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
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.
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

\[
\text{val } N: \mathbb{N};
\]
\[
\text{ axiom } \min N \iff N \geq 1;
\]
\[
\text{ type } \text{Proc} = \mathbb{N}[\mathbb{N}-1];
\]

\[
\text{shared system } S
\]
\{
\]
\[
\text{ var } \text{critical}: \text{Array}[N, \text{Bool}] = \text{Array}[N, \text{Bool}](\bot);
\]
\[
\text{ var } \text{next}: \mathbb{Z}[-1, N] = 0;
\]

\[
\text{ invariant } 0 \leq \text{next} \land \text{next} < N;
\]
\[
\text{ invariant } \forall i_1: \text{Proc}, i_2: \text{Proc}. \text{critical}[i_1] \land \text{critical}[i_2] \Rightarrow i_1 = i_2;
\]

\[
\text{ ltl } \forall i_1: \text{Proc}, i_2: \text{Proc}. \square[\text{critical}[i_1] \land \text{critical}[i_2] \Rightarrow i_1 = i_2];
\]
\[
\text{ ltl[ fairness]} \forall i: \text{Proc}. \square\diamond[\text{next} = i];
\]
\[
\text{ ltl[ fairness]} \forall i: \text{Proc}. \square\diamond[\text{critical}[i]];\]

\[
\text{ action arbiter() with } \forall j: \text{Proc}. \neg \text{critical}[j];
\]
\[
\text{ fairness strong};
\]
\{
\]
\[
\text{ next := if next = N - 1 then 0 else next + 1; }
\]

\[
\text{ action enter(i: Proc) with i = next \land } \forall j: \text{Proc}. \neg \text{critical}[j];
\]
\[
\text{ fairness strong, all; }
\]
\{
\]
\[
\text{ critical}[i] := \top; }
\]

\[
\text{ action exit(i: Proc) with critical[i];}
\]
\{
\]
\[
\text{ critical}[i] := \bot; }
\]
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:
- $K \models p$
- $K \models X q$
- $K \models G(\neg(r \land p))$
- $K \models (p U r) \lor (G p)$

and some, which do not:
- $K \not\models F(r \land p)$
- $K \not\models p U r$

**Definition**

*Model checking problem*
Given a Kripke-structure $K = (S, I, T, L)$ and an LTL formula $f$ determine whether $K \models f$, and if not, provide a trace $\pi$ 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 \(\Sigma\)
- a labelling of the states \(\mathcal{L} : S \to 2^\Sigma\)
- a transition relation \(\to \subseteq S \times S\)
- set of accepting sets \(\mathcal{F} \subseteq 2^S, \mathcal{F} = \{F_1, F_2, \ldots, F_n\}\).

**Definition**

A Büchi automaton \(\mathcal{A}\) accepts a word \(w = a_0a_1a_2\ldots \in \Sigma^\omega\) if there exists \(\sigma = s_0s_1s_2\ldots \in S^\omega\) 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 \(\sigma\).
The LTL to Büchi automaton algorithm

- Preprocessing:
  - Introduce new temporal operator $V$, defined as the dual of $U$:
    $$f V g \equiv \neg (\neg f U \neg g).$$
  - Replace the temporal operators $F$ and $G$ using $F p \equiv \top U p$ and $G p \equiv \bot V p$.
  - 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:
  - $X p$ holds if $p$ holds in the next state
  - $p \land q$ holds if $p$ and $q$ hold in the current state
  - $p \lor q$ holds if either $p$ or $q$ holds in the current state
  - $p U q$ holds if either $q$ holds in the current state or $p$ holds in the current state and $p U q$ holds in the next state
  - $p V q$ holds if either both $p$ and $q$ hold in the current state or if $q$ holds in the current state and $p V q$ holds in the next state

- This construction was first described by Gerth et al. [2]
The LTL to Büchi automaton algorithm

procedure CREATE_GRAPH(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 ∈ nodesSet
            with n.old = node.old and n.next = node.next then
            n.incoming ← n.incoming ∪ node.incoming
            return nodesSet
        else
            return EXPAND({incoming: {node}, new: node.next, old: {}, next: {}},
                           nodesSet ∪ {node})
        end if
    else
        let f ∈ node.new
        node.new.remove(f)
        if f = p_i or f = ¬p_i or f = ⊤ or f = ⊥ then
            if f = ⊥ or ¬f ∈ node.old then
                return nodesSet
            else
                node.old ← node.old ∪ {f}
                return EXPAND(node, nodesSet)
            end if
        end if
    end if
end procedure
The LTL to Büchi automaton algorithm II

end if
else if \( f = Xg \) then
    return \( \text{EXPAND}(\{\text{incoming}: \text{node.incoming}, \text{new}: \text{node.new}, \)
    \old: \text{node.old} \cup \{f\}, \text{next}: \text{node.next} \cup \{g\}, \text{nodesSet} \cup \{\text{node}\}) \)
else if \( f = g \land h \) then
    return \( \text{EXPAND}(\{\text{incoming}: \text{node.incoming}, \text{new}: \text{node.new} \cup (\{g, h\} \setminus \text{node.old}), \)
    \old: \text{node.old} \cup \{f\}, \text{next}: \text{node.next}, \text{nodesSet} \cup \{\text{node}\}) \)
else if \( f = g \lor h \) or \( f = g \mathbf{U} h \) or \( f = g \mathbf{V} h \) then
    \text{node1} \leftarrow \{ \text{incoming}: \text{node.incoming}, \text{new}: \text{node.new} \cup (\text{new1}(f) \setminus \text{node.old}), \)
    \old: \text{node.old} \cup \{f\}, \text{next}: \text{node.next} \cup \text{next1}(f) \} \)
    \text{node2} \leftarrow \{ \text{incoming}: \text{node.incoming}, \text{new}: \text{node.new} \cup (\text{new2}(f) \setminus \text{node.old}), \)
    \old: \text{node.old} \cup \{f\}, \text{next}: \text{node.next} \} \)
    return \( \text{EXPAND}(\text{node2}, \text{EXPAND}(\text{node1}, \text{nodesSet})) \)
end if
end if
end procedure

<table>
<thead>
<tr>
<th>( f )</th>
<th>\text{new1}(f)</th>
<th>\text{next1}(f)</th>
<th>\text{new2}(f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( g \lor h )</td>
<td>{g}</td>
<td>\emptyset</td>
<td>{h}</td>
</tr>
<tr>
<td>( g \mathbf{U} h )</td>
<td>{g}</td>
<td>{g \mathbf{U} h}</td>
<td>{h}</td>
</tr>
<tr>
<td>( g \mathbf{V} h )</td>
<td>{h}</td>
<td>{g \mathbf{V} h}</td>
<td>{g, h}</td>
</tr>
</tbody>
</table>
Generated automaton

Figure: LGBA corresponding to the formula $p \cup q$
Strongly connected components

**Proposition**

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

**Definition**

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

**Proposition**

The language described by a generalized Büchi automaton $A$ is non-empty if and only if there exists an SCC $C$ reachable from $I$ such that $C \cap F \neq \emptyset$ 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 ← ∅
            repeat
                (u, C) ← roots.pop()
                B ← B ∪ C
                if B = K then report cycle
            until u.dfsnum ≤ t.dfsnum
        end if
    end for

    if roots.top() = (s, _) then
        roots.pop()
        repeat
            u ← 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:
  - WeakFairness $a \equiv (\text{FG} \; \text{Enabled} \; a) \implies (\text{GF} \; \text{Executed} \; a)$. 
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 ∈ weakFairness do
        if for all states s ∈ A a is enabled in s and a is not executed in s then
            return false
        end if
    end for
    A' ← A
    for all action a ∈ strongFairness do
        if for all states s ∈ A a is not executed in s then
            A' ← {s ∈ A' : a is not enabled in s}
        end if
    end for
    if A' = A then return true
    end if
    for all A_i ∈ 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 $\forall i_1:Proc, i_2:Proc. ([] [[. (critical[i1] \land ...$

  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 $\forall i:Proc. ([]<>[. \ next = i. ]])...$

  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$^+$

<table>
<thead>
<tr>
<th>Model</th>
<th>Property</th>
<th>RISCAL</th>
<th>TLA$^+$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternating Bit</td>
<td>Liveness</td>
<td>2.7</td>
<td>11</td>
</tr>
<tr>
<td>Peterson $N = 2$</td>
<td>Safety Inv.</td>
<td>$&lt; 0.1$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Safety LTL</td>
<td>$&lt; 0.1$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Liveness</td>
<td>$&lt; 0.1$</td>
<td>14</td>
</tr>
<tr>
<td>Peterson $N = 3$</td>
<td>Safety Inv.</td>
<td>1.4</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Safety LTL</td>
<td>2.1</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Liveness</td>
<td>4.6</td>
<td>-</td>
</tr>
<tr>
<td>Resource Allocator</td>
<td>Safety Inv.</td>
<td>1.1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Safety LTL</td>
<td>3.0</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Liveness 1</td>
<td>3.0</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>Liveness 2</td>
<td>7.1</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>Liveness 3</td>
<td>5.0</td>
<td>7</td>
</tr>
</tbody>
</table>

**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

