NSWI101: System Behaviour Models and Verification

8. Bounded, Infinite-State MC, Compositional Reasoning

Jan Kofroň

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

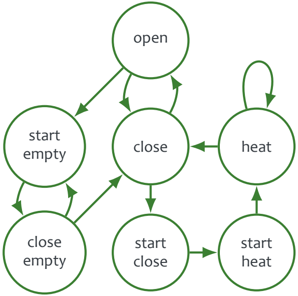
Department of
Distributed and
Dependable
Systems

Today

- Bounded model checking
- Infinite-state model checking
- Compositional reasoning

Part I: Bounded Model Checking

BOUNDED MODEL CHECKING



System model

$$\textbf{AG (start} \rightarrow \textbf{AF heat)}$$

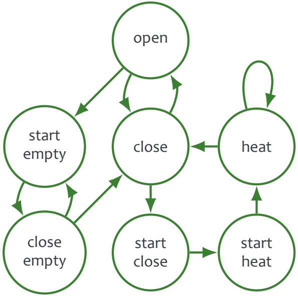
Property specification

Model Checker

Property satisfied

Property violated

Bounded Model Checking



System model

$$\boxed{\textbf{AG (start} \rightarrow \textbf{AF heat)}}$$

Property specification

Model Checker

Property satisfied

Property violated

Bounded Model Checking

- Let $M = \{S, I, R, L\}$ be Kripke structure
- Define predicate $Reach(s, s') \equiv R(s, s')$
- $[\![M]\!]^k = \bigwedge_{i=0}^{k-1} Reach(s_i, s_{i+1})$
- $[\![M]\!]^k$ contains states reachable in exactly k steps
- Then search for counterexamples formed by k states

Bounded Model Checking – Procedure

Input: $M, \neg\varphi$

1. $k = 0$
2. Is $\neg\varphi$ satisfiable in $[\![M]\!]^k$?
   - YES: $M \models \neg\varphi$, terminate
3. Is $k <$ threshold?
   - NO: $M \not\models_k \neg\varphi$, terminate
4. Increment k
5. Go to 2.

Bounded Model Checking For Programs

Realized by constructing formula capturing transitions in program

- trying to reach assertion violation, i.e., violation of formula 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

```
int i=4;
int s=0;
while (1) {
   s+=i;
   if (i>0)
      i--;
   assert(s<10);
}
```

BMC – Example

First unwind loops up to bound (k).

```
int i=4;
int s=0;
while (1) {
   s+=i;
   if (i>0)
      i--;
   assert(s<10);
}
```

```
int i=4;
int s=0;

s+=i;
if (i>0)
   i--;
assert(s<10);
s+=i;
if (i>0)
   i--;
assert(s<10);
...
```

BMC – Example

```
int i=4;
int s=0;

s+=i;
if (i>0)
    i--;

assert(s<10);
s+=i;
if (i>0)
    i--;

assert(s<10);
```

BMC – EXAMPLE

Transform each line of code into (CNF) formula.

```
 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);
```

$f_1 : (pc_1 = 1) \wedge (i_2 = 4) \wedge (pc_2 = 2)$

$f_2 : (pc_2 = 2) \wedge (i_3 = i_2) \wedge (s_3 = 0) \wedge (pc_3 = 3)$

$f_3 : (pc_3 = 3) \wedge (i_4 = i_3) \wedge (s_4 = s_3) \wedge (pc_4 = 4)$

$f_4 : (pc_4 = 4) \wedge (i_5 = i_4) \wedge (s_5 = s_4 + i_4) \wedge (pc_5 = 5)$

$f_5 : (pc_5 = 5) \wedge (i_6 = i_5) \wedge (s_6 = s_5) \wedge (pc_6 = 6)$

$f_6 : (pc_6 = 6) \wedge (((i_6 > 0) \wedge (i_7 = i_6 - 1)) \vee$
$((i_6 \leq 0) \wedge (i_7 = i_6))) \wedge (s_7 = s_6) \wedge (pc_7 = 7)$

$f_7 : (pc_7 = 7) \wedge (s_7 \geq 10) \wedge (pc_8 = 8)$

$f_8 : (pc_8 = 8) \wedge (i_9 = i_8) \wedge (s_9 = s_8 + i_8) \wedge (pc_9 = 9)$

$f_9 : (pc_9 = 9) \wedge (i_{10} = i_9) \wedge (s_{10} = s_9) \wedge (pc_{10} = 10)$

$f_{10} : (pc_{10} = 10) \wedge (((i_{10} > 0) \wedge (i_{11} = i_{10} - 1)) \vee$
$((i_{10} \leq 0) \wedge (i_{11} = i_{10}))) \wedge (s_{11} = s_{10}) \wedge (pc_{11} = 11)$

$f_{11} : (pc_{11} = 11) \wedge (s_{11} \geq 10) \wedge (pc_{12} = 12)$

BMC – Example

Transform each line of code into (CNF) formula.

```
 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);
```

$$f_1 : (pc_1 = 1) \wedge (i_2 = 4) \wedge (pc_2 = 2)$$

$$f_2 : (pc_2 = 2) \wedge (i_3 = i_2) \wedge (s_3 = 0) \wedge (pc_3 = 3)$$

$$f_3 : (pc_3 = 3) \wedge (i_4 = i_3) \wedge (s_4 = s_3) \wedge (pc_4 = 4)$$

$$f_4 : (pc_4 = 4) \wedge (i_5 = i_4) \wedge (s_5 = s_4 + i_4) \wedge (pc_5 = 5)$$

$$f_5 : (pc_5 = 5) \wedge (i_6 = i_5) \wedge (s_6 = s_5) \wedge (pc_6 = 6)$$

$$f_6 : (pc_6 = 6) \wedge (((i_6 > 0) \wedge (i_7 = i_6 - 1)) \vee ((i_6 \leq 0) \wedge (i_7 = i_6))) \wedge (s_7 = s_6) \wedge (pc_7 = 7)$$

$$f_7 : (pc_7 = 7) \wedge (s_7 \geq 10) \wedge (pc_8 = 8)$$

$$f_8 : (pc_8 = 8) \wedge (i_9 = i_8) \wedge (s_9 = s_8 + i_8) \wedge (pc_9 = 9)$$

$$f_9 : (pc_9 = 9) \wedge (i_{10} = i_9) \wedge (s_{10} = s_9) \wedge (pc_{10} = 10)$$

$$f_{10} : (pc_{10} = 10) \wedge (((i_{10} > 0) \wedge (i_{11} = i_{10} - 1)) \vee ((i_{10} \leq 0) \wedge (i_{11} = i_{10}))) \wedge (s_{11} = s_{10}) \wedge (pc_{11} = 11)$$

$$f_{11} : (pc_{11} = 11) \wedge (s_{11} \geq 10) \wedge (pc_{12} = 12)$$

BMC – EXAMPLE

- 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

BMC Applications

- 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

BMC – REMARKS

- 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

MOTIVATION

- 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 Families

- 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

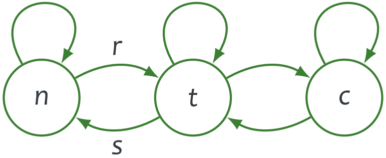
- Indexed CTL (ICTL) – formula for each system component
  - i-th formula applied onto i-th component
  - allows for special expressions: $\wedge_i f(i)$, $\vee_i f(i)$, $\bigwedge_{j \neq i} f(j)$, and $\bigvee_{j \neq i} f(j)$

Infinite Families – Token Ring Example

Simple token ring

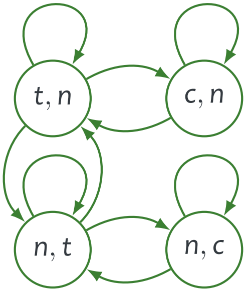
- atomic propositions:
  non-critical section, keeping **t**oken, **c**ritical section, **r**eceive token, **s**end token

One process Q originally keeping token (t), several processes P_i originally in state n



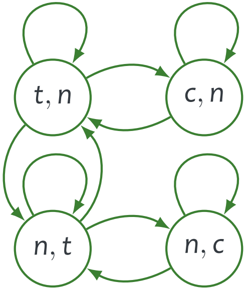
Infinite Families – Token Ring Example

Synchronous composition $Q||P$ with natural synchronization of s and r



Infinite Families – Token Ring Example

Synchronous composition $Q||P$ with natural synchronization of s and r



Generally, token ring family: $\mathcal{F} = \{Q||P_i\}_{i=1}^{\infty}$, desired property: $\bigwedge_i \mathrm{AG}\,(c_i \implies \bigwedge_{j \neq i} \neg c_j)$

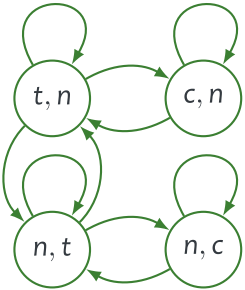
Infinite Families

How to prove the property when there are infinitely many P processes?
We have to find generalizing structure – *invariant*:

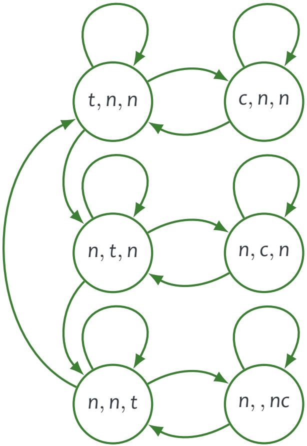
- Let $\mathcal{F} = \{Q||P_i\}_{i=1}^{\infty}$ be family of structures
- Let $\geq$ be reflexive, transitive relation on structures
- *Invariant* I is structure such that $\forall i : I \geq M_i$
- Relation $\geq$ determine properties that can be checked:
  - $\geq$ is bisimulation $\implies$ strong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



$$
\begin{aligned}
(t, n) &\mapsto (t, n, n)\\
(c, n) &\mapsto (c, n, n)\\
(n, t) &\mapsto (n, t, n)\\
(n, t) &\mapsto (n, n, t)\\
(n, c) &\mapsto (n, c, n)\\
(n, c) &\mapsto (n, n, c)
\end{aligned}
$$



Systematic Approach to Finding Invariants

Definition: Composition $||$ is monotonic w.r.t. relation $\geq \Leftrightarrow$

$$\forall P_1, P'_1, P_2, P'_2 : P_1 \geq P'_1 \wedge P_2 \geq P'_2 \implies P_1||P_2 \geq P'_1||P'_2$$

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

This is more like:

“This holds once we have the relation” than “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

Motivation

- 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

Example – Producer-Consumer Model

- 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: $(\langle g\rangle M'\langle f\rangle) \wedge (\langle true\rangle M\langle g\rangle) \implies \langle true\rangle M||M'\langle f\rangle$
- Can be expressed as inference rule:

$$\frac{\begin{array}{c}\langle true\rangle M\langle g\rangle \\ \langle g\rangle M'\langle f\rangle\end{array}}{\langle true\rangle M||M'\langle f\rangle}$$

Assume-Guarantee Formally

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

$$\frac{\begin{array}{c}\langle f \rangle M \langle g \rangle \\ \langle g \rangle M' \langle f \rangle\end{array}}{M || M' \models f \wedge g}$$

Again: This is not incorrect!

Assume-Guarantee – Application to Software Components

- 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

ASSUME-GUARANTEE – REMARKS

- 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