A Temporal Logic Extension of the RISCAL Model Checker

Wolfgang Schreiner, Ágoston Sütő

Research Institute for Symbolic Computation

Johannes Kepler University, Linz

November 23, 2022

Introduction

- Our paper describes the extension of RISCAL with model checking capabilities for concurrent systems
- Model checking is a method used for verifying whether a finite model of a system meets a given specification
- The systems are non-deterministic and the specification is formulated using a variant of temporal logic, such as linear temporal logic (LTL)
- Various model checking approaches exist, our implementation is an automaton-based explicit model checker, which consists of the following main components:
  1. translation of LTL formulas to automata,
  2. on-the-fly expansion of the product automaton of the system and the formula in search of violations,
  3. validation of violations against fairness constraints

RISCAL

Figure: The RISCAL GUI

- Developed at the JKU by Wolfgang Schreiner et al., freely available
- Used to describe algorithms over finite structures, and to automatically check verification conditions (e.g. loop invariants) before attempting proof-based verification
- Intended primarily for didactic purposes
- Extended to support concurrent systems and to check their invariants
- More about RISCAL in the manual: (Schreiner, 2021)

Mutual exclusion modelled in RISCAL

```
val N: ℕ;
axiom minN ⇔ N ≥ 1;
type Proc = ℕ[N−1];

shared system S
{
    var critical: Array[N, Bool] = Array[N, Bool](⊥);
    var next: ℤ[−1, N] = 0;

    invariant 0 ≤ next ∧ next < N;
    invariant ∀i1: Proc, i2: Proc. critical[i1] ∧ critical[i2] ⇒ i1 = i2;

    ltl ∀i1: Proc, i2: Proc. □⟦ critical[i1] ∧ critical[i2] ⇒ i1 = i2 ⟧;
    ltl[fairness] ∀i: Proc. □◊⟦ next = i ⟧;
    ltl[fairness] ∀i: Proc. □◊⟦ critical[i] ⟧;

    action arbiter() with ∀j: Proc. ¬critical[j];
        fairness strong;
    { next := if next = N − 1 then 0 else next + 1; }

    action enter(i: Proc) with i = next ∧ ∀j: Proc. ¬critical[j];
        fairness strong_all;
    { critical[i] := ⊤; }

    action exit(i: Proc) with critical[i];
    { critical[i] := ⊥; }
}
```

Basic concepts

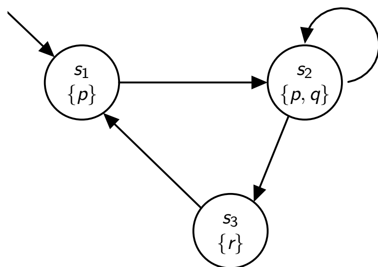

Figure: *Kripke structure K* modelling a non-deterministic system

LTL formulas which hold for the system:

- $K \models p$
- $K \models \mathbf{X}\, q$
- $K \models \mathbf{G}\neg(r \wedge p)$
- $K \models (p\,\mathbf{U}\, r) \vee (\mathbf{G} p)$

and some, which do not:

- $K \not\models \mathbf{F}(r \wedge p)$
- $K \not\models p\,\mathbf{U}\, r$

Definition

Model checking problem

Given a Kripke-structure $K = (S, I, T, \mathcal{L})$ and an LTL formula f determine whether $K \models f$, and if not, provide a trace π of K such that $\pi \not\models f$.

Labelled Büchi automata

Definition

A *labelled generalized Büchi automaton (LGBA)* is defined as the tuple $(S, I, \Sigma, \mathcal{L}, T, \mathcal{F})$ consisting of the following components:

- a finite set of states S
- a set of initial states $I \subseteq S$, $I \neq \emptyset$
- an input alphabet Σ
- a labelling of the states $\mathcal{L} \colon S \to 2^{\Sigma}$
- a transition relation $\to \subseteq S \times S$
- set of accepting sets $\mathcal{F} \subseteq 2^{S}$, $\mathcal{F} = \{F_1, F_2, ..., F_n\}$.

Definition

A Büchi automaton $\mathcal{A}$ *accepts* a word $w = a_0 a_1 a_2 ... \in \Sigma^{\omega}$ if there exists $\sigma = s_0 s_1 s_2 ... \in S^{\omega}$ such that for each $i \geq 0$, $a_i \in \mathcal{L}(s_i)$, $s_0 \in I$, $s_i \to s_{i+1}$, and for each acceptance set $F_j \in \mathcal{F}$ there exists at least one state $s_j \in F_j$ which appears infinitely often in σ.

The LTL to Büchi automaton algorithm

- Preprocessing:
  - Introduce new temporal operator $\mathbf{V}$, defined as the dual of $\mathbf{U}$: $f \mathbf{V} g \equiv \neg(\neg f \mathbf{U} \neg g)$.
  - Replace the temporal operators $\mathbf{F}$ and $\mathbf{G}$ using $\mathbf{F}p \equiv \top \mathbf{U} p$ and $\mathbf{G}p \equiv \bot \mathbf{V} p$.
  - Convert $\neg f$ into negation normal form
- Two step construction: first a directed graph (tableau), which is then converted into an automaton.
- Uses the expansion formulas of temporal operators:
  - $\mathbf{X}p$ holds if p holds in the next state
  - $p \wedge q$ holds if p and q hold in the current state
  - $p \vee q$ holds if either p or q holds in the current state
  - $p \mathbf{U} q$ holds if either q holds in the current state or p holds in the current state and $p \mathbf{U} q$ holds in the next state
  - $p \mathbf{V} q$ holds if either both p and q hold in the current state or if q holds in the current state and $p \mathbf{V}$ q holds in the next state
- This construction was first described by (Gerth et al., 1996)

Emptiness checking

Proposition

The language described by a generalized Büchi automaton $\mathcal{A}$ is non-empty if and only if there exists a cycle $\mathcal{C}$ reachable from I such that $\mathcal{C} \cap F \neq \emptyset$ for all $F \in \mathcal{F}$.

Definition

A *strongly connected component (SCC)* of a directed graph $\mathcal{G} = (V, E)$ is a subset $S \subseteq V$ such that for any pair $s, t \in S$ we have that $s \rightarrow_S^* t$.
An SCC is called *trivial* if $S = \{s\}$ and $s \not\rightarrow s$.

Proposition

The language described by a generalized Büchi automaton $\mathcal{A}$ is non-empty if and only if there exists an SCC $\mathcal{C}$ reachable from I such that $\mathcal{C} \cap F \neq \emptyset$ for all $F \in \mathcal{F}$.

The ASCC algorithm

- For both of these equivalent definitions there exist algorithms for checking emptiness based on them
- The ones based on cycles require the automaton to be transformed into a simple Büchi automaton (with only a single acceptance set)
- According to the comparisons by (Gaiser and Schwoon, 2009) and our own experiments, the *ASCC* algorithm has the best run-time performance at the cost of a small increase in memory use
- It is the adaptation of Tarjan's SCC algorithm to automata
- Besides avoiding a potential polynomial increase in the number of automaton states, it has one further advantage: makes fast fairness checking possible

Fairness

- Most interesting liveness properties for concurrent systems don't hold in all possible executions
- We need certain assumptions on the behaviour of the scheduler
- These conditions are called *fairness constraints*

  Weak fairness is when all actions which are (from some point on) always enabled eventually executed

  Strong fairness is when all actions which are infinitely often enabled eventually executed
- They can be modelled in LTL:

$$\text{WeakFairness}\, a \equiv (\mathbf{FG}\ \text{Enabled}\, a) \implies (\mathbf{GF}\ \text{Executed}\, a)$$
$$\text{StrongFairness}\, a \equiv (\mathbf{GF}\ \text{Enabled}\, a) \implies (\mathbf{GF}\ \text{Executed}\, a)$$

Fairness checking

- We could naively add the fairness constraints to the formula.
- This works, but the size of the automaton (thus also the run-time) is exponential in the length of the formula.
- Adding a few of these constraints already results in automata which are too large to construct.
- This can be avoided by instead examining the SCC for fairness.
- An algorithm for this is described in (Lichtenstein and Pnueli, 1985), and is only linear in the number of fairness constraints.
- We have to modify ASCC so that before reporting a counter-example, it first checks if the SCC is fair.

Example output of the model checker

- Verification of the first LTL formula for $N = 3$ in the example on the 4th slide yields:

```
Checking LTL formula ∀i1:Proc, i2:Proc. ([][[.(critical[i1] ∧ ...
Formula automaton with 37 states generated.
6 system states and 90 product automaton states investigated.
LTL formula is satisfied (model checking time: 10 ms).
Execution completed (21 ms).
```

- Verification of the second LTL formula, but without fairness yields the error trace:

```
Checking LTL formula ∀i:Proc. ([](<>[[. next = i. ]]))...
Formula automaton with 15 states generated.
4 system states and 19 product automaton states investigated.
LTL formula is NOT satisfied (model checking time: 11 ms).
Counterexample execution:
Action: init() values: [critical:[false,false,false],next:0]
...
> Loop start
    Action: enter(2) values: [critical:[false,false,true],next:2]
    Action: exit(2) values: [critical:[false,false,false],next:2]
> Loop end
ERROR encountered in execution (30 ms).
```

Measured performance of the RISCAL model checker

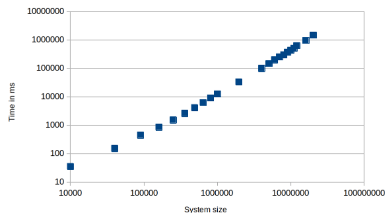

Figure: Timings for a simple property

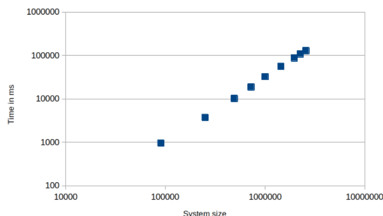

Figure: Timings for a simple property with fairness

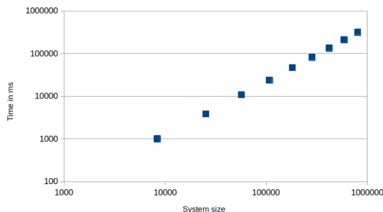

Figure: Timings for a more complex property with fairness

Comparison of RISCAL to TLA$^+$

Model	Property	RISCAL	TLA$^+$
Alternating Bit	Liveness	2.7	11
Peterson $N = 2$	Safety Inv.	< 0.1	1
	Safety LTL	< 0.1	1
	Liveness	< 0.1	14
Peterson $N = 3$	Safety Inv.	1.4	7
	Safety LTL	2.1	7
	Liveness	4.6	-
Resource Allocator	Safety Inv.	1.1	3
	Safety LTL	3.0	3
	Liveness 1	3.0	11
	Liveness 2	7.1	20
	Liveness 3	5.0	7

Figure: RISCAL versus TLA$^+$ (times in seconds)

Conclusions and further work

- Conclusions:
  - With the inclusion of the LTL model checker into RISCAL version 4.2.0, it is now a full-fledged systems checker.
  - Much slower than SPIN for checking safety properties, but has a higher level specification language and can handle more fairness constraints.
  - Comparable in speed and abstraction level to TLA^+, but again better fairness handling.

- Potential improvements
  - Implementation of partial order reduction, which could decrease the number of states to be checked by an order of magnitude
  - Decreasing the memory use (currently up to 1000 bytes per system state)
  - Implementation of a search with user defined maximum depth, which could be used to find shorter counterexamples
  - Implementation of a concurrent model checker

Bibliography I

Schreiner, W. (2021). The RISC Algorithm Language (RISCAL). https://www3.risc.jku.at/research/formal/software/RISCAL/manual/main.pdf

Gerth, R., Peled, D., Vardi, M. Y., & Wolper, P. (1996). Simple on-the-fly automatic verification of linear temporal logic. In P. Dembiński & M. Średniawa (Eds.), *Protocol specification, testing and verification XV: Proceedings of the fifteenth IFIP WG6.1 international symposium on protocol specification, testing and verification, Warsaw, Poland, June 1995* (pp. 3–18). Springer US.

Gaiser, A., & Schwoon, S. (2009). Comparison of algorithms for checking emptiness on Buechi automata. https://doi.org/10.48550/ARXIV.0910.3766

Bibliography II

Lichtenstein, O., & Pnueli, A. (1985). Checking that finite state concurrent programs satisfy their linear specification. *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 97–107. https://doi.org/10.1145/318593.318622