_5 : (pc_5 = 5) \land (i_6 = i_5) \land (s_6 = s_5) \land (pc_6 = 6) \]

6: `i--;`  
   \[ f_6 : (pc_6 = 6) \land (((i_6 > 0) \land (i_7 = i_6 - 1)) \lor ((i_6 \leq 0) \land (i_7 = i_6))) \land (s_7 = s_6) \land (pc_7 = 7) \]

7: `assert(s < 10);`  
   \[ f_7 : (pc_7 = 7) \land (s_7 \geq 10) \land (pc_8 = 8) \]

8: `s += i;`  
   \[ f_8 : (pc_8 = 8) \land (i_9 = i_8) \land (s_9 = s_8 + i_8) \land (pc_9 = 9) \]

9: `if (i > 0)`  
   \[ f_9 : (pc_9 = 9) \land (i_{10} = i_9) \land (s_{10} = s_9) \land (pc_{10} = 10) \]

10: `i--;`  
   \[ f_{10} : (pc_{10} = 10) \land (((i_{10} > 0) \land (i_{11} = i_{10} - 1)) \lor ((i_{10} \leq 0) \land (i_{11} = i_{10}))) \land (s_{11} = s_{10}) \land (pc_{11} = 11) \]

11: `assert(s < 10);`  
   \[ f_{11} : (pc_{11} = 11) \land (s_{11} \geq 10) \land (pc_{12} = 12) \]
Assertion expressions are negated – we are searching for violations

Formula to be checked for satisfiability: \( f = \bigwedge_{i=0..k} f_i \)

Found satisfying assignment correspond to violation of original formula

If \( f \) is unsatisfiable, there is no violation in \( k \) steps
When applied on software, BMC itself cannot prove general absence of assertion violations

- it is useful to discover them
- there are extensions to BMC (unbounded model checking) aiming at proving absence of violations

When applied on pieces of hardware, it can prove their absence

- number of steps (its upper bound) of particular operations is known
Bounds can be useful – finding shortest counter-examples

By including loop invariants (which are difficult to compute, though) into BMC, infinite paths can be verified
Part II: Infinite-State Model Checking
Finite models are sometimes insufficient

- Protocols and circuits specification can be parametrized by size of int type (CPU), number of processors in multicore environment, of communicating network nodes, ...

- Even though model checking of general infinite-state models is impossible, special cases can be model-checked
Infinite family of systems: $\mathcal{F} = \{M_i\}_{i=1}^{\infty}$

Verification task: assume $f$ to be temporal formula, verify: $\forall i : M_i \models f$

Generally, this is still undecidable – we have to add more assumptions later

Indexed CTL (ICTL) – formula for each system component
- $i$-th formula applied onto $i$-th component
- allows for special expressions: $\land_i f(i)$, $\lor_i f(i)$, $\land_{j \neq i} f(j)$, and $\lor_{j \neq i} f(j)$
Simple token ring

- atomic propositions:
  - non-critical section, keeping token, critical section, receive token, send token

One process Q originally keeping token (t), several processes $P_i$ originally in state n
Synchronous composition $Q \parallel P$ with natural synchronization of $s$ and $r$
Synchronous composition $Q \parallel P$ with natural synchronization of $s$ and $r$

Generally, token ring family: $\mathcal{F} = \{Q \parallel P_i\}_{i=1}^\infty$, desired property: $\bigwedge_i \text{AG} (c_i \implies \bigwedge_{j \neq i} \neg c_j)$
How to prove the property when there are infinitely many $P$ processes?
We have to find generalizing structure – *invariant*:

- Let $\mathcal{F} = \{Q || P_i\}_{i=1}^{\infty}$ be family of structures
- Let $\geq$ be reflexive, transitive relation on structures
- *Invariant* $I$ is structure such that $\forall i : I \geq M_i$
- Relation $\geq$ determine properties that can be checked:
  - $\geq$ is bisimulation $\implies$ strong preservation: $I \models f \iff M \models f$
  - $\geq$ is simulation preorder $\implies$ weak preservation: $I \models f \implies M \models f$
  - Similarly for language-level preorder and equivalence

**Token ring example:** Token rings of size $n$ and 2 are in simulation preorder $\implies$
sufficient to verify just whether $(P || Q) \models f$
INFINITE FAMILIES – TOKEN RING EXAMPLE

(t, n) ↦ (t, n, n)
(c, n) ↦ (c, n, n)
(n, t) ↦ (n, t, n)
(n, t) ↦ (n, n, t)
(n, c) ↦ (n, c, n)
(n, c) ↦ (n, n, c)
SYSTEMATIC APPROACH TO FINDING INVARIANTS

**Definition:** Composition $\parallel$ is monotonic w.r.t. relation $\geq$ $\iff$

$$\forall P_1, P_1', P_2, P_2' : P_1 \geq P_1' \land P_2 \geq P_2' \implies P_1 \parallel P_2 \geq P_1' \parallel P_2'$$

**Lemma:** Let $\geq$ be a reflexive, transitive relation and let $\parallel$ be a composition operator that is monotonic w.r.t. $\geq$. If $I \geq P$ and $I \geq I \parallel P$, then $\forall i : I \geq F_i$, where $F = \{P^i\}_{i=1}^{\infty}$.

This is more like:

“\(\text{This holds once we have the relation}\)” than “\(\text{How to find the relation}\)”

Finding suitable relation is hard and not possible in algorithmic way – problem is undecidable in general.
Part III: Compositional Reasoning
Efficient verification algorithms can extend applicability of formal methods

Many systems can be decomposed into parts
- verifying properties of each part separately
- if conjunction of parts properties implies overall specification, we are done
- the entire system never analysed as whole
Three communication-protocol actors: sender, network, receiver

Overall specification:
- Data correctly transmitted from sender to receiver

Partial specifications:
- Data correctly sent from sender to network
- Data correctly transmitted via network
- Data correctly transmitted from network to receiver

Verification of partial specifications typically much easier
- sum of state spaces much smaller than state space of entire system (impact of state space explosion mitigated)
ASSUME-GUARANTEE PRINCIPLE

- Verifies each component separately

- Based on specification of
  - **Assumptions** – requirements on behaviour of environment
  - **Guarantees** – provisions offered to environment if assumptions are met
    - environment = the other components

- By combining assumptions and guarantees of particular parts, it is possible to establish correctness of entire system

- Full transition graph never constructed
**Assume-Guarantee Formally**

- Formula capturing assume-guarantee principle is triple $\langle g \rangle M \langle f \rangle$ where $g, f$ are temporal formulae and $M$ is program
  - whenever $M$ is part of system satisfying $g$, system also guarantees $f$
- Composition of proofs: $(\langle g \rangle M' \langle f \rangle) \land (\langle true \rangle M \langle g \rangle) \implies \langle true \rangle M | | M' \langle f \rangle$
- Can be expressed as inference rule:

\[
\begin{array}{c}
\langle true \rangle M \langle g \rangle \\
\langle g \rangle M' \langle f \rangle \\
\hline \\
\langle true \rangle M | | M' \langle f \rangle
\end{array}
\]
ASSUME-GUARANTEE FORMALLY

Necessary to avoid circular dependencies making reasoning **unsound**:

\[
\frac{\langle f \rangle M \langle g \rangle \\
\langle g \rangle M' \langle f \rangle}{M | | M' |= f \land g}
\]

Again: This is not incorrect!
Each component specifies not only provided (implemented) interfaces
  similarly as objects do
But also required ones
  in addition to objects
Syntactic (type) information may or may not consider interface/type inheritance
Semantic (behaviour) specification – usage protocols, restrictions beyond language capabilities, ...
  can cover various aspects of component functional and extra-functional properties:
  allowed sequences of messages/calls, timing, reliability, resource usage, security, ...
  composability verification based on the same principle as syntax: each component should provide at least as much (as good, fast, reliable, ...) as its environment requires
ASSUME-GUARANTEE – APPLICATION TO CODE

- Syntax – usually checked by compiler and no additional effort required
- Semantics – code annotations (code contracts):
  - at level of functions/methods
  - assumptions – preconditions
  - guarantees – postconditions
  - usually also invariants – loop invariants

Verification is modular:
- each function is verified separately – whether execution of each function really guarantees its postcondition if precondition is satisfied upon function entry
- if function is called from within another function, its contract is used
  - precondition checked
  - postcondition is assumed
It is not easy to specify contracts:
- too weak preconditions make it difficult to guarantee postconditions
- too strong preconditions are hard to be satisfied by callers
- too strong postconditions are hard to be proven
- too weak postconditions usually do not “satisfy” callers

One has to know and tune...

There are approaches for real programming languages
- Spec#, JML, Code Contracts, Nagini, Dafny...
- backed by verification tools – model checkers, SAT/SMT solvers, theorem provers