Explicit-state model checking for LTL

Doina Bucur

RuG

Dec 2012
In this lecture...

Omega-acceptance and Büchi automata
- Standard finite-state automata with finite runs
- Some intuition for what follows
- FSA with infinite accepting runs: Büchi automata

Automata-based model checking
- Unified formalism: $\omega$-automata
- Model checking algorithm
- Graph searching algorithm

Historical and bibliographical notes

Assignment

In this lecture, you see how one of the main model checking algorithms works. It (i) rewrites the Kripke structure into an equivalent Finite-State Automata (FSA), and (ii) translates the LTL formula into an equivalent FSA (where an accepting word expresses a violation of the original formula). Model-checking is then simply checking whether the intersection of these two automata is empty.
(Explicit-state) model checking

Specification language: a temporal logic (LTL) formula, i.e. safety, liveness
Checking algorithm: exhaustive search of the state space of the finite-state system, to prove whether the specification is true or not

Advantages:
▶ The specification language expresses complex properties.
▶ Many model checking tools are implemented.
▶ The runtime is fast in many practical cases.
▶ Counterexamples are provided by many model checkers.

Disadvantages:
▶ The state-explosion problem remains present; it is caused by large number of threads, or large data space.
Alternative: Automata as specifications

You can also express temporal specifications as automata$^1$.

However, the complexity of expressing a specification with automata is higher than with temporal logics. The same goes for operating on a specification (think how hard it is to complement an automaton). This is why temporal logics are more widely used as specification formalisms.

$^1$Also, with other logics, calculi, regular expressions.
Why automata as specification language?

Nevertheless, automata specifications are central to the main checking algorithm for temporal-logic model checking. SPIN is one such case, and we see this in these two lectures.

Key idea: Liveness properties cannot be expressed by finite-state automata on finite accepted words. Thus, in order to write temporal properties, we have to update the classical definition of automata.
Standard finite-state automata

Definition (Finite-state automata, FSA)

A FSA \( \mathcal{A} \) is a tuple \((\Sigma, Q, \Delta, Q_0, F)\), where

- The state space \( Q \) is finite.
- The initial states are \( Q_0 \subseteq Q \), and the final states \( F \subseteq Q \).
- \( \Sigma \) is a finite alphabet of symbols.
- \( \Delta \) is the transition relation, \( \Delta \subseteq Q \times \Sigma \times Q \).

\( \mathcal{A} \) is called deterministic iff \( \forall q \in Q, \forall a \in \Sigma, ((q, a, q') \in \Delta \text{ and } (q, a, q'') \in \Delta) \rightarrow q' \text{ is } q'' \).

Here, \( \Sigma = \{a, b\} \).
Words and runs over FSA

Say \( v \) is a word (also: string or sequence) of length \( |v| \) over the alphabet \( \Sigma \) in the automaton \( \mathcal{A} \). The \( i \)th symbol in \( v \) is written \( v[i] \).

A run of \( \mathcal{A} \) over the word \( v \) is a sequence of states \( q_i \) starting in an initial state \( q_0 \in Q_0 \), so that a transition in \( \mathcal{A} \) leads from state \( q_i \) to state \( q_{i+1} \) over symbol \( v[i] \), i.e.

\[
\forall 0 \leq i < |v| \quad (q_i, v[i], q_{i+1}) \in \Delta.
\]

- We say that \( \mathcal{A} \) reads word \( v \) if a run of \( \mathcal{A} \) over \( v \) exists.
- In general, runs may be infinite.
Standard acceptance over FSA

Definition (Standard acceptance)

An accepting run of a FSA $A$ is a finite run which ends in a final state from set $F$.

$A$ accepts word $v$ iff there exists an accepting run of $A$ over $v$.

This FSA accepts the empty word, $a$, $ba$, $aba$, $aabba$, etc.

The language of $A$, denoted $L(A) \subseteq \Sigma^*$, is the set of all words accepted by $A$. Here, $L(A)$ is described by the regular expression $\epsilon | (a \mid b)^* aa^*$.\(^2\)

\(^2\)The * operator is the Kleene star, a unary operator meaning “zero or any finite number of”. | is the binary operator indicating choice.
Some intuition (1)

The FSA symbols $a \in \Sigma$ don’t have particular meaning in the theory of FSA; what is their meaning for us?

Remember: LTL formulas describe properties of infinite execution paths in Kripke structures with state labels. A state label is a $p \in AP$ which holds in that state; label $true$ will hold in any state, and label $false$ will hold in no state.
Some intuition (2)

A formula is true or false depending on the sequence of state labels on that path:

\[ X_p \]
\[ G_p \]
\[ F_p \]
\[ pUq \]

Let’s write these same sequences as the following equivalent forms:

- true $p$ true \(^{\text{infinitely many}}\)
- $p$ \(^{\text{infinitely many}}\)
- true \(^{\text{finitely many}}\) $p$ true \(^{\text{infinitely many}}\)
- $p$ \(^{\text{finitely many}}\) $q$ true \(^{\text{infinitely many}}\)
Some intuition (3)

These look exactly like infinite FSA runs, where the FSA symbols from $\Sigma$ are the atomic propositions from $AP$.

We can write small FSA having such a run. We also try to add accepting states; the original formula holds if the run reaches an accepting state:

- true $p$ true infinitely many

![Diagram of automaton for $X p$]

- $p$ infinitely many

![Diagram of automaton for $G p$]
Some intuition (4)

- True finitely many $p$ true infinitely many

true
\[ q_0 \xrightarrow{p} q_1 \]
automaton for $Fp$

- $p$ finitely many $q$ true infinitely many

$p$
\[ q_0 \xrightarrow{q} q_1 \]
automaton for $pUq$
Some intuition (5)

- Also, a negated mutual exclusion $\neg G \neg (p_0 \land p_1)$ is equivalent to $F(p_0 \land p_1)$ and is written as:

$$\neg (p_0 \land p_1)^\text{infinitely many}$$

Automaton for not mutual exclusion:
Towards FSA with infinite accepting runs

Standard FSA can only have finite accepting runs, but we are also interested in infinite accepting runs, such as $aaa\ldots$ (infinitely many $a$, or $a$ infinitely many).

An infinite run is often called omega-run, written $\omega$-run. For an infinite run $\sigma$:

- $\sigma^\omega$ denotes the set of states which appear infinitely often;
- $\sigma^+$ denotes the set of states which appear only finitely many times.
Büchi acceptance

Infinite runs over FSA need a new definition for acceptance; the most used one is Büchi acceptance.

Definition (Büchi acceptance)

An accepting $\omega$-run of FSA $A$ is any $\omega$-run $\sigma$ such that
\[ \exists q_f. \ q_f \in F \text{ and } q_f \in \sigma^\omega. \]

This means that a run which never ends is accepted iff it visits a final state infinitely many times.

This FSA Büchi-accepts the $\omega$-runs $aaa...$, $bababa...$, $bbb...ba$, etc.
Unifying standard and Büchi acceptance

For convenience, we need a way to also include standard acceptance into Büchi acceptance.

This is done by artificially extending finite runs into infinite runs. We extend the alphabet $\Sigma$ with a new “null” symbol $\epsilon$, to label a transition which can always execute.

**Definition (Stutter extension)**

The stutter extension of a finite run $\sigma$ with final state $q_n$ is the $\omega$-run $\sigma \cdot (q_n, \epsilon, q_n)^\omega$. \(^3\)

This makes the final state of the finite run persist forever. It is easily proven that standard acceptance with stutter extension equals Büchi acceptance.

---

\(^3\)A matter of notation: $\omega$ used as a superscript represents infinite repetition, and $\cdot$ represents concatenation.
Büchi automata

Accepting $\omega$-runs can always provably be written as
- A finite prefix regular expression executed once, e.g. $(ba)^3$, and
- A finite suffix regular expression repeated infinitely, e.g. $a^\omega$ or $\epsilon^\omega$.

These expressions are called $\omega$-regular expressions.
The class of properties they express is called $\omega$-regular properties.
Automata with accepting $\omega$-runs are called $\omega$-automata.
Automata with Büchi acceptance conditions are called Büchi automata$^4$.

---

$^4$There are other formalisms for $\omega$-acceptance and $\omega$-automata: Muller, Rabin, Streett acceptance and automata. They all express the same class of $\omega$-regular properties as Büchi automata.
Properties of Büchi automata

It has been shown that the following properties of Büchi automata are decidable:

- **Language emptiness**, i.e. whether the set of accepting runs of a given Büchi automaton is empty;
- **Language intersection**, i.e. generating a new Büchi automaton which accepts precisely those \( \omega \)-runs accepted by all Büchi automata from a given set.

Key idea in what follows: The **SPIN** model checking problem is equivalent to the emptiness test for an intersection of two given Büchi automata. This emptiness test boils down to a depth-first search algorithm.
**Spin use**

It is proven that for every LTL formula there exists a Büchi automaton which accepts precisely the runs which satisfy the formula.

**Spin** takes LTL formulas natively in **Promela**\(^5\); these LTL formulas are taken as **positive** properties that must be satisfied by the program.

**Spin** implements a LTL-to-Büchi translation, see e.g. `spin -f '[]p'` (not guaranteed to be minimal)\(^6\). A given LTL formula is first **negated** internally, then translated into an automaton written as a **never** claim with accept state labels. The existence of an accepting run in a never claim is signalled as a violation of the original LTL formula.

---

\(^5\) Have a look at the relevant manual page, [http://spinroot.com/spin/Man/ltl.html](http://spinroot.com/spin/Man/ltl.html).

\(^6\) The translation algorithm is quite complex; see Clarke’s Model Checking book, p. 132.
More examples

![Automaton for $\text{FG } p$](image1)

![Automaton for $\text{!FG } p$](image2)

![Automaton for $\text{G}(p \rightarrow \text{F } q)$](image3)

![Automaton for $\text{F}(p \land \text{G } !q)$](image4)
Automata-based model checking

Roadmap for automata-based model checking

Specification (LTL)

F (x=3 ∧ y=3) → ! F (x=3 ∧ y=3)

Intersection of system and negated specification

System (Kripke)

Intersection (i.e. synchronous product)

If nonempty, counterexample exists

Kripke structure Buchi automaton

Doina Bucur (RuG) LTL model checking Dec 2012 21 / 48
Kripke structures to $\omega$-automata (1)

Any Kripke structure $M$ has an equivalent $\omega$-automaton $A$

$M$ and $A$ are equivalent iff the sequence of state labels on any path $\pi$ in $M$ has a correspondent word in $\mathcal{L}(A)$.

Kripke structure | Buchi automaton
--- | ---
$s_0 \{x=0, y=0\}$ | $s_0 \{x=0, y=0\}$
$s_1 \{x=1, y=2\}$ | $s_1 \{x=1, y=2\}$
$s_2 \{x=2, y=3\}$ | $s_2 \{x=2, y=3\}$
$s_3 \{x=0, y=1\}$ | $s_3 \{x=0, y=1\}$
Kripke structures to $\omega$-automata (2)

The translation algorithm is simple:

1. Take Kripke structure $M = (S, S_0, T, L)$ where $L : S \rightarrow \mathcal{P}(AP)$.
2. Write the FSA $A = (\Sigma, Q, \Delta, Q_0, F)$, where:
   - The **alphabet** is the set of subsets of $AP$, i.e. $\Sigma := \mathcal{P}(AP)$.
   - We add an **initial state** $\nu$, i.e. $Q_0 := \{\nu\}$, $Q := S \cup \{\nu\}$ and $F := S \cup \{\nu\}$.
   - The new **transitions** are:
     - $(\nu, \alpha, s) \in \Delta$ iff $s \in S_0$ and $\alpha = L(s)$, and
     - $(s, \alpha, s') \in \Delta$ iff $s, s' \in S$, $(s, s') \in T$ and $\alpha = L(s')$. 
LTL formulas to $\omega$-automata (reminder)

![Diagram](attachment:image.png)

Specification $\neg F \ (x=3 \land y=3)$ as automaton

Note: The upper bound of the state space of the automata is **exponential** in the size of the LTL formula (i.e., the number of atomic propositions in the LTL formula)\(^7\).

---

\(^7\) [http://en.wikipedia.org/wiki/Linear_temporal_logic_to_Buchi_automaton](http://en.wikipedia.org/wiki/Linear_temporal_logic_to_Buchi_automaton)
Transition labels: some necessary clarifications

We wrote transition labels on FSA both as:

- subsets of $AP$, $\{x = 3, y = 3\}$ — this follows the definitions — and
- boolean formulas $x = 3 \land y = 3$.

Say that $AP = \{p, q, r, t\}$. When a transition is labelled with a boolean formula, this is in fact a short representation for a number of actual transitions:

- The label $p \lor q$ means any of the transitions labelled
  $\{p\}$ or $\{q\}$ or $\{p, q\}$ or $\{p, q, r\}$, etc.
- The label $p \land q$ means any of the transitions labelled
  $\{p, q\}$ or $\{p, q, r\}$ or $\{p, q, t\}$, etc.
- The label $\neg p$ means any of the transitions labelled
  $\{q\}$ or $\{q, r\}$ or $\{q, r, t\}$, etc.
- The label true means any transition.
Now to automata-based model checking:
Basic idea of the algorithm

▶ Any **temporal specification** can be represented as an \( \omega \)-automaton \( S \) (as we have seen last time).
▶ The **system model**, also: a Kripke structure directly corresponds to an \( \omega \)-automaton \( A \) (as we have just seen now).
▶ The system \( A \) satisfies the positive property \( S \) when

\[
\mathcal{L}(A) \subseteq \mathcal{L}(S)
\]
Satisfaction of a specification

The system $\mathcal{A}$ satisfies a positive property $S$ (both as $\omega$-automata) when

$$\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(S)$$

i.e., each system execution is among the behaviours accepted by the specification.

If we define the complement of language $\mathcal{L}(S)$ to be

$$\overline{\mathcal{L}(S)} := \Sigma^\omega \setminus \mathcal{L}(S)$$

we can write the satisfaction of a specification in terms of intersection of languages:

$$\mathcal{L}(\mathcal{A}) \cap \overline{\mathcal{L}(S)} = \emptyset$$
Model-checking procedure

As already suggested:

1. Take a specification as automaton $S$. This describes a **positive** property which should not be violated.
   - ($$\text{SPIN}$$ takes this either in positive LTL form, or directly in negative Buchi automaton form.)

2. **Complement** automaton $S$, i.e. construct an automaton $\overline{S}$ which accepts the language $\overline{L(S)}$.
   - ($$\text{SPIN}$$ calculates this complement automaton directly out of the positive LTL formula above, by first negating it, and then translating this negation into an automaton $\overline{S}$.)

3. Construct the automaton which accepts the **intersection** of languages $L(A)$ and $L(S)$.

4. If this intersection is empty, $S$ **holds** for $A$.

5. Otherwise, a **counterexample** is provided. An infinite counterexample (i.e. an $\omega$-run) is expressed finitely as $uv^\omega$, with $u$ a finite prefix and $v$ a finite suffix (as on slide 17).
Example (step 1)

System $\mathcal{A}$ with positive liveness specification $\mathcal{S}$ in LTL:

$$\mathcal{F}(x = 3 \land y = 3)$$
Example (step 2)

System $\mathcal{A}$ with **negative** specification $\bar{S}$ as automaton:

$$
\begin{align*}
\mathcal{A} & : (s_0) \\
& \quad \xrightarrow{x=0, y=0} (s_1) \\
& \quad \xrightarrow{x=1, y=2} (s_2) \\
& \quad \xrightarrow{x=2, y=3} (s_3) \\
& \quad \xrightarrow{x=3, y=3} (q_0)
\end{align*}
$$

System as automaton

$F(x = 3 \land y = 3)$ is negated as $\neg F(x = 3 \land y = 3)$ which is translated as:

$$
\neg (x = 3 \land y = 3)
$$

Specification $\neg F (x = 3 \land y = 3)$ as automaton
Calculating the intersection of two automata

Take two FSA,

- $B_1 = (\Sigma, Q_1, \Delta_1, Q_0^1, F_1)$ and
- $B_2 = (\Sigma, Q_2, \Delta_2, Q_0^2, F_2)$.

We want $B_1 \cap B_2$ which accepts $\mathcal{L}(B_1) \cap \mathcal{L}(B_2)$. We give an algorithm which is not general\(^8\), but is sufficient when all states of one automaton are accepting (here, say the first automata $B_1$ has $Q_1 = F_1$).

$B_1 \cap B_2 := (\Sigma, Q := Q_1 \times Q_2, \Delta, Q_0 := Q_0^1 \times Q_0^2, F := Q_1 \times F_2)$,

where for any symbol $a \in \Sigma$,

an edge $((r_i, q_j), a, (r_m, q_n)) \in \Delta$ iff

$(r_i, a, r_m) \in \Delta_1$ and $(q_j, a, q_n) \in \Delta_2$.

---

\(^8\)For the general algorithm, and information on complementing Büchi automata, see Ch. 9, *Model Checking and Automata Theory*, from *Model Checking*, E. M. Clarke, O. Grumberg, D. A. Peled.
Example (step 3)

Some number-crunching for the calculation of the intersection in this example:

\[
\Sigma := \mathcal{P}\{x = 0, x = 1, x = 2, x = 3, y = 0, y = 1, y = 2, y = 3\}
\]

\[
Q := \{\iota, s_0, s_1, s_2, s_3\} \times \{q_0\}
\]

\[
Q_0 := \{\iota, q_0\}
\]

\[
F := Q
\]
Example (result from step 3)

And the intersection $\mathcal{A} \cap \overline{S}$ is:

Intersection of system and negated specification
Example (step 2 again, for comparison)

System $\mathcal{A}$ with **negative** specification $\overline{S}$ as automaton:

![Automaton Diagram]

$F(x = 3 \land y = 3)$ is negated as $\neg F(x = 3 \land y = 3)$
which is translated as:

Specification $! (x = 3 \land y = 3)$ as automaton
Emptiness of language

Reminder—steps 4 and 5:

4. If this intersection is empty, $S$ holds for $A$.

5. Otherwise, a counterexample is provided. An infinite counterexample (i.e. an $\omega$-run) is expressed finitely as $uv^\omega$, with $u$ a finite prefix and $v$ a finite suffix.

A language $L(B)$ is **empty** if $B$ has no accepting $\omega$-runs.

You can then reason that:

A language $L(B)$ is **nonempty** iff there exists a reachable accepting state with a cycle back to itself.
Example (step 4)
The intersection $A \cap \overline{S}$ is...

Intersection of system and negated specification

... nonempty. Three accepting states exist with a cycle back. This means that the original specification $F(x = 3 \land y = 3)$ does not hold.
And finally, the counterexample?

This algorithm is constructed in this way so that the **counterexample** is exactly that run reaching an accepting state, then cycling back to it.

The nonemptiness test thus solves both steps 4 and 5.

This test can be implemented as a **depth-first search algorithm** (DFS), with two searches: one for any accepting state, and the other for a cycle back to that state. The resulting counterexample will not be the shortest.
Example (step 5)

An accepting run and counterexample is:

\[(\iota, q_0), \{x = 0, y = 0\}, (s_0, q_0)\],
\[(s_0, q_0), \{x = 1, y = 2\}, (s_1, q_0)\],
\[(s_1, q_0), \{x = 2, y = 3\}, (s_2, q_0)\],
\[(s_2, q_0), \{x = 0, y = 1\}, (s_3, q_0)\],
\[(s_3, q_0), \{x = 1, y = 2\}, (s_1, q_0)\)]^\omega.

Intersection of system and negated specification
The double depth-first search algorithm, $O(|Q| + |\Delta|)$

\(\text{nonemptiness()}\)

\(\text{for all } q_0 \in Q_0\)
\(\quad \text{dfs1}(q_0)\)
\(\text{return } \text{False, i.e. no counterexample exists}\)

\(\text{dfs1}(q)\)
\(\quad \text{local } q'\)
\(\quad \text{hash}(q)\)
\(\quad \text{for all successors } q' \text{ of } q\)
\(\quad \quad \text{if } q' \text{ not in the hash table then } \text{dfs1}(q')\)
\(\quad \quad \text{if } \text{accept}(q) \text{ then } \text{dfs2}(q)\)

\(\text{dfs2}(q)\)
\(\quad \text{local } q'\)
\(\quad \text{flag}(q)\)
\(\quad \text{for all successors } q' \text{ of } q\)
\(\quad \quad \text{if } q' \text{ on dfs1 stack then return True, i.e. counterexample found}\)
\(\quad \quad \text{else if } q' \text{ not flagged then } \text{dfs2}(q')\)
Model-checking “on-the-fly”

This model checking procedure allows to skip on generating the entire model $A$ from a (concurrent) system at each verification run.

We can instead generate the specification automaton $S$, and compute the intersection while also doing the nonemptiness check. This may result in constructing only a small portion of the Kripke structure before a counterexample is found.

*Final note:* this depth-first search (DFS) can also be implemented as a breadth-first search (BFS). However, a DFS algorithm has the great advantage that the counterexample searched is exactly the trace of states analyzed by the algorithm.
End of roadmap for automata-based model checking

Specification (LTL)
\[ F(x=3 \land y=3) \quad \neg F(x=3 \land y=3) \]

System (Kripke)

Intersection (i.e. synchronous product)

Intersection of system and negated specification

If nonempty, counterexample exists
The **Spin** implementation

- **Spin** cuts corners compared to the algorithm above, when given a safety specification $S$ (assertions, deadlock, invariants):

  Spin safety checking is simply a standard DFS algorithm over the system model $A$ itself, where at each state, the state labels are evaluated with regard to $S$.

- **Spin** can also take a **depth bound** for the DFS search. All guarantees of finding a violation of the safety specification disappear now, even within the depth bound. To solve this, it also implements breadth-first search (BFS), which brings back the guarantees within the depth bound.
Historical and bibliographical notes

Historical notes

The basic theory of finite automata was developed in the 50s. The theory of $\omega$-automata is almost as old: Büchi’s work started in 1960. The correspondence between temporal logic and Büchi automata was first described in 1983.

The first DFS algorithm linear in the size of the graph is due to Tarjan (1972). The nested DFS algorithm dates from 1990.
Bibliographical notes

The content in these lectures covers Ch. 9, *Model Checking and Automata Theory*, from *Model Checking*, E. M. Clarke, O. Grumberg, D. A. Peled.

This covers the automata-related text from Ch. 6, *Automata and Logic*, and Ch. 8, *Search Algorithms*, from *The SPIN Model Checker*, G. J. Holzmann.

This subject is described in Ch. 4, *Regular Properties*, and Section 5.2, *Automata-Based LTL Model Checking* from *Principles of Model Checking*, C. Baier, J.-P. Katoen.
The Pathfinder released a robot to roam on Mars in 1997. The robot’s software controls occasionally failed during the mission, causing loss of contact. The essence of the problem (an unseen conflict between prioritizing two threads and mutually excluding them) was modelled in Promela (code in the Spin sources, Test/pathfinder.pml). The thread priorities are modelled with a provided (...)\(^9\), and the allowed end states with end labels\(^{10}\).

Run Spin and display the trail to a deadlock. Draw the Kripke structure (the program has 3 variables) and check whether it is the only possible deadlock state. Explain why any deadlock states can appear.

\(^9\)See the manual, http://spinroot.com/spin/Man/provided.html
\(^{10}\)See the manual, .../Man/end.html
Assignment (2)

(25%) **PROMELA** never claims express negative properties as FSA; check the manual .../Man/never.html. Final states have accept labels, .../Man/accept.html; when these are reached, the program has matched the FSA and a violation was found.

Do [LMC’11] Assignment 4 (on Nestor under Course Material), points 2 and 3 (you can use spin -f and LTL to either generate or check the claims at point 2).
Assignment (3)

(50%) The Alternating-bit protocol (ABP)\(^{11}\) is a simple data-exchange network protocol between a sender A and a receiver B on a shared line, using retransmission of lost or corrupted packets. **Model** the protocol, both with a perfect communication medium and with a lossy one\(^{12}\). Simulate both. **Specify** at least two behaviour(s) for this protocol, for example: Regardless of the number of errors introduced by the medium, every message sent by A is received error-free at least once; It is accepted at most once by B; Messages are not reordered; There are no non-progress cycles. **Verify** these on both types of channels and explain the results; you can use assertions, LTL, progress labels, etc. Notes: Channel operations in Promela are reviewed under e.g. .../Man/chan.html, etc. You can check progress with either LTL or progress labels, .../progress.html. You may try Promela’s remote references .../remoterefs.html to check the state of a process.


\(^{12}\)Existing models at .../Man/Exercises.html, Ex. 2, and the Spin sources Samples/p123.pml
Assignment administration

Handing in:

▶ in lab session or electronically;
▶ either way, before the end of Fri Dec 21; that’s about 10 days.

Grading:

▶ this is worth 25% of your Practical grade (and 15% of the final grade).