Verification of non-deterministic systems using model checking in RISCAL

Master Thesis

Sütő Ágoston

Research Institute for Symbolic Computation

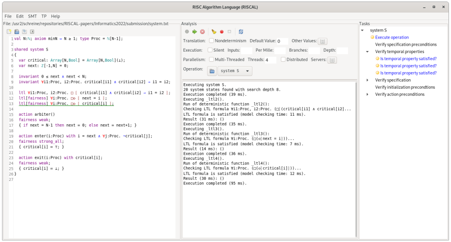
Thesis supervisor: Prof. Wolfgang Schreiner

November 8, 2022

Introduction

- Model checking is a method used for verifying whether a system meets a given specification
- Actually: only verifies a finite model of the system
- The systems are usually non-deterministic, mostly due to concurrency
- LTL is a logic that allows us to talk about the future of paths and is used for the specification
- RISCAL is a software for describing and analyzing mathematical theories and algorithms over discrete structure
- This thesis describes the extension of RISCAL with model checking capabilities for concurrent systems

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()
  fairness weak;
  { if next = N-1 then next = 0; else next = 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];
  fairness weak;
  { critical[i] = ⊥; }
}
```

Translation: ☐ Nondeterminism Default Value: 0 Other Values:
Execution: ☐ Silent Inputs: Per Mille: Branches: Depth:
Parallelism: ☐ Multi-Threaded Threads: 4 ☐ Distributed Servers:
Operation: system S

```
Executing system S.
20 system states found with search depth 8.
Execution completed (39 ms).
Executing _ltl2().
Run of deterministic function _ltl2():
Checking LTL formula ∀i1:Proc, i2:Proc. (□[(critical[i1] ∧ critical[i2]...
LTL formula is satisfied (model checking time: 11 ms).
Result (31 ms): ()
Execution completed (35 ms).
Executing _ltl3().
Run of deterministic function _ltl3():
Checking LTL formula ∀i:Proc. (□(◇[next = i]))...
LTL formula is satisfied (model checking time: 7 ms).
Result (14 ms): ()
Execution completed (36 ms).
Executing _ltl4().
Run of deterministic function _ltl4():
Checking LTL formula ∀i:Proc. (□(◇[critical[i]]))...
LTL formula is satisfied (model checking time: 12 ms).
Result (30 ms): ()
Execution completed (95 ms).
```

Figure: The RISCAL GUI

- Developed at the JKU by prof. Wolfgang Schreiner, freely available at `https://risc.jku.at/research/formal/software/RISCAL/`
- Intended primarily for didactic purposes
- Can automatically check verification conditions before attempting proof-based verification
- Extended to support concurrent systems and to check their invariants
- More about RISCAL in the manual: [1]

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] := ⊥; }
}
```

Outcomes of the thesis

1. Implementation of a full-fledged LTL model checking extension of RISCAL. The model checker consists of the following components:
   1. the translation of LTL formulas to generalized Büchi automata,
   2. the on-the-fly expansion of the state space to find SCCs (potential violations) in the product automaton of the system and the formula,
   3. the validation of SCCs against the fairness constraints to check whether they are indeed violations
2. Experimental evaluation and benchmarking of the implementation

The remainder of the presentation will be structured around these four main topics.

Basic concepts

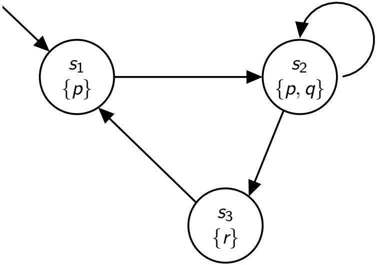


Figure: *Kripke structure K* modelling a non-deterministic system

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 **V**, defined as the dual of **U**: $f \mathbf{V} g \equiv \neg(\neg f \mathbf{U} \neg g)$.
  - Replace the temporal operators **F** and **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. [2]

The LTL to Büchi automaton algorithm I

```
procedure CREATE_GRAPH($f$)                                        ▷ LTL formula $f$
    return EXPAND({incoming: init, new: {f}, old: {}, next: {}}, {})
end procedure
procedure EXPAND(node, nodesSet)
    if node.new is empty then
        if there is a graph node n $\in$ nodesSet
              with n.old = node.old and n.next = node.next then
            n.incoming $\leftarrow$ n.incoming $\cup$ node.incoming
            return nodesSet
        else
            return EXPAND({incoming: {node}, new: node.next, old: {}, next: {}},
              nodesSet $\cup$ {node})
        end if
    else
        let f $\in$ node.new
        node.new.remove(f)
        if f = $p_i$ or f = $\neg p_i$ or f = $\top$ or f = $\bot$ then
            if f = $\bot$ or $\neg$ f $\in$ node.old then
                return nodesSet
            else
                node.old $\leftarrow$ node.old $\cup$ {f}
                return EXPAND(node, nodesSet)
```

The LTL to Büchi automaton algorithm II

```
            end if
        else if $f = \mathbf{X}\, g$ then
            return EXPAND({incoming: node.incoming, new: node.new,
              old: node.old ∪ {f}, next: node.next ∪ {g}}, nodesSet ∪ {node})
        else if $f = g \wedge h$ then
            return EXPAND({incoming: node.incoming, new: node.new ∪ ({g, h} \ node.old),
              old: node.old ∪ {f}, next: node.next}, nodesSet ∪ {node})
        else if $f = g \vee h$ or $f = g\, \mathbf{U}\, h$ or $f = g\, \mathbf{V}\, h$ then
            node1 ← { incoming: node.incoming, new: node.new ∪ (new1(f) \ node.old),
              old: node.old ∪ {f}, next: node.next ∪ next1(f) }
            node2 ← { incoming: node.incoming, new: node.new ∪ (new2(f) \ node.old),
              old: node.old ∪ {f}, next: node.next }
            return EXPAND(node2, EXPAND(node1, nodesSet))
        end if
    end if
end procedure
```

f	**new1**(f)	**next1**(f)	**new2**(f)
$g \vee h$	$\{g\}$	$\emptyset$	$\{h\}$
$g\ \mathbf{U}\ h$	$\{g\}$	$\{g\ \mathbf{U}\ h\}$	$\{h\}$
$g\ \mathbf{V}\ h$	$\{h\}$	$\{g\ \mathbf{V}\ h\}$	$\{g, h\}$

Generated automaton

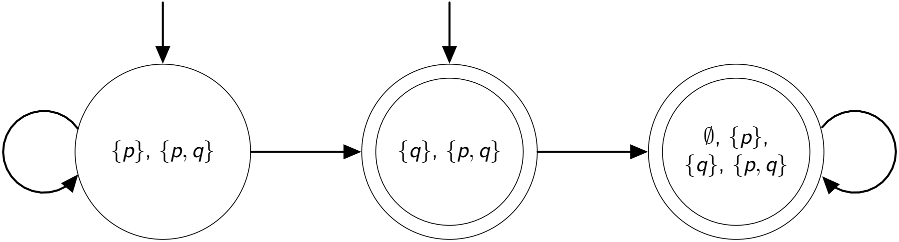


Figure: LGBA corresponding to the formula $p\ \mathbf{U}\ q$

Strongly connected components

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 $F \in \mathcal{F}$.

Emptiness check comparisons

- For both of these equivalent definitions there exist algorithms for checking emptiness based on them
- Some of these require the automaton to be transformed into a simple Büchi automaton (with only a single acceptance set)
- This can result in a polynomial blowup in the number of states
- According to the comparisons by Gaiser & Schwoon 2009 [3] and our own experiments, the *ASCC* algorithm has the best run-time performance at the cost of a small increase in memory use

The ASCC algorithm

- The ASCC algorithm works by finding the strongly connected components of the automaton and checking if they contain at least one state in each final set.
- Avoids a potential polynomial increase in the number of states if there are multiple acceptance sets.
- In reality most properties have a corresponding automaton with one or zero final sets (90-95% according to [4], 92% in the test-set of [3]), so it doesn't help that much.
- But it has one big advantage: makes fast fairness checking possible
- It is the adaptation of Tarjan's SCC algorithm to automata

The ASCC algorithm

```
procedure FIND_CYCLES($s, d$)                              ▷ state $s$, search depth $d$
    $s$.dfsnum ← $d$
    $s$.current ← true
    $roots$.push($s$, $A(s)$)
    $active$.push($s$)
    for all successors $t$ of $s$ do
        if $t$.dfsnum $= 0$ then FIND_CYCLES($t, d + 1$)
        else if $t$.current then
            $B \leftarrow \emptyset$
            repeat
                $(u, C) \leftarrow$ $roots$.pop()
                $B \leftarrow B \cup C$
                if $B = K$ then report cycle
            until $u$.dfsnum $\leq$ $t$.dfsnum
        end if
    end for
    if $roots$.top() $= (s, \_)$ then
        $roots$.pop()
        repeat
            $u \leftarrow$ $active$.pop()
            $u$.current ← false
        until $u = s$
    end if
end procedure
```

How it works

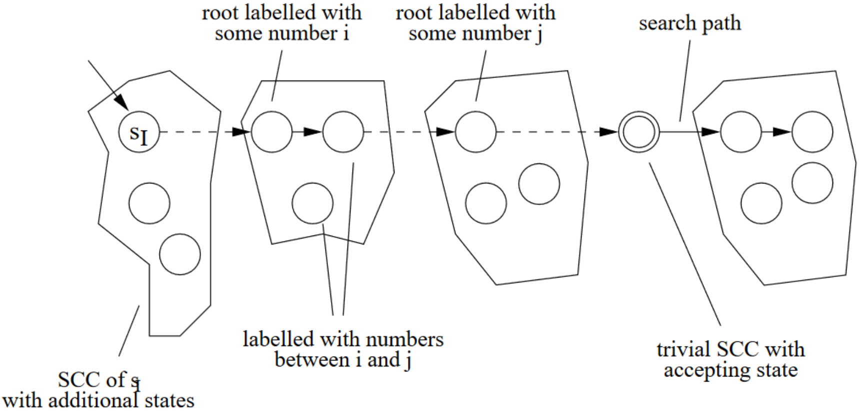


Figure: Shape of the active graph taken from [3]

Fairness

- Most interesting liveness conditions 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:

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

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 [5], 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.

Fairness checking algorithm

```
▷ A: strongly connected subgraph of the product automaton
▷ weakFairness: set of actions with weak fairness constraints
▷ strongFairness: set of actions with strong fairness constraints
procedure IS_SCC_FAIR(A, weakFairness, strongFairness)
    for all action a $\in$ weakFairness do
        if for all states s $\in$ A a is enabled in s and a is not executed in s then
            return false
        end if
    end for
    A' $\leftarrow$ A
    for all action a $\in$ strongFairness do
        if for all states s $\in$ A a is not executed in s then
            A' $\leftarrow$ {s $\in$ A' : a is not enabled in s}
        end if
    end for
    if A' = A then return true
    end if
    for all $A_i$ $\in$ DECOMPOSE_INTO_SCCS(A') do
        if IS_SCC_FAIR($A_i$, weakFairness, strongFairness) then
            return true
        end if
    end for
    return false
end procedure
```

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

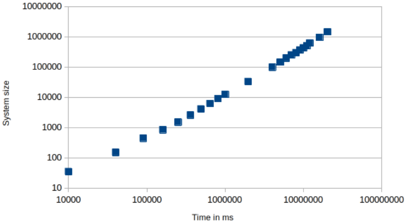


Figure: Timings for a simple property $T(n) = O(n^{1.448})$

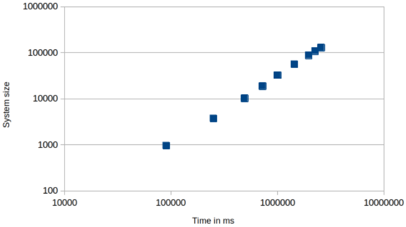


Figure: Timings for a simple property with fairness $T(n) = O(n^{1.536})$

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 concurrent model checker

Bibliography I

[1] Wolfgang Schreiner. *The RISC Algorithm Language (RISCAL)*. `https://www3.risc.jku.at/research/formal/software/RISCAL/manual/main.pdf`. 2021.

[2] R. Gerth et al. “Simple On-the-fly Automatic Verification of Linear Temporal Logic”. In: *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*. Ed. by Piotr Dembiński and Marek Średniawa. Boston, MA: Springer US, 1996, pp. 3–18. ISBN: 978-0-387-34892-6.

[3] Andreas Gaiser and Stefan Schwoon. *Comparison of Algorithms for Checking Emptiness on Buechi Automata*. 2009. DOI: `10.48550/ARXIV.0910.3766`. URL: `https://arxiv.org/abs/0910.3766`.

Bibliography II

[4] Ivana Cerna and Radek Pelánek. “Relating Hierarchy of Temporal Properties to Model Checking”. In: vol. 2747. Aug. 2003, pp. 318–327. ISBN: 978-3-540-40671-6. DOI: 10.1007/978-3-540-45138-9_26.

[5] Orna Lichtenstein and Amir Pnueli. “Checking That Finite State Concurrent Programs Satisfy Their Linear Specification”. In: *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’85. New Orleans, Louisiana, USA: Association for Computing Machinery, 1985, pp. 97–107. ISBN: 0897911474. DOI: 10.1145/318593.318622.