LTL Model Checking in RISCAL

Master Thesis Topic

Sütő Ágoston

Research Institute for Symbolic Computation

Thesis supervisor: Prof. Wolfgang Schreiner

November 29, 2021

Introduction

Model checking is a method used for checking whether a system meets a given specification

RISCAL is a software used to model algorithms (deterministic and non-deterministic)

LTL is a logic that allows us to talk about the future of paths

Several methods for LTL model checking exist

The goal of the thesis is to research these methods and choose one to implement it as an extension to RISCAL

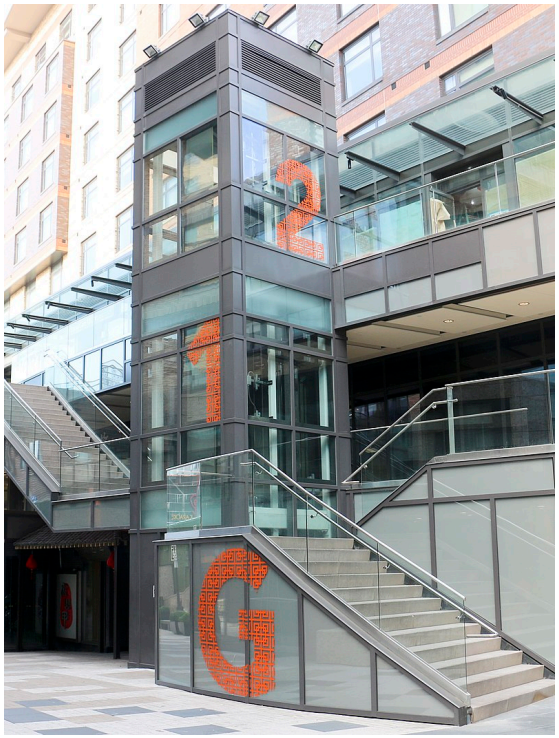
Concurrency

Non-deterministic system: may exhibit different behaviours on different runs given the same input.

Concurrent system: can execute several computations at the same time.

Due to race conditions, concurrency might lead to non-determinism.

Concurrent systems in real life



Downtown Flushing By Chris Hamby, CC BY-SA 2.0

System consisting of the mechanism responsible for moving the cabin and the mechanism for controlling the cabin door. We would expect it to satisfy:

Safety properties: cabin never moves when the doors are open.

Liveness properties: whenever the call button on a certain floor was pressed, the cabin will eventually stop there, and open the door.

The problem

There is a rather large body of sad experience to indicate that a concurrent program can withstand very careful scrutiny without revealing its errors.[1]

[1] S. Owicki and L. Lamport. *Proving liveness properties of concurrent programs*

RISCAL

- language and associated software system
- developed at RISC by Prof. Wolfgang Schreiner
- used to describe mathematical algorithms over finite structures
- used as a preliminary step in computer assisted proofs
- limited support for non-determinism: "shared" and "distributed" systems

Non-determinism in RISCAL

```
val N: ℕ; axiom minN ⇔ N ≥ 4;
type elem = ℕ[N];

shared system S1
{
    var x: elem = 0;
    var y: elem = 0
    init(a:elem) with a > N / 2;
        { x := a; y := N - a; }
    action incx() with x < N;
        { x := x + 1; y := y - 1; }
    action decx() with x > y + 1;
        { x := x - 1; y := y + 1; }
    action swap() with x > y + 1;
        { var tmp := x; x := y; y := tmp; }
}
```

Non-determinism in RISCAL

What we can do: check if a property holds in all states using "invariants".

E.g.: `invariant x + y = N;`

Non-determinism in RISCAL

What we can do: check if a property holds in all states using "invariants".
E.g.: `invariant x + y = N;`

What we cannot do: check properties of runs. E.g. "$x > y$ until the action `swap` is taken"

It's not immediately clear how to specify this constraint.

Kripke structures

Definition

A Kripke-structure K over a set of atomic propositions $\mathcal{A}$ is defined as the tuple $(S, I, T, \mathcal{L})$ consisting of the following components:

- a set of states S
- a set of initial states $I \subseteq S$, $I \neq \emptyset$
- a total transition relation $T \subseteq S \times S$
- a labelling function $\mathcal{L}\colon S \to \mathbb{P}(\mathcal{A})$

Definition

A *trace* π is a finite or infinite sequence of states of a Kripke-structure, such that $\forall i\colon s_i \xrightarrow{T} s_{i+1}$

$$\pi = (s_0, s_1, \ldots)$$

Linear Temporal Logic

The alphabet of LTL consists of atomic propositions $\mathcal{A}$, the standard logical operators ($\neg$, $\vee$, $\wedge$ etc.) and special temporal operators **X** (next), **F** (finally), **G** (globally), and **U** (until). The language of LTL formulas is defined inductively as follows:

- If p is an atomic proposition, then it is an LTL formula
- If g and h are LTL formulas, then $\neg g$, $g \vee h$, $g \wedge h$ etc. are LTL formulas
- If g and h are LTL formulas, then $\mathbf{X}g$, $\mathbf{F}g$, $\mathbf{G}g$, and $g\,\mathbf{U}\,h$ are LTL formulas.

Linear Temporal Logic

The semantics of LTL for infinite paths π of a Kripke-structure $K = (S, I, T, \mathcal{L})$ are defined as follows:

$$
\begin{array}{lll}
\pi \models p & \text{iff} & p \in \mathcal{L}(\pi(0)) \\
\pi \models \neg g & \text{iff} & \pi \not\models g \\
\pi \models g \vee h & \text{iff} & \pi \models g \text{ or } \pi \models h \\
\pi \models g \wedge h & \text{iff} & \pi \models g \text{ and } \pi \models h \\
\pi \models \mathbf{X} g & \text{iff} & \pi^1 \models g \\
\pi \models \mathbf{F} g & \text{iff} & \exists\, i \in \mathbb{N} \colon \pi^i \models g \\
\pi \models \mathbf{G} g & \text{iff} & \forall\, i \in \mathbb{N} \colon \pi^i \models g \\
\pi \models g \,\mathbf{U}\, h & \text{iff} & \exists\, i \in \mathbb{N} \colon \pi^i \models h \wedge \forall\, j \in \mathbb{N} \colon j < i \rightarrow \pi^j \models g
\end{array}
$$

LTL examples

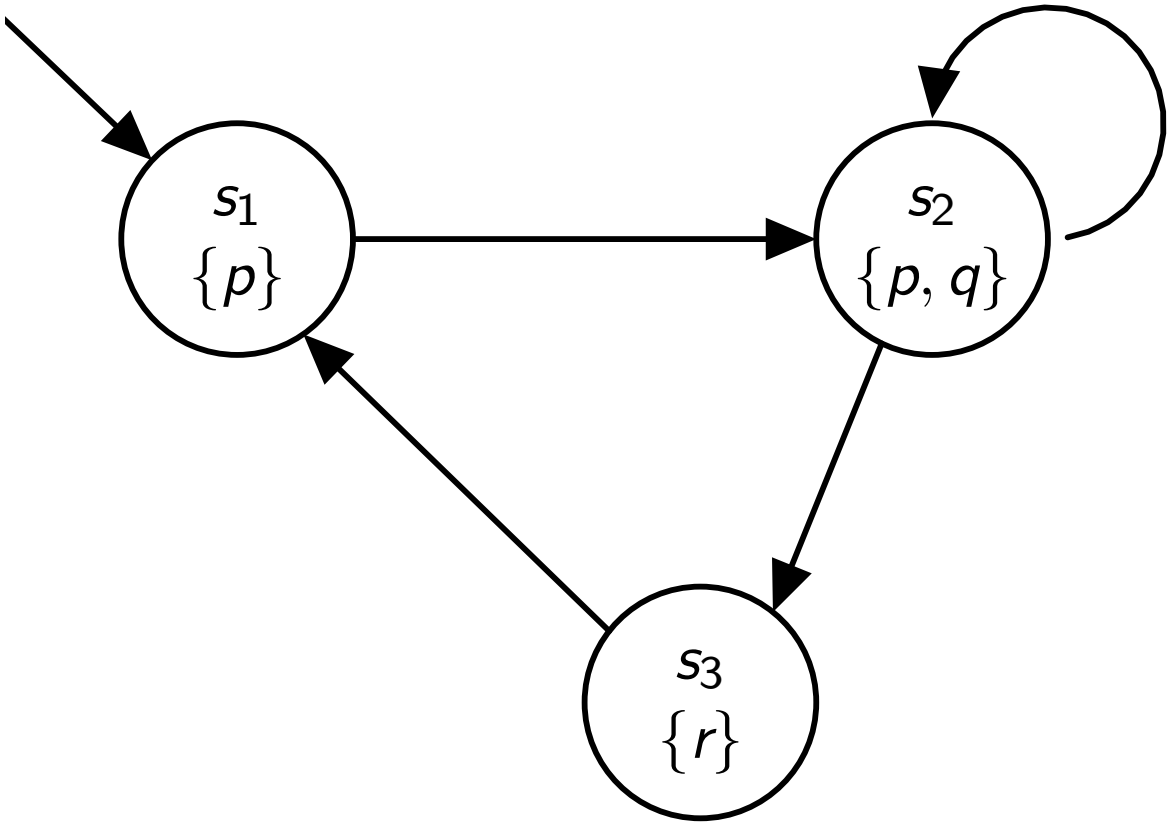


Figure: Example of a Kripke structure

Model checking

Goal: given a model of a program as a Kripke structure check if any given LTL formula holds for the system

Solution: use automaton and graph theory (only one of many possible approaches)

Model checking

Goal: given a model of a program as a Kripke structure check if any given LTL formula holds for the system

Solution: use automaton and graph theory (only one of many possible approaches)

1. Construct an automaton S_A from the system whose language $\mathcal{L}(S_A)$ describes all system runs

Model checking

Goal: given a model of a program as a Kripke structure check if any given LTL formula holds for the system

Solution: use automaton and graph theory (only one of many possible approaches)

1. Construct an automaton S_A from the system whose language $\mathcal{L}(S_A)$ describes all system runs
2. Construct an automaton P_A from the LTL property whose language $\mathcal{L}(P_A)$ describes runs satisfying the formula

Model checking

Goal: given a model of a program as a Kripke structure check if any given LTL formula holds for the system

Solution: use automaton and graph theory (only one of many possible approaches)

1. Construct an automaton S_A from the system whose language $\mathcal{L}(S_A)$ describes all system runs
2. Construct an automaton P_A from the LTL property whose language $\mathcal{L}(P_A)$ describes runs satisfying the formula
3. Check $\mathcal{L}(S_A) \subseteq \mathcal{L}(P_A)$

Transforming the system

Given a Kripke-structure $K = (S, I, T, \mathcal{L})$ we have to construct an automaton such that:

If $s_0 \to s_1 \to s_2 \to \ldots$ is a run of the system

Then $\iota \xrightarrow{\mathcal{L}(s_0)} s_0 \xrightarrow{\mathcal{L}(s_1)} s_1 \xrightarrow{\mathcal{L}(s_2)} s_2 \to \ldots$ is accepted by the automaton.

Transforming the system

Given a Kripke-structure $K = (S, I, T, \mathcal{L})$ we have to construct an automaton such that:

If $s_0 \to s_1 \to s_2 \to \ldots$ is a run of the system

Then $\iota \xrightarrow{\mathcal{L}(s_0)} s_0 \xrightarrow{\mathcal{L}(s_1)} s_1 \xrightarrow{\mathcal{L}(s_2)} s_2 \to \ldots$ is accepted by the automaton.

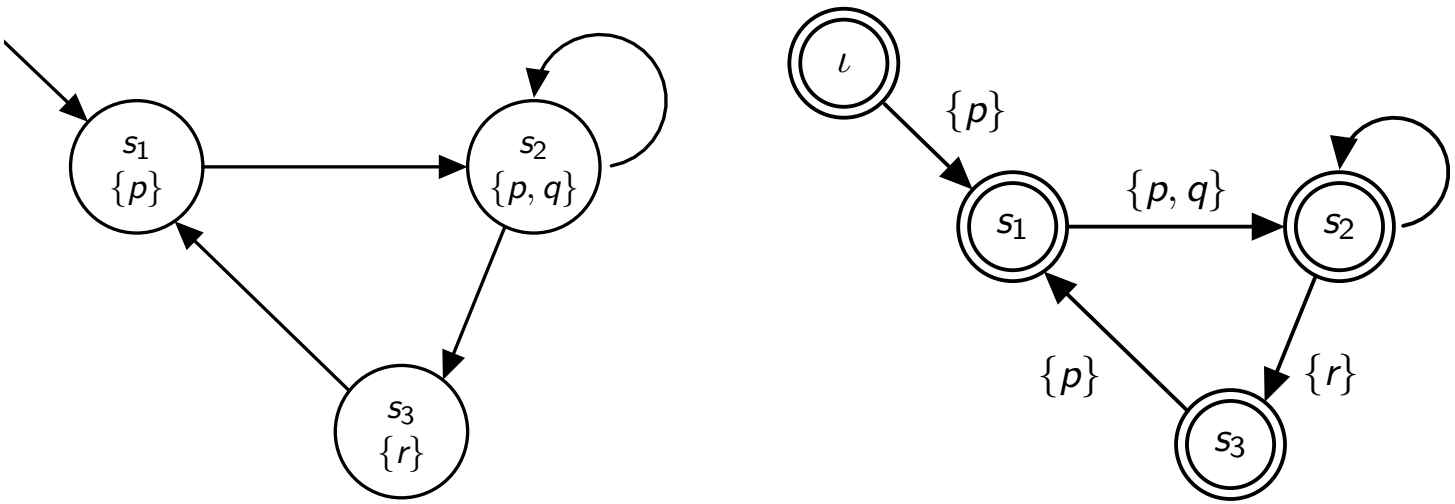


Figure: Kripke structure and corresponding automaton

Transforming the formula

Given an LTL formula f we have to construct an automaton such that

If $(p_0, p_1, p_2, ...) \models f$

Then $s_0 \xrightarrow{p_0} s_1 \xrightarrow{p_1} s_2 \xrightarrow{p_2} s_3 \rightarrow ...$ is accepted by the automaton.

Transforming the formula

Given an LTL formula f we have to construct an automaton such that

If $(p_0, p_1, p_2, ...) \models f$

Then $s_0 \xrightarrow{p_0} s_1 \xrightarrow{p_1} s_2 \xrightarrow{p_2} s_3 \rightarrow ...$ is accepted by the automaton.

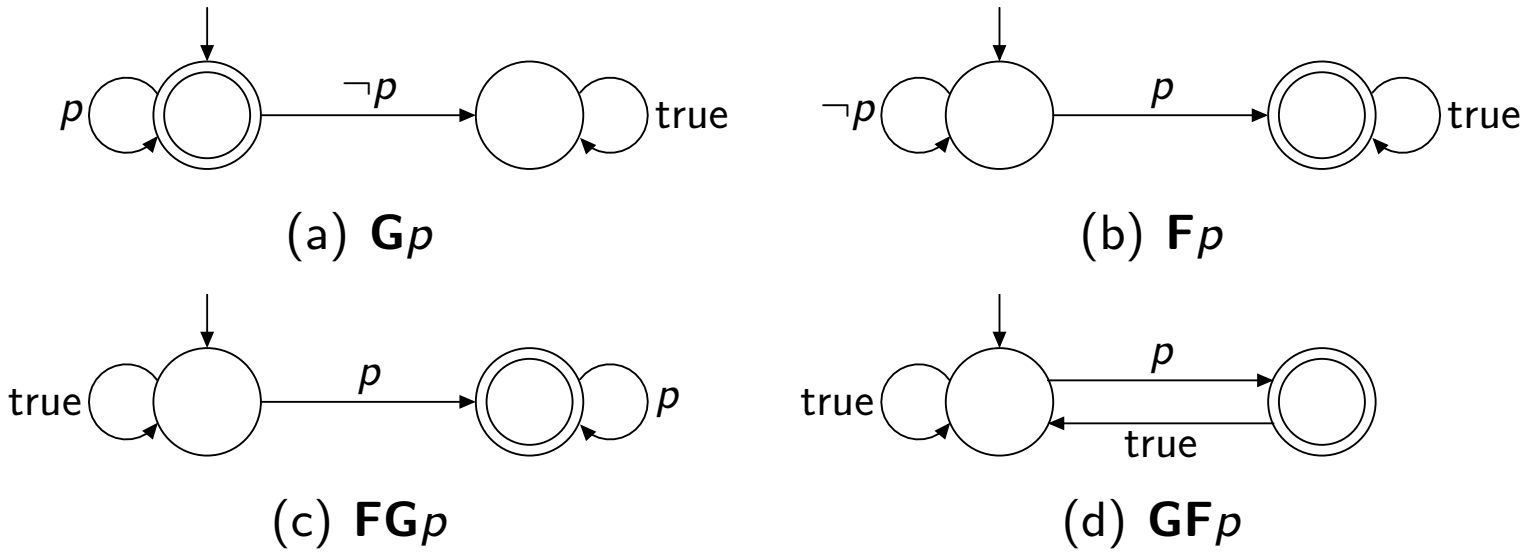


Figure: Examples

The next steps

The original problem was to check if $\mathcal{L}(S_A) \subseteq \mathcal{L}(P_A)$.

$$\Leftrightarrow \mathcal{L}(S_A) \cap \overline{\mathcal{L}(P_A)} = \emptyset$$
$$\Leftrightarrow \mathcal{L}(S_A) \cap \mathcal{L}((\neg P)_A) = \emptyset$$

The next steps

The original problem was to check if $\mathcal{L}(S_A) \subseteq \mathcal{L}(P_A)$.

$$\Leftrightarrow \mathcal{L}(S_A) \cap \overline{\mathcal{L}(P_A)} = \emptyset$$
$$\Leftrightarrow \mathcal{L}(S_A) \cap \mathcal{L}((\neg P)_A) = \emptyset$$

An equivalent problem is to check if $\mathcal{L}(S_A) \cap \mathcal{L}((\neg P)_A) = \emptyset$

The synchronized product automaton of two automata A and B, $A \otimes B$, has the property $\mathcal{L}(A) \cap \mathcal{L}(B) = \mathcal{L}(A \otimes B)$

The next steps

The original problem was to check if $\mathcal{L}(S_A) \subseteq \mathcal{L}(P_A)$.

$$\Leftrightarrow \mathcal{L}(S_A) \cap \overline{\mathcal{L}(P_A)} = \emptyset$$
$$\Leftrightarrow \mathcal{L}(S_A) \cap \mathcal{L}((\neg P)_A) = \emptyset$$

An equivalent problem is to check if $\mathcal{L}(S_A) \cap \mathcal{L}((\neg P)_A) = \emptyset$

The synchronized product automaton of two automata A and B, $A \otimes B$, has the property $\mathcal{L}(A) \cap \mathcal{L}(B) = \mathcal{L}(A \otimes B)$

Its states consist of pairs (s_A, s_B) where s_A and s_B are states of A and B, and a transition $(s_A, s_B) \to (s'_A, s'_B)$ is possible if $s_A \to s'_A$ and $s_B \to s'_B$ is possible in A and B

The next steps

The original problem was to check if $\mathcal{L}(S_A) \subseteq \mathcal{L}(P_A)$.

$\Leftrightarrow \mathcal{L}(S_A) \cap \overline{\mathcal{L}(P_A)} = \emptyset$

$\Leftrightarrow \mathcal{L}(S_A) \cap \mathcal{L}((\neg P)_A) = \emptyset$

An equivalent problem is to check if $\mathcal{L}(S_A) \cap \mathcal{L}((\neg P)_A) = \emptyset$

The synchronized product automaton of two automata A and B, $A \otimes B$, has the property $\mathcal{L}(A) \cap \mathcal{L}(B) = \mathcal{L}(A \otimes B)$

Its states consist of pairs (s_A, s_B) where s_A and s_B are states of A and B, and a transition $(s_A, s_B) \to (s'_A, s'_B)$ is possible if $s_A \to s'_A$ and $s_B \to s'_B$ is possible in A and B

The final problem is then to check if $\mathcal{L}(S_A \otimes (\neg P)_A) = \emptyset$

Checking emptiness

- In our case (automata with infinite words = Büchi automata) a word r is accepted if it contains infinitely many occurences of an accepting state s

Checking emptiness

- In our case (automata with infinite words = Büchi automata) a word r is accepted if it contains infinitely many occurences of an accepting state s
- Since the state space is finite, it must contain a cycle $s \to \ldots \to s$

Checking emptiness

- In our case (automata with infinite words = Büchi automata) a word r is accepted if it contains infinitely many occurences of an accepting state s
- Since the state space is finite, it must contain a cycle $s \to \ldots \to s$
- Which means we have too look for a *reachable* accepting state also reachable from itself

Checking emptiness

- In our case (automata with infinite words = Büchi automata) a word r is accepted if it contains infinitely many occurences of an accepting state s
- Since the state space is finite, it must contain a cycle $s \to \ldots \to s$
- Which means we have too look for a *reachable* accepting state also reachable from itself
- Finding such a state s together with the path to it from an initial state and the cycle is the same as finding a counterexample to the original LTL formula in the system

Checking emptiness

- In our case (automata with infinite words = Büchi automata) a word r is accepted if it contains infinitely many occurences of an accepting state s
- Since the state space is finite, it must contain a cycle $s \to \ldots \to s$
- Which means we have too look for a *reachable* accepting state also reachable from itself
- Finding such a state s together with the path to it from an initial state and the cycle is the same as finding a counterexample to the original LTL formula in the system
- The problem is then reduced to a simple graph-theoretical problem, which can be solved efficiently using depth-first search

Implementation in RISCAL

- `"satisfies" <LTL>` clause added to RISCAL for non-deterministic systems
- also fairness constraints (next presentation)
- rich LTL language, including also quantifiers, variable bindings, weak until
- first step: transform the input to a simple form of LTL
- then construct property automaton
- system automaton is constructed ”on the fly”

Thesis structure

- Introduction (∼7 pages)
- State of the art (∼22 pages)
  - Tableau-based explicit-state model checking
  - Symbolic model checking
  - Bounded model checking
- Automata-based model checking (∼15 pages)
- Implementation (∼15 pages)
- Optimizations (∼12 pages)
- Results and measurements (∼10 pages)
- Future work and conclusions (∼5 pages)

Time schedule

October - November 2021 thesis proposal, reading about automata based model checking

December 2021 - January 2022 learning about other model checking approaches, state of the art chapter

February - April 2022 basic implementation, automata based model checking and implementation chapters

May 2022 implementing and describing optimizations, results and measurements chapter

June 2022 writing the future work and conclusions chapters, the introduction and finalizing the thesis

July 2022 defense of the thesis and final examination.

Bibliography

[1] Mordechai Ben-Ari. *Principles of the Spin Model Checker*. Springer Science & Business Media London, 2008. ISBN: 978-1-84628-769-5.

[2] Armin Biere et al. "Symbolic Model Checking without BDDs". In: *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 1999*. Springer, Berlin, Heidelberg, 1999, pp. 193–207.

[3] Edmund M. Clarke et al. *Handbook of Model Checking*. Springer, Cham, 2018. ISBN: 978-3-319-10575-8.

[4] Leslie Lamport. *Specifying Systems*. The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, 2002. ISBN: 0-321-14306-X.

[5] Kenneth L. McMillan. *Symbolic model checking*. Springer, Boston, 1993. ISBN: 978-0-7923-9380-1.

[6] Moshe Y. Vardi. "An automata-theoretic approach to linear temporal logic". In: *Logics for Concurrency: Structure versus Automata*. Springer-Verlag, 1996. ISBN: 978-3-540-60915-5.