

Bounded Model Checking

Luca Geatti

University of Udine, Italy

FBK, Trento, Italy

Automatic System Verification

UniUD, May 18st, 2020, Udine, Italy

Introduction

- Automatic formal verification techniques: great progress in the last decades;

- Automatic formal verification techniques: great progress in the last decades;
- big chip or software companies have integrated them in their development or quality assurance process;

- Automatic formal verification techniques: great progress in the last decades;
- big chip or software companies have integrated them in their development or quality assurance process;
- **Intel**: FDIV bug, error in the floating point division instruction on some Intel®Pentium® processors.

- Automatic formal verification techniques: great progress in the last decades;
- big chip or software companies have integrated them in their development or quality assurance process;
- **Intel**: FDIV bug, error in the floating point division instruction on some Intel®Pentium® processors.
 - it costed \approx US \$475 million;

- Automatic formal verification techniques: great progress in the last decades;
- big chip or software companies have integrated them in their development or quality assurance process;
- **Intel**: FDIV bug, error in the floating point division instruction on some Intel®Pentium® processors.
 - it costed \approx US \$475 million;
 - big investment in formal verification.

The most used formal verification technique is **Model Checking** (MC, for short).

The most used formal verification technique is **Model Checking** (MC, for short).

- the system to verify is modeled as a finite-state machine (*i.e.*, Kripke structure) and the specification is expressed by means of a temporal logic formula;

The most used formal verification technique is **Model Checking** (MC, for short).

- the system to verify is modeled as a finite-state machine (*i.e.*, Kripke structure) and the specification is expressed by means of a temporal logic formula;
- distinctive features:
 - fully automatic;

The most used formal verification technique is **Model Checking** (MC, for short).

- the system to verify is modeled as a finite-state machine (*i.e.*, Kripke structure) and the specification is expressed by means of a temporal logic formula;
- distinctive features:
 - fully automatic;
 - exhaustive;

The most used formal verification technique is **Model Checking** (MC, for short).

- the system to verify is modeled as a finite-state machine (*i.e.*, Kripke structure) and the specification is expressed by means of a temporal logic formula;
- distinctive features:
 - fully automatic;
 - exhaustive;
 - it generates a counterexample trace if the specification does not hold.

We consider **LTL** model checking.

- LTL syntax:

$$\begin{aligned} p &| \neg\phi &| \phi_1 \vee \phi_2 &| \phi_1 \wedge \phi_2 \\ &| X\phi_1 &| \phi_1 \mathcal{U} \phi_2 \\ &| \phi_1 \mathcal{R} \phi_2 &| F\phi_1 &| G\phi_1 \end{aligned}$$

We consider **LTL** model checking.

- LTL syntax:

$$\begin{aligned} p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \\ \mid X\phi_1 \mid \phi_1 \mathcal{U} \phi_2 \\ \mid \phi_1 \mathcal{R} \phi_2 \mid F\phi_1 \mid G\phi_1 \end{aligned}$$

- shortcuts:

- $\phi_1 \mathcal{R} \phi_2 \equiv \neg(\neg\phi_1 \mathcal{U} \neg\phi_2)$,
- $F\phi_1 \equiv \top \mathcal{U} \phi_1$
- $G\phi_1 \equiv \neg F\neg\phi_1$

Linear Temporal Logic

- Semantics. LTL formulas are interpreted over **infinite state sequences** $\sigma = \langle \sigma_0, \sigma_1, \dots \rangle \in (2^\Sigma)^\omega$ of sets of propositions $\sigma_i \in 2^\Sigma$:

$$\sigma \models_i p \quad \text{iff} \quad p \in \sigma_i$$

$$\sigma \models_i X\phi \quad \text{iff} \quad \sigma \models_{i+1} \phi$$

$$\sigma \models_i \phi_1 \mathcal{U} \phi_2 \quad \text{iff} \quad \text{there exists } j \geq i \text{ such that} \\ \sigma \models_j \phi_2 \text{ and } \sigma \models_k \phi_1 \text{ for all} \\ i \leq k < j$$

...

- LTL model checking:
 - decide if $\mathcal{M}, s \models \phi$, where $\mathcal{M} = (S, I, T, L)$ is a Kripke structure, $s \in I$ is an initial state and ϕ is an LTL formula; in many contexts, you may find the notation: $\mathcal{M}, s \models A\phi$;
 - **PSPACE-complete**.

Classical Approach to LTL MC

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} ;

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 $L(\mathcal{A}_{\mathcal{M}} \times \mathcal{A}_{\neg\phi}) \neq \emptyset$, what can we say about \mathcal{M} and ϕ ?

State-space Explosion Problem

- the previous algorithm belongs to the class of **explicit** model checking algorithms:
 - the Kripke Structure \mathcal{M} is represented as a set of memory locations, pointers ecc...

State-space Explosion Problem

- the previous algorithm belongs to the class of **explicit** model checking algorithms:
 - the Kripke Structure \mathcal{M} is represented as a set of memory locations, pointers ecc...
- MC suffers from the **state-space explosion problem**: the number of states of

$$\mathcal{M} = \mathcal{M}_1 \times \mathcal{M}_2 \times \cdots \times \mathcal{M}_n$$

is exponential in n ;

State-space Explosion Problem

- the previous algorithm belongs to the class of **explicit** model checking algorithms:
 - the Kripke Structure \mathcal{M} is represented as a set of memory locations, pointers ecc...
- MC suffers from the **state-space explosion problem**: the number of states of

$$\mathcal{M} = \mathcal{M}_1 \times \mathcal{M}_2 \times \cdots \times \mathcal{M}_n$$

is exponential in n ;

- the size of system that could be verified by explicit model checkers was restricted to $\approx 10^6$ states.

Tackling the explosion...

Three main techniques have been proposed:

- BDD-based symbolic model checking, next lessons
- partial order reduction, next lessons :-)
- SAT-based symbolic model checking, aka **Bounded Model Checking**.

Tackling the explosion...

Three main techniques have been proposed:

- BDD-based symbolic model checking, next lessons
- partial order reduction, next lessons :-)
- SAT-based symbolic model checking, aka **Bounded Model Checking**.

They allowed for the verification of systems with $> 10^{20}$ states.

Symbolic Transition Systems

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;

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 $s := \{s(0), \dots, s(n)\}$ be a set of state (Boolean) variables;

Symbolic Transition Systems

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

- let $s := \{s(0), \dots, s(n)\}$ be a set of state (Boolean) variables;
- $S = \{0, 1\}^n$, i.e., a state is an **assignment** to all the state variables;

Symbolic Transition Systems

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

- let $s := \{s(0), \dots, s(n)\}$ be a set of state (Boolean) variables;
- $S = \{0, 1\}^n$, i.e., a state is an **assignment** to all the state variables;
- $m \models f_l(s)$ 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 $s := \{s(0), \dots, s(n)\}$ be a set of state (Boolean) variables;
- $S = \{0, 1\}^n$, i.e., a state is an **assignment** to all the state variables;
- $m \models f_I(s)$ is true iff $m \in I$
- $m, m' \models f_T(s, s')$ is true iff $(m, m') \in T$

Symbolic Transition Systems

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

- let $s := \{s(0), \dots, s(n)\}$ be a set of state (Boolean) variables;
- $S = \{0, 1\}^n$, i.e., a state is an **assignment** to all the state variables;
- $m \models f_I(s)$ is true iff $m \in I$
- $m, m' \models f_T(s, s')$ is true iff $(m, m') \in T$
- $m \models f_p(s)$ is true iff $p \in L(m)$, for all labels $p \in \text{RANGE}(L)$

Symbolic Transition Systems

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

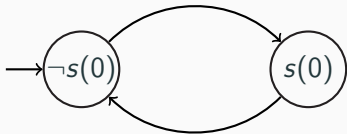
- let $s := \{s(0), \dots, s(n)\}$ be a set of state (Boolean) variables;
- $S = \{0, 1\}^n$, i.e., a state is an **assignment** to all the state variables;
- $m \models f_I(s)$ is true iff $m \in I$
- $m, m' \models f_T(s, s')$ is true iff $(m, m') \in T$
- $m \models f_p(s)$ is true iff $p \in L(m)$, for all labels $p \in \text{RANGE}(L)$

The corresponding **symbolic** Kripke structure is the tuple $(S, f_I, f_T, \{f_{p_1}, \dots, f_{p_k}\})$.

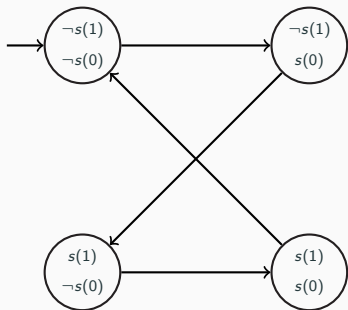
- we will write simply $\mathcal{M} = (S, I, T, L)$, meaning a **symbolic** transition system
- a path (or **trace**) $\pi = m_0, m_1, \dots$ is an infinite sequence of assignment to the state variables such that:
 - $m_0 \models I(s)$;
 - $m_i, m_{i+1} \models T(s, s')$ holds, for all $i \geq 0$.

where $s' := \{s'(0), \dots, s'(n)\}$.

simple-example.smv



modulo-4-counter.smv



Bounded Model Checking

- recall that we can reduce $\mathcal{M}, s \models \psi$ to checking the emptiness of $\mathcal{M} \times \mathcal{A}_{\neg\psi}$;

- recall that we can reduce $\mathcal{M}, s \models \psi$ to checking the emptiness of $\mathcal{M} \times \mathcal{A}_{\neg\psi}$;
 - 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

- recall that we can reduce $\mathcal{M}, s \models \psi$ to checking the emptiness of $\mathcal{M} \times \mathcal{A}_{\neg\psi}$;
 - 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** an 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, $k++$.

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

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

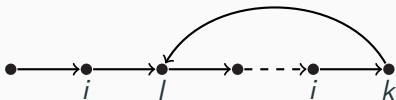
Crucial observation:

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

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**): apply standard LTL semantics to check if $\pi \models \phi$;
- or π is **loop-free**: apply bounded semantics.

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.

If a path π is a k -loop, then we can check if $\pi \models \phi$ with the standard LTL semantics.

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 $\pi \models_k^i \phi_2$

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:

- $\pi \models_k^i X \phi_1$ iff $i < k$ and $\pi \models_k^{i+1} \phi_1$
- $\pi \models_k^i \phi_1 \mathcal{U} \phi_2$ 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:

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

- $\pi \models_k^i \mathbf{G} \phi_1$ *is always false*
- $\pi \models_k^i \mathbf{F} \phi_1$ *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 .

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:

$$\llbracket \mathcal{M} \rrbracket_k := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$$

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 .

Definition (Unfolding of the Transition Relation)

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

$$\llbracket \mathcal{M} \rrbracket_k := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$$

What does a model of $\llbracket \mathcal{M} \rrbracket_k$ represent?

Encoding of the LTL formula

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

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.

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:

- ${}_l L_k := T(s_k, s_l)$
- $L_k := \bigvee_{l=0}^k {}_l L_k$

Encoding of a loop

Definition (Loop Encoding)

Let $l \leq k$. We define:

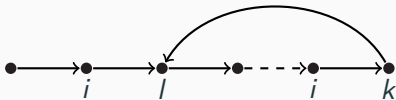
- ${}_l L_k := T(s_k, s_l)$
- $L_k := \bigvee_{l=0}^k {}_l L_k$

Definition (Successor in a Loop)

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

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

Encoding in case of Loop

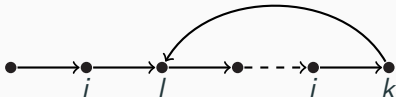


Definition (Encoding of an LTL formula for a (k, l) -loop)

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

- ${}_l\llbracket p \rrbracket_k^i := p(s_i)$
- ${}_l\llbracket \neg p \rrbracket_k^i := \neg p(s_i)$

Encoding in case of Loop

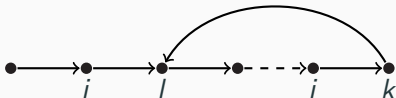


Definition (Encoding of an LTL formula for a (k, l) -loop)

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

- ${}_l\llbracket\phi_1 \vee \phi_2\rrbracket_k^i := {}_l\llbracket\phi_1\rrbracket_k^i \vee {}_l\llbracket\phi_2\rrbracket_k^i$
- ${}_l\llbracket\phi_1 \wedge \phi_2\rrbracket_k^i := {}_l\llbracket\phi_1\rrbracket_k^i \wedge {}_l\llbracket\phi_2\rrbracket_k^i$

Encoding in case of Loop

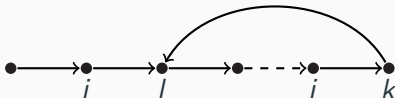


Definition (Encoding of an LTL formula for a (k, l) -loop)

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

- ${}_l\llbracket X \phi_1 \rrbracket_k^i := {}_l\llbracket \phi_1 \rrbracket_k^{\text{succ}(i)}$
- ${}_l\llbracket \phi_1 \mathcal{U} \phi_2 \rrbracket_k^i := {}_l\llbracket \phi_2 \rrbracket_k^i \vee ({}_l\llbracket \phi_1 \rrbracket_k^i \wedge {}_l\llbracket \phi_1 \mathcal{U} \phi_2 \rrbracket_k^{\text{succ}(i)})$

Encoding in case of Loop

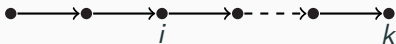


Definition (Encoding of an LTL formula for a (k, l) -loop)

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

- ${}_l\llbracket\mathbf{G}\phi_1\rrbracket_k^i := {}_l\llbracket\phi_1\rrbracket_k^i \wedge {}_l\llbracket\mathbf{G}\phi_1\rrbracket_k^{\text{succ}(i)}$
- ${}_l\llbracket\mathbf{F}\phi_1\rrbracket_k^i := {}_l\llbracket\phi_1\rrbracket_k^i \vee {}_l\llbracket\mathbf{F}\phi_1\rrbracket_k^{\text{succ}(i)}$

Encoding in case of NO Loops

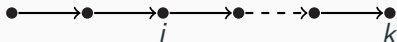


Definition (Encoding of an LTL formula for a loop-free path)

Let ϕ be an LTL formula and $i, k \geq 0$. We define $\llbracket \phi \rrbracket_k^i$ recursively as follows:

- $\llbracket \phi \rrbracket_k^{k+1} := \perp$

Encoding in case of NO Loops



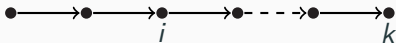
Definition (Encoding of an LTL formula for a loop-free path)

Let ϕ be an LTL formula and $i, k \geq 0$. We define $\llbracket \phi \rrbracket_k^i$ recursively as follows:

- $\llbracket p \rrbracket_k^i := p(s_i)$
- $\llbracket \neg p \rrbracket_k^i := \neg p(s_i)$

with $i \leq k$

Encoding in case of NO Loops



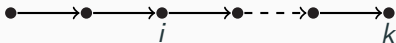
Definition (Encoding of an LTL formula for a loop-free path)

Let ϕ be an LTL formula and $i, k \geq 0$. We define $\llbracket \phi \rrbracket_k^i$ recursively as follows:

- $\llbracket \phi_1 \vee \phi_2 \rrbracket_k^i := \llbracket \phi_1 \rrbracket_k^i \vee \llbracket \phi_2 \rrbracket_k^i$
- $\llbracket \phi_1 \wedge \phi_2 \rrbracket_k^i := \llbracket \phi_1 \rrbracket_k^i \wedge \llbracket \phi_2 \rrbracket_k^i$

with $i \leq k$

Encoding in case of NO Loops



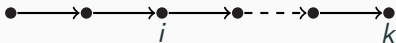
Definition (Encoding of an LTL formula for a loop-free path)

Let ϕ be an LTL formula and $i, k \geq 0$. We define $\llbracket \phi \rrbracket_k^i$ recursively as follows:

- $\llbracket X \phi_1 \rrbracket_k^i := \llbracket \phi_1 \rrbracket_k^{i+1}$
- $\llbracket \phi_1 \mathcal{U} \phi_2 \rrbracket_k^i := \llbracket \phi_2 \rrbracket_k^i \vee (\llbracket \phi_1 \rrbracket_k^i \wedge \llbracket \phi_1 \mathcal{U} \phi_2 \rrbracket_k^{i+1})$

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 $\llbracket \phi \rrbracket_k^i$ recursively as follows:

- $\llbracket \mathbf{G} \phi_1 \rrbracket_k^i := \llbracket \phi_1 \rrbracket_k^i \wedge \llbracket \mathbf{G} \phi_1 \rrbracket_k^{i+1}$
- $\llbracket \mathbf{F} \phi_1 \rrbracket_k^i := \llbracket \phi_1 \rrbracket_k^i \vee \llbracket \mathbf{F} \phi_1 \rrbracket_k^{i+1}$

with $i \leq k$

Definition (Overall encoding)

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

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

Definition (Overall encoding)

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

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

Theorem (Soundness)

$\llbracket \mathcal{M}, \phi \rrbracket_k$ 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: BMC is not complete;

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: BMC is not complete;
- BMC is mainly used as a bug finder, rather than as a prover.

Completeness

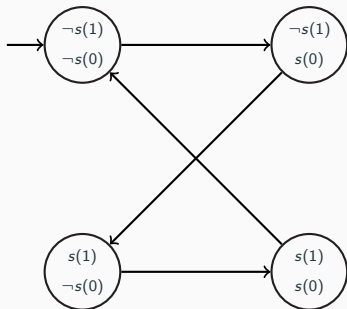
Algorithm:

- 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: BMC is not complete;
- BMC is mainly used as a bug finder, rather than as a prover.
- several techniques have been proposed to make BMC complete:
 - recurrence diameter
 - Simple Bounded Model Checking (SBMC)

modulo-4-counter.smv



- $\phi_1 := GF(s(0) \wedge s(1))$ ✓
- $\phi_2 := FG(\neg s(0) \wedge \neg s(1))$ ✗

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?

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:

$$\llbracket \mathcal{M} \rrbracket_k \wedge \left((\neg L_k \wedge \llbracket \phi \rrbracket_k^0) \vee \bigvee_{l=0}^k (l L_k \wedge l \llbracket \phi \rrbracket_k^0) \right)$$

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:

$$\llbracket \mathcal{M} \rrbracket_k \wedge \left((\neg L_k \wedge \llbracket \phi \rrbracket_k^0) \vee \bigvee_{l=0}^k (l L_k \wedge l \llbracket \phi \rrbracket_k^0) \right)$$

- satisfiability checking

$$\top \wedge \left((\neg L_k \wedge \llbracket \phi \rrbracket_k^0) \vee \bigvee_{l=0}^k (l L_k \wedge l \llbracket \phi \rrbracket_k^0) \right)$$

- remember the one-pass and tree-shaped tableau for LTL of the previous lessons?
- we developed this tool:

BLACK = one-pass tree-shaped tableau + BMC

- BLACK = Bounded Ltl sAtisfiability ChecKer ¹

¹ <https://github.com/black-sat/black>