

Bounded Model Checking

Motivation

Abstraction works using overapproximation

The technique considers all real executions plus additional, “spurious” ones

Question: Is a (LTL) property satisfied?

Answer: Yes (if actual system satisfies it), no (if *possibly* violated)

→ answer is precise on the “safe” side

Goal of overapproximation: **verification** (= proof of correctness)

Complementary approach: underapproximation

Considers a subset of the real executions

Answer: no (if error found), “don’t know” (if no error found)

Goal: **falsification** (finding errors, gaining confidence)

akin to testing methods, but more systematic (and costly)

Bounded Model Checking

Example of an underapproximation technique: **Bounded Model Checking** (BMC)

Consider only runs of length k , for some fixed value of k

Introduced by **Biere et al 1999**

Survey: Biere et al, *Bounded Model Checking*. In: *Advances in Computers*, 2003

Overview

Given: (compact description of) Kripke structure \mathcal{K} , LTL formula ϕ , bound $k \geq 0$

Translation of $\neg\phi$ and all paths of length k of \mathcal{K} into a propositional-logic formula F

If F satisfiable, then \mathcal{K} does *not* satisfy ϕ .
(The reverse implication does not hold!)

Use a SAT checker for the latter.

Paths as PL formulae

In the following, let $\mathcal{K} = (S, \rightarrow, r, AP, \nu)$ be a Kripke structure, where S is finite.

Assumption: $S = \{0, 1\}^m$ for some m (wlog.)

Another assumption: \mathcal{K} does not have deadlocks.

Any state from S can be described as a vector \vec{s} of m atomic propositions.

Use vectors $\vec{s}_0, \dots, \vec{s}_k$ to describe paths of length k
($(k + 1) \cdot m$ atomic propositions).

In the following we identify \vec{s}_i with the state s_i it represents.

Let $I(s_0)$ be a formula s.t. $I(s_0)$ is true iff $s_0 = r$.

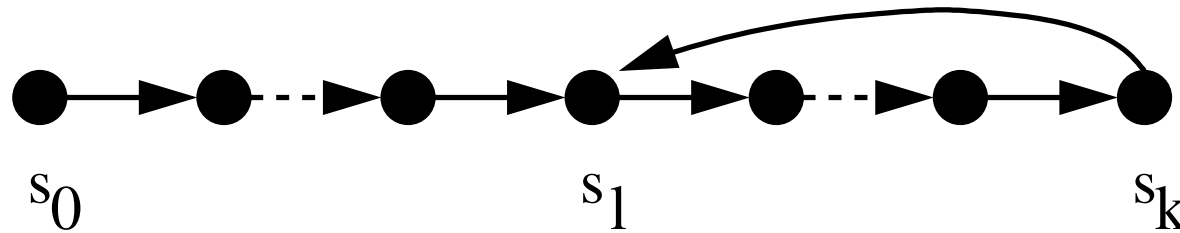
Let $T(s_i, s_j)$ be true iff $s_i \rightarrow s_j$.

Then $F_P := I(s_0) \wedge \bigwedge_{i=1}^k T(s_{i-1}, s_i)$ is true iff the corresponding states form a path of length k .

k -loops

Let $p = s_0, \dots, s_k$ be a path of length k .

Let $0 \leq \ell \leq k$. We call p a (k, ℓ) -loop if $s_k \rightarrow s_\ell$.



We call p a **loop** if p is a (k, ℓ) -loop for some value of ℓ .

p is **non-looping** if p is not a loop.

LTL semantics for loops

Let p be a (k, ℓ) -loop, and let ϕ be an LTL formula in negative normal form (all negations pushed inside).

p represents an infinite path, by repeating the loop over and over:

$$\pi = s_0 \dots s_{\ell-1} (s_\ell \dots s_k)^\omega.$$

We shall write: $p \models^k \phi$ iff $\pi \models \phi$.

Let $\sigma(i) := i + 1$ for $i < k$ and $\sigma(k) := \ell$ (successor position).

Let $0 \leq i, j \leq k$. We define $i \preceq_\ell j$ iff $i \leq j$ or $i, j \geq \ell$.

(Intuition: s_j appears after s_i in π .)

Let $i \leq k$. Observation:

$$\pi^i \models \phi \quad \Leftrightarrow \quad \pi^{i+j \cdot (k-\ell+1)} \models \phi \quad \text{for any } j \geq 0 \text{ and } i \geq \ell$$

$$\pi^i \models a \quad \Leftrightarrow \quad a \in \nu(s_i) \quad \text{for } a \in AP$$

$$\pi^i \models \neg a \quad \Leftrightarrow \quad a \notin \nu(s_i) \quad \text{for } a \in AP$$

$$\pi^i \models \phi \vee \psi \quad \Leftrightarrow \quad \pi^i \models \phi \vee \pi^i \models \psi$$

$$\pi^i \models \phi \wedge \psi \quad \Leftrightarrow \quad \pi^i \models \phi \wedge \pi^i \models \psi$$

$$\pi^i \models \mathbf{X} \phi \quad \Leftrightarrow \quad \pi^{\sigma(i)} \models \phi$$

$$\pi^i \models \phi \mathbf{U} \psi \quad \Leftrightarrow \quad \pi^i \models \psi \vee (\pi^i \models \phi \wedge \pi^{\sigma(i)} \models \phi \mathbf{U} \psi)$$

$$\pi^i \models \phi \mathbf{R} \psi \quad \Leftrightarrow \quad \pi^i \models \phi \wedge \psi \vee (\pi^i \models \psi \wedge \pi^{\sigma(i)} \models \phi \mathbf{R} \psi)$$

Conclusion: $\pi \models \phi$ (and therefore $p \models^k \phi$) depends only on p .

LTL semantics for finite paths

Let p be a path (possibly non-looping) of length k and ϕ an LTL formula in negative normal form.

Since p does not have any deadlocks (by assumption), p can be extended to at least one infinite path π .

Let us define $p \models^k \phi$ in such a manner that $p \models^k \phi$ implies: $\pi \models \phi$ for any infinite extension of p .

The following definition attains this goal (where $i \leq k$):

$$\begin{aligned} p^{k+1} \not\models^k \phi & \quad \text{for any formula } \phi \\ p^i \models^k a & \Leftrightarrow a \in \nu(s_i) \quad \text{for } a \in AP \\ p^i \models^k \neg a & \Leftrightarrow a \notin \nu(s_i) \quad \text{for } a \in AP \\ p^i \models^k \phi \vee \psi & \Leftrightarrow p^i \models^k \phi \vee p^i \models^k \psi \\ p^i \models^k \phi \wedge \psi & \Leftrightarrow p^i \models^k \phi \wedge p^i \models^k \psi \\ p^i \models^k \mathbf{X} \phi & \Leftrightarrow p^{i+1} \models^k \phi \\ p^i \models^k \phi \mathbf{U} \psi & \Leftrightarrow p^i \models^k \psi \vee (p^i \models^k \phi \wedge p^{i+1} \models^k \phi \mathbf{U} \psi) \\ p^i \models^k \phi \mathbf{R} \psi & \Leftrightarrow p^i \models^k \phi \wedge \psi \vee (p^i \models^k \psi \wedge p^{i+1} \models^k \phi \mathbf{R} \psi) \end{aligned}$$

In the above, p^i denotes the sequence $s_i \dots s_k$.

Properties of BMC

Let p be a path of length k and ϕ an LTL formula.

We have: If $p \models^k \neg\phi$, then $\mathcal{K} \not\models \phi$.

Moreover: If $\mathcal{K} \not\models \phi$, then there are $k \geq 0$ and p s.t. $p \models^k \neg\phi$.

Translation into propositional logic

We generate a formula F with the following properties:

F is satisfiable iff there exists p with $p \models^k \neg\phi$.

Observation: Fix a path p . To check whether, ob $p \models^k \neg\phi$, we need to check the subformulae of ϕ only in finitely many places.

Translating a loop

Let $l \leq k$ and p a (k, l) -loop, and let π be the corresponding infinite path.
Let ψ be a subformula of (the NNF of) $\neg\phi$ and $i \leq k$.

We introduce atomic propositions of the form ${}_l\llbracket\psi\rrbracket_k^i$.

${}_l\llbracket\psi\rrbracket_k^i$ should be true iff $\pi^i \models \psi$.

Generate subformulae of F by exploiting the previous observations, e.g.

$$\begin{aligned} {}_l\llbracket\psi_1 \vee \psi_2\rrbracket_k^i &\leftrightarrow {}_l\llbracket\psi_1\rrbracket_k^i \vee {}_l\llbracket\psi_2\rrbracket_k^i \\ {}_l\llbracket\mathbf{X}\psi\rrbracket_k^i &\leftrightarrow {}_l\llbracket\psi\rrbracket_k^{\sigma(i)} \end{aligned}$$

Finally, conjoin all those subformulae.

Attention: For **U**, something special must be done:

$$e[\psi_1 \mathbf{U} \psi_2]_k^i \leftrightarrow \left(e[\psi_2]_k^i \vee (e[\psi_1]_k^i \wedge e[\psi_1 \mathbf{U} \psi_2]_k^{\sigma(i)}) \right) \\ \wedge \bigvee_{i \leq j} e[\psi_2]_k^j$$

Without the last clause it may happen that the right-hand side of a **U**-formula is never satisfied.

By analogy, we introduce atomic propositions $[\psi]_k^i$ for arbitrary paths p .

Overview of the translation

F is defined as follows:

$$F := I(s_0) \wedge \bigwedge_{i=1}^k T(s_{i-1}, s_i) \wedge \left(\llbracket \neg\phi \rrbracket_k^0 \vee \bigvee_{\ell=0}^k (T(s_k, s_\ell) \wedge \ell \llbracket \neg\phi \rrbracket_k^0) \right)$$

If F is satisfiable, then $\mathcal{K} \not\models \phi$.

Remark: size of F is polynomial in $|\mathcal{K}|$, $|\phi|$, and k .

Satisfiability checking for PL formulae

First option: generate a BDD for F , check whether that BDD contains only the 0-node.

Second option: use a dedicated SAT checker.

SAT checkers

Find *some* satisfying valuation, often very quickly.
(BDDs would construct *all* satisfying valuations.)

Enormous progress in this area since about 2000:

Hundreds of thousands of variables and clauses manageable

Examples: zChaff, MiniSAT, ...

Advantages: memory efficiency!

SAT and BMC

Advantages for software model checking: all language features can (in principle) be handled

Usual approach: user fixed a bound for the number of loop unrollings. BMC checker starts with small value of k and tries successively higher values as long as feasible.

In practice still efficient for values up to 80.

Well-known implementation: CBMC (for C programs)

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