Department of Mathematics, Computer Science and Physics, University of Udine

Symbolic Model Checking and Bounded Model Checking

Luca Geatti
luca.geatti@uniud.it
Angelo Montanari
angelo.montanari@uniud.it

May 8th, 2024

SYMBOLIC MODEL CHECKING

Tackling the state-space explosion problem

Symbolic Model Checking

- Classical algorithms for model checking belong to the class of explicit-state model checking algorithms:
  - the Kripke Structure M is represented as a set of memory locations, pointers ecc...
  - states are considered individually

Symbolic Model Checking

- Classical algorithms for model checking belong to the class of explicit-state model checking algorithms:
  - the Kripke Structure M is represented as a set of memory locations, pointers ecc...
  - states are considered individually
- MC suffers from the state-space explosion problem: the number of states of

$$M = M_1 \times M_2 \times \cdots \times M_n$$

is exponential in n.

Symbolic Model Checking

- Classical algorithms for model checking belong to the class of explicit-state model checking algorithms:
  - the Kripke Structure M is represented as a set of memory locations, pointers ecc...
  - states are considered individually
- MC suffers from the state-space explosion problem: the number of states of

$$M = M_1 \times M_2 \times \cdots \times M_n \qquad : \qquad \text{Cartesian product}$$

is exponential in n.

Symbolic Model Checking

- Classical algorithms for model checking belong to the class of explicit-state model checking algorithms:
  - the Kripke Structure M is represented as a set of memory locations, pointers ecc...
  - states are considered individually
- MC suffers from the state-space explosion problem: the number of states of

$$M = M_1 \times M_2 \times \cdots \times M_n \qquad : \qquad \text{Cartesian product}$$

  is exponential in n.
- The exploration of such a a huge state space may be prohibitive even for an algorithm running in linear time in the size of the model.

Symbolic Model Checking

- Classical algorithms for model checking belong to the class of explicit-state model checking algorithms:
  - the Kripke Structure M is represented as a set of memory locations, pointers ecc...
  - states are considered individually
- MC suffers from the state-space explosion problem: the number of states of

$$M = M_1 \times M_2 \times \cdots \times M_n \qquad : \qquad \text{Cartesian product}$$

  is exponential in n.
- The exploration of such a a huge state space may be prohibitive even for an algorithm running in linear time in the size of the model.
- the size of system that could be verified by explicit model checkers was restricted to $\approx 10^6$ states. Solution: Symbolic Model Checking

State-space explosion problem

Citation for the 2007 Turing Award

Although the 1981 paper demonstrated that the model checking was possible in principle, its application to practical systems was severely limited. The most pressing limitation was the number of states to search. Early model checkers required explicitly computing every possible configuration of values the program might assume. For example, if a program counts the millimeters of rain at a weather station each day of the week, it will need 7 storage locations. Each location will have to be big enough to hold the largest rain level expected in a single day. If the highest rain level in a day is 1 meter, this simple program will have 10^{21} possible states, slightly less than the number of stars in the observable universe. Early model checkers would have to verify that the required property was true for every one of those states.

https://amturing.acm.org/award_winners/clarke_1167964.cfm

Symbolic model checking has been proposed by McMillan (1993).

Reference:

Kenneth L McMillan (1993). "Symbolic model checking". In: *Symbolic Model Checking*. Springer, pp. 25–60

It replaces fixed-point computations over individual states by manipulations of definitions of state sets. It allows an exhaustive implicit enumeration of a huge number of states

Main definition: *symbolic* transition system.

Symbolic Transition Systems

Consider a (explicit) Kripke structure $\mathcal{M} = (S, I, T, L)$. We can give a Boolean encoding of it:

Consider a (explicit) Kripke structure $\mathcal{M} = (S, I, T, L)$. We can give a Boolean encoding of it:

- let $\overline{x} := \{x_0, \ldots, x_{n-1}\}$ be a set of state (Boolean) variables;

Consider a (explicit) Kripke structure $\mathcal{M} = (S, I, T, L)$. We can give a Boolean encoding of it:

- let $\overline{x} \coloneqq \{x_0, \ldots, x_{n-1}\}$ be a set of state (Boolean) variables;
- $S = \{0,1\}^n$, *i.e.*, a state is an assignment to all the state variables;
  - a state is a bit vector, *e.g.*, $\langle 0, 1, 1, \ldots, 0 \rangle$
  - with n variables we represent 2^n states

Symbolic Transition Systems

Consider a (explicit) Kripke structure $\mathcal{M} = (S, I, T, L)$. We can give a Boolean encoding of it:

- let $\overline{x} := \{x_0, \ldots, x_{n-1}\}$ be a set of state (Boolean) variables;
- $S = \{0, 1\}^n$, *i.e.*, a state is an assignment to all the state variables;
  - a state is a bit vector, *e.g.*, $\langle 0, 1, 1, \ldots, 0 \rangle$
  - with n variables we represent 2^n states
- $m \models f_I(\overline{x})$ is true iff $m \in I$

Symbolic Transition Systems

Consider a (explicit) Kripke structure $\mathcal{M} = (S, I, T, L)$. We can give a Boolean encoding of it:

- let $\overline{x} := \{x_0, \ldots, x_{n-1}\}$ be a set of state (Boolean) variables;
- $S = \{0,1\}^n$, *i.e.*, a state is an assignment to all the state variables;
  - a state is a bit vector, *e.g.*, $\langle 0, 1, 1, \ldots, 0 \rangle$
  - with n variables we represent 2^n states
- $m \models f_I(\overline{x})$ is true iff $m \in I$
- $m, m' \models f_T(\overline{x}, \overline{x}')$ is true iff $(m, m') \in T$

Consider a (explicit) Kripke structure $\mathcal{M} = (S, I, T, L)$. We can give a Boolean encoding of it:

- let $\overline{x} := \{x_0, \ldots, x_{n-1}\}$ be a set of state (Boolean) variables;
- $S = \{0,1\}^n$, *i.e.*, a state is an assignment to all the state variables;
  - a state is a bit vector, *e.g.*, $\langle 0, 1, 1, \ldots, 0 \rangle$
  - with n variables we represent 2^n states
- $m \models f_I(\overline{x})$ is true iff $m \in I$
- $m, m' \models f_T(\overline{x}, \overline{x}')$ is true iff $(m, m') \in T$
- $m \models f_p(\overline{x})$ is true iff $p \in L(m)$, for all labels $p \in RANGE(L)$

Consider a (explicit) Kripke structure $\mathcal{M} = (S, I, T, L)$. We can give a Boolean encoding of it:

- let $\overline{x} := \{x_0, \ldots, x_{n-1}\}$ be a set of state (Boolean) variables;
- $S = \{0, 1\}^n$, *i.e.*, a state is an assignment to all the state variables;
  - a state is a bit vector, *e.g.*, $\langle 0, 1, 1, \ldots, 0\rangle$
  - with n variables we represent 2^n states
- $m \models f_I(\overline{x})$ is true iff $m \in I$
- $m, m' \models f_T(\overline{x}, \overline{x}')$ is true iff $(m, m') \in T$
- $m \models f_p(\overline{x})$ is true iff $p \in L(m)$, for all labels $p \in RANGE(L)$

The corresponding symbolic Kripke structure is the tuple $(\overline{x}, f_I, f_T, \{f_{p_1}, \ldots, f_{p_k}\})$.

- we will write simply $\mathcal{M} = (S, I, T, L)$, meaning a symbolic transition system
- a path (or trace) $\pi = m_0, m_1, \ldots$ is an infinite sequence of assignment to the state variables such that:
  - $m_0 \models I(\overline{x})$;
  - $m_i, m'_{i+1} \models T(\overline{x}, \overline{x}')$ holds, for all $i \geq 0$.

  where $\overline{x}' := \{x'_0, \ldots, x'_{n-1}\}$.

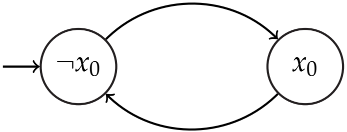
Symbolic Transition Systems

Three main techniques have been proposed:

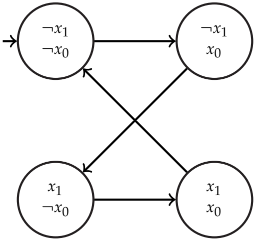
- partial order reduction
- BDD-based symbolic model checking
  - kind of *compressed truth tables*
- SAT-based symbolic model checking, aka *Bounded Model Checking*.

They allowed for the verification of systems with $> 10^{120}$ states.

- substantially larger than the number of atoms in the observable universe (around 10^{80})




```
MODULE main
VAR
  x0 : boolean;
INIT
  !x0;
TRANS
  x0 <-> next(!x0);
```




```
MODULE main
VAR
  x0 : boolean;
  x1 : boolean;
INIT
  !x0 & !x1;
TRANS
  (next(x0) <-> !x0)
    &
  (next(x1) <-> ((x0 & !x1) | (!x0
    & x1)));
```

```
while true do
  if x < 200 then
    x := x + 1
od
```

```
while true do
  if x > 0 then
    x := x-1
od
```

```
while true do
  if x = 200 then
    x := 0
od
```

```
MODULE main
VAR
  x    : 0 .. 200;
INIT
  x = 199;
TRANS
  (x<200 & next(x)=x+1) |
  (x>0 & next(x)=x+(-1)) |
  (x=200 & next(x)=0);
```

An example of Symbolic Model Checking

- Let $S = \{0,1\}^3$ and let the `target` be defined by the function $x_0 \wedge x_1 \wedge \neg x_2$. Let R be defined by:

$$(\bigwedge_{i=0}^{2}(x_i \rightarrow x'_i)) \wedge (x_0 \leftrightarrow x'_0 \vee x_1 \leftrightarrow x'_1)$$

An example of Symbolic Model Checking

- Let $S = \{0,1\}^3$ and let the `target` be defined by the function $x_0 \wedge x_1 \wedge \neg x_2$. Let R be defined by:

$$(\bigwedge_{i=0}^{2}(x_i \rightarrow x_i')) \wedge (x_0 \leftrightarrow x_0' \vee x_1 \leftrightarrow x_1')$$

- Compute the Boolean formula which defines the set of states in the system in which the formula $EF(\texttt{target})$ holds, written as $[EF(\texttt{target})]$.

An example of Symbolic Model Checking

- Let $S = \{0, 1\}^3$ and let the `target` be defined by the function $x_0 \wedge x_1 \wedge \neg x_2$. Let R be defined by:

$$(\bigwedge_{i=0}^{2}(x_i \rightarrow x'_i)) \wedge (x_0 \leftrightarrow x'_0 \vee x_1 \leftrightarrow x'_1)$$

- Compute the Boolean formula which defines the set of states in the system in which the formula $EF(\texttt{target})$ holds, written as $[EF(\texttt{target})]$.
- $[\texttt{target}]$ is defined by $x_0 \wedge x_1 \wedge \neg x_2$.

An example of Symbolic Model Checking

- Let $S = \{0,1\}^3$ and let the target be defined by the function $x_0 \wedge x_1 \wedge \neg x_2$. Let R be defined by:

$$(\bigwedge_{i=0}^{2}(x_i \rightarrow x'_i)) \wedge (x_0 \leftrightarrow x'_0 \vee x_1 \leftrightarrow x'_1)$$

- Compute the Boolean formula which defines the set of states in the system in which the formula $EF(\mathtt{target})$ holds, written as $[EF(\mathtt{target})]$.
- $[\mathtt{target}]$ is defined by $x_0 \wedge x_1 \wedge \neg x_2$.
- $[\mathtt{target}] \cup [EX(\mathtt{target})]$ is defined by

$$(x_0 \wedge x_1 \wedge \neg x_2) \vee (\neg x_0 \wedge x_1 \wedge \neg x_2) \vee (x_0 \wedge \neg x_1 \wedge \neg x_2)$$

An example of Symbolic Model Checking

- Let $S = \{0,1\}^3$ and let the target be defined by the function $x_0 \wedge x_1 \wedge \neg x_2$. Let R be defined by:

$$(\bigwedge_{i=0}^{2}(x_i \rightarrow x'_i)) \wedge (x_0 \leftrightarrow x'_0 \vee x_1 \leftrightarrow x'_1)$$

- Compute the Boolean formula which defines the set of states in the system in which the formula $EF(\texttt{target})$ holds, written as $[EF(\texttt{target})]$.
- $[\texttt{target}]$ is defined by $x_0 \wedge x_1 \wedge \neg x_2$.
- $[\texttt{target}] \cup [EX(\texttt{target})]$ is defined by

$$(x_0 \wedge x_1 \wedge \neg x_2) \vee (\neg x_0 \wedge x_1 \wedge \neg x_2) \vee (x_0 \wedge \neg x_1 \wedge \neg x_2)$$

- $[\texttt{target}] \cup [EX(\texttt{target})] \cup [EXEX(\texttt{target})]$ is defined by:

$$(x_0 \wedge x_1 \wedge \neg x_2) \vee (\neg x_0 \wedge x_1 \wedge \neg x_2) \vee (x_0 \wedge \neg x_1 \wedge \neg x_2) \vee (\neg x_0 \wedge \neg x_1 \wedge \neg x_2)$$

$$\equiv \neg x_2$$

An example of Symbolic Model Checking

- Let $S = \{0,1\}^3$ and let the target be defined by the function $x_0 \wedge x_1 \wedge \neg x_2$. Let R be defined by:

$$(\bigwedge_{i=0}^{2}(x_i \rightarrow x'_i)) \wedge (x_0 \leftrightarrow x'_0 \vee x_1 \leftrightarrow x'_1)$$

- Compute the Boolean formula which defines the set of states in the system in which the formula $EF(\texttt{target})$ holds, written as $[EF(\texttt{target})]$.
- $[\texttt{target}]$ is defined by $x_0 \wedge x_1 \wedge \neg x_2$.
- $[\texttt{target}] \cup [EX(\texttt{target})]$ is defined by

$$(x_0 \wedge x_1 \wedge \neg x_2) \vee (\neg x_0 \wedge x_1 \wedge \neg x_2) \vee (x_0 \wedge \neg x_1 \wedge \neg x_2)$$

- $[\texttt{target}] \cup [EX(\texttt{target})] \cup [EXEX(\texttt{target})]$ is defined by:

$$(x_0 \wedge x_1 \wedge \neg x_2) \vee (\neg x_0 \wedge x_1 \wedge \neg x_2) \vee (x_0 \wedge \neg x_1 \wedge \neg x_2) \vee (\neg x_0 \wedge \neg x_1 \wedge \neg x_2)$$

$\equiv \neg x_2$ fixpoint: $[EF(\texttt{target})]$

Key step:

- proceed from a set of states T to the set of its predecessors:

$$\{s \in S \mid \exists (s, s') \in R, s' \in T\}$$

- proceed from a formula $\beta_T(\overline{x})$ to:

$$\beta'(\overline{x}) = \exists \overline{y}(f_R(\overline{x}, \overline{y}) \wedge \beta_T(\overline{y}))$$

Problem:

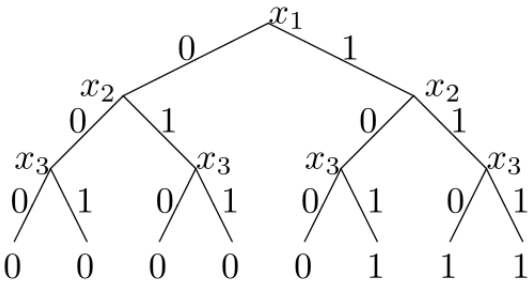
what is a good normal form for the representation of Boolean functions which allows efficient application of $\neg$, $\wedge$, $\vee$, and $\exists$?

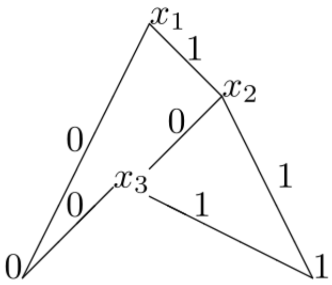
Ordered Binary Decision Diagrams (OBDDs)

Ordered Binary Decision Diagrams (OBDDs) are based on work by Akers (1978) and Bryant (1986). They are reduced versions of decision trees for Boolean functions.

Example: $x_1 \wedge (x_2 \vee x_3)$

Decision tree vs. OBDD





Basic Results on OBDDs

1. Using the two reduction rules:
   - identify isomorphic "subgraphs"
   - for $x \in \{0, 1\}$, replace the paths $x0$ and $x1$ by the arc x, whenever $x0$ and $x1$ lead to the same value (cf. previous example)
   - one obtains a canonical (unique) OBDD for a given Boolean function and a given order of variables
2. The operations $\neg, \wedge, \vee, \exists$ can be performed efficiently on OBDDs

Partial order reduction can be used to reduce the size of the state space making use of the following observation:

computations that differ in the ordering of independently executed events are usually indistinguishable by the specification and thus they can be considered equivalent

It suffices to check a reduced state space, which contains (at least) one representative computation for each class of equivalent computations.

Advanced Techniques

- Bounded model checking: SAT solvers + symbolic model checking + bounded models
- Satisfiability Modulo Theories (SMT solvers): satisfiability of logical formulas with respect to one or more background theories formulated in first-order logic with equality (integers, real numbers, ..)
- K-Liveness: a simple but effective technique for LTL verification; it checks the absence of lasso-shaped counterexamples by trying to prove that bad states are visited at most k times, for increasing values of k (liveness checking as a sequence of safety checks)
- IC3: a very successful SAT-based model checking algorithm based on induction

BOUNDED MODEL CHECKING

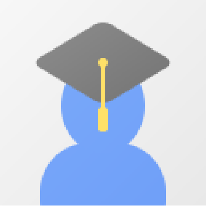
Reference:

Armin Biere et al. (1999). "Symbolic model checking without BDDs". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, pp. 193–207

(a) A. Biere

(b) A. Cimatti

(c) E. Clarke



(d) Y. Zhu

The SAT problem

- given a Boolean formula f, establish if f is satisfiable;

The SAT problem

- given a Boolean formula f, establish if f is satisfiable;
- f is normally given in CNF:

$$f := (L_{1,1} \vee \cdots \vee L_{1,k}) \wedge \cdots \wedge (L_{n,1} \vee \cdots \vee L_{n,m})$$

where each literal $L_{i,j}$ is either a variable or a negation of a variable.

The SAT problem

- given a Boolean formula f, establish if f is satisfiable;
- f is normally given in CNF:

$$f := (L_{1,1} \vee \cdots \vee L_{1,k}) \wedge \cdots \wedge (L_{n,1} \vee \cdots \vee L_{n,m})$$

  where each literal $L_{i,j}$ is either a variable or a negation of a variable.
- why not in DNF?

$$f := (L_{1,1} \wedge \cdots \wedge L_{1,k}) \vee \cdots \vee (L_{n,1} \wedge \cdots \wedge L_{n,m})$$

The SAT problem

- first NP-complete problem, but ...

The SAT problem

- first NP-complete problem, but ...
- there are several efficient algorithms for solving SAT (*e.g.*, DPLL, CDCL...) along with many heuristics (*e.g.*, 2 watching literals, glue clauses...)

The SAT problem

- first NP-complete problem, but ...
- there are several efficient algorithms for solving SAT (*e.g.*, DPLL, CDCL...) along with many heuristics (*e.g.*, 2 watching literals, glue clauses...)
- some numbers:
  - $> 100'000$ variables;
  - $> 1'000'000$ clauses;

In order to decide if $\mathcal{M}, s \models \phi$:

Classical Approach to LTL MC

In order to decide if $\mathcal{M}, s \models \phi$:

- build the Büchi automaton $\mathcal{A}_{\mathcal{M}}$ that accepts all and only the words corresponding to computations of $\mathcal{M}$;

Classical Approach to LTL MC

In order to decide if $\mathcal{M}, s \models \phi$:

- build the Büchi automaton $\mathcal{A}_{\mathcal{M}}$ that accepts all and only the words corresponding to computations of $\mathcal{M}$;
- build the Büchi automaton $\mathcal{A}_{\neg\phi}$ that accepts all and only the words corresponding to models of $\neg\phi$;

Classical Approach to LTL MC

In order to decide if $\mathcal{M}, s \models \phi$:

- build the Büchi automaton $\mathcal{A}_\mathcal{M}$ that accepts all and only the words corresponding to computations of $\mathcal{M}$;
- build the Büchi automaton $\mathcal{A}_{\neg\phi}$ that accepts all and only the words corresponding to models of $\neg\phi$;
- check the (non-)emptiness of the product automaton $\mathcal{A}_\mathcal{M} \times \mathcal{A}_{\neg\phi}$.

MC=universal problem, EMPTINESS= existential problem

Classical Approach to LTL MC

In order to decide if $\mathcal{M}, s \models \phi$:

- build the Büchi automaton $\mathcal{A}_\mathcal{M}$ that accepts all and only the words corresponding to computations of $\mathcal{M}$;
- build the Büchi automaton $\mathcal{A}_{\neg\phi}$ that accepts all and only the words corresponding to models of $\neg\phi$;
- check the (non-)emptiness of the product automaton $\mathcal{A}_\mathcal{M} \times \mathcal{A}_{\neg\phi}$.

  MC=universal problem, EMPTINESS= existential problem

- if $L(\mathcal{A}_\mathcal{M} \times \mathcal{A}_{\neg\phi}) \neq \varnothing$, then $\mathcal{M}, s \overset{?}{\models} \phi$.

In order to decide if $\mathcal{M}, s \models \phi$:

- build the Büchi automaton $\mathcal{A}_{\mathcal{M}}$ that accepts all and only the words corresponding to computations of $\mathcal{M}$;
- build the Büchi automaton $\mathcal{A}_{\neg\phi}$ that accepts all and only the words corresponding to models of $\neg\phi$;
- check the (non-)emptiness of the product automaton $\mathcal{A}_{\mathcal{M}} \times \mathcal{A}_{\neg\phi}$.

MC=universal problem, EMPTINESS= existential problem

- if $L(\mathcal{A}_{\mathcal{M}} \times \mathcal{A}_{\neg\phi}) \neq \varnothing$, then $\mathcal{M}, s \overset{?}{\models} \phi$.

$$\mathcal{M}, s \not\models \phi$$

- the universal problem $\mathcal{M}, s \models A\psi$ is reduced to the existential problem $\mathcal{M}, s \models E\phi$, where $\phi := \neg\psi$;

Bounded Model Checking (BMC) solves the problem $\mathcal{M}, s \models E\phi$ by proceeding incrementally:

- we start with $k = 0$;
- check if there exists and execution π of $\mathcal{M}$ of length k that satisfies ϕ; encode this problem into a SAT instance and call a SAT-solver;
- if so, we have found a counterexample to ψ; if not, $k{+}{+}$.

Loop-backs

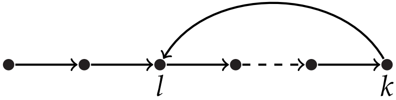
- BMC checks only bounded/finite traces of the system;
- ...but LTL formulas are defined over infinite state sequences;

Loop-backs

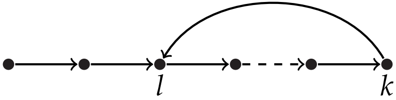
- BMC checks only bounded/finite traces of the system;
- ...but LTL formulas are defined over infinite state sequences;

Crucial observation:

- a finite trace can still represent an infinite state sequence, if it contains a loop-back.



k-loop, aka Lasso-Shaped Models



Definition (k-loop)

A path π is a (k, l)-loop, with $l \leq k$, if $T(\pi(k), \pi(l))$ holds and $\pi = u \cdot v^\omega$, where:

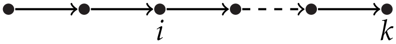
- $u = \pi(1) \dots \pi(l-1)$;
- $v = \pi(l) \dots \pi(k)$.

We call π a k-loop if there exists $l \leq k$ for which π is a (k, l)-loop.

Given a finite trace π of the system $\mathcal{M}$, BMC distinguishes between two cases:

- either π contains a loop-back (π is lasso-shaped):
  - $\Rightarrow$ apply standard LTL semantics to check if $\pi \models \phi$;
- or π is loop-free:
  - $\Rightarrow$ apply bounded semantics
  - $\Rightarrow$ if a path is a model of ϕ under bounded semantics then any extension of the path is a model of ϕ under standard semantics (conservative semantics)

Bounded Semantics for LTL

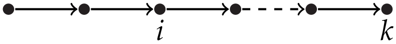


If π is not a k-loop, we introduce bounded semantics for LTL.

Definition (Bounded semantics for LTL)

Let $k \geq 0$ and π a path that is not a k-loop. An LTL formula ϕ is valid along π with bound k, written $\pi \models_k^0 \phi$, iff:

- $\pi \models_k^i p$ iff $p \in L(\pi(i))$
- $\pi \models_k^i \neg p$ iff $p \notin L(\pi(i))$



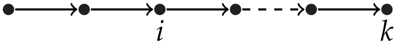
If π is not a k-loop, we introduce bounded semantics for LTL.

Definition (Bounded semantics for LTL)

Let $k \geq 0$ and π a path that is not a k-loop. An LTL formula ϕ is valid along π with bound k, written $\pi \models_k^0 \phi$, iff:

- $\pi \models_k^i \phi_1 \vee \phi_2$ iff $\pi \models_k^i \phi_1$ or $\pi \models_k^i \phi_2$
- $\pi \models_k^i \phi_1 \wedge \phi_2$ iff $\pi \models_k^i \phi_1$ and $\pi \models_k^i \phi_2$

Bounded Semantics for LTL

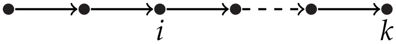


If π is not a k-loop, we introduce bounded semantics for LTL.

Definition (Bounded semantics for LTL)

Let $k \geq 0$ and π a path that is not a k-loop. An LTL formula ϕ is valid along π with bound k, written $\pi \models_k^0 \phi$, iff:

- $\pi \models_k^i \mathsf{X}\phi_1$ iff $i < k$ and $\pi \models_k^{i+1} \phi_1$
- $\pi \models_k^i \phi_1 \,\mathsf{U}\, \phi_2$ iff $\exists i \leq j \leq k$ such that $\pi \models_k^j \phi_2$ and $\forall i \leq n < j$ it holds that $\pi \models_k^n \phi_1$

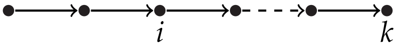


If π is not a k-loop, we introduce bounded semantics for LTL.

Definition (Bounded semantics for LTL)

Let $k \geq 0$ and π a path that is not a k-loop. An LTL formula ϕ is valid along π with bound k, written $\pi \models_k^0 \phi$, iff:

- $\pi \models_k^i \mathsf{G}\phi_1$ iff ???
- $\pi \models_k^i \mathsf{F}\phi_1$ iff ???



If π is not a k-loop, we introduce bounded semantics for LTL.

Definition (Bounded semantics for LTL)

Let $k \geq 0$ and π a path that is not a k-loop. An LTL formula ϕ is valid along π with bound k, written $\pi \models_k^0 \phi$, iff:

- $\pi \models_k^i \mathsf{G}\phi_1$ is always false
- $\pi \models_k^i \mathsf{F}\phi_1$ iff $\exists i \leq j \leq k$ such that $\pi \models_k^j \phi_1$

Now we see how to reduce BMC to SAT.

- the first thing to do is to define a Boolean formula that encodes all the paths of $\mathcal{M}$ of length k.

SAT-based encoding of BMC

Now we see how to reduce BMC to SAT.

- the first thing to do is to define a Boolean formula that encodes all the paths of $\mathcal{M}$ of length k.

Definition (Unfolding of the Transition Relation)

For a Kripke structure $\mathcal{M}$ and $k \geq 0$, we define:

$$[\![\mathcal{M}]\!]_k := I(\overline{x}^0) \wedge \bigwedge_{i=0}^{k-1} T(\overline{x}^i, \overline{x}^{i+1})$$

N.B.: For each $i \geq 0$, with $\overline{x}^i$ we represent the i^{th}-stepped version of the set of variables $\overline{x}$. For example, $\overline{x}^1 := \overline{x}'$.

Encoding of the LTL formula

So far, we have seen how to encode paths of length k of the model $\mathcal{M}$.

So far, we have seen how to encode paths of length k of the model $\mathcal{M}$.

- intuitively, this corresponds to the left-hand side of the automaton $\mathcal{A}_{\mathcal{M}} \times \mathcal{A}_{\neg\psi}$
- now we see how to encode the right-hand side.

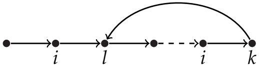
Encoding of the LTL formula

So far, we have seen how to encode paths of length k of the model $\mathcal{M}$.

- intuitively, this corresponds to the left-hand side of the automaton $\mathcal{A}_{\mathcal{M}} \times \mathcal{A}_{\neg\psi}$
- now we see how to encode the right-hand side.

We have seen that BMC distinguishes between lasso-shaped (k-loop) and loop-free paths:

- we start with the encoding in case of k-loops.

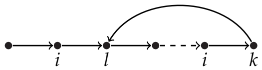


Definition (Loop Encoding)

Let $l \leq k$. We define:

$$_{l}L_k := T(\overline{x}^k, \overline{x}^l) \qquad L_k := \bigvee_{l=0}^{k} {}_{l}L_k$$

Encoding of a loop



Definition (Loop Encoding)

Let $l \leq k$. We define:

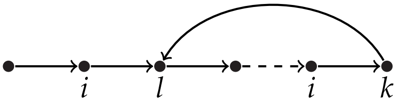
$$ {}_lL_k := T(\overline{x}^k, \overline{x}^l) \qquad\qquad L_k := \bigvee_{l=0}^{k} {}_lL_k $$

Definition (Successor in a Loop)

Let $l, i \leq k$ and π be a (k, l)-loop. We define the successor $succ(i)$ of i in π as:

- $succ(i) := i + 1$ if $i < k$;
- $succ(i) := l$ if $i = k$.

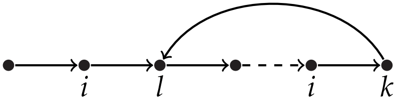
Encoding in case of Loop



Definition (Encoding of an LTL formula for a (k, l)-loop)

Let ϕ be an LTL formula and $l, i, k \geq 0$ such that $l, i \leq k$. We define ${}_l[\![\phi]\!]_k^i$ recursively as follows:

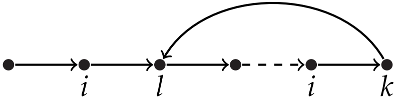
- ${}_l[\![p]\!]_k^i := p(\overline{x}^i)$
- ${}_l[\![\neg p]\!]_k^i := \neg p(\overline{x}^i)$



Definition (Encoding of an LTL formula for a (k, l)-loop)

Let ϕ be an LTL formula and $l, i, k \geq 0$ such that $l, i \leq k$. We define ${}_l[\![\phi]\!]_k^i$ recursively as follows:

- ${}_l[\![\phi_1 \vee \phi_2]\!]_k^i := {}_l[\![\phi_1]\!]_k^i \vee {}_l[\![\phi_2]\!]_k^i$
- ${}_l[\![\phi_1 \wedge \phi_2]\!]_k^i := {}_l[\![\phi_1]\!]_k^i \wedge {}_l[\![\phi_2]\!]_k^i$

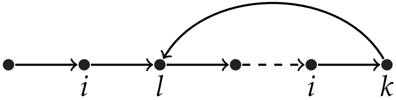


Definition (Encoding of an LTL formula for a (k, l)-loop)

Let ϕ be an LTL formula and $l, i, k \geq 0$ such that $l, i \leq k$. We define ${}_l[\![\phi]\!]_k^i$ recursively as follows:

- ${}_l[\![\mathsf{X}\phi_1]\!]_k^i := {}_l[\![\phi_1]\!]_k^{succ(i)}$
- ${}_l[\![\phi_1 \,\mathsf{U}\, \phi_2]\!]_k^i := {}_l[\![\phi_2]\!]_k^i \vee \left({}_l[\![\phi_1]\!]_k^i \wedge {}_l[\![\phi_1 \,\mathsf{U}\, \phi_2]\!]_k^{succ(i)}\right)$

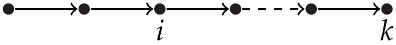
Encoding in case of Loop



Definition (Encoding of an LTL formula for a (k, l)-loop)

Let ϕ be an LTL formula and $l, i, k \geq 0$ such that $l, i \leq k$. We define ${}_l[\![\phi]\!]_k^i$ recursively as follows:

- ${}_l[\![\mathsf{G}\phi_1]\!]_k^i := {}_l[\![\phi_1]\!]_k^i \wedge {}_l[\![\mathsf{G}\phi_1]\!]_k^{succ(i)}$
- ${}_l[\![\mathsf{F}\phi_1]\!]_k^i := {}_l[\![\phi_1]\!]_k^i \vee {}_l[\![\mathsf{F}\phi_1]\!]_k^{succ(i)}$

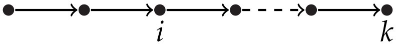


Definition (Encoding of an LTL formula for a loop-free path)

Let ϕ be an LTL formula and $i, k \geq 0$. We define $[\![\phi]\!]_k^i$ recursively as follows:

- $[\![\phi]\!]_k^{k+1} := \bot$

Encoding in case of NO Loops

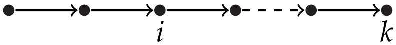


Definition (Encoding of an LTL formula for a loop-free path)

Let ϕ be an LTL formula and $i, k \geq 0$. We define $[\![\phi]\!]_k^i$ recursively as follows:

- $[\![p]\!]_k^i := p(\overline{x}^i)$
- $[\![\neg p]\!]_k^i := \neg p(\overline{x}^i)$

with $i \leq k$



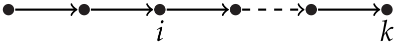
Definition (Encoding of an LTL formula for a loop-free path)

Let ϕ be an LTL formula and $i, k \geq 0$. We define $[\![\phi]\!]_k^i$ recursively as follows:

- $[\![\phi_1 \vee \phi_2]\!]_k^i := [\![\phi_1]\!]_k^i \vee {}_l[\![\phi_2]\!]_k^i$
- $[\![\phi_1 \wedge \phi_2]\!]_k^i := [\![\phi_1]\!]_k^i \wedge {}_l[\![\phi_2]\!]_k^i$

with $i \leq k$

Encoding in case of NO Loops

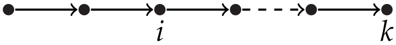


Definition (Encoding of an LTL formula for a loop-free path)

Let ϕ be an LTL formula and $i, k \geq 0$. We define $[\![\phi]\!]_k^i$ recursively as follows:

- $[\![\mathsf{X}\phi_1]\!]_k^i := [\![\phi_1]\!]_k^{i+1}$
- ${}_l[\![\phi_1 \ \mathsf{U}\ \phi_2]\!]_k^i := {}_l[\![\phi_2]\!]_k^i \lor ({}_l[\![\phi_1]\!]_k^i \land {}_l[\![\phi_1 \ \mathsf{U}\ \phi_2]\!]_k^{i+1})$

with $i \leq k$



Definition (Encoding of an LTL formula for a loop-free path)

Let ϕ be an LTL formula and $i, k \geq 0$. We define $[\![\phi]\!]_k^i$ recursively as follows:

- $_l[\![\mathsf{G}\phi_1]\!]_k^i := {}_l[\![\phi_1]\!]_k^i \wedge {}_l[\![\mathsf{G}\phi_1]\!]_k^{i+1}$
- $_l[\![\mathsf{F}\phi_1]\!]_k^i := {}_l[\![\phi_1]\!]_k^i \vee {}_l[\![\mathsf{F}\phi_1]\!]_k^{i+1}$

with $i \leq k$

Overall encoding

Definition (Overall encoding)

Let ϕ be an LTL formula, $\mathcal{M}$ be a Kripke structure and $k \geq 0$:

$$[\![M, \phi]\!]_k := \underbrace{[\![\mathcal{M}]\!]_k}_{\substack{\text{encoding of} \\ \text{the machine}}} \wedge \left(\underbrace{\left(\neg L_k \wedge [\![\phi]\!]_k^0\right)}_{\substack{\text{loop-free} \\ \text{models}}} \vee \underbrace{\bigvee_{l=0}^{k} \left({}_lL_k \wedge {}_l[\![\phi]\!]_k^0\right)}_{\substack{\text{lasso-shaped} \\ \text{models}}} \right)$$

Definition (Overall encoding)

Let ϕ be an LTL formula, $\mathcal{M}$ be a Kripke structure and $k \geq 0$:

$$[\![M, \phi]\!]_k := \underbrace{[\![\mathcal{M}]\!]_k}_{\substack{\text{encoding of} \\ \text{the machine}}} \wedge \left(\underbrace{(\neg L_k \wedge [\![\phi]\!]_k^0)}_{\substack{\text{loop-free} \\ \text{models}}} \vee \underbrace{\bigvee_{l=0}^{k} ({}_l L_k \wedge {}_l[\![\phi]\!]_k^0)}_{\substack{\text{lasso-shaped} \\ \text{models}}} \right)$$

Theorem (Soundness)

$[\![\mathcal{M}, \phi]\!]_k$ is satisfiable iff $\mathcal{M} \models_k E\phi$.

Completeness

Algorithm:

- start with $k = 0$
- call a SAT-solver on $[\![\mathcal{M}, \phi]\!]_k$
- if it is SAT, stop; otherwise, $k{+}{+}$.

Completeness

Algorithm:

- start with $k = 0$
- call a SAT-solver on $[\![\mathcal{M}, \phi]\!]_k$
- if it is SAT, stop; otherwise, $k{+}{+}$.

What happens if $\mathcal{M} \not\models \phi$?

Completeness

Algorithm:

- start with $k = 0$
- call a SAT-solver on $[\![\mathcal{M}, \phi]\!]_k$
- if it is SAT, stop; otherwise, $k{+}{+}$.

What happens if $\mathcal{M} \not\models \phi$?

- the procedure does not terminate

Completeness

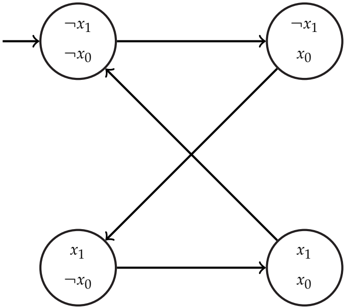
Algorithm:

- start with $k = 0$
- call a SAT-solver on $[\![\mathcal{M}, \phi]\!]_k$
- if it is SAT, stop; otherwise, $k{+}{+}$.

What happens if $\mathcal{M} \not\models \phi$?

- the procedure does not terminate
- in order to be complete, BMC needs to compute the recurrence diameter: very costly
- BMC is mainly used as a bug finder, rather than as a prover.

Example



- $\phi_1 \coloneqq \mathsf{GF}(x_0) \wedge x_1)$ ✓
- $\phi_2 \coloneqq \mathsf{FG}(\neg x_0 \wedge \neg x_1)$ ✗

Solving LTL-SAT with BMC

LTL-SAT is the problem of establishing if, given an LTL formula ϕ, there exists an infinite state sequence σ such that $\sigma \models \phi$.

Solving LTL-SAT with BMC

LTL-SAT is the problem of establishing if, given an LTL formula ϕ, there exists an infinite state sequence σ such that $\sigma \models \phi$.

- how can one solve LTL-SAT with BMC?

LTL-SAT is the problem of establishing if, given an LTL formula ϕ, there exists an infinite state sequence σ such that $\sigma \models \phi$.

- how can one solve LTL-SAT with BMC?
- model checking:

$$[\![\mathcal{M}]\!]_k \wedge \left((\neg L_k \wedge [\![\phi]\!]_k^0) \vee \bigvee_{l=0}^{k} ({}_l L_k \wedge {}_l[\![\phi]\!]_k^0) \right)$$

Solving LTL-SAT with BMC

LTL-SAT is the problem of establishing if, given an LTL formula ϕ, there exists an infinite state sequence σ such that $\sigma \models \phi$.

- how can one solve LTL-SAT with BMC?
- model checking:

$$[\![\mathcal{M}]\!]_k \wedge \left((\neg L_k \wedge [\![\phi]\!]_k^0) \vee \bigvee_{l=0}^{k} ({}_l L_k \wedge {}_l[\![\phi]\!]_k^0) \right)$$

- satisfiability checking

$$\top \wedge \left((\neg L_k \wedge [\![\phi]\!]_k^0) \vee \bigvee_{l=0}^{k} ({}_l L_k \wedge {}_l[\![\phi]\!]_k^0) \right)$$

BLACK

- we developed this tool based on the idea of *bounded satisfiability checking*
- BLACK = Bounded Ltl sAtisfiability ChecKer
- `https://www.black-sat.org/en/stable/`
- Examples

Reference:

Luca Geatti, Nicola Gigante, and Angelo Montanari (2021). "BLACK: A Fast, Flexible and Reliable LTL Satisfiability Checker". In: *Proceedings of the 3rd Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis*. Ed. by Dario Della Monica, Gian Luca Pozzato, and Enrico Scala. Vol. 2987. CEUR Workshop Proceedings. CEUR-WS.org, pp. 7–12. URL: `http://ceur-ws.org/Vol-2987/paper2.pdf`

REFERENCES

Bibliography I

Armin Biere et al. (1999). "Symbolic model checking without BDDs". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, pp. 193–207.

Luca Geatti, Nicola Gigante, and Angelo Montanari (2021). "BLACK: A Fast, Flexible and Reliable LTL Satisfiability Checker". In: *Proceedings of the 3rd Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis*. Ed. by Dario Della Monica, Gian Luca Pozzato, and Enrico Scala. Vol. 2987. CEUR Workshop Proceedings. CEUR-WS.org, pp. 7–12. URL: http://ceur-ws.org/Vol-2987/paper2.pdf.

Kenneth L McMillan (1993). "Symbolic model checking". In: *Symbolic Model Checking*. Springer, pp. 25–60.