# Model Checking Concurrent Systems Under Fairness Constraints in RISCAL

*Author:*
Ágoston Sütő

*Supervisor:*
A.Univ.-Prof. DI Dr.
Wolfgang Schreiner

Master's Thesis
for the acquisition of the academic degree
Diplom-Ingenieur
in the master's program
Computer Mathematics

Research Institute for Symbolic Computation

Johannes Kepler University Linz
Altenbergerstraße 69, 4040 Linz, Österreich

May 2023

# CONTENTS

# ABSTRACT

Model checking is a method for verifying that a program satisfies certain desirable properties formalised using mathematical logic. It is a rigorous method, similar to theorem proving, but it is generally applied when theorem proving would be too difficult due to the complexity of the algorithm, such as in concurrent systems. Model checking is used in the software industry. RISCAL (RISC Algorithm Language) is a language and software system that can be used to describe algorithms over a finite domain, specify their behaviour and then validate the specification. While it mainly focuses on deterministic algorithms, it has limited support for non-deterministic systems as well.

The thesis extends the support for non-deterministic systems in RISCAL by allowing the user to specify complex properties about their behaviour in the language of *Linear Temporal Logic* (LTL) and then to validate them. The core contribution is a model checker implemented in Java using the so-called automaton-based explicit state model checking approach. The software is capable of verifying certain properties that could not be handled by a well-known model checker used in the industry. While in most cases it has underperformed its competitors, our implementation is promising, especially when it comes to properties with certain side conditions, called *fairness constraints*. The majority of the thesis is be concerned with the theoretical aspects of the automaton-based model checking approach, which is followed by a description of the implementation and various benchmarks.

# ZUSAMMENFASSUNG

Modellprüfung (Model Checking) ist eine Methode zur Überprüfung, ob ein Programm bestimmte wünschenswerte Eigenschaften erfüllt, die mit Hilfe der mathematischen Logik formalisiert sind. Die Methode ist exakt, ähnlich wie das Beweisen von Theoremen, wird aber im Allgemeinen angewendet, wenn das Beweisen aufgrund der Komplexität des Algorithmus zu schwierig wäre, z.B. bei nebenläufigen Systemen. Modellprüfung wird in der Softwareindustrie eingesetzt. RISCAL (RISC Algorithm Language) ist eine Sprache und ein Softwaresystem, das verwendet werden kann Algorithmen über einen endlichen Bereich zu beschreiben, ihr Verhalten zu spezifizieren und dann die Spezifikation zu validieren. RISCAL konzentriert sich hauptsächlich auf deterministische Algorithmen, unterstützt in begrenztem Umfang aber auch nicht-deterministische Systeme.

Diese Arbeit erweitert die Unterstützung für nicht-deterministische Systeme in RISCAL, indem sie dem Benutzer die Möglichkeit gibt, komplexe Eigenschaften ihres Verhaltens in der Sprache der *linearen Temporalen Logik* (LTL) zu spezifizieren und diese dann zu validieren. Der Kernbeitrag ist ein Modellprüfer, der in Java implementiert ist und den Automaten-basierten Ansatz der Modellprüfung mit expliziter Darstellung der Zustände umsetzt. Die Software ist in der Lage, bestimmte Eigenschaften zu verifizieren, die von einem bekannten, in der Industrie verwendeten Modellprüfprogramm nicht bewältigt werden können. Obwohl sie in den meisten Fällen schlechter abschneidet als ihre Konkurrenten, ist unsere Implementierung vielversprechend, insbesondere wenn es um Eigenschaften mit bestimmten Nebenbedingungen geht, so genannte *Fairness-Einschränkungen*. Der Hauptteil der Arbeit befasst sich mit den theoretischen Aspekten des Automaten-basierten Modellprüfungsansatzes, gefolgt von einer Beschreibung der Implementierung und verschiedener Benchmarks.

# 1 INTRODUCTION

The goal of software engineering is to create correct programs, that is, turn the specification into a software that fulfills it without errors. To achieve this the individual software engineer has a variety of tools at their disposal, such as the compiler and static analyzers. Nevertheless these do not ensure adherence to the specification, they can only find common error patterns. Adherence to the specification is most commonly checked using tests. While testing offers immense value for the relatively low cost, it is only ever going to verify a small subset of potential input-output pairs or potential behaviours.

The specification of a program can be formalized using mathematics, making it possible to reason with the help of axioms and rules of inference, that under certain assumptions (i.e. in a model environment) the given program satisfies them. These mathematically rigorous techniques are called formal methods, and their use was pioneered by Robert Floyd [Flo67] and Antony Hoare [Hoa69] in their seminal papers.

As opposed to the deterministic programs presented in these papers, modern systems often involve nondeterminism and concurrency. A nondeterministic program, unlike a deterministic one, may exhibit different behaviours on different runs given the same input. A concurrent system is a system that can execute several computations at the same time which might lead to nondeterminism due to so called race conditions (when the order in which computations finish matters for the overall state of the system).

While it might still be possible to fully understand such systems and provide a mathematical description of their behaviour through theorem proving, that might be prohibitively time consuming. An alternative approach would be to model them as finite state machines, formulate the requirements and then algorithmically verify that these conditions hold for every possible execution of the system. This is called *model checking*.

An example for such a concurrent system would be an elevator with the mechanism for controlling the cabin door and the mechanism for moving the cabin as separate components. In this case we might expect the system to satisfy certain *safety properties* (the cabin never moves when the doors are open) and *liveness properties* (whenever the call button on a given floor was pressed, the cabin will eventually stop there and open the door).

To describe general safety and liveness properties we use variants of temporal logic, which can reason about a sequence of events, such as the different states a computer program reaches during its execution. In the thesis we will work with Linear Temporal Logic (LTL), whose temporal operators such as *always*, *eventually* and *until* make it possible to describe, for example, if an event occurs infinitely often, or if it always happens before another event.

RISCAL is a software system developed at the Research Institute for Symbolic Computation (RISC) of the Johannes Kepler University Linz. Its primary goal is to be a preliminary step in computer assisted proofs by checking that the verification conditions hold in a finite domain before attempting the proof in a more general setting.

RISCAL was originally intended for use with deterministic algorithms, but it was later extended to support non-determinism in the form of so called *shared* and *distributed* systems. These can be used to model concurrent algorithms, such as mutual exclusion or data transfer protocols.

Using a simple search to discover all states of a non-deterministic system, RISCAL is able to verify invariants, that is properties which hold in every state of the system. Many important but simple safety properties, such as mutual exclusion can be expressed using invariants, but for more complex safety and liveness properties one needs a more general language, such as LTL.

Some important liveness properties do not hold in all possible executions, only under certain assumptions, for example that the operating system will eventually schedule all processes that make up the system. These are called *fairness constraints*, and while they are expressible directly in temporal logic, due to their prevalence and importance they should be handled in a special way for performance reasons and ease-of-use.

The primary goal of this thesis is to extend the RISCAL software system with LTL model checking capabilities, making it able to verify complex safety and liveness properties, potentially under various fairness constraints. The model checker should be efficient enough to verify non-trivial concurrent algorithms.

To achieve this goal, we have started by studying the theory of model checking, then continued by exploring one approach, the so called automaton-based explicit state model checking in detail. As even this particular approach has a lot of possible variants, we had to make several choices along the way. Such choices were between the variants of the automaton creation algorithm, and the different approaches for checking the emptiness of the language of an automaton. Finally we had to figure out how to efficiently handle many fairness constraints. This involves an algorithm that was independently discovered by the author before it was found in existing literature.

After the decision has been made on which particular combination of algorithms to use, these have been implemented in Java. The RISCAL software, which includes the LTL model checker since version 4.2.0 (contained in the package `riscal.ltl`) can be freely downloaded from https://www.risc.jku.at/research/formal/software/RISCAL/. The thesis documents the choices made during the implementation. Finally, to verify whether the model checking extension is adequate, its performance was measured using a set of real-world and purpose made concurrent algorithms, then compared it to other widely used model checkers.

It was found that the RISCAL LTL model checker performs very well for liveness properties involving many fairness constraints, often outperforming the model checker which served as the main basis of comparison for our implementation. For simpler safety properties it usually performed worse, but nevertheless it was able to verify non-trivial systems.

With some further improvements, such as concurrency, it could compete with model checkers used in the software industry.

The rest of the thesis is structured as follows: Chapter 2 introduces the RISCAL software and the Spin and TLC model checkers, which will serve as a basis of comparison for our implementation. Chapter 3 gives the detailed description of the syntax and semantics of LTL. Chapter 4 introduces the basics of automata theory, first for simple finite automata then for so-called Büchi automata. Chapter 5 contains a detailed description of the automaton-based model checking process. Chapter 6 compares this to alternative model checking approaches, such as symbolic model checking. Then the practical part starts with Chapter 7, containing a user-oriented demonstration of the model checking extension. This is followed by a description of the implementation with example code snippets in Chapter 8. Chapter 9 contains the results of the measurements taken over a series of benchmarks and their interpretations, and finally the the thesis is closed off in Chapter 10 by a series of concluding remarks and potential future improvements.

The core results of this thesis were published in [SS22].

# 2 STATE OF THE ART

This chapter describes the purpose and current capabilities of the RISCAL software which will be extended by an LTL model checker. It also describes two established model checkers, both used in the software industry, which will later serve as a basis of comparison in Chapter 9.

## 2.1 RISCAL

RISCAL has been developed at the Research Institute for Symbolic Computation (RISC) of the Johannes Kepler University Linz; it has been described as "a language and associated software system for describing mathematical algorithms over discrete structures, formally specifying their behaviour by logical formulas and formulating the mathematical theories on which these specifications depend." [Sch21] [Sch23]

### 2.1.1 *Support for deterministic algorithms*

Formal proofs of the correctness of software usually require human assistance and are very time consuming. The majority of this time is spent trying to prove statements which do *not* hold because the assumptions were either too weak or incorrect. The goal of the RISCAL language and software system is to be a preliminary step in computer assisted proofs by checking that the verification conditions hold in a finite domain. To facilitate this RISCAL provides an extensive collection of built-in operators and data types (e.g. sets and maps).

To demonstrate the workings and capabilities of RISCAL, let us consider the binary search algorithm described in Program 1. The first line defines two values, $N$ and $M$, both natural numbers. These parameters can be set from outside (i.e. from the GUI); in the next three lines we can see how they are used to specify the types, that is the finite domain, in which the algorithms in this file will be executed.

This is followed by the definition of the binary search algorithm, in a language very similar to other modern programming languages. The only surprising element is the use of non-ASCII symbols $\wedge$ and $\leq$ in the syntax. Indeed, RISCAL supports convenient single-character forms for many well-known operators, which make long specifications much more legible.

But an algorithm defined like this is not of much use in RISCAL. Since our goal is to check if it is correct, first we have to define what *correct* means. To do this, we need to

```
val N: ℕ; val M: ℕ;

type elem = ℕ[M];
type array = Array[N, elem];
type int = ℤ[-1, N];

proc bsearch(a: array, x: elem, from: int, to: int): int {
  var r: int := -1;
  var low: int := from; var high: int := to;
  while r = -1 ∧ low ≤ high do {
    val mid = (low + high) / 2;
    if a[mid] = x then
      r := mid;
    else if a[mid] < x then
      low := mid + 1;
    else
      high := mid - 1;
  }
  return r;
}
```

Program 1: Binary search algorithm in RISCAL

*specify* the output of the function: the return value is the index of the element $x$, or $-1$, if it is not contained in the array $a$ between positions *from* and *to*. To do this, we may write the following after the procedure declaration:

```
ensures (result = -1 ∧ ∀ i: int with from ≤ i ∧ i ≤ to. a[i] ≠ x)
     ∨ (from ≤ result ∧ result ≤ to ∧ a[result] = x);
```

These formulas are specified using first-order logic, so they may contain both existential and universal quantifiers. RISCAL is able to evaluate these formulas because all the domains are finite. In the example above, the universally quantified variable $i$ of type *int* can only take up values between $-1$ and $N$.

Now we can click on the green arrow in the RISCAL GUI to check if the algorithm is behaving correctly for all the values in our domain. But if we do so, we will find that it is trying to execute the algorithm for nonsensical values, because there are no requirements defined on the input parameters. For the binary search algorithm to work, the array must be sorted, and the inequality $0 \leq \text{from} \leq \text{to}$ must hold. This can be specified by adding

```
requires (0 ≤ from ∧ from < N ∧ 0 ≤ to ∧ to < N ∧ from ≤ to)
    ∧ (∀ i: int with from < i ∧ i ≤ to. a[i - 1] ≤ a[i]);
```

between the procedure declaration and the *ensures* clause. Now running the procedure will give something like

```
Execution completed for ALL inputs (44084 ms, 6796 checked, 30068 inadmissible).
```

If we make a mistake, say change `r := mid;` to `r := x;`, then running the algorithm will provide a counter-example, a set of inputs that satisfy the input requirements, but where the result does not meet the output requirements:

```
postcondition is violated by result 1 for application bsearch([1,0,0,0], 1, 0, 0)
```

These capabilities make it a very useful tool to heuristically *validate the specification*, before a formal proof is attempted.

Loop invariants and loop measures may also be defined and from these RISCAL can generate verification conditions:

- the invariant holds before entering the loop
- the invariant is preserved after each iteration of the loop
- the measure is always positive
- the measure is decreased after each iteration of the loop

The validity of these implies the correctness of the program and they can also be checked automatically.

Besides the brute force approach to checking validity, recent versions of the software include a support for several SMT solvers, which can be used to verify the validity of formulas over much larger domains than what is possible with the RISCAL checker alone [Rei20].

### 2.1.2 *Support for non-deterministic systems*

RISCAL also supports so called "shared" and "distributed" systems with *concurrent* (non-deterministic) behavior. A shared system consists of a single component with multiple actions, which are executed non-deterministically, while a distributed system consists of multiple components that communicate with each other using messages.

The shared system presented in Program 2 works by first executing the `init` action with a parameter `a` which fulfills the condition `a > N / 2`. Then it non-deterministically chooses one of the actions `incx`, `decx` or `swap` whose guard condition is satisfied.

It is not hard to see that the sum of the variables `x` and `y` stays constant. We can annotate the system with an invariant which specifies this:

```
...
var y: elem = 0;

invariant x + y = N;

init(a: elem) with a > N / 2; {
...
```

Then we can run RISCAL with a given value of `N` to check if this invariant will really hold in every state in every possible execution, again by clicking the green arrow in the RISCAL GUI. Doing so results in the following output for $N = 10$:

```
Executing system S1.
11 system states found with search depth 11.
Execution completed (4 ms).
```

If we make a mistake, for example if we accidentaly write

```
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: elem := x; x := y; y := tmp;
    }
}
```

Program 2: Example of a shared system in RISCAL

```
    action decx() with x > y + 1; {
        x := x − 1; y := y;
    }
```

then the output changes and shows us how an execution of the system reaches violating state:

```
Executing system S1.
ERROR in execution of system S1: evaluation of
  invariant (x+y) = N;
at line 10 in file example.txt:
  invariant is violated
The system run leading to this error:
  0:[x:6,y:4]->incx()->
  1:[x:7,y:3]->incx()->
  2:[x:8,y:2]->incx()->
  3:[x:9,y:1]->incx()->
  4:[x:10,y:0]->decx()->
  5:[x:9,y:0]
ERROR encountered in execution (2 ms).
```

If we suspect that a shorter violating run exists, we can set the value of the *Depth* variable in the UI and check only executions up to that length. In our case even setting it to 1 will result in a violating run:

```
Executing system S1.
ERROR in execution of system S1: evaluation of
  invariant (x+y) = N;
at line 10 in file gcd.txt:
  invariant is violated
The system run leading to this error:
  0:[x:6,y:4]->decx()->
  1:[x:5,y:4]
ERROR encountered in execution (2 ms).
```

RISCAL checks the invariants using a simple depth-first search to explore the set of reachable states. The pseudocode representation in Algorithm 2.1 and also the actual implementation represents the system only by its initial states, and each state consists of the list of its successors and the values of the system variables. The system is expanded *lazily*.

**Algorithm 2.1** Algorithm for verifying system invariants

```
procedure CHECK_INVARIANT(initialStates, maxDepth)
    visited ← $\emptyset$
    for all $s \in$ initialStates do
        if invariant doesn't hold in state $s$ then
            return false
        end if
        if CHECK_INVARIANT(s, visited, 0, maxDepth) is false then
            return false
        end if
    end for
end procedure

procedure CHECK_INVARIANT(currentStatte, visited, currentDepth, maxDepth)
    if currentState $\in$ visited then
        return true
    end if
    visited ← visited $\cup \{s\}$
    if maxDepth is set and maxDepth $+ 1 >$ depth then
        return true
    end if
    for all $s \in$ currentState.successors do
        if invariant doesn't hold in state $s$ then
            return false
        end if
        if CHECK_INVARIANT(s, visited, depth + 1, maxDepth) is false then
            return false
        end if
    end for
end procedure
```

The search algorithm is split into two procedures. The first one initializes the set of visited states to the empty set, iterates through the set of initial states, checks that the invariant holds for the given state, then starts a new search from this state.

The second procedure checks if the current state is already visited, otherwise it is added to the set of visited states. If there is a maximum depth set and it is exceeded at the current recursion level, then the function returns. Otherwise it iterates through the successors of the current state, checks if the invariant holds in them, then starts from each a new depth first search, making sure that the recursion depth is increased.

## 2.2 THE SPIN MODEL CHECKER

Spin (Simple Promela INterpreter) is a software verification tool focusing on distributed systems. Its development was started by Gerard J. Holzmann in the 1980s at the Computing Sciences Research Center at Bell Labs and it continues into the present. The first version was released in 1989, the current version presented in this chapter is 6.5.2 released on Dec 6, 2019.

Similarly to RISCAL, Spin allows the user to describe a model of the system to be verified in a domain specific programming language, which is in this case Promela (Process or Protocol Meta Language). Promela is a modelling language with built-in support for many constructs used in the development of distributed systems, such as processes, atomic constructs and message queues.

The details of Promela and the verification capabilities of Spin will be explained through a concrete example, mutual exclusion. Multiple components accessing the same resource, such as the same location in memory at the same time can lead to unintended consequences due to race conditions, i.e. when the behaviour of the program depends on the sequence of uncontrollable events. Mutual exclusion is a property of concurrent systems which ensures, that at any given time only one component is accessing a shared resource, or more generally, only one component is in the so called *critical section*.

Dekker's algorithm is the first known correct solution for the mutual exclusion problem [Dij63], Program 3 contains its description in Promela.

The first three lines declare global variables, the first two of them being two element arrays. A variable's type may be one of the basic data types `bit`, `bool`, `byte`, `short` or `int`, a user defined data type declared with the `typedef` keyword, or a named constants declared using `mtype`. Promela also has built-in support for message channels.

This is followed by the `init` process, which is created at the beginning; it is usually used to set up the system by initializing the variables and starting other process-instances. In our case it starts two instances of the `component` process, initialized with the identifiers 0 and 1 respectively.

Process types are declared using the `proctype` keyword followed by an identifier and the parameters which are set when a new process of this type is initialized using `run`. The instructions of the processes running simultaneously are executed in a non-deterministically interleaved manner.

In Promela every statement can be either enabled or blocked. Some statements, such as `skip`, declaring or assigning a value to a variable are always enabled. A simple expression is enabled if and only if it evaluates to a non-zero value (or `true`, in case of a boolean

```
 1 bool wants_to_enter[2];
 2 bool critical[2];
 3 bit turn;

 5 init
 6 {
 7         run component(0);
 8         run component(1);
 9 }

11 proctype component(bit id)
12 {
13         bit other = 1 - id;

15         do :: true ->
16                 wants_to_enter[id] = true;

18                 do
19                 :: wants_to_enter[other] ->
20                         if
21                         :: turn == other ->
22                                 wants_to_enter[id] = false;
23                                 (turn == id);
24                                 wants_to_enter[id] = true;
25                         :: else -> skip;
26                         fi;
27                 :: else -> break;
28                 od;

30                 critical[id] = true;
31                 // critical section
32                 critical[id] = false;

34                 turn = other;
35                 wants_to_enter[id] = false;
36         od;
37 }
```

Program 3: Promela representation of Dekker's solution to the mutual exclusion problem

expression). Execution may only continue with an enabled statements, so if at a given time all the running processes are blocked, the system is in a deadlock. This means that the busy waiting loop, which in C would look like `while (turn != id) {}` can be implemented in Promela by `turn == id` (see line 23, the outer brackets are optional).

The two important control structures in Promela, selection and repetition are quite different from the conditional statements and loops of ordinary programming languages.

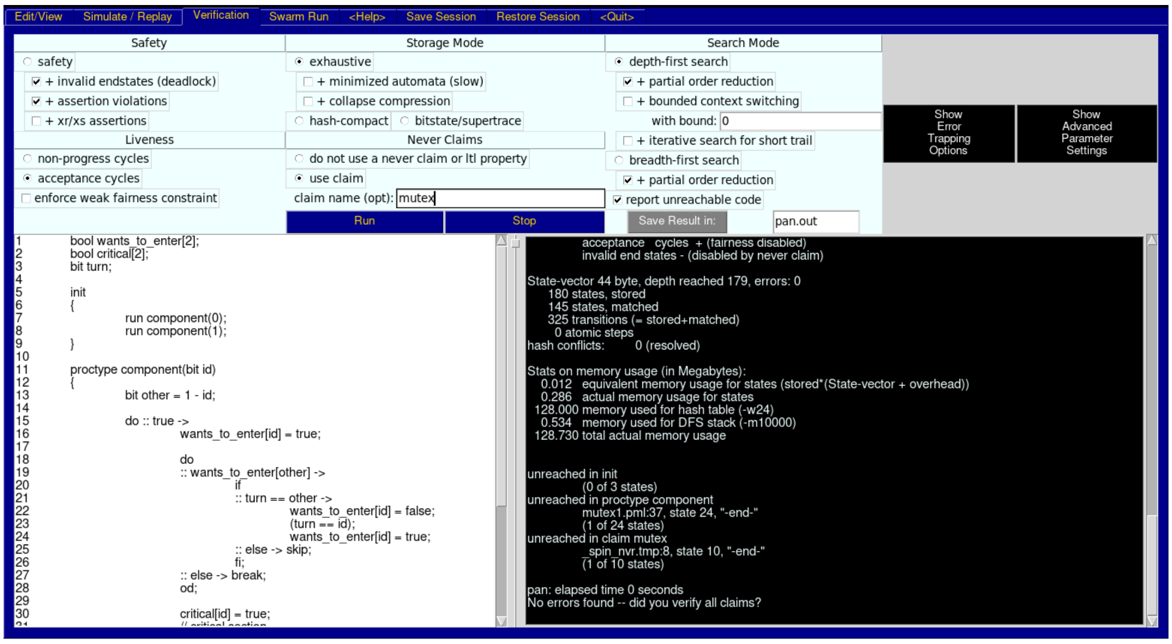
```
if
:: guard -> statements
...
:: guard -> statements
[ :: else -> statements ]
fi
```

Promela syntax for the selection statement

```
do
:: guard -> statements
...
:: guard -> statements
[ :: else -> statements ]
od
```

Promela syntax for the repetition statement

In both of these structures, one of the options (introduced by `::`), with an enabled guard condition is selected non-deterministically, and is executed. Selection then proceeds with



Safety
- [ ] safety
- [✓] + invalid endstates (deadlock)
- [✓] + assertion violations
- [ ] + xr/xs assertions

Liveness
- [ ] non-progress cycles
- [✓] acceptance cycles
- [ ] enforce weak fairness constraint

Storage Mode
- [✓] exhaustive
- [ ] + minimized automata (slow)
- [ ] + collapse compression
- [ ] hash-compact
- [ ] bitstate/supertrace

Never Claims
- [ ] do not use a never claim or ltl property
- [✓] use claim

claim name (opt): mutex

Search Mode
- [✓] depth-first search
- [✓] + partial order reduction
- [ ] + bounded context switching

with bound: 0
- [ ] + iterative search for short trail
- [ ] breadth-first search
- [✓] + partial order reduction
- [✓] report unreachable code

pan.out

```
bool wants_to_enter[2];
bool critical[2];
bit turn;

init
{
        run component(0);
        run component(1);
}

proctype component(bit id)
{
        bit other = 1 - id;

        do :: true ->
                wants_to_enter[id] = true;

                do
                :: wants_to_enter[other] ->
                        if
                        :: turn == other ->
                                wants_to_enter[id] = false;
                                (turn == id);
                                wants_to_enter[id] = true;
                        :: else -> skip;
                        fi;
                :: else -> break;
                od;

                critical[id] = true;
```

```
        acceptance   cycles  + (fairness disabled)
        invalid end states - (disabled by never claim)

State-vector 44 byte, depth reached 179, errors: 0
     180 states, stored
     145 states, matched
     325 transitions (= stored+matched)
       0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.012   equivalent memory usage for states (stored*(State-vector + overhead))
    0.286   actual memory usage for states
  128.000 memory used for hash table (-w24)
    0.534   memory used for DFS stack (-m10000)
  128.730 total actual memory usage


unreached in init
        (0 of 3 states)
unreached in proctype component
        mutex1.pml:37, state 24, "-end-"
        (1 of 24 states)
unreached in claim mutex
        _spin_nvr.tmp:8, state 10, "-end-"
        (1 of 10 states)

pan: elapsed time 0 seconds
No errors found -- did you verify all claims?
```

Figure 2.1: Verification tab of `ispin`

the next instruction, while repetition is executed repeatedly until a `break` statement is encountered. An optional else option can also be present, which may be executed only when all of the guard conditions are blocked. Both of these structures will block the execution until one of the guard conditions become enabled, which is why we needed to add the `else` options in lines 25 and 27.

For the purposes of verification we can use the `ispin` graphical interface for Spin. The *Edit/View* tab can be used to edit the opened Promela file and to view its automaton representation. In the *Verification* tab (Fig. 2.1) we can exhaustively check all states of the system for a violation of an `assert` statement or the presence of a deadlock, but we can also select an inline claim. These properties can be specified in the source code, outside of the `init` and `proctype` blocks, using the syntax `ltl [ name ] { formula }`. Any solution of the mutual exclusion problem must satisfy two important properties: first of all, it must be correct, i.e. only one component may be in the critical section at any given time, and secondly both components should enter the critical section infinitely often. In Promela these can be formulated as follows:

```
ltl mutex { [] !(critical[0] && critical[1]) }
ltl recurrence { ([] <> critical[0]) && ([] <> critical[1]) }
```

If in the *Verification* tab under *Never Claims* we select *use claim*, set `mutex` as the claim name and click run, we can successfully verify the safety property of the system, see the output in Listing 6. But if we try to check the second property, we get an error (acceptance cycle). We can select the *iterative search for short trail* option under *Search Mode* and after running again switch to the *Simulate / Replay* tab to examine the violating run. On the trail it is visible that only one component is being executed, this is because fairness constraints are not enabled by default in Spin. We can check the *enforce weak fairness constraint* under *Liveness* settings, then run again to verify that it also holds given this constraint.

```
pan: ltl formula mutex

(Spin Version 6.5.2 -- 6 December 2019)
        + Partial Order Reduction

Full statespace search for:
        never claim             + (mutex)
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)

State-vector 44 byte, depth reached 179, errors: 0
      180 states, stored
      145 states, matched
      325 transitions (= stored+matched)
        0 atomic steps
hash conflicts:          0 (resolved)

Stats on memory usage (in Megabytes):
    0.012       equivalent memory usage for states
                (stored*(State-vector + overhead))
    0.287       actual memory usage for states
  128.000       memory used for hash table (-w24)
    0.534       memory used for DFS stack (-m10000)
  128.730       total actual memory usage

pan: elapsed time 0 seconds
No errors found -- did you verify all claims?
```

Listing 6: Output of Spin for the verification of the `mutex` property

## 2.3 THE TLC MODEL CHECKER

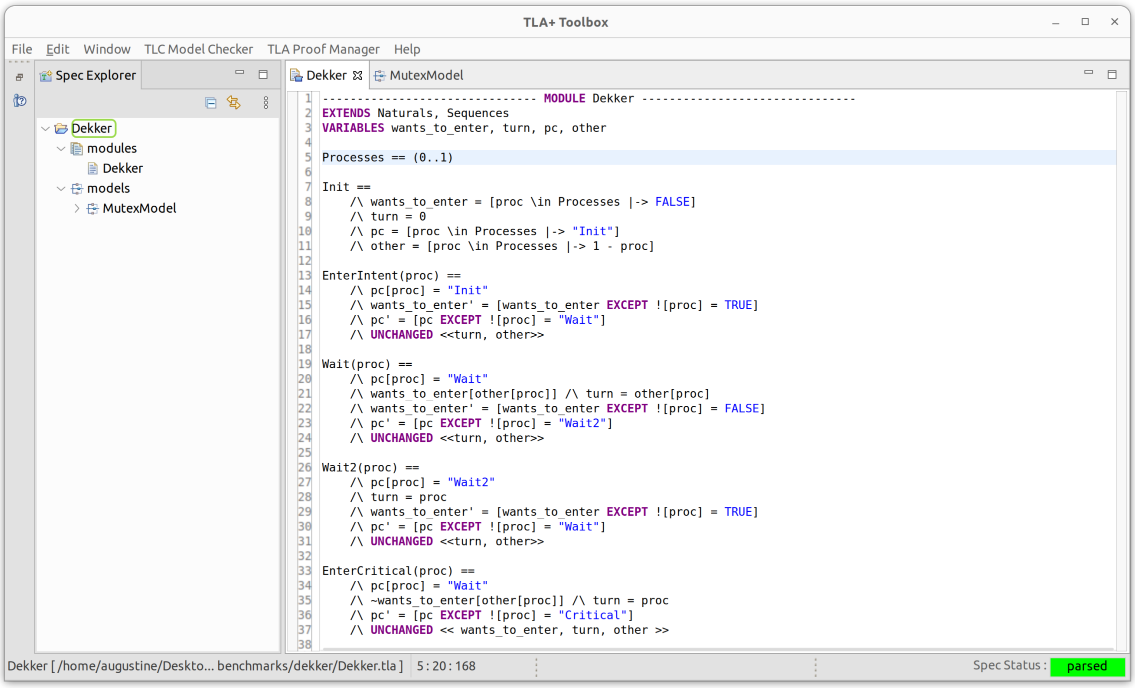
TLC is an explicit state model checker for specifications written in the TLA$^+$ language [Lam02]. It was written by Yuan Yu and extended by Markus Kuppe, and it can efficiently check both safety and liveness properties. TLC has good support for parallelization, it achieves an approximately linear speedup for safety properties on modern computers with multiple cores. It can also be ran on a network of computers.

The specification language TLA$^+$ was developed by Leslie Lamport for "modeling software above the code level and hardware above the circuit level" [Lam]. It is based on mathematical logic and does not resemble traditional programming languages. The TLA$^+$ representation of Dekker's algorithm, the same example as in the previous section, is given in Listing 7.

In TLA$^+$ transitions between system states have to be explicitly described as mathematical formulas. These have to contain their guard condition (including the program counter, which is implicit in Promela), the new values of the changed variables (the changed values of the variables are marked with a prime) and also the unchanged variables. To update one value of an array, we can use the `EXCEPT!` syntax, for example changing the value of the program counter array at the index `proc` to the value `"Critical"` is done by `pc' = [pc EXCEPT ![proc] = "Critical"]`. To specify that the variable `turn` is unchanged we could say either `turn' = turn`, or more clearly `UNCHANGED <<turn>>`.

```
------------------------------- MODULE Dekker -------------------------------
EXTENDS Naturals, Sequences
VARIABLES wants_to_enter, turn, pc, other

Processes == (0..1)

Init ==
    /\ wants_to_enter = [proc \in Processes |-> FALSE]
    /\ turn = 0
    /\ pc = [proc \in Processes |-> "Init"]
    /\ other = [proc \in Processes |-> 1 - proc]

EnterIntent(proc) ==
    /\ pc[proc] = "Init"
    /\ wants_to_enter' = [wants_to_enter EXCEPT ![proc] = TRUE]
    /\ pc' = [pc EXCEPT ![proc] = "Wait"]
    /\ UNCHANGED <<turn, other>>

Wait(proc) ==
    /\ pc[proc] = "Wait" /\ wants_to_enter[other[proc]] /\ turn = other[proc]
    /\ wants_to_enter' = [wants_to_enter EXCEPT ![proc] = FALSE]
    /\ pc' = [pc EXCEPT ![proc] = "Wait2"]
    /\ UNCHANGED <<turn, other>>

Wait2(proc) ==
    /\ pc[proc] = "Wait2" /\ turn = proc
    /\ wants_to_enter' = [wants_to_enter EXCEPT ![proc] = TRUE]
    /\ pc' = [pc EXCEPT ![proc] = "Wait"]
    /\ UNCHANGED <<turn, other>>

EnterCritical(proc) ==
    /\ pc[proc] = "Wait" /\ ~wants_to_enter[other[proc]] /\ turn = proc
    /\ pc' = [pc EXCEPT ![proc] = "Critical"]
    /\ UNCHANGED <<wants_to_enter, turn, other>>

ExitCritical(proc) ==
    /\ pc[proc] = "Critical"
    /\ turn' = other[proc]
    /\ wants_to_enter' = [wants_to_enter EXCEPT ![proc] = FALSE]
    /\ pc' = [pc EXCEPT ![proc] = "Init"]
    /\ UNCHANGED <<other>>

Next(proc) == \/ EnterIntent(proc) \/ Wait(proc) \/ Wait2(proc)
              \/ EnterCritical(proc) \/ ExitCritical(proc)

vars == <<wants_to_enter, turn, pc, other>>

Dekker == Init /\ [][\E proc \in Processes : Next(proc)]_vars
=============================================================================
```

Listing 7: TLA$^+$ representation of Dekker's solution to the mutual exclusion problem



```
---------------------------- MODULE Dekker ----------------------------
EXTENDS Naturals, Sequences
VARIABLES wants_to_enter, turn, pc, other

Processes == (0..1)

Init ==
    /\ wants_to_enter = [proc \in Processes |-> FALSE]
    /\ turn = 0
    /\ pc = [proc \in Processes |-> "Init"]
    /\ other = [proc \in Processes |-> 1 - proc]

EnterIntent(proc) ==
    /\ pc[proc] = "Init"
    /\ wants_to_enter' = [wants_to_enter EXCEPT ![proc] = TRUE]
    /\ pc' = [pc EXCEPT ![proc] = "Wait"]
    /\ UNCHANGED <<turn, other>>

Wait(proc) ==
    /\ pc[proc] = "Wait"
    /\ wants_to_enter[other[proc]] /\ turn = other[proc]
    /\ wants_to_enter' = [wants_to_enter EXCEPT ![proc] = FALSE]
    /\ pc' = [pc EXCEPT ![proc] = "Wait2"]
    /\ UNCHANGED <<turn, other>>

Wait2(proc) ==
    /\ pc[proc] = "Wait2"
    /\ turn = proc
    /\ wants_to_enter' = [wants_to_enter EXCEPT ![proc] = TRUE]
    /\ pc' = [pc EXCEPT ![proc] = "Wait"]
    /\ UNCHANGED <<turn, other>>

EnterCritical(proc) ==
    /\ pc[proc] = "Wait"
    /\ ~wants_to_enter[other[proc]] /\ turn = proc
    /\ pc' = [pc EXCEPT ![proc] = "Critical"]
    /\ UNCHANGED << wants_to_enter, turn, other >>

```

Dekker [ /home/augustine/Deskto... benchmarks/dekker/Dekker.tla ] 5 : 20 : 168 Spec Status : parsed

Figure 2.2: The TLA$^+$ Toolbox

The specification of the entire system has to be described as an LTL formula. In this case the value of `Dekker` tells us that the initial values of the variables are given by `Init` and afterwards for either one of the two processes one of the actions (listed in `Next`) has to be executed.

The same properties as before (mutual exclusion and progress) can be formulated as follows:

```
Mutex == []~(\E proc1 \in Processes, proc2 \in Processes :
          proc1 # proc2 /\ pc[proc1] = "Critical" /\ pc[proc2] = "Critical")

Recurrence == (\A proc \in Processes: []<> (pc[proc] = "Critical"))
```

While TLC can be used from the command line, just like Spin has `ispin`, TLC also has a graphical user interf2ace, namely the *TLA$^+$ Toolbox*, depicted in Fig. 2.2. For checking the progress property, we have to add weak fairness constraints to the system. As opposed to Spin, here fairness constraints are not a checkbox in the graphical interface, they are part of the model itself. To assume weak fairness on the execution of all actions we have to change the formula `Dekker` to

```
Dekker == Init /\ [][\E proc \in Processes : Next(proc)]_vars
    /\ (\A proc \in Processes :
        /\ WF_vars(EnterIntent(proc))
        /\ WF_vars(Wait(proc))
        /\ WF_vars(Wait2(proc))
        /\ WF_vars(EnterCritical(proc))
        /\ WF_vars(ExitCritical(proc))
       )
```

```
$ tlc Dekker.tla
TLC2 Version 2.17 of 02 February 2022 (rev: 3c7caa5)
...
Implied-temporal checking--satisfiability problem has 1 branches.
Computing initial states...
Finished computing initial states: 1 distinct state generated
    at 2023-04-30 21:20:29.
Progress(10) at 2023-04-30 21:20:29: 31 states generated,
    20 distinct states found, 0 states left on queue.
Checking temporal properties for the complete state space
    with 20 total distinct states at (2023-04-30 21:20:29)
Finished checking temporal properties in 00s at 2023-04-30 21:20:29
Model checking completed. No error has been found.
    Estimates of the probability that TLC did not check all reachable states
    because two distinct states had the same fingerprint:
    calculated (optimistic):  val = 1.2E-17
31 states generated, 20 distinct states found, 0 states left on queue.
The depth of the complete state graph search is 10.
The average outdegree of the complete state graph is 1
    (minimum is 0, the maximum 2 and the 95th percentile is 2).
Finished in 00s at (2023-04-30 21:20:29)
```

Listing 8: Output of TLC for the verification of the `Recurrence` property

The output of running TLC from the command line for the progress property is displayed in Listing 8.

From the output it is clear that TLC follows a probabilistic approach, the equivalent of the optional "Hash Compact" representation in Spin. This means that TLC will not store the visited states in their entirety, only their *fingerprints*. The fingerprint of a state is a hash of the values of all the variables which determine the given state. Storing only the fingerprints saves memory, since the full representation usually takes up much more memory than a single integer. It also avoids the costly comparison of states, as only these integers have to be compared to each other. However this approach may overlook some states with a low but non-zero probability.

In our implementation we make sure that no states are missed, but this of course means it will be inherently slower than TLC.

# 3 LINEAR TEMPORAL LOGIC

In this chapter we introduce the mathematical framework that will be used in the following, namely Kripke-structures and linear temporal logic, and finally we use these to formalise the model checking problem.

## 3.1 KRIPKE-STRUCTURES

Kripke-structures, proposed by Saul Kripke in 1963 [Kri63] are a variant of transition systems used to model the behaviour of non-deterministic systems. A Kripke-structure describes the transitions between the states and also the properties which hold in each state.

**Definition 3.1.1.** A relation $R \subseteq X \times Y$ is *total* if $\forall x \in X\ \exists y \in Y\colon xRy$.

**Definition 3.1.2.** A *Kripke-structure* $K$ over a set of atomic propositions $\mathcal{P}$ 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 $\mathcal{L}\colon S \to \mathbb{P}(\mathcal{P})$

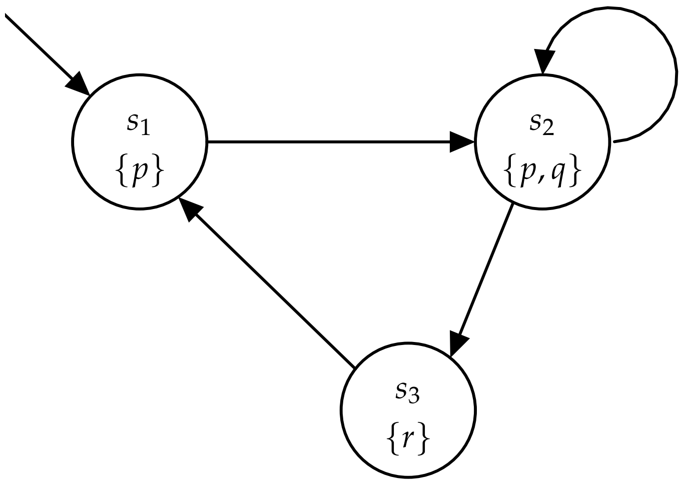


Figure 3.1: Example of a Kripke structure

For example, the Kripke-structure in Fig. 3.1 has three states $S = \{s_1, s_2, s_3\}$ with the initial state being $s_1$. The transition relation is $T = \{(s_1, s_2), (s_2, s_2), (s_2, s_3), (s_3, s_1)\}$ and the labelling function $\mathcal{L}$ maps $s_1$ to $\{p\}$, $s_2$ to $\{p, q\}$ and $s_3$ to $\{r\}$.

The atomic propositions are chosen to be the system properties we care about in this model, for example *cabinStopped* or *doorOpen*, while the labelling function determines in which states they hold.

In the following we want to study the behaviour of the system as it evolves, and this is done by examining its runs. When modelled by a Kripke-structure, the equivalent of a system run is a sequence of states, called a *trace*.

**Definition 3.1.3.** A *trace* $\pi = (s_0, s_1, ...)$ of a Kripke-structure $K = (S, I, T, \mathcal{L})$ is a finite or infinite sequence of states of $K$, such that $\forall i \colon s_i \xrightarrow{T} s_{i+1}$:
$|\pi|$ is the *length* of $\pi$ (e.g. $|\pi| = 2$ for $\pi = (s_0, s_1, s_2)$ and $|\pi| = \infty$ for infinite traces).
$\pi(i)$ is the i-th state $s_i$ of $\pi$ for $i \leq |\pi|$.
$\pi^i = (s_i, s_{i+1}, ...)$ denotes the suffix of $\pi$ starting with the i-th state $s_i$ for $i \leq |\pi|$.

## 3.2 LINEAR TEMPORAL LOGIC

Temporal logic, introduced by Jerzy Łoś and Arthur Prior [TJ19] in the mid-20th century, is any mathematical formalism used to represent and reason about propositions qualified in terms of time (e.g. *eventually* the thesis will be finished).

In general we may have two different views of the future: one in which the sequence of events is determined right at the beginning, which is described by linear temporal logic (LTL). In the other, the future is at every point indeterminate and branching, this is described by computation tree logic (CTL). Neither of these is stronger than the other, because there exist properties which can be described in only one of them. The logic combining these two views is called CTL*, a superset of LTL and CTL. For more about temporal logics and LTL, see [DGL16].

For the purposes of formal verification, LTL was first proposed by Amir Pnueli [Pnu77]. Since LTL is strong enough to describe most interesting properties of system runs and humans generally have an easier time reasoning about it than about CTL, in the future we restrict ourselves to linear temporal logic. Now we define the syntax and semantics of a typical version of LTL.

**Definition 3.2.1.** The alphabet of LTL consists of a set of atomic propositions $\mathcal{P}$, 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.

**Definition 3.2.2.** The semantics of LTL for an infinite trace $\pi$ of a Kripke-structure $K = (S, I, T, \mathcal{L})$ is defined as follows:

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

**Definition 3.2.3.** For a Kripke-structure $K = (S, I, T, \mathcal{L})$ and an LTL formula $f$ we have $K \models f$ if $\pi \models f$ for all infinite traces $\pi$ of $K$ such that $\pi(0) \in I$.

Let us examine the Kripke-structure in Fig. 3.1 and describe some of its properties using LTL:

- Since $p \in \mathcal{L}(s_1)$ we have $K \models p$, but because $q \notin \mathcal{L}(s_1)$, $K \not\models q$.
- Because the second state in every trace is $s_2$ and $p, q \in \mathcal{L}(s_2)$, $K \models \mathbf{X}p$ and $K \models \mathbf{X}q$.
- Again by the same reasoning $K \models \mathbf{F}q$, but since no state has both $r$ and $p$ true, $K \not\models \mathbf{F}(r \wedge p)$. This last one we could also formulate as $K \models \neg\mathbf{F}(r \wedge p)$ or as $K \models \mathbf{G}\neg(r \wedge p)$, since LTL has a variant of De Morgan's rule.
- Since $(s_1, s_2, s_2, s_2, ...)$ is a valid trace, $K \not\models p\,\mathbf{U}\,r$, but $K \models (p\,\mathbf{U}\,r) \vee (\mathbf{G}p)$. This property can also be written as $K \models p\,\mathbf{W}\,r$ using the "weak until" operator $\mathbf{W}$.

Now we have all the mathematical groundwork we need to formally describe the problem we are attempting to solve, called the model checking problem.

**Definition 3.2.4.** *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 $\pi$ of $K$ such that $\pi \not\models f$.

This is the problem whose solution will be studied in the next chapters.

# 4 AUTOMATA THEORY

We continue with the basics of automata theory, which will later be needed to describe the model checking algorithm.

## 4.1 FINITE AUTOMATA

**Definition 4.1.1.** A *finite automaton* is defined as the tuple $(S, I, \Sigma, T, F)$ 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 transition relation $T \subseteq S \times \Sigma \times S$
- a set of accepting states $F \subseteq S$

We write $s \xrightarrow{a} s'$ if there is a transition from state $s$ to $s'$ with the label $a$, i.e. $(s, a, s') \in T$. The theory of finite automata is concerned with the labels of traces going from an initial state to an accepting state. Formally:

**Definition 4.1.2.** The *set of words* $\Sigma^*$ over an alphabet $\Sigma$ is the set

$$\Sigma^* = \bigcup_{n \geq 0} \{(a_1, a_2, ..., a_n) : a_1, a_2, ..., a_n \in \Sigma\}$$

**Definition 4.1.3.** A finite automaton $A = (S, I, \Sigma, T, F)$ *accepts* a word $w = a_1 a_2 ... a_n \in \Sigma^*$ iff there exist $s_0, ..., s_n \in S$ with

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} ... \xrightarrow{a_{n-1}} s_{n-1} \xrightarrow{a_n} s_n$$

where $n \geq 0$, $s_0 \in I$, and $s_n \in F$.

**Definition 4.1.4.** The *language* $\mathscr{L}(A)$ of the finite automaton $A$ is the set of words accepted by $A$.

On a first glance it might seem that the language is a tool to study the automata, but in most cases it is the opposite: automata are usually constructed to accept a given language.

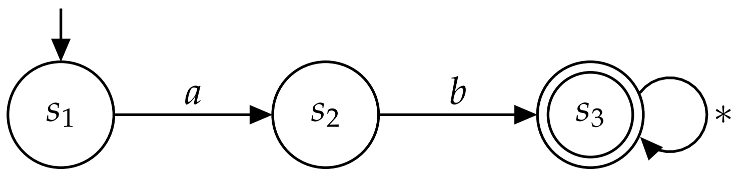
Then they can be used to efficiently check if a word is part of that language. One example would be to construct the automaton that accepts the intersection of two languages.

**Definition 4.1.5.** The *product automaton* $A = A_1 \times A_2$ of two finite automata $A_1 = (S_1, I_1, \Sigma_1, T_1, F_1)$ and $A_2 = (S_2, I_2, \Sigma_2, T_2, F_2)$ over the same alphabet $\Sigma_1 = \Sigma_2$ is defined as the tuple $(S, I, \Sigma, T, F)$ consisting of the following components:

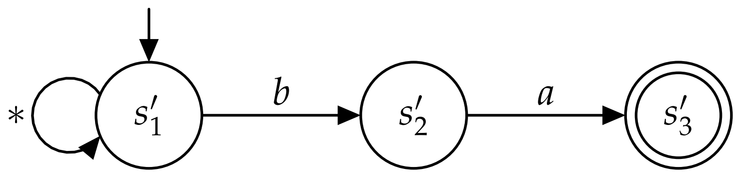
- $S = S_1 \times S_2$
- $I = I_1 \times I_2$
- $\Sigma = \Sigma_1 = \Sigma_2$
- $F = F_1 \times F_2$
- $T((s_1, s_2), a, (s'_1, s'_2))$ iff $T_1(s_1, a, s'_1)$ and $T_2(s_2, a, s'_2)$

**Proposition 4.1.1.** Given two finite automata $A_1$ and $A_2$, if $A = A_1 \times A_2$, then $\mathscr{L}(A) = \mathscr{L}(A_1) \cap \mathscr{L}(A_2)$.

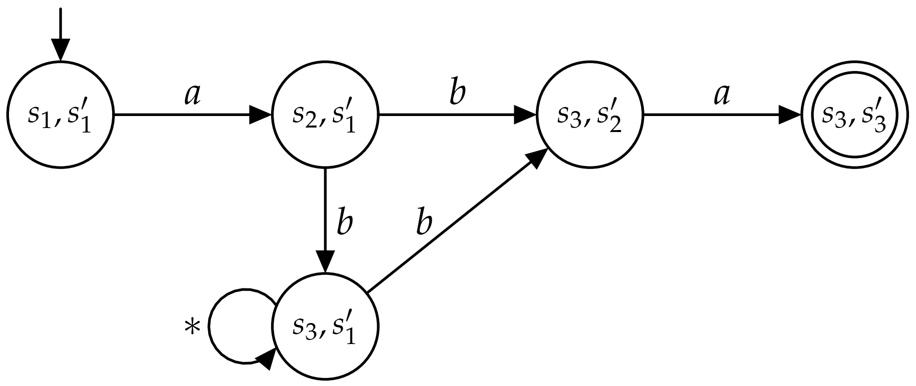
As an example, let $\Sigma$ be the Latin alphabet, and let us construct the automaton $A_1$ which accepts all words with prefix *ab* ($\mathscr{L}(A_1) = \{ab\mathbf{w} : \mathbf{w} \in \Sigma^*\}$), the automaton $A_2$ which accepts all words with the suffix *ba* ($\mathscr{L}(A_2) = \{\mathbf{w}ba : \mathbf{w} \in \Sigma^*\}$) and the automaton $A$ which accepts the intersection of the two languages.



(a) Finite automaton $A_1$ with the language $\mathscr{L}(A_1) = \{ab\mathbf{w} : \mathbf{w} \in \Sigma^*\}$



(b) Finite automaton $A_2$ with the language $\mathscr{L}(A_2) = \{\mathbf{w}ba : \mathbf{w} \in \Sigma^*\}$



(c) Finite automaton $A$ with the language $\mathscr{L}(A) = \{ab\mathbf{w} : \mathbf{w} \in \Sigma^*\} \cap \{\mathbf{w}ba : \mathbf{w} \in \Sigma^*\}$, unreachable states are not shown.

Figure 4.1: Examples of finite automata

While in the first automaton in Fig. 4.1 for each character in the word we know how to proceed to check if it is accepted, the second and third automata involve non-deterministic choices.

**Definition 4.1.6.** For a finite automaton $A = (S, I, \Sigma, T, F)$, a state $s \in S$ and a character $a \in \Sigma$, let $s \xrightarrow{a}$ denote the *set of successors* of $s$ defined as

$$s \xrightarrow{a} = \{s' \in S : T(s, a, s')\}.$$

**Definition 4.1.7.** A finite automaton $A = (S, I, \Sigma, T, F)$ is *complete* if $|I| > 0$ and $|s \xrightarrow{a}| > 0$ for all $s \in S$ and $a \in \Sigma$.

**Definition 4.1.8.** A finite automaton $A = (S, I, \Sigma, T, F)$ is *deterministic* if $|I| \leq 1$ and $|s \xrightarrow{a}| \leq 1$ for all $s \in S$ and $a \in \Sigma$.

Being non-deterministic or incomplete is not a problem in general, still deterministic and complete automata have a few desirable properties. It turns out that given a non-deterministic automaton $A_n$, we can construct a deterministic and complete automaton $A_d$ with the property that $\mathscr{L}(A_d) = \mathscr{L}(A_n)$.

**Definition 4.1.9.** The *power-automaton* of a finite automaton $A = (S, I, \Sigma, T, F)$, denoted by $\mathcal{P}(A)$ consists of the components $(S', I', \Sigma', T', S)$ where:

- $S' = 2^S$
- $I' = \{I\}$
- $\Sigma' = \Sigma$
- $F' = \{F_0 \subseteq S : F_0 \cap F \neq \emptyset\}$
- $T(s, a, s')$ iff $s' = \bigcup_{s_0 \in s} s \xrightarrow{a}$

**Proposition 4.1.2.** The power-automaton $\mathcal{P}(A)$ of a finite automaton $A$ is deterministic, complete and has the property $\mathscr{L}(\mathcal{P}(A)) = \mathscr{L}(A)$.

Unfortunately the above construction involves an exponential blowup in the number of states, but we can use the power-automaton as an intermediate step for creating one which accepts the complement of a language.

**Definition 4.1.10.** The *complement-automaton* of a finite automaton $A = (S, I, \Sigma, T, F)$, denoted by $\mathcal{C}(A)$, consists of the components $(S, I, \Sigma, T, S \setminus F)$.

**Proposition 4.1.3.** The complement-automaton $\mathcal{C}(A)$ of a deterministic and complete finite automaton $A$ accepts the complement of the language of $A$:

$$\mathscr{L}(\mathcal{C}(A)) = \overline{\mathscr{L}(A)} = \Sigma^* \setminus \mathscr{L}(A)$$

Combining these two steps indeed yields the automaton that accepts the complement of a language defined by a non-deterministic automaton. We might hope that the intermediate step is not necessary, but as shown by [MF71] the exponential blowup in the worst case is unavoidable.

## 4.2 BÜCHI AUTOMATA

Since we attempt to verify infinite system runs, for our purposes the previously described automata are not exactly the right tool. Nevertheless a basic familiarity with the finite case is a very useful in understanding the infinite one. Büchi automata were introduced by Julius Richard Büchi in 1960 for the purposes of studying infinite words [Büc60]. The thesis will present a slightly modified version, which makes the algorithms a bit easier to understand.

**Definition 4.2.1.** A *labelled Büchi automaton* is defined as the tuple $(S, I, \Sigma, \mathcal{L}, T, F)$ 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 $\mathcal{L} \colon S \to 2^{\Sigma}$
- a transition relation $T \subseteq S \times S$
- a set of accepting states $\mathcal{F}$

Notice that as opposed to the regular finite automata, here not the transitions but the states are labelled with potentially multiple labels. This construction is therefore essentially the extension of a Kripke structure by an acceptance condition. Just like it was the case for finite automata, we are again interested in the language accepted by the automaton.

**Definition 4.2.2.** An *accepting execution* $\sigma$ of a Büchi automaton $\mathcal{A} = (S, I, \Sigma, \mathcal{L}, T, F)$ is an infinite sequence of states $\sigma = s_0 s_1 s_2 ... \in S^{\omega}$ such that $s_0 \in I$ and there exists at least one state $s \in F$ which appears infinitely often in $\sigma$.

The definition above considers only sequences of states, but in the following we will need acceptance based on the labelling.

**Definition 4.2.3.** The *set of infinite words* $\Sigma^{\omega}$ over an alphabet $\Sigma$ is the set

$$\Sigma^{\omega} = \{(a_1, a_2, ...) : a_1, a_2, ... \in \Sigma\}$$

**Definition 4.2.4.** A Büchi automaton $\mathcal{A} = (S, I, \Sigma, \mathcal{L}, T, F)$ *accepts* a word $w = a_0 a_1 a_2 ... \in \Sigma^{\omega}$ if there exists an accepting execution $\sigma = s_0 s_1 s_2 ... \in S^{\omega}$ such that for each $i \geq 0$, $a_i \in \mathcal{L}(s_i)$.

One might be tempted to think that constructing the Büchi automaton which accepts the intersection of the languages of two Büchi automata can be done like it was done for finite automata, but unfortunately that is not the case. Consider for example $\mathcal{A}_1 = (\{s_1, t_1\}, \{s_1\}, \{a\}, \mathcal{L}_1, \{(s_1, t_1), (t_1, s_1)\}, \{t_1\})$ with $\mathcal{L}_1(s_1) = \mathcal{L}_1(t_1) = \{a\}$ and $\mathcal{A}_2 = (\{s_2, t_2\}, \{s_2\}, \{a\}, \mathcal{L}_2, \{(s_2, t_2), (t_2, s_2)\}, \{s_2\})$ with $\mathcal{L}_2(s_2) = \mathcal{L}_2(t_2) = \{a\}$. Then clearly the intersection of the languages contains the word $a^\omega$, but the language of the naively constructed product automaton is empty, since no run ever goes through the only accepting state $(t_1, s_2)$.

Nevertheless it is possible to construct the automaton corresponding to the intersection of the languages, but for that first we introduce a variant of Büchi automata with a generalized acceptance condition.

**Definition 4.2.5.** A *labelled generalized Büchi automaton (LGBA)* consists of the same components $(S, I, \Sigma, \mathcal{L}, T, \mathcal{F})$ as a simple labelled Büchi automaton, except that the accepting set is replaced by a set of accepting sets $\mathcal{F} \subseteq 2^S$, $\mathcal{F} = \{F_1, F_2, ..., F_n\}$.

**Definition 4.2.6.** An *accepting execution* $\sigma$ of an LGBA $\mathcal{A} = (S, I, \Sigma, \mathcal{L}, T, \mathcal{F})$ is an infinite sequence of states $\sigma = s_0 s_1 s_2 ... \in S^\omega$ such that $s_0 \in I$ and for each $i \geq 0$, $s_i \to s_{i+1}$, 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 $\sigma$.

Any labelled Büchi automaton can be trivially mapped to an LGBA, by replacing $F$ with $\{F\}$. Note that the set of sets of accepting states may be empty. In this case every execution is accepting. Using LGBAs we can finally give a simple construction for the automaton corresponding to the intersection of languages.

**Proposition 4.2.1.** Given two LGBA $\mathcal{A}_i = (S_i, I_i, \Sigma, \mathcal{L}_i, T_i, \mathcal{F}_i)$, $i \in \{1, 2\}$, there exists an LGBA $\mathcal{A} = (S, I, \Sigma, \mathcal{L}, T, \mathcal{F})$ such that $\mathscr{L}(\mathcal{A}) = \mathscr{L}(\mathcal{A}_1) \cap \mathscr{L}(\mathcal{A}_2)$ consisting of the following components:

- $S = S_1 \times S_2$
- $I = I_1 \times I_2$
- $\mathcal{L}(s_1, s_2) = \mathcal{L}_1(s_1) \cap \mathcal{L}_2(s_2)$
- $((s_1, s_2), (t_1, t_2)) \in T$ iff $(s_1, t_1) \in T_1$ and $(s_2, t_2) \in T_2$
- $\mathcal{F} = \{F \times S_2 : F \in \mathcal{F}_1\} \cup \{S_1 \times F : F \in \mathcal{F}_2\}$

Now we just need a way to translate an LGBA back to a regular labelled Büchi automaton. Fortunately there exists a simple construction with size $O(|S| \cdot |\mathcal{F}|)$.

**Proposition 4.2.2.** Given an LGBA $\mathcal{A} = (S, I, \Sigma, \mathcal{L}, T, \mathcal{F})$ there exists a Büchi automaton $\mathcal{A}' = (S', I', \Sigma, \mathcal{L}', T', F')$ such that $|S'| = |S| \times k$, where $k = |\mathcal{F}|$ and $\mathscr{L}(\mathcal{A}) = \mathscr{L}(\mathcal{A}')$ consisting of the following components:

1. $S' = S \times \{1, 2, ..., k\}$
2. $I' = I \times \{1\}$
3. $\mathcal{L}'((s, i)) = \mathcal{L}(s)$
4. $((s, i), (t, j)) \in T'$ iff $(s, t) \in T$ and $s \notin F_i \wedge i = j$ or $s \in F_i \wedge j \equiv i + 1 \pmod{k}$
5. $F' = F_1 \times \{1\}$

In the next chapter we will describe a procedure which uses Büchi automata and their languages to decide the validity of a temporal formula for a given system.

# 5 AUTOMATON-BASED MODEL CHECKING

Using the notions defined in the previous chapter we may now describe the automaton-based model checking procedure:

1. Construct an LGBA $\mathcal{A}_{\neg f}$ which accepts the set of words that satisfy the negation of the formula $f$.
2. Given the Kripke-structure $K = (S, I, T, \mathcal{L})$ of the system, construct the LGBA $\mathcal{A}_K = (S, I, 2^{\mathcal{P}}, \mathcal{L}', T, \emptyset)$ with $\mathcal{L}'(s) = \{\mathcal{L}(s)\}$ for any $s \in S$.
3. Construct the automaton which accepts the intersection of the languages of $\mathcal{A}_{\neg f}$ and $\mathcal{A}_K$.
4. Check if the language of the resulting automaton is non-empty: if so, there is a counterexample to the property, otherwise the property holds.

In the next sections we describe the algorithms necessary provide a complete implementation of this procedure, the translation and the emptiness checking.

## 5.1 TRANSLATION OF LTL FORMULAS TO BÜCHI AUTOMATA

This section provides the details of an efficient algorithm solving the following problem: given an LTL formula $f$ over the set of atomic propositions $\mathcal{P}$, construct an LGBA $\mathcal{A}$ such that for every trace $\pi = s_0 s_1 s_2 \ldots$ we have that $\pi \models f$ if and only if there exists a word $w = a_0 a_1 a_2 \ldots$ accepted by $\mathcal{A}$ such that the label of state $s_i$ is exactly $a_i$. This algorithm was first described in [Ger+96] and it has proven to be practical in the model checker SPIN.

First given the formula $f$, we replace the temporal operators **F** and **G** using $\mathbf{F}p \equiv \top \mathbf{U}\, p$ and $\mathbf{G}p \equiv \bot \mathbf{V}\, p$, where **V** is defined as the dual of **U**: $f \mathbf{V} g \equiv \neg(\neg f \mathbf{U} \neg g)$. Then we convert $f$ into negation normal form by pushing the negations to in front of propositions using De Morgan's laws. The operator **V** was introduced to make sure this step can be done while avoiding an exponential blowup in the size of the formula.

Algorithm 5.1 gives the pseudocode representation of the algorithm. The data structure for the graph nodes contains the following fields:

**Incoming** contains the incoming edges as references to the nodes with outgoing edges to the current node. A special value, **init** is used to denote initial nodes, it does not represent a real edge.

**New** is a set of temporal properties that must hold at in current node and have not been processed yet.

**Old** is the set of temporal properties that must hold in the current node and have been already processed. During the execution of the algorithm, every formula from **New** will be processed and moved to **Old**.

**Next** is the set of temporal properties that must hold in all states that are immediate successors of states satisfying the properties in **Old**.

The nodes whose construction was already finished are stored in the set `nodesSet` which is used to optimize the constructed automaton by merging equivalent nodes.

The algorithm starts with a node (lines 1 – 3) with **init** as its only incoming edge. As mentioned before, having **init** among the incoming edges denotes that it is an initial node. The set `new` is initialized to the formula for which we want to construct the automaton, while old and next are initialized to an empty set. `nodesSet` is also initialized to empty, since no node has been finalized yet.

The bulk of the algorithm is contained in the `expand` function (lines 4 – 32), which splits or transforms graph nodes based on the yet unprocessed constraints. It starts by checking if the currently processed node has any requirements it still needs to satisfy (line 5). If not, it means that the construction of the current node was finished. This is the point where the `nodesSet` is used to check if an equivalent node with the same `old` and `next` already exists (line 6). If so, the two nodes are merged by taking the union of the `incoming` set, and returning `nodesSet` unchanged (lines 7 – 8). Otherwise the current node is added to `nodesSet` and the construction of a new child node is started from the constraints stored in the `next` set of the current node (line 10).

In the second part of the function (lines 13 – 30) we take an arbitrary element of the yet unprocessed constraints and remove it from the set. The node is then processed based on the shape of the formula.

In the first case (lines 15 – 21) the formula $f$ is an atomic proposition, the negation of one, or it is one of the literals $\top$ or $\bot$. If the formula is $\bot$, or `old` contains the negation of the formula (lines 16 – 17) then this node contains a contradiction, so we discard it and return `nodesSet` unchanged. Otherwise we simply add the formula to `old` and continue expanding the node (lines 18 – 20).

In the second case (lines 22 – 23) the formula has at its root the temporal operator $\mathbf{X}$. The formula is simply moved to `old` and the subformula under $\mathbf{X}$ is added to `next` and we continue the expansion.

In the third case (lines 24 – 25) the formula is a conjunction. The formula is once again moved to `old` and the two subformulas of the conjunction are added to `new`, excepting the ones which are already processed and are therefore in `old`. Then we continue the expansion.

The fourth case (lines 26 – 30) handles the operators $\vee$, $\mathbf{U}$ and $\mathbf{V}$. In these cases the current node will be split into two new nodes based on the operator at the root of the formula.

**Algorithm 5.1** Construction of a tableau for an LTL formula

```
1: procedure CREATE_GRAPH($f$)                                   ▷ LTL formula $f$
2:     return EXPAND({incoming: init, new: {f}, old: {}, next: {}}, {})
3: end procedure

4: procedure EXPAND(node, nodesSet)
5:     if node.new is empty then
6:         if there is a graph node n $\in$ nodesSet
               with n.old = node.old and n.next = node.next then
7:             n.incoming $\leftarrow$ n.incoming $\cup$ node.incoming
8:             return nodesSet
9:         else
10:            return EXPAND({incoming: {node}, new: node.next, old: {}, next: {}},
                 nodesSet $\cup$ {node})
11:        end if
12:    else
13:        let f $\in$ node.new
14:        node.new $\leftarrow$ node.new $\setminus \{f\}$
15:        if f $= p_i$ or f $= \neg p_i$ or f $= \top$ or f $= \bot$ then
16:            if f $= \bot$ or $\neg$ f $\in$ node.old then
17:                return nodesSet
18:            else
19:                node.old $\leftarrow$ node.old $\cup$ {f}
20:                return EXPAND(node, nodesSet)
21:            end if
22:        else if $f = \mathbf{X}\, g$ then
23:            return EXPAND({incoming: node.incoming, new: node.new,
                 old: node.old $\cup$ {f}, next: node.next $\cup$ {g}}, nodesSet $\cup$ {node})
24:        else if $f = g \wedge h$ then
25:            return EXPAND({incoming: node.incoming, new: node.new $\cup$ ({g, h}
                 $\setminus$ node.old), old: node.old $\cup$ {f}, next: node.next}, nodesSet $\cup$ {node})
26:        else if $f = g \vee h$ or $f = g\,\mathbf{U}\,h$ or $f = g\,\mathbf{V}\,h$ then
27:            node1 $\leftarrow$ { incoming: node.incoming, new: node.new $\cup$ (new1(f) $\setminus$ node.old),
                 old: node.old $\cup$ {f}, next: node.next $\cup$ next1(f) }
28:            node2 $\leftarrow$ { incoming: node.incoming, new: node.new $\cup$ (new2(f) $\setminus$ node.old),
                 old: node.old $\cup$ {f}, next: node.next }
29:            return EXPAND(node2, EXPAND(node1, nodesSet))
30:        end if
31:    end if
32: end procedure
```

$f = g \vee h$ : the node is split by adding $g$ to the `new` set of `node1` and $h$ to `node2`. These correspond to the two ways $g \vee h$ can be made to hold.

$f = g\,\mathbf{U}\,h$ : the node is split by adding $g$ to the `new` set and $g\,\mathbf{U}\,h$ to the `next` set of `node1` and $h$ to the `new` set of `node2`. These correspond to the fact that $g\,\mathbf{U}\,h$ is equivalent to $h \vee (g \wedge \mathbf{X}(g\,\mathbf{U}\,h))$.

$f = g\,\mathbf{V}\,h$ : the node is split by adding $h$ to the `new` and $g\,\mathbf{V}\,h$ to the `next` of `node1`, while adding both $g$ and $h$ to the `new` set of `node2`. These correspond to the fact that $g\,\mathbf{V}\,h$ is equivalent to $h \wedge (g \vee \mathbf{X}(g\,\mathbf{V}\,h))$.

The three sub-cases above are represented in the algorithm succinctly using the functions **new1**, **next1**, and **new2** which are defined in the following table:

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

Listing 9 depicts the behaviour of the algorithm through a concrete example by transforming the property $p\,\mathbf{U}\,q$ to an automaton. After these steps we end up with the structure presented in Fig. 5.1.

```
Processing is started with node0 = { incoming: [init], new: [p U q] }
node0 is split into
        node1 = { incoming: [init], new: [p], old: [p U q], next: [p U q] }
        and node2 = { incoming: [init], new: [q], old: [p U q] }
Moving literal p in node1 to the set old.
        Result: node1 = { incoming: [init], old: [p U q, p], next: [p U q] }
node1 has no more properties in the set new.
        New child node is added: node3 = { incoming: [node1], new: [p U q] }
node3 is split into
        node4 = { incoming: [node1], new: [p], old: [p U q], next: [p U q] }
        and node5 = { incoming: [node1], new: [q], old: [p U q] }
Moving literal p in node4 to the set old.
        Result: node4 = { incoming: [node1], old: [p U q, p], next: [p U q] }
node4 is equivalent to node1. Merging incoming edges.
        Result: node1 = { incoming: [node1, init], old: [p U q, p], next: [p U q] }
Moving literal q in node5 to the set old.
        Result: node5 = { incoming: [node1], old: [p U q, q] }
node5 has no more properties in the set new.
        New child node is added: node6 = { incoming: [node5] }
node6 has no more properties in the set new.
        New child node is added: node7 = { incoming: [node6] }
node7 is equivalent to node6. Merging incoming edges.
        Result: node6 = { incoming: [node6, node5] }
Moving literal q in node2 to the set old.
        Result: node2 = { incoming: [init], old: [p U q, q] }
node2 is equivalent to node5. Merging incoming edges.
        Result: node5 = { incoming: [node1, init], old: [p U q, q] }
```

Listing 9: Execution trace of the algorithm for the formula $p\,\mathbf{U}\,q$

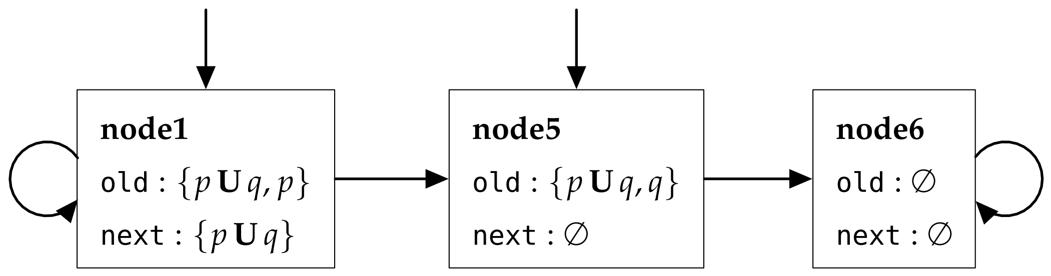

node1
old : $\{p\,\mathbf{U}\,q, p\}$
next : $\{p\,\mathbf{U}\,q\}$
node5
old : $\{p\,\mathbf{U}\,q, q\}$
next : $\emptyset$
node6
old : $\emptyset$
next : $\emptyset$

Figure 5.1: Resulting structure after execution of the algorithm for $p\,\mathbf{U}\,q$

It remains to be shown how the LGBA $\mathcal{A}_f = (S, I, \Sigma, \mathcal{L}, T, \mathcal{F})$ of a formula $f$ results from the graph created by the algorithm.

- The set of states $S$ is the set of nodes returned by the algorithm.
- The set of initial states $I$ is the set of all the nodes n for which **init** $\in$ `n.incoming`.
- For the transition $T$ we have that $(\mathtt{n}_1, \mathtt{n}_2) \in T$ iff $\mathtt{n}_1 \in \mathtt{n}_2$`.incoming`.
- The alphabet $\Sigma$ is the set of all combinations in which the atomic propositions $\mathcal{P}$ may hold: $\Sigma = 2^{\mathcal{P}}$
- The labelling function $\mathcal{L}$ maps each state n to the set of all sets of atomic propositions, which are compatible with the literals in `n.old`.
  Formally let $\mathrm{pos}(\mathtt{n}) = \{p \in \mathcal{P} : p \in \mathtt{n.old}\}$ and $\mathrm{neg}(\mathtt{n}) = \{p \in \mathcal{P} : \neg p \in \mathtt{n.old}\}$ then $\mathcal{L}(\mathtt{n}) = \{l \in \Sigma : \mathrm{pos}(\mathtt{n}) \in l \wedge \mathrm{neg}(\mathtt{n}) \cap l = \emptyset\}$.
- Finally the acceptance set $\mathcal{F}$ consists of sets $F_i \subseteq S$ for each subformula of $f$ of the form $g\,\mathbf{U}\,h$ such that $\mathtt{n} \in F_i$ if $g\,\mathbf{U}\,h \notin \mathtt{n.old}$ or $h \in \mathtt{n.old}$.

Given the structure returned by the algorithm on Fig. 5.1 and the rules above let us construct the labelled generalized Büchi automaton $\mathcal{A}_{p\,\mathbf{U}\,q} = (S, I, \Sigma, \mathcal{L}, T, \mathcal{F})$ for the formula $p\,\mathbf{U}\,q$ over the set of atomic predicates $\mathcal{P} = \{p, q, r\}$.

- $S = \{\mathtt{node1}, \mathtt{node5}, \mathtt{node6}\}$
- $I = \{\mathtt{node1}, \mathtt{node5}\}$
- $T = \{(\mathtt{node1}, \mathtt{node1}), (\mathtt{node1}, \mathtt{node5}), (\mathtt{node5}, \mathtt{node6}), (\mathtt{node6}, \mathtt{node6})\}$
- $\Sigma = \{\emptyset, \{p\}, \{q\}, \{r\}, \{p, q\}, \{p, r\}, \{q, r\}, \{p, q, r\}\}$
- $\mathcal{L}(\mathtt{node1}) = \{\{p\}, \{p, q\}, \{p, r\}, \{p, q, r\}\}$
  $\mathcal{L}(\mathtt{node5}) = \{\{q\}, \{p, q\}, \{q, r\}, \{p, q, r\}\}$
  $\mathcal{L}(\mathtt{node6}) = \Sigma$
- $\mathcal{F} = \{\{\mathtt{node5}, \mathtt{node6}\}\}$

This automaton is depicted in Fig. 5.2.

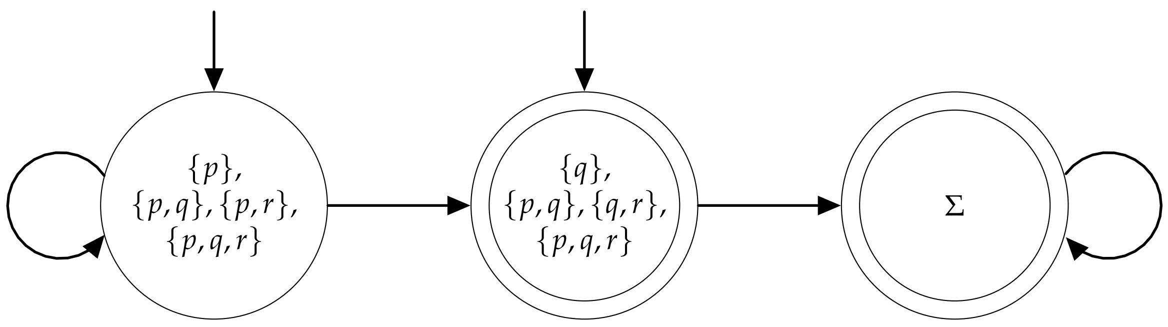

$\{p\}$, $\{p,q\}$, $\{p,r\}$, $\{p,q,r\}$
$\{q\}$, $\{p,q\}$, $\{q,r\}$, $\{p,q,r\}$
$\Sigma$

Figure 5.2: LGBA corresponding to the formula $p\,\mathbf{U}\,q$

### 5.1.1 *Proof of correctness*

In this subsection we will show that the automaton generated using this algorithm is correct, meaning

(i) every word accepted by the automaton $\mathcal{A}_\varphi$ is a model of $\varphi$

(ii) every model of $\varphi$ is accepted by the automaton $\mathcal{A}_\varphi$

The proof follows the one described in [Ger+96], with slight changes in notation and some added detail. Where a proof is not provided, it is immediate from the structure of the algorithm.

Given the LGBA $\mathcal{A} = (S, I, \Sigma, \mathcal{L}, T, \mathcal{F})$ for any $s \in S$ we will denote by $\Delta(s)$ the value of `q.old`, where `q` is the node corresponding to the state $s$ in the structure generated by the algorithm.

**Lemma 5.1.1.** Let $\sigma = s_0 s_1 s_2 ... \in S^\omega$ be an execution of $\mathcal{A}$, and let $g\,\mathbf{U}\,h \in \Delta(s_0)$. Then one of the following conditions holds:

(i) $\forall i \geq 0\colon \{g, g\,\mathbf{U}\,h\} \subseteq \Delta(s_i) \wedge h \notin \Delta(s_i)$

(ii) $\exists j \geq 0\colon h \in \Delta(s_j) \wedge \forall i$ with $0 \leq i < j\colon \{g, g\,\mathbf{U}\,h\} \subseteq \Delta(s_i)$

*Proof.* Notice how no formula is ever removed from `old`, and when splitting nodes containing the temporal operator $\mathbf{U}$, one child will have $g$ in its `old`, while the other has both $f\,\mathbf{U}\,g$ and $f$, and it's descendants again start with $f\,\mathbf{U}\,g$ in their `new`. This ensures that the only two possibilities for any execution are the ones listed above. $\square$

**Lemma 5.1.2.** Let `q` be a node during the construction which is split into two new nodes $\mathtt{q}_1$ and $\mathtt{q}_2$ in lines 26 – 29. Then the following holds:

$$(\bigwedge \mathtt{q.old} \wedge \bigwedge \mathtt{q.new} \wedge \mathbf{X}(\bigwedge \mathtt{q.next})) \Leftrightarrow$$
$$(\bigwedge \mathtt{q}_1\mathtt{.old} \wedge \bigwedge \mathtt{q}_1\mathtt{.new} \wedge \mathbf{X}(\bigwedge \mathtt{q}_1\mathtt{.next})) \vee (\bigwedge \mathtt{q}_2\mathtt{.old} \wedge \bigwedge \mathtt{q}_2\mathtt{.new} \wedge \mathbf{X}(\bigwedge \mathtt{q}_2\mathtt{.next}))$$

*Proof.* Follows from the fact that for any LTL formula $f$ of the form $g\,\mathbf{U}\,h$, $g\,\mathbf{V}\,h$ or $g \vee h$ we have $f \equiv (\bigwedge \mathbf{new1}(f) \wedge \mathbf{X}(\bigwedge \mathbf{next1}(f)) \vee \bigwedge \mathbf{new2}(f)$ $\square$

**Lemma 5.1.3.** Let $\mathtt{q}$ be a node during the construction which is updated to $\mathtt{q}'$ in lines 22 – 23 or 24 – 25. Then the following holds:

$$(\bigwedge \mathtt{q.old} \wedge \bigwedge \mathtt{q.new} \wedge \mathbf{X}(\bigwedge \mathtt{q.next})) \Leftrightarrow (\bigwedge \mathtt{q}'\mathtt{.old} \wedge \bigwedge \mathtt{q}'\mathtt{.new} \wedge \mathbf{X}(\bigwedge \mathtt{q}'\mathtt{.next}))$$

We may split the nodes into two categories: rooted nodes, which are either created at the start or when the construction of a node was finished and the algorithm moves on to handling the next field (line 10), and non-rooted nodes, which are created from other nodes by splitting and transforming them (in the outermost else branch, lines 12-31).

**Lemma 5.1.4.** Let $\mathtt{p}$ be a rooted node, let $\mathtt{q_1}, ..., \mathtt{q_n}$ be its descendants when the algorithm terminates and let $\Xi$ be the set of formulas in $\mathtt{p.new}$ when it is created. Then the following holds:

$$\bigwedge \Xi \Leftrightarrow \bigvee_{1 \leq i \leq n} (\bigwedge \mathtt{q_i.old} \wedge \mathbf{X}(\bigwedge \mathtt{q_i.next}))$$

*Proof.* Using lemmas 5.1.2 and 5.1.3 by induction on the construction; considering the fact that when $\mathtt{p}$ is created, the fields $\mathtt{old}$ and $\mathtt{next}$ are empty, whereas when the construction of a node is finished, the field $\mathtt{new}$ is empty. □

**Lemma 5.1.5.** Let $\mathtt{q_1}, ..., \mathtt{q_n}$ be as before. If $\xi \models \bigvee_{1 \leq i \leq n}(\bigwedge \mathtt{q_i.old} \wedge \mathbf{X} \bigwedge \mathtt{q_i.next})$ then there exists $i \in \{1, ..., n\}$ such that $\xi \models \bigwedge \mathtt{q_i.old} \wedge \mathbf{X} \bigwedge \mathtt{q_i.next}$ such that for each $g \mathbf{U} h \in \mathtt{q_i.old}$ with $\xi \models h$, $h$ is also in $\mathtt{q_i.old}$.

*Proof.* Every node with $g \mathbf{U} h$ in its $\mathtt{old}$ will appear with a sibling such that one has $g$ and the other has $h$ in its $\mathtt{old}$ as long as both $g$ and $h$ are satisfiable, which is provided by the extra condition $\xi \models h$. □

**Lemma 5.1.6.** Let $\mathtt{q}$ be a node such that $\xi \models \bigwedge \mathtt{q.old} \wedge \mathbf{X}(\bigwedge \mathtt{q.next})$ and let $\Gamma = \{h : g \mathbf{U} h \in \mathtt{q.old} \wedge h \notin \mathtt{q.old} \wedge \xi^1 \models h\}$. Then it has a successor node $\mathtt{q'}$ such that $\xi^1 \models \bigwedge \mathtt{q'.old} \wedge \mathbf{X} \bigwedge \mathtt{q'.next}$ and $\Gamma \subseteq \mathtt{q'.old}$.

*Proof.* From the definition of LTL we have that if $\xi \models \bigwedge \mathtt{q.old} \wedge \mathbf{X}(\bigwedge \mathtt{q.next})$ then $\xi^1 \models \bigwedge \mathtt{q.next}$. When the construction of the node $\mathtt{q}$ is finished, a successor $\mathtt{p}$ is created such that $\Xi = \mathtt{p.new} = \mathtt{q.next}$. Then lemmas 5.1.4 and 5.1.5 guarantee the existence of such a $\mathtt{q'}$ successor node resulting from the processing of $\mathtt{p}$. □

**Lemma 5.1.7.** For every initial state $s \in I$ of the automaton $\mathcal{A}$ generated from the formula $\varphi$, we have that $\varphi \in \Delta(s)$.

**Lemma 5.1.8.** Let $\mathcal{A}$ be an automaton generated from the formula $\varphi$. Let $\mathtt{q_s}$ be the graph node corresponding to the automaton node $s$. Then the following holds:

$$\varphi \Leftrightarrow \bigvee_{s \in I} (\bigwedge \mathtt{q_s.old} \wedge \mathbf{X}(\bigwedge \mathtt{q_s.next}))$$

*Proof.* Using lemma 5.1.2, since for the initial node $\mathtt{p}$ we have $\mathtt{p.new} = \{\varphi\} = \Xi$ when it is created. □

**Lemma 5.1.9.** Let $\sigma = s_0 s_1 s_2 ... \in S^\omega$ be an execution of $\mathcal{A}$ which accepts the word $\xi$. Then $\xi \models \bigwedge \Delta(s_0)$.

*Proof.* Using structural induction on the formula $\bigwedge \Delta(s_0)$. The base case is atomic formulas $p$, $\neg p$ with $p \in \mathcal{P}$, which follow trivially from the construction of the automaton from the graph. Here we will only present the induction step for the temporal operator $\mathbf{U}$. According to lemma 5.1.1 we have two cases for $g \, \mathbf{U} \, h$, but due to the acceptance conditions imposed upon the automaton only case (ii) can happen. Then by the induction hypothesis we have that $\xi^j \models h$ for some $j \in \mathbb{N}$ and $\xi^i \models g$ for all $0 \le i < j$. Thus by the definition of the semantics of LTL, $\xi \models g \, \mathbf{U} \, h$. □

**Lemma 5.1.10.** Let $\sigma = s_0 s_1 s_2 ... \in S^\omega$ be an execution of the automaton $\mathcal{A}$, constructed for $\varphi$ that accepts the word $\xi$. Then $\xi \models \varphi$.

*Proof.* By lemma 5.1.9 we have that $\xi \models \bigwedge \Delta(s_0)$. Since $s_0 \in I$, by lemma 5.1.7 we have that $\varphi \in \Delta(s_0)$. Thus $\xi \models \varphi$. □

**Lemma 5.1.11.** Let $\xi \models \varphi$. Then there exists an execution $\sigma$ of the automaton $\mathcal{A}$, constructed for $\varphi$ which accepts $\xi$.

*Proof.* By lemma 5.1.8, there exists an $s_0 \in I$ such that $\xi \models \bigwedge \mathtt{q}_{s_0}\mathtt{.old} \wedge \mathbf{X} \bigwedge \mathtt{q}_{s_0}\mathtt{.next}$. Now we can construct the execution $\sigma$ by repeated application of lemma 5.1.6. That is, if $\xi^i \models \bigwedge \mathtt{q}_{s_i}\mathtt{.old} \wedge \mathbf{X} \bigwedge \mathtt{q}_{s_i}\mathtt{.next}$ then choose $s_{i+1}$ such that $\xi^{i+1} \models \bigwedge \mathtt{q}_{s_{i+1}}\mathtt{.old} \wedge \mathbf{X} \bigwedge \mathtt{q}_{s_{i+1}}\mathtt{.next}$ (this part provides us with an execution).

Furthermore, lemma 5.1.6 also guarantees that we can choose $s_{i+1}$ such that for any $g \, \mathbf{U} \, h \in \Delta(s_i)$ with $\xi^{i+1} \models h$ we have $h \in \Delta(s_{i+1})$. By lemma 5.1.1 we have that $g \, \mathbf{U} \, h \in \Delta(s_{i+1})$, unless $h \in \Delta(s_{i+1})$. Since $\xi^i \models g \, \mathbf{U} \, h$, there must be some minimal $j \ge i$ such that $\xi^j \models h$. Thus we can choose the execution $\sigma$ such that $h \in \Delta(s_j)$ (this part provides us with an *accepting* execution). □

## 5.2 NON-EMPTYNESS PROBLEM FOR BÜCHI AUTOMATA

It remains to show how to solve the non-emptyness problem for Büchi automata. Since the automaton only has a finite set of states, if the language of the automaton contains a word, then it contains a word which is eventually repeating (i.e. it consists of a finite prefix and of a recurring cycle). Thus instead of all the infinite executions, it enough to consider only the cycles:

**Proposition 5.2.1.** 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 $F \in \mathcal{F}$.

This can again be reformulated using the strongly connected components of the automaton

**Definition 5.2.1.** 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 5.2.2.** 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 $F \in \mathcal{F}$.

For both of these equivalent definitions there exist a family of algorithms that checks emptiness based on them. We will present one algorithm for each definition.

### 5.2.1 *Emptiness checking based on cycles*

The algorithm used as an example for the cycle based approach, Algorithm 5.2, is taken from [Cou+92].

**Algorithm 5.2** Loop-based non-emptyness check for Büchi automata

```
procedure IS_LANGUAGE_EMPTY(initialStates, acceptingStates)
    $S_1 \leftarrow$ Stack(initialStates)
    $S_2 \leftarrow$ Stack()
    $M_1 \leftarrow \emptyset$
    $M_2 \leftarrow \emptyset$
    while $S_1 \neq \emptyset$ do
        $x \leftarrow S_1$.top()
        if there is a state y $\in$ x.next with y $\notin M_1$ then
            $M_1 \leftarrow M_1 \cup \{y\}$
            $S_1$.push(y)
        else
            $S_1$.pop()
            if x $\in$ acceptingStates then $S_2$.push(x)
                while $S_2 \neq \emptyset$ do
                    $v \leftarrow S_2$.top()
                    if x $\in$ v.next then
                        return false
                    end if
                    if there is a state w $\in$ v.next with w $\notin M_2$ then
                        $M_2 \leftarrow M_2 \cup \{w\}$
                        $S_2$.push(w)
                    else
                        $S_2$.pop()
                    end if
                end while
            end if
        end if
    end while
    return true
end procedure
```

Cycle based emptiness checks usually require the automaton to be transformed into a simple Büchi automaton using Proposition 4.2.2. This can lead to a large increase in the number of product automaton states, but according to experiments done by [GS09] and [CP03] only a small portion of real-world automata have this property. These algorithms generally use less auxiliary memory, but given that the memory use is dominated by the representation of the states this isn't a significant advantage.

The data structure used to represent nodes of the simple Büchi automaton consists only of the set of its successors. The only additional data we need is the set of initial and accepting states, which are passed as sets to the IS_LANGUAGE_EMPTY function. The algorithm consists of two interleaved depth-first search procedures and uses two sets to record the visited states and two stacks to keep track of the current position in the search.

The first depth-first search, starting at line 6 looks for states which are reachable from the set of initial states. If it reaches an accepting state `x` with no unvisited successors (lines 11 – 13) it starts another depth first search, this time looking for loops. If during the search it finds a state `v` such that `x` can be reached from `v` (lines 16 – 17), then it has found a loop and the procedure returns `false`, since there is an accepting execution of the automaton. If there is no such reachable state also reachable from itself, then the language of the automaton is empty, and the algorithm returns `true` (line 29).

The violating system run can be easily extracted from the stacks $S_1$ and $S_2$.

### 5.2.2 *Emptiness checking based on strongly connected components*

The algorithm used as an example for the SCC based approach, Algorithm 5.3, is taken from [GS09], according to whom it was the fastest algorithm among the ones they tested. It is an improvement upon Coevreur's algorithm [Cou99], which in turn is a variant of Tarjan's algorithm [Tar72] specialized for automata.

SCC based emptiness checks don't require the generalized Büchi automaton to be converted into a simple one, thus it has to process fewer states when there are multiple accepting sets. These generally require more auxiliary memory than algorithms of the first kind.

The data structure for generalized Büchi automata remains the same as it was in the previous algorithm, but in addition the states now contain some extra information in the `dfsnum` and `current` fields, needed for the execution of the algorithm. To represent the acceptance conditions, we assume that $K = \{1, ..., n\}$ represents each accepting set by an integer and the function $A \colon S \to 2^K$ maps each state to the accepting sets in which the state is contained.

The algorithm operates on two global stacks *roots* and *active* which are both initialized to empty. When invoked as IS_LANGUAGE_EMPTY$(s, 0)$ on every initial state $s$, the algorithm finds successively larger SCCs by initializing trivial single element SCCs (when pushing $s$ onto the *roots* stack) and then by combining existing ones when a connection between them is found (in the first **repeat** loop). During this combination, it is checked whether the current SCC contains a state from each accepting set; if so, a cycle is reported. Otherwise, once the current SCC cannot be augmented further (it is maximal), it is removed from the set of active nodes and from the *roots* stack (second **repeat** loop).

**Algorithm 5.3** SCC-based non-emptyness check for Büchi automata

```
1: procedure IS_LANGUAGE_EMPTY($s, d$)
2:     $s$.dfsnum ← $d$
3:     $s$.current ← true
4:     $roots$.push($s$, $A(s)$)
5:     $active$.push($s$)
6:     for all $t \in s$.next do
7:         if $t$.dfsnum = 0 and IS_LANGUAGE_EMPTY($t, d + 1$) is false then
8:             return false
9:         else if $t$.current then
10:            $B \leftarrow \emptyset$
11:            repeat
12:                $(u, C) \leftarrow roots$.pop()
13:                $B \leftarrow B \cup C$
14:                if $B = K$ then return false
15:                end if
16:            until $u$.dfsnum $\leq t$.dfsnum
17:        end if
18:    end for
19:    if $roots$.top() = $(s, \_)$ then
20:        $roots$.pop()
21:        repeat
22:            $u \leftarrow active$.pop()
23:            $u$.current ← false
24:        until $u = s$
25:    end if
26:    return true
27: end procedure
```

As this is a recursive algorithm, the stack (i.e. the prefix of the counterexample) is not immediately available. And since it finds a strongly connected component instead of a cycle, the looping part of the counterexample has to be extracted separately as well. Despite these drawbacks, this algorithm has one significant advantage, namely it makes model checking with many fairness constraints possible, which will be described in detail in the next section.

## 5.3 model checking under fairness constraints

It was mentioned already in the introduction that certain desirable properties of concurrent system do not hold in all possible executions, only under certain assumptions, for example that the operating system will eventually schedule each process in the system. These assumptions are called *fairness constraints* and in this thesis we consider two of them:

**Definition 5.3.1.** A system run being *weakly fair* with regard to a given action means that if the action is eventually permanently enabled, then it is executed infinitely often.

**Definition 5.3.2.** A system run being *strongly fair* with regard to a given action means that if the action is infinitely often enabled, then it is executed infinitely often.

An efficient solution for model checking under fairness constraints is given in Algorithm 5.4. It was first described in [LP85], but was discovered independently by the author. In the following we sketch how one might come up with this idea, and by doing so we explain how the algorithm works.

**Algorithm 5.4** Fairness checking for strongly connected components

```
 1: ▷ A: strongly connected subgraph of the product automaton
 2: ▷ weakFairness: set of actions with weak fairness constraints
 3: ▷ strongFairness: set of actions with strong fairness constraints
 4: procedure IS_SCC_FAIR(A, weakFairness, strongFairness)
 5:     for all action a ∈ weakFairness do
 6:         if for all states s ∈ A a is enabled in s and a is not executed in s then
 7:             return false
 8:         end if
 9:     end for
10:     A′ ← A
11:     for all action a ∈ strongFairness do
12:         if for all states s ∈ A a is not executed in s then
13:             A′ ← {s ∈ A′ : a is not enabled in s}
14:         end if
15:     end for
16:     if A′ = A then return true
17:     end if
18:     for all $A_i$ ∈ DECOMPOSE_INTO_SCCS(A′) do
19:         if IS_SCC_FAIR($A_i$, weakFairness, strongFairness) then
20:             return true
21:         end if
22:     end for
23:     return false
24: end procedure
```

The definitions of the fairness constraints can be reformulated using LTL:

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

Thus if we want to say that the fairness of some actions implies that an LTL formula $g$ holds, we could convert the constraints to LTL formulas $f_1, ..., f_n$ and then apply model checking to $(f_1 \wedge ... f_n) \implies g$.

This approach is certainly correct, but it is in almost all cases unfeasible. The translation algorithm we presented (and in fact all other similar algorithms) produce an automaton whose size is, in the worst case, exponential in the length of the formula. While a formula of the form $\mathbf{GF}p$ is translated into an automaton with 3 states, with just one weak fairness constraint added, the automaton consists of 20 states, with 5 weak fairness constraints 5120 states and with 5 strong fairness constraints 11423 states. Note that while 5 fairness constraints might seem a lot, in a system with multiple components you often have to apply fairness to all of them (e.g. one has to specify separately that `process1` and `process2` both eventually release the lock they hold).

Fortunately there is a much better solution to this problem. Let $g$ be the formula we want to check under the weak fairness constraints $wf(a_1), ..., wf(a_n)$ and strong fairness constraints $sf(b_1), ..., sf(b_n)$. Thus the full formula-to-be-checked is $(wf(a_1) \land ... \land wf(a_n) \land sf(b_1) \land ... \land sf(b_n)) \implies g$ and we are looking for a counter-example, meaning a formula which satisfies $wf(a_1) \land ... \land wf(a_n) \land sf(b_1) \land ... \land sf(b_n) \land \neg g$.

We know already that for falsifying an LTL formula it is sufficient to consider only the infinite traces consisting of a finite prefix and an infinitely repeating (finite) loop. As the fairness constraints are only concerned with the eventual behaviour of the system, the prefix can be ignored, and we can concentrate on the cycles in the automaton. So we could find all counterexamples for $g$ with distinct cycles, then check the cycles one-by-one:

- for a constraint $wf(a_i)$ the cycle is a valid counterexample, if either there is a state in which the action $a_i$ is executed, or if there is one in which it is not enabled,
- for a constraint $sf(b_j)$ the cycle is a valid counterexample, if either there is a state in which the action $b_j$ is executed, or if it is not enabled in any of the states.

Unfortunately the cycles in the automaton are again too numerous, but this approach is on the right track. Instead of searching for all the cycles, we start by searching for the maximal SCCs which could contain a counterexample for $g$ by using, for example, the algorithm from before. Every cycle is part of some SCC, so we do not lose anything by doing this. Then we examine if this SCC can produce a fair cycle.

First, for weak fairness constraints $wf(a_i)$: if there is a state $s$ in the SCC $A$ in which the action $a_i$ is executed, or if there is one in which it is not enabled we can easily construct a fair cycle from any cycle in $A$ that satisfies $g$, because from any state in $A$ we can reach $s$ and vice versa. If there is no such state, the SCC can be rejected, as it cannot contain a fair cycle.

Second, for strong fairness constraints $sf(b_j)$: if $b_j$ is nowhere enabled then all cycles are fair with respect to $b_j$. If $b_j$ is executed at least once, then by the previous logic we can extend any cycle to be fair. But what if $b_j$ is never executed, but there are some states in which is is enabled? No fair cycle may go through these states, so we can simply remove them from the SCC. What we get is a (potentially disconnected) sub-graph $G$ in which we want to find cycles. To do so, we can apply our algorithm recursively. First find the maximal SCCs of $G$ which contain a counterexample for $g$, then check them using this same method. The procedure eventually terminates, since in each execution we make sure that at least one more action with a strong fairness attached to it is never enabled, and there is only a finite number of them.

# 6 ALTERNATIVE MODEL CHECKING APPROACHES

There exist various approaches to LTL model checking, a detailed overview of the state of the art is given in [Cla+18]. In this chapter we will describe three further algorithms which were considered for implementation within this thesis.

## 6.1 TABLEAU-BASED EXPLICIT STATE MODEL CHECKING

The tableau-based model checking algorithm is described in chapter 5 of [MP95]; it is implemented in the TLC model checker which is part of the TLA+ software package [Lam02].

The method shares many similarities with the automaton-based approach, since both are explicit state algorithms, the difference being the way they transform the LTL formula to a structure they can operate on. As opposed to the property automaton, here we construct a *tableau*, i.e. a directed graph in which the edges represent the necessary conditions for a certain set of formulas to hold. The idea behind tableau construction is the decomposition of temporal formulas that was already discussed during the automaton construction..

First we define the *closure* of a formula $\varphi$, denoted as $\Phi_\varphi$, as the smallest set of formulas satisfying the following properties:

- $\varphi \in \Phi_\varphi$.
- For every $p \in \Phi_\varphi$ and every subformula $q$ of $p$, $q \in \Phi_\varphi$
- For every $p \in \Phi_\varphi$, $\neg p \in \Phi_\varphi$ (to keep the closure finite, $\neg\neg p$ is identified with $p$).
- For every $\psi \in \{\mathbf{G}p, \mathbf{F}p, p\,\mathbf{U}\,q\}$, if $\psi \in \Phi_\varphi$ then $\mathbf{X}\psi \in \Phi_\varphi$.

A subset $A$ of $\Phi_\varphi$ is called an *atom* of $\varphi$ if it satisfies the following properties:

- For every $p \in \Phi_\varphi$, $p \in A$ iff $\neg p \notin A$.
- A formula of the form $p \wedge q$ is in $A$ iff both $p \in A$ and $q \in A$.
- A formula of the form $\mathbf{G}p$ is in $A$ iff both $p \in A$ and $\mathbf{XG}p \in A$.
- A formula of the form $p \vee q$ is in $A$ iff either $p \in A$ or $q \in A$.
- A formula of the form $\mathbf{F}p$ is in $A$ iff either $p \in A$ or $\mathbf{XF}p \in A$.
- A formula of the form $p\,\mathbf{U}\,q$ is in $A$ iff either $q \in A$ or both $p \in A$ and $\mathbf{X}(p\,\mathbf{U}\,q) \in A$.

A set of formulas $S \subseteq \Phi_\varphi$ is called *mutually satisfiable* if there exists a trace $\pi$ such that $\pi \models p$ for every $p \in S$. Atoms are maximal for the mutually satisfiable subsets, meaning that every mutually satisfiable set is a subset of some atom, though they themselves need not be mutually satisfiable.

A formula is called *basic* if it is an atomic proposition $p$ or if it has the form $\mathbf{X}\psi$ for some arbitrary formula $\psi$. The presence of basic formulas uniquely determines the atom, which means we have a simple algorithm for generating all atoms: let $p_1, ..., p_b \in \Phi_\varphi$ be all the basic formulas in the closure of $\varphi$, construct all $2^b$ combinations in the form $q_1, ..., q_b$ where $q_i$ is either $p_i$ or $\neg p_i$ and complete each combination into a full atom using the defining properties of an atom listed above.

Now given the formula $\varphi$ we may finally construct the tableau $T_\varphi$, which is a directed graph, whose nodes are the atoms of $\varphi$, and atom $A$ is connected to atom $B$ if for every $\mathbf{X}p \in \Phi_\varphi$ we have $\mathbf{X}p \in A \Leftrightarrow p \in B$. This means a connection is a necessary condition for the situation that all formulas contained in $A$ hold at some position $j$ while all the formulas in $B$ hold in the immediately next position $j + 1$.

For a formula $\varphi$ and a trace $\pi$, the infinite atom path $A_0, A_1, ...$ in $T_\varphi$ is said to be *induced* by $\pi$ if, for every closure formula $p \in \Phi_\varphi$ and $j \in \mathbb{N}$ we have $\pi^j \models p$ iff $p \in A_j$. Every trace induces such an infinite path, furthermore if $\pi \models \varphi$ then $\varphi \in A_0$. The converse of this is not true: not every infinite atom path is induced by some trace.

A formula $\psi \in \Phi_\varphi$ is said to *promise* the formula $r$ if $\psi$ has one of the forms $\mathbf{F}r$, $p\,\mathbf{U}\,r$, $\neg\mathbf{G}\neg r$ or if $r$ is the negation $\neg q$ and $\psi$ has the form $\neg\mathbf{G}q$. Each of these formulas implies $\mathbf{F}r$, which may be interpreted as a promise that $r$ will eventually hold. Such a formula $\psi$ is called a *promising* formula.

An atom $A$ *fulfills* a formula $\psi$ that promises $r$ if $\neg\psi \in A$ or $r \in A$. A path $A_0, A_1, ...$ in the tableau $T_\varphi$ is called *fulfilling* if for every promising formula $\psi \in \Phi_\varphi$ it contains infinitely many atoms that fulfill $\psi$. Every trace induces a fulfilling path and the converse is also true: every fulfilling path induces a trace. This means we can check if the formula $\varphi$ is satisfiable by checking if the tableau $T_\varphi$ contains a fulfilling path $A_0, A_1, ...$ such that $\varphi \in A_0$.

Unfortunately this does not immediately yield a decision procedure, since the tableau may contain infinitely many paths. Similarly to the automaton-based emtyness check, here we again rely on maximal strongly connected components. An *MSCC* is fulfilling if it is not just a single atom not connected to itself and every promising formula $\psi \in \Phi_\varphi$ is fulfilled by some atom $A \in S$. It is *$\varphi$-reachable* if there exists a finite path $B_0, B_1, ..., B_k$ such that $\varphi \in B_0$ and and $B_k \in S$. Claim 5.7 in [MP95] shows that the existence of a $\varphi$-reachable *MSCS* is equivalent to the existence of a fulfilling path $A_0, A_1, ...$ such that $\varphi \in A_0$, which means that it is also equivalent to the satisfiability of the formula.

Using these facts we get a simple decision algorithm: first construct the tableau $\Phi_\varphi$ then remove all atoms not reachable from $\varphi$. Now decompose $\Phi_\varphi$ into *MSCS*s (using the algorithm DECOMPOSE from section 3.6 of [MP95]) and check if any of them is fulfilling. If yes, then $\varphi$ is satisfiable, otherwise it is not.

To check if a formula is valid for a given Kripke-structure $K$, we first note that $\varphi$ is $K$-valid, if $\neg\varphi$ is not $K$-satisfiable. Then we proceed similarly to automaton-based model checking by constructing the product of the tableau $T_\varphi$ and the Kripke-structure $K$ called the behaviour-graph $B_{(K,\varphi)}$ in such a way that every state $(s, A)$ in $B_{(K,\varphi)}$ is compatible, meaning every state formula in $A$ holds in the state $s$.

Then we can similarly decompose $B_{(K,\varphi)}$ into *MSCCs*; if we find one which is fulfilling, then $\varphi$ is satisfiable by a trace of $K$, otherwise it is not.

The size of the tableau is exponential in the size of the formula, but there are optimizations which enable us to *prune* it, reducing the number of nodes (see algorithm PRUNE in section 5.1 of [MP95]).

## 6.2 BDD-BASED SYMBOLIC MODEL CHECKING

The BDD-based symbolic model checking approach is described in [McM93] and chapter 8 of [Cla+18]. It is used by the SMV[1] model checker for CTL.

A *Boolean algebra* is an algebraic structure $(B; \wedge, \vee, \neg, 0, 1)$ such that $\wedge$ and $\vee$ are associative, commutative, and distributive to each other, and obey the axioms of absorption ($a \vee (a \wedge b) = a$, $a \wedge (a \vee b) = a$), identity ($a \vee 0 = a$, $a \wedge 1 = a$), and complementation ($a \vee \neg a = 1$, $a \wedge \neg a = 0$). Note that $(B; \wedge, \vee)$ is a lattice. In particular the structure $(\mathcal{P}(S); \cap, \cup, \complement, \emptyset, S)$, whose elements are subsets of a given set $S$, is a Boolean algebra.

*Functionals*, as defined in $\lambda$-calculus, are objects denoted by $\lambda y.f$, where $y$ is a variable and $f$ is a formula. When a functional $\tau = \lambda y.f$ is evaluated at $p$, denoted by $\tau(p)$, it yields $f$ with $y$ substituted for $p$. A functional $\tau$ is monotonic if $p \subseteq q$ implies $\tau(p) \subseteq \tau(q)$. Any $p$ such that $\tau(p) = p$ is called the fixpoint of the functional $\tau$. Not all functionals have fixpoints, and if they do it is not necessarily unique. But monotonic functionals have the useful property that they always have a least and greatest fixpoint. For $\lambda y.f$ these are denoted by $\mu y.f$ and $\nu y.f$ respectively. The study of these fixpoint-operators is called *modal $\mu$-calculus* [Sti01].

For example $\lambda y.(x \vee y)$ is monotonic, since $p \subseteq q$ implies $x \vee p \subseteq x \vee q$, and it has the fixpoint $x \vee y$, since $\tau(x \vee y) = x \vee (x \vee y) = x \vee y$.

If we identify an LTL formula $f$ over the Kripke-structure $K = (S, I, T, \mathcal{L})$ as the set of states $\{s \in S : s \models f\}$ then we can give the operators of temporal logic fixpoint-characterizations. For example $\mathbf{G}p$ is the greatest fixpoint of the functional $\lambda y.(p \wedge \mathbf{X}y)$, i.e. $\mathbf{G}p = \nu y.(p \wedge \mathbf{X}y)$.

Since there exists an iterative method of computing fixpoints (described in chapter 2.4 of [McM93]), we can use this to construct a model checking algorithm: given a formula $f$, if the set of initial states is fixed by the fixpoint representation of $f$, then it holds for the system. Notice the similarity between the fixpoint representations and the decomposition formulas used when constructing the tableau. In fact, this algorithm and the tableau-based is essentially the same, and it suffers from the same problem of state-space explosion.

1 http://www.cs.cmu.edu/~modelcheck/smv.html

However whenever the Kripke-structure can be represented with formulas instead of naively enumerating all the states and transitions between them, for example in the case of circuits, then this algorithm can be applied more efficiently. This leads to the approach called *symbolic model checking*.

For a Kripke-structure $K = (S, I, T, \mathcal{L})$ over the set of atomic predicates $\mathcal{A} = \{a_1, ..., a_n\}$ we will consider the Boolean algebra of subsets over $\mathcal{P}(\mathcal{P}(\mathcal{A}))$ and vector functionals in the form $\lambda(v_1, ..., v_n).f$. In this way we can identify any state with a Boolean vector $\{\text{true}, \text{false}\}^n$ and then any subset of $S$ with a vector functional $\lambda(v_1, ..., v_n).f$ where the formula $f$ is closed, meaning that it maps any such vector to either true or false. Similarly the transition relation can be represented by a functional $\lambda((v_1, ..., v_n), (v'_1, ..., v'_n)).f$ with a closed formula $f$ that evaluates to true iff there is a transition from state $a = (a_1, ..., a_n)$ to $a' = (a'_1, ..., a'_n)$.

To characterize the operators of temporal logic using this symbolic representation of the states, we introduce the Boolean quantification operators. The formula $\exists(v_1, ..., v_n).f$ evaluates to true if there is an assignment under which $\lambda(v_1, ..., v_n).f$ is true; similarly $\forall(v_1, ..., v_n).f$ is true if it evaluates to true for all assignments. Formally

$$\exists(v_1, ..., v_n).f = \bigvee_{a \in \{\text{true}, \text{false}\}^n} (\lambda(v_1, ..., v_n).f)(a)$$

$$\forall(v_1, ..., v_n).f = \bigwedge_{a \in \{\text{true}, \text{false}\}^n} (\lambda(v_1, ..., v_n).f)(a)$$

For example

$$\mathbf{G}p = \nu v.(p(v) \wedge \forall v'.(R(v, v') \to p(v')))$$

This means in order to use the iterative algorithm of computing the fixpoints, we need an efficient way of manipulating Boolean formulas, i.e. applying $\wedge$, $\vee$, $\exists$, $\forall$ etc. to functionals. The tool used for this in the SMV model checker are *ordered Boolean decision diagrams*, or *OBDD*s, which are directed acyclic graphs. If two formulas are represented by OBDDs with $n$ and $m$ nodes respectively, then the time complexity of computing the OBDD representation of their conjunction and disjunction is $O(mn)$, and all other operations are similarly efficient. OBBDs are described in chapter 7 of [Cla+18].

OBBDs can be obtained from the ordered decision tree of the formula, which is a complete binary tree with each level representing a variable and the leaves representing evaluations. From any node going towards the so called 0-child means the variable is assigned false, whereas going to the 1-child means it is assigned true. This way when we reach the leaves every variable is assigned a value and the formula can be evaluated. In the following diagrams the 0-child is represented by dashed, while the 1-child is represented by solid lines.

The OBDD representation can be obtained in linear time from the ordered decision tree by applying the following two rules from the bottom up:

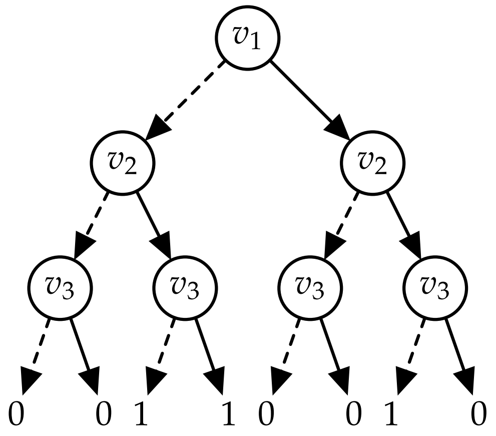


Figure 6.1: Ordered decision tree of the formula $f = (v_1 \wedge v_2 \wedge \neg v_3) \vee (\neg v_1 \wedge v_3)$

1. if two nodes have the same 0 and 1 children (isomorphic subtrees), then remove one of the nodes, and direct all incoming edges to the other,
2. if the 0 and 1 children of a node are the same (irrelevant node), then remove the node and direct all incoming edges to the child.

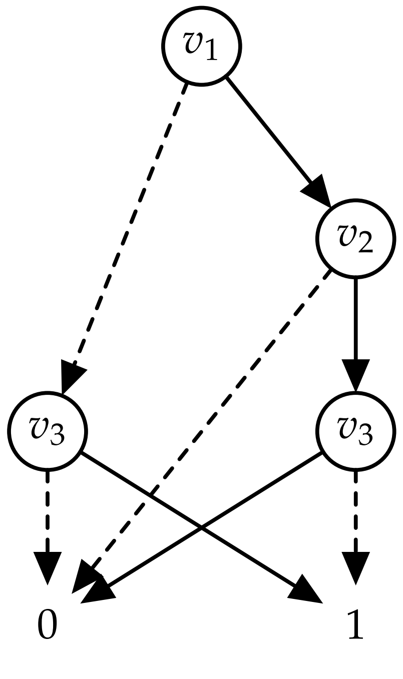


Figure 6.2: Ordered binary decision diagram of the formula $f = (v_1 \wedge v_2 \wedge \neg v_3) \vee (\neg v_1 \wedge v_3)$

## 6.3 BOUNDED MODEL CHECKING

Bounded model checking as described in [Bie+99] is used in the NuSMV[2] model checker for both CTL and LTL.

The *Boolean satisfiability problem*, or *SAT*, asks to determine whether for a given propositional formula there exists an assignment of the propositional variables under which the formula evaluates to true. SAT plays a critical role in theoretical computer science, it was the first problem to be proven NP-complete [Coo71], meaning that all problems in the complexity class NP may be reduced to SAT in polynomial time. Karp's famous paper [Kar72] lists 20 such problems, and similarly, bounded model checking is the reduction of the model checking problem to SAT.

Although there exists no known polynomial time algorithm for SAT, due to its importance much progress was made in creating tools, so called SAT solvers, which are capable of handling large formulas arising naturally, for example in circuit verification. A detailed description of the state of the art in Boolean satisfiability can be found in [Bie+21].

Bounded model checking is also a symbolic approach, meaning that it requires the states and transitions to be encoded using Boolean formulas like in 6.2. It proves the validity of LTL formulas by attempting to find counterexamples of the negation of the given formula ($K \models f \Leftrightarrow K \not\models \neg f$) by progressively considering longer and longer finite prefixes of the traces.

Bounded model checking gives an interpretation for $\pi \models_k f$, meaning $f$ holds for the length $k$ prefix of $\pi$. Crucially, even finite prefixes may represent a witness for formulas of the form $\mathbf{G}p$, provided they contain a back-loop from the last state to a previous one. One can then show that $\pi \models_k f$ implies $\pi \models f$.

The final step in showing that bounded model checking is not only correct, but also complete, is to prove that for any $f$ and any Kripke-structure $K = (S, I, T, \mathcal{L})$ if there exists a trace $\pi$ such that $\pi \models f$, then we must have a $k \in \mathbb{N}$ and a trace $\pi'$ such that $\pi' \models_k f$. This means it is always enough to examine traces of finite length. [Bie+99] gives the upper bound of $k \leq |S| \cdot 2^{|f|}$, where $|f|$ is the length of the formula $f$.

Bounded model checking finds counter-examples of minimal length very quickly compared to other approaches thanks to the optimized SAT solvers and due to considering traces of increasing length during the search. An advantage over alternative symbolic model checkers is that bounded model checking uses less memory than BDD based approaches and it needs no manual ordering of the variables or time consuming dynamic reordering.

## 6.4 COMPARISON OF MODEL CHECKING APPROACHES

All of the methods discussed in this and the previous chapter have their strengths and weaknesses; and they all used in practice for the purposes of verifying real software.

2 https://nusmv.fbk.eu/

While bounded model checking is very efficient for certain applications due to its reliance on optimized SAT solvers, and in theory it is capable of verification if you consider a sufficiently large bound, in practice this is often unfeasible, thus the approach is mainly used for falsification, i.e. finding errors ([Bie+21] Chapter 18). This means it is not suitable for our implementation.

Symbolic model checking was originally proposed in the context of hardware verification and it has proven to be successful in this respect, verifying systems with orders of magnitude more states then explicit state model checking approaches [Bur+90]. More recently it has been used for the purposes of software verification, such as Boolean programs and push-down systems. Unfortunately, the encoding of the states and transitions of complex systems with many different data types (such as the ones in RISCAL) using only binary variables, while theoretically possible, is very inefficient.

Using OBDDs with the right variable ordering in some sense understands the the underlying circuit of the boolean function, representing it efficiently. But if we were to encode numeric data, or a more complex data type such as a map using independent binary variables, we would lose information about how these new variables relate to each other. This is also a reason why bounded model checking is not applicable in our case.

Explicit state model checking approaches do not have the encoding problem, since they interpret the system as a directed graph of states, irrespective of the data contained within. This also means that in the worst case scenario they have to traverse all the states. For example if we only have two components, each with a 32 bit counter which they increment in a loop, that is already $2^{32+32} \approx 10^{19}$ system states, so it is not possible to keep all the states in memory, let alone visit them all. This means that when model checking nondeterministic systems, the first step is to model only the relevant interactions between components, while abstracting away the unnecessary information. As an example model checking is often used for the verification of communication protocols, and in this case the actual value of the transmitted data is not relevant [Wol86].

With a correctly modelled system explicit state model checking *can* be used efficiently for verification, as proven by SPIN and TLA+ both being used extensively in the software development industry. This is especially true when applying certain optimizations, such as partial order reduction, which reduces the actual number of states which need to be visited. Moreover, explicit state model checking can also be used to find counter-examples of minimal length, or to heuristically verify large systems for which exhaustive verification is unfeasible.

The choice therefore is between the automaton-based and the tableau-based algorithms. These two are essentially the same, they even use the same expansion formulas for temporal operators to construct the structure they operate on (e.g. $p\,\mathbf{U}\,q \equiv q \vee [p \wedge \mathbf{X}(p\,\mathbf{U}\,q)]$).

Within the scope of this thesis, the decision was made to implement the automaton-based approach for the following reasons:

- it has a much simpler and faster algorithm for constructing the Büchi-automaton, compared to the tableau construction,

- in most common applications it produces an automaton with fewer states than the equivalent tableau, without resorting to a secondary pruning step,
- finally, there is more literature covering the details of the automaton-based approach.

In the next chapters we describe the LTL model checking extension based on the automaton-based explicit-state approach: its usage, implementation, and performance.

# 7 THE RISCAL LTL EXTENSION

In this chapter we present the LTL language as implemented by RISCAL, the user interface of the software and how the model checker can be used to find and correct bugs in a system. The actual implementation of the model checker will be discussed in Chapter 8.

## 7.1 THE LTL LANGUAGE

For both shared and distributed systems we can describe their properties using LTL clauses of the following form:

```
"ltl" <LTL>
```

Here the LTL formulas accepted by the system are given by the following grammar in Backus-Naur form:

```
<LTL> ::=
    <LTL-atom>                                      atomic formulas
  | "~" <LTL>                                       negation
  | <LTL> "/\" <LTL>                                conjunction
  | <LTL> "\/" <LTL>                                disjunction
  | <LTL> "=>" <LTL>                                implication
  | <LTL> "<=>" <LTL>                               equivalence
  | "if" <LTL> "then" <LTL> "else" <LTL>            conditional operator
  | "Next" <LTL>                                    next-time
  | "Globally" <LTL>                                always
  | "Finally" <LTL>                                 eventually
  | <LTL> "Until" <LTL>                             strong until
  | <LTL> "WeakUntil" <LTL>                         weak until
  | "let" <binders> "in" <LTL>                      local variable bindings (sequential)
  | "letpar" <binders> "in" <LTL>                   local variable bindings (parallel)
  | "forall" <quantified-variables> "." <LTL>       universal quantification
  | "exists" <quantified-variables> "." <LTL>       existential quantification
```

Some operators also have equivalent symbolic representations accepted by the system: $\neg$ (`~`), $\wedge$ (`/\`), $\vee$ (`\/`), $\Rightarrow$ (`=>`), $\Leftrightarrow$ (`<=>`), $\circ$ and $\bigcirc$ (`Next`), `[]` and $\Box$ (`Globally`), `<>` and $\Diamond$ (`Finally`), $\forall$ (`forall`), $\exists$ (`exists`).

The binders are comma-separated lists of the form `<identifier> "=" <expression>` where `<expression>` is an arbitrary expression in the RISCAL syntax. They bind the variables named by the identifiers to the values of the expression on the right hand side of the equals sign. The `let` operator binds each value in turn (i.e. each subsequent binding can already refer to the previously introduced ones), while `letpar` binds them simultaneously (no binding can refer to previously introduced ones).

The quantified variables are comma-separated lists of the form `<identifier> ":" <type>` or `<identifier> ":" <type> "with" <expression>`. They are equivalent to a conjunction or disjunction of the LTL formula with the identifiers bound to all possible values which make up the type (or in the second case only the ones for which the expression following `with` evaluates to true). Since RISCAL operates over a finite domain, they are guaranteed to expand to a finite conjunction or disjunction.

The atomic formulas are defined as follows:

```
<LTL-atom> ::=
    "[[." <boolean-expression> ".]]"
```
state predicate (can alternatively be enclosed within $[\![$ and $]\!]$)
```
  | "Enabled" <action>
```
some instance of the action is enabled
```
  | "Enabled" <action> "(" <xs> ")"
```
the instance of the action for the given arguments is enabled
```
  | "Executed" <action>
```
some instance of the action is executed
```
  | "Executed" <action> "(" <xs> ")"
```
the instance of the action for the given arguments is executed
```
  | "WeakFairness" <action>
```
equivalent to `(<>[] Enabled action)` $\Rightarrow$ `([]<> Executed action)`
```
  | "WeakFairness" <action> "(" <xs> ")"
```
equivalent to `(<>[] Enabled action(xs))` $\Rightarrow$ `([]<> Executed action(xs))`
```
  | "StrongFairness" <action>
```
equivalent to `([]<> Enabled action)` $\Rightarrow$ `([]<> Executed action)`
```
  | "StrongFairness" <action> "(" <xs> ")"
```
equivalent to `([]<> Enabled action(xs))` $\Rightarrow$ `([]<> Executed action(xs))`

The fairness predicates above should be avoided whenever possible, as they expand into a long formula, thus producing a very large automaton. To take advantage of the fast fairness checking algorithm the actions themselves need to be annotated with fairness specifications.

These are:

- `weak` or `weak_some` (equivalent to `(<>[] Enabled action)` ⇒ `([]<> Executed action)`),
- `strong` or `strong_some` (equivalent to `([]<> Enabled action)` ⇒ `([]<> Executed action)`),
- `weak_all` (equivalent to ∀`exps. (<>[] Enabled action(exps))` ⇒ `([]<> Executed action(exps))`),
- `strong_all` (equivalent to ∀`exps. ([]<> Enabled action(exps))` ⇒ `([]<> Executed action(exps))`)

The next section gives examples of the application of fairness specifications.

## 7.2 THE LTL MODEL CHECKING INTERFACE

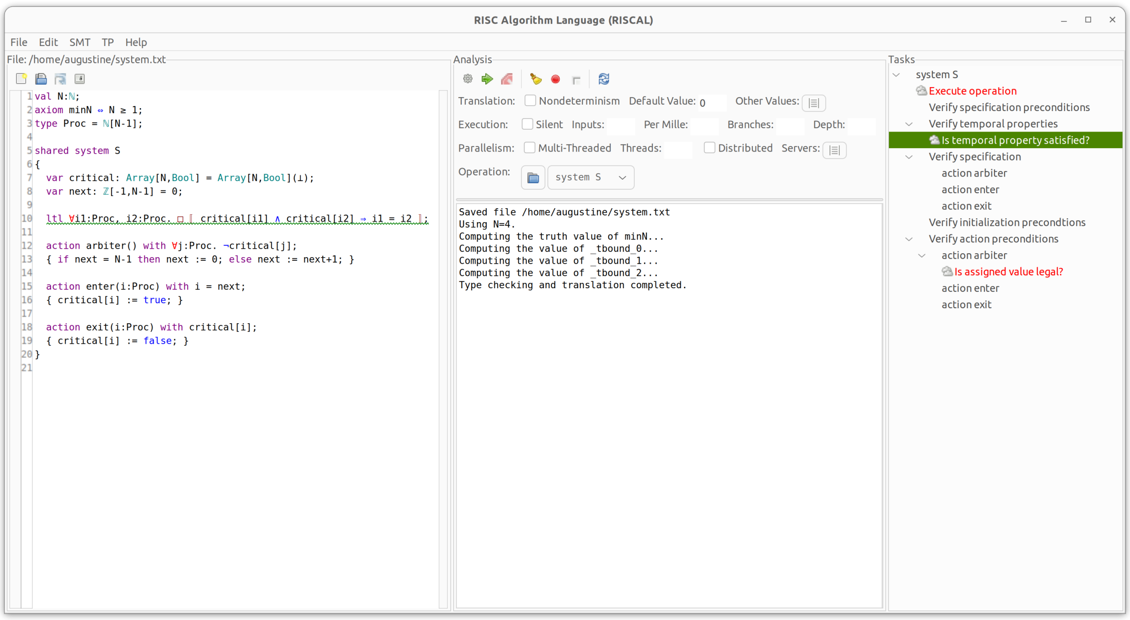
In this section we show how to use the RISCAL model checker to verify and to debug a non-deterministic system. The example, taken from [Sch21], is a simple mutual exclusion algorithm.

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

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

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

  action enter(i: Proc) with i = next ∧ ¬critical[i];
  { critical[i] := true; }

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

Program 10: Simple mutual exclusion algorithm in RISCAL

The model depicted as Program 10 describes a concurrent system with one arbiter and $N$ worker processes operating on a shared system state consisting of the variables *critical* and *next*. *critical* is a Boolean array of $N$ elements, the $i$-th value being true means that the $i$-th worker process is currently in the critical section. *next* denotes the index of the worker process which may next access the critical section.

The arbiter process repeatedly cycles through the values $0..N-1$ and assigns it to the variable *next* (*arbiter* action). However it must first wait until no worker process is in the critical section (guard condition introduced using *with*). The worker processes can enter


RISC Algorithm Language (RISCAL)
File Edit SMT TP Help
Analysis
Tasks
system S
Execute operation
Verify specification preconditions
Verify temporal properties
Is temporal property satisfied?
Verify specification
action arbiter
action enter
action exit
Verify initialization precondtions
Verify action preconditions
action arbiter
Is assigned value legal?
action enter
action exit

File: /home/augustine/system.txt

```
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-1] = 0;

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

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

  action enter(i:Proc) with i = next;
  { critical[i] := true; }

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

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

```
Saved file /home/augustine/system.txt
Using N=4.
Computing the truth value of minN...
Computing the value of _tbound_0...
Computing the value of _tbound_1...
Computing the value of _tbound_2...
Type checking and translation completed.
```

Figure 7.1: Example system with mutual exclusion formula

and exit the critical section using the respective actions, only if they are assigned as next, or they are currently in the critical section.

The most important property a mutual exclusion protocol has to satisfy is that no two processes are in the critical section at the same time. We can describe this requirement using the RISCAL LTL language as follows:

```
ltl ∀i1: Proc, i2: Proc. □ ⟦ critical[i1] ∧ critical[i2] ⇒ i1 = i2 ⟧;
```

The LTL formulas have to be added between the variable declarations and the actions.

In Fig. 7.1 we see the graphical user interface of the RISCAL system. On the left we have a code editor with the mutual exclusion example loaded into it. In the center we have the Analysis menu with a variety of settings. If we want to check the system on the left, we should first assign some value to the constants appearing in the definition. Under *Other values* we should set $N$ to some small integer, in our example 4. Below this is the output view which prints the result of the model checking operations.

To check the formula we have to first save, select *System S* in the menu "Operation" and then click the "Folder" button to open the Tasks menu (see Fig. 7.2) on the right. Here we can select the task "Is temporal property satisfied?" which will highlight the respective LTL formula on the left hand side. Double clicking the task checks the formula and yields the following output:

```
Executing _ltl0().
Run of deterministic function _ltl0():
Checking LTL formula ∀i1:Proc, i2:Proc. ([]⟦(critical[i1] ∧ critical[i2]...
Formula automaton with 65 states generated.
8 system states and 208 product automaton states investigated.
LTL formula is satisfied (model checking time: 5 ms).
Execution completed (6 ms).
```

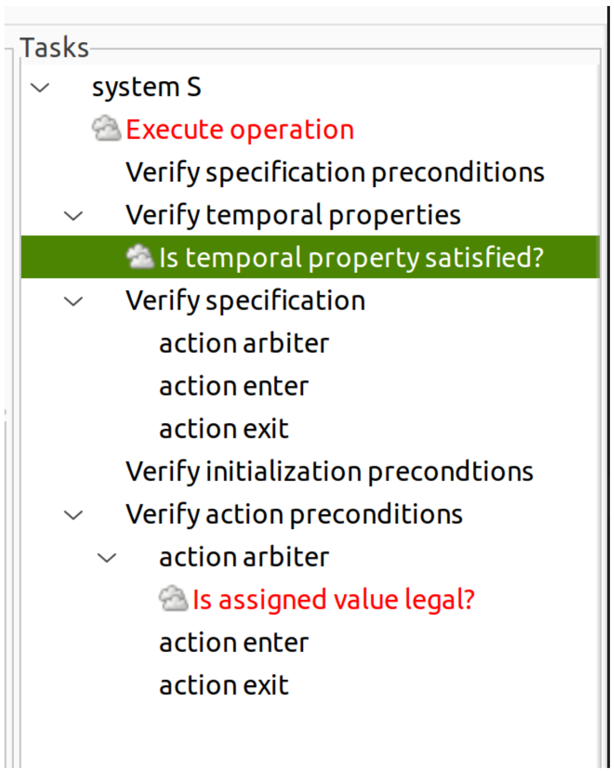

Tasks
system S
Execute operation
Verify specification preconditions
Verify temporal properties
Is temporal property satisfied?
Verify specification
action arbiter
action enter
action exit
Verify initialization precondtions
Verify action preconditions
action arbiter
Is assigned value legal?
action enter
action exit

Figure 7.2: Tasks menu with model checking commands

The output in this case is self-explanatory: some information is given about the complexity of the problem (size of the automaton generated from the formula, number of visited system states and number of visited product automaton states during the emptiness checking) and then the result: the formula is satisfied, our system is correct (with respect to this requirement).

Checking this simple safety property is not enough to ensure that the system really works as we expect it to. It is not enough to know that no two processes are in the critical section at the same time, we would also like to make sure that each process infinitely often enters it. This can be expressed using the following LTL formula:

```
ltl ∀i: Proc. □◇ ⟦ critical[i] ⟧;
```

However running this yields an error:

```
Executing _ltl1().
Run of deterministic function _ltl1():
Checking LTL formula ∀i:Proc. ([](<>⟦critical[i]⟧))...
Formula automaton with 20 states generated.
7 system states and 33 product automaton states investigated.
ERROR in execution of _ltl1(): evaluation of
  ∀i:Proc. ([](<>⟦critical[i]⟧))
at line 11 in file system.txt:
  LTL formula is NOT satisfied (model checking time: 3 ms).
Counterexample execution:
Action: init() values: [critical:[false,false,false,false],next:0]
```

```
Action: arbiter() values: [critical:[false,false,false,false],next:1]
> Loop start
    Action: arbiter() values: [critical:[false,false,false,false],next:2]
    Action: arbiter() values: [critical:[false,false,false,false],next:3]
    Action: arbiter() values: [critical:[false,false,false,false],next:0]
    Action: arbiter() values: [critical:[false,false,false,false],next:1]
> Loop end
ERROR encountered in execution (6 ms).
```

As the semantics of LTL is defined over infinite system runs, any counterexample must also be infinite and as discussed in the theory section, the counterexamples provided by the system will consist of a prefix and an infinitely repeating loop. The counterexample describes which actions are executed (*init* is the initial action, implicit in this case, but it can be overridden) and what the values of the variables are after each action.

In this counterexample we can observe that only the arbiter action is ever executed, and if we check the system definition, we find that this is indeed a correct system run, meaning our property does not hold. Only under certain assumptions about the fairness of the underlying scheduler of the processes can we make sure that every worker process reaches the critical section. As discussed in Section 7.1 we can express such assumptions using the `WeakFairness` and `StrongFairness` formulas.

We can start by adding a `WeakFairness` constraint to the enter action of each process:

```
ltl ∀(i1:Proc. WeakFairness enter(i1)) ⇒ □◇ ⟦ critical[i] ⟧;
```

If we run this, after a rather long wait, we get a counterexample just like before. That is because while the arbiter process is cycling through the values for the variable *next*, no worker process has its enter action always enabled. We need to strengthen our assumptions to strong fairness, which only requires the action to be infinitely often enabled:

```
ltl ∀(i1:Proc. StrongFairness enter(i1)) ⇒ □◇ ⟦ critical[i] ⟧;
```

This is still not correct, but at least it yields a different counterexample:

```
Counter example execution:
Action: init() values: [critical:[false,false,false,false],next:0]
Action: arbiter() values: [critical:[false,false,false,false],next:1]
> Loop start
    Action: enter(1) values: [critical:[false,true,false,false],next:1]
    Action: enter(1) values: [critical:[false,true,false,false],next:1]
> Loop end
```

Now the problem is that the arbiter action is never executed, because the first worker process keeps entering and exiting the critical section, blocking access for every other process. The solution to this is to add a strong fairness assumption to the arbiter action (weak fairness is not enough here either, since the arbiter cannot execute while any process is in the critical section):

```
ltl ∀((i1:Proc. StrongFairness enter(i1)) ∧ StrongFairness arbiter)
        ⇒ □◇ ⟦ critical[i] ⟧;
```

However this property generates a huge (45692 states) automaton already for 4 processes, and running it takes a very long time. One more process and it is almost guaranteed that we run out of memory or time before model checking completes. This means that checking a more complex system with fairness constraints is impossible by using the naive approach.

As discussed in Section 5.3, there is a better algorithm for model checking under fairness constraints, but for that algorithm to work we need a list of the fairness formulas. While these could be extracted from a simple formula automatically, no such optimization is implemented, because there is a much simpler way of specifying which actions are fair.

The previous section describes how each action can be annotated with one of `fairness weak`, `fairness strong`, `fairness weak_all` or `fairness strong_all` and what LTL formulas they are equivalent to. It is then trivial to collect the list of constraints denoted by $wf(a_i)$ and $sf(b_j)$ in Section 5.3 by going through the actions and expanding `fairness weak_all` and `fairness strong_all` the same way we would expand a quantification in the formula.

For example the strong fairness on the *enter* action of all processes, which was written as `∀i1:Proc. StrongFairness enter(i1)` when part of the formula can be described using the annotation

```
action enter(i: Proc) with i = next ∧ ¬critical[i];
  fairness strong_all;
```

We expect some properties to hold also without fairness and there is an overhead associated with the use of Algorithm 5.4, therefore checking under fairness is not enabled by default. If we want a specific formula to be checked with the fairness constraints, it has to be introduced by `ltl [fairness]` instead of `ltl`. The final code with the actions annotated and the liveness property prefixed is presented in Program 11.

When checking the safety LTL formula for this implementation nothing changes compared to the previous version. However checking the liveness formula outputs

```
Executing _ltl1().
Run of deterministic function _ltl1():
Checking LTL formula ∀i:Proc. ([](<>⟦critical[i]⟧))...
Formula automaton with 20 states generated.
8 system states and 140 product automaton states investigated.
LTL formula is satisfied (model checking time: 3 ms).
Result (5 ms): ()
Execution completed (5 ms).
```

This is at least 100000 times faster than the naive approach!

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

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

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

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

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

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

Program 11: Mutual exclusion algorithm in RISCAL, with LTL formulas and fairness annotations

# 8 IMPLEMENTATION

This chapter presents how the algorithms from Chapter 5 were implemented in Java to yield the software demonstrated in Chapter 7. The implementation is collected in the package `riscal.ltl`. It consists of 32 classes, 148 methods and ca. 1500 lines of source code (excluding comments and blank lines). The package is part of the RISCAL software since version 4.2.0.

## 8.1 PREPROCESSING

The automaton creation algorithm Algorithm 5.1 on page 31 expects the LTL formula to contain only a small subset of logical and temporal operators allowed by the RISCAL system. Thus every disallowed operator needs to be replaced with equivalent representations using the allowed operators.

The RISCAL parser returns an abstract syntax tree, in which the nodes are inner classes of and extend the class `AST.LTL`, e.g. `AST.LTL.ForallLTL`, `AST.LTL.IfThenElseLTL`, and `AST.LTL.WeakUntilLTL`. The model checker module, however, uses a different tree representation for the simplified formulas. This representation consists of nodes which are inner classes of and extend the `LTLFormula` class, e.g. `LTLFormula.NegationNode`, `LTLFormula.OperatorNode`, and `LTLFormula.ActionNode`.

The translation between the two representations is facilitated by the `LTLPreprocessor` class, which defines methods that recursively translate each of the nodes from the abstract syntax tree to the simplified representation. The entry-point to the translation is the `LTLFormula preprocess(AST.LTL ast, Types.Context context)` method, which dispatches the actual translation based on the type of the `ast` variable to one of the methods.

For operators which are allowed by the automaton creation algorithm, the translator methods are trivial:

```java
@Override
public LTLFormula translate(AST.LTL.UntilLTL ast, Types.Context context) {
    return new LTLFormula.OperatorNode(
            UNTIL,
            preprocess(ast.ltl1, context),
            preprocess(ast.ltl2, context)
    );
}
```

For others the translation is a bit more involved. In the case of the weak until operator there are multiple possibilities for an equivalent representation using the allowed operators. The following implementation was chosen, because it results in an optimal automaton after executing the automaton creation algorithm (see the comment):

```
@Override
public LTLFormula translate(AST.LTL.WeakUntilLTL ast, Types.Context context) {
    // f W g == (f U g) || G f == f U (g || G f) == g V (g || f)
    // here we use the last equivalence as that produces the simplest
    // automaton using this algorithm

    LTLFormula left = preprocess(ast.ltl1, context);
    LTLFormula right = preprocess(ast.ltl2, context);

    return new LTLFormula.OperatorNode(
            RELEASE,
            right,
            new LTLFormula.OperatorNode(OR, right, left)
    );
}
```

Indeed, the translation of $f\ \mathbf{W}\ g$ to $(f\,\mathbf{U}\,g) \vee \mathbf{G}f$ results in 13 automaton states, the translation $f\,\mathbf{U}\,(g \vee \mathbf{G}f)$ in 7, while the chosen representation $g\,\mathbf{V}\,(g \vee f)$ results in only 5.

The expansion of universal and existential quantifiers uses a helper method, since the only difference between the two are the default value if there are no elements in the set over which the quantification is done (true for universal, false for existential), and the operator which connects the expanded subformulas ($\wedge$ for universal, $\vee$ for existential quantification):

```
private LTLFormula expandQuantifier(AST.LTL ltl, AST.QuantifiedVariable quantifier,
        Types.Context context, LTLFormula def, LTLFormula.LTLOperator connective) {
    List<Types.Context> contexts = LTLChecker.getContexts(context, quantifier);
    if (contexts.size() == 0) {
        return def;
    }

    LTLFormula result = preprocess(ltl, contexts.get(0));
    for (int i = 1; i < contexts.size(); i++) {
        LTLFormula current = preprocess(ltl, contexts.get(i));
        result = new LTLFormula.OperatorNode(connective, result, current);
    }

    return result;
}

@Override
public LTLFormula translate(AST.LTL.ForallLTL ast, Types.Context context) {
    return expandQuantifier(ast.ltl, ast.qvar, context,
        LTLFormula.BooleanNode.TRUE, AND);
}

@Override
public LTLFormula translate(AST.LTL.ExistsLTL ast, Types.Context context) {
    return expandQuantifier(ast.ltl, ast.qvar, context,
        LTLFormula.BooleanNode.FALSE, OR);
}
```

The `Types.Context` object contains a mapping from the names bound at the current point in the translation (either by `let`, `letpar` or a quantifier) to their actual values.

The most involved translation is the one for the fairness formulas, as we have to construct either **FG** Enabled $a$ or **GF** Enabled $a$ and also **GF** Executed $a$ and connect them using an implication (which itself needs translation to negation and disjunction).

```
// WeakFairness a == (<>[] Enabled a) => ([]<> Executed a)
// StrongFairness a == ([]<> Enabled a) => ([]<> Executed a)
LTLFormula lhs;
if (kind == AST.ActionLTLTag.Kind.weakfairness) {
    // equivalent to !(<>[] Enabled a)
    lhs = new LTLFormula.OperatorNode(
            RELEASE,
            LTLFormula.BooleanNode.FALSE,
            new LTLFormula.OperatorNode(
                    UNTIL,
                    LTLFormula.BooleanNode.TRUE,
                    new LTLFormula.NegationNode(
                            new LTLFormula.ActionNode(AST.ActionLTLTag.Kind.enabled,
                                    symbol, instance, args)
                    )
            )
    );
} else {
    // equivalent to !([]<> Enabled a)
    lhs = ...
}
```

Working on `LTLFormulas` is done using implementations of the `LTLProcessor` interface. The interface defines a `process` method for each of the possible nodes in the formula:

```
public interface LTLProcessor<T, Q> {
        T process(LTLFormula.BooleanNode LTL, Q context);
        T process(LTLFormula.NegationNode LTL, Q context);
        T process(LTLFormula.NextNode LTL, Q context);
        T process(LTLFormula.OperatorNode LTL, Q context);
        T process(LTLFormula.StateNode LTL, Q context);
        T process(LTLFormula.ActionNode LTL, Q context);
}
```

The `LTLFormula` abstract class defines the process method:

```
public abstract <T, Q> T process(LTLProcessor<T, Q> processor, Q context);
```

This is then implemented simply by each of the subclasses:

```
public static class OperatorNode extends LTLFormula {
    ...

    @Override
    public <T, Q> T process(LTLProcessor<T, Q> processor, Q context) {
        return processor.process(this, context);
    }
}
```

After the translation of the abstract syntax tree to a simplified LTL formula, this formula has to be converted into negation normal form. This can be done by simply using an implementation of the `LTLProcessor` interface in which the context is a Boolean value, denoting whether we should be "negating" at the current level (i.e. there have been an odd number of negations before this point):

```
private static class NNFConverterImpl implements LTLProcessor<LTLFormula, Boolean> {
    ...

    @Override
    public LTLFormula process(LTLFormula.NegationNode LTL, Boolean negated) {
        return LTL.child.process(this, !negated);
    }

    @Override
    public LTLFormula process(LTLFormula.OperatorNode LTL, Boolean negated) {
        LTLFormula.LTLOperator operator = negated ? getDual(LTL.operator) : LTL.operator;
        LTLFormula left = LTL.left.process(this, negated);
        LTLFormula right = LTL.right.process(this, negated);

        return new LTLFormula.OperatorNode(operator, left, right);
    }
}
```

After this conversion the formula is ready to be passed on to the automaton creation algorithm.

## 8.2 AUTOMATON CREATION

The core of the automaton creation, the `createTableau` method, is a straightforward implementation of Algorithm 5.1 on page 31. It returns a set of objects of type `TableauNode`, which is defined as follows:

```
public static class TableauNode {
    public final Set<TableauNode> incoming;
    public final Set<LTLFormula> newFormulas;
    public final Set<LTLFormula> oldFormulas;
    public final Set<LTLFormula> nextFormulas;

    public TableauNode(...) { ... }
}
```

The `then` branch of the first condition in `expand` does not differ from the pseudocode, but the else branch uses another implementation of the `LTLProcessor` interface:

```
private static Set<TableauNode> expand(TableauNode node, Set<TableauNode> nodesSet) {
    if (node.newFormulas.isEmpty()) {
        ...
    } else {
        LTLFormula f = node.newFormulas.stream().findAny().get();
        node.newFormulas.remove(f);
        return f.process(new FormulaProcessor(), new ProcessingContext(node, nodesSet));
    }
}
```

The `ProcessingContext` is a simple class grouping the current node and the already finalized nodes (`nodesSet`).

The `FormulaProcessor` handles contradictions arising during the expansion and the splitting of the node when the root of the currently handled formula is a binary operator. When splitting the node, instead of creating two new instances, the old node is reused.

```
@Override
public Set<TableauNode> process(LTLFormula.OperatorNode LTL,
        ProcessingContext context) {
    if (LTL.operator == LTLFormula.LTLOperator.AND) {
        ...
    } else { // OR, UNTIL, RELEASE
        // compute the new and next sets of the first created node
        Set<LTLFormula> node1new = new1(LTL, context.node.newFormulas,
                context.node.oldFormulas);
        Set<LTLFormula> node1next = next1(LTL, context.node.nextFormulas);

        // for the second created node we reuse the old one
        // by just modifying the old and new sets
        context.node.oldFormulas.add(LTL);
        new2(LTL, context.node.newFormulas, context.node.oldFormulas);

        TableauNode node1 = new TableauNode(
                new HashSet<>(context.node.incoming),
                node1new,
                new HashSet<>(context.node.oldFormulas),
                node1next
        );

        return expand(context.node, expand(node1, context.nodesSet));
    }
}
```

The automaton construction is a two-step algorithm: after the construction of the tableau, the automaton is constructed by computing the final sets. In this implementation the automaton nodes are represented as follows:

```
public class AutomatonNode {

        public List<AutomatonNode> children;
        public List<LTLFormula> formulas;
        // bitset, the i-th bit being set means this node is part of
        // the i-th final set of the automaton
        public long finalSets;
}
```

The conversion is done by "flipping the edges" (in the graph each node refers to its ancestors, while in the automaton they refer to their successors) and by determining which nodes are contained within each final set. As described in Section 5.1, there is one accepting set $F_i \subseteq S$ for each subformula of $f$ of the form $g\,\mathbf{U}\,h$, and for a state `n` we have $\mathtt{n} \in F_i$ if $g\,\mathbf{U}\,h \notin$ `n.old` or $h \in$ `n.old`. To find all subformulas with an until operator as root, again an implementation of the `LTLProcessor` interface is used, namely the `UntilSubformulaFinder` class.

## 8.3 EMPTINESS CHECKING

The product of the system automaton and the property automaton is not constructed prior to the emptiness checking, it is expanded *on-the-fly*, which makes it possible to find counterexamples without considering all reachable states.

The states in the product automaton are represented by the `CheckingState` class, which consists of

1. the respective formula automaton and system states,
2. auxiliary information needed for the emptiness checking algorithm,
3. a list of successors, or null, if they haven't been computed yet,
4. a method which computes said successors.

It is worth to take a look at this method, as it is the one which makes sure that there is only one instance of any given state of the product automaton. Without this restriction it is impossible to store any additional information within the checking state and they would have to be stored in additional lookup tables (e.g. one mapping each state to its successors). This way there is only one cache containing the already constructed states (`checkingStateCache` in `LTLChecker`) and the existing one is retrieved if an equivalent is encountered. For technical reasons, this has to be a `Map` instead of a `Set`. The code of this method is as follows:

```java
public List<CheckingState> getSuccessors() {
    if (successors == null) {
        successors = new ArrayList<>();

        for (Systems.StateClass systemState : ltlChecker.getSuccessors(stateNode)) {
            for (AutomatonNode automatonState : automatonNode.children) {
                if (!automatonState.isCompatibleWith(ltlChecker, systemState)) {
                    continue;
                }

                successors.add(
                        ltlChecker.checkingStateCache.computeIfAbsent(
                                new CheckingState(ltlChecker, automatonState, systemState),
                                x -> x
                        )
                );
            }
        }
    }
    return successors;
}
```

For emptiness checking the *ASCC* algorithm (see Algorithm 5.3 on page 39) was chosen, because:

1. it handles generalized Büchi automata without requiring any extra care when there is more than one accepting set,
2. it makes it possible to apply the fast fairness checking algorithm,
3. the extra memory costs are negligible compared to the memory required for the representation of the system states and the product automaton.

Unlike the automaton creation, this is not a straightforward implementation, as here the recursive algorithm is translated into an iterative one. This is done to make sure it only uses constant stack space and to make the path to the SCC explicit. For this conversion to work we need to store explicitly all the information which would otherwise be implicit in the call stack: the values of the local variables at the point of the recursive call. The variable $s$, the current state will be stored on the stack we introduce, but there is one more local variable hidden in the algorithm, namely the loop variable of the loop at line 6 of Algorithm 5.3. In our implementation this value (the number of already visited children) is stored within the checking state object.

The first few lines of the implementation, which are the one affected by these changes, are as follows:

```
private CounterExample findSCCs(CheckingState initialState) {
    int count = 0;
    Deque<CheckingState> searchPath = new ArrayDeque<>();

    Deque<Pair<CheckingState, Long>> roots = new ArrayDeque<>();
    Deque<CheckingState> actives = new ArrayDeque<>();

    searchPath.push(initialState);

    while (!searchPath.isEmpty()) {
        CheckingState state = searchPath.peek();

        if (state.isUnprocessed()) {
            count++;
            state.dfsnum = count;
            state.current = true;

            roots.push(new Pair<>(state, state.automatonNode.finalStates));
            actives.push(state);
        }

        for (CheckingState successor = state.getFirstSuccessor();
             successor != null; successor = state.getNextSuccessor()) {
             ...
```

The remainder of the method has no meaningful modifications except for further processing of the SCCs in case fairness checking is enabled.

## 8.4 FAIRNESS CHECKING AND COUNTEREXAMPLE GENERATION

The fairness checking algorithm is enabled when the formula is introduced by the keyword `ltl [fairness]`. In this case all the fairness constraints are collected into a list containing formulas of the form `WeakFairness action`, or `StrongFairness action` or universally quantified versions of these.

The looping part of the counterexample has to satisfy these fairness constraints and also it has to contain at least one state from each final set. These requirements (`FairnessConstraint` and final state constraints) are stored in classes implementing the `LoopConstraint` interface. This is a functional interface, its `checkState` method takes a state and returns true iff including this state in the loop satisfies the given constraint. For example the `FairnessConstraint` class implements it as follows:

```
@Override
public boolean checkState(CheckingState state, LTLChecker ltlChecker) {
    switch (type) {
    case Weak:
        return !enabled.isCompatibleWith(state.stateNode, ltlChecker)
                || executed.isCompatibleWith(state.stateNode, ltlChecker);
    case Strong:
        return executed.isCompatibleWith(state.stateNode, ltlChecker);
    default:
        throw new RuntimeException("Invalid␣fairness␣constraint");
    }
}
```

Thus the first step in fairness checking is to translate the list of formulas to a list of loop constraints, which is done by `FairnessConstraint::makeFairnessConstraints`.

Then the `findSCCs` method from the previous section collects all the states of the SCC into a list and passes them on to the `FairnessChecker`'s `findFairSCC` method.

```
public class FairnessChecker {
    private final List<FairnessConstraint> weakConstraints;
    private final List<FairnessConstraint> strongConstraints;

    public Pair<List<CheckingState>, List<LoopConstraint>> findFairSCC(
            List<CheckingState> SCC) {
        return findFairSCC(SCC, new ArrayList<>(strongConstraints));
    }

    private Pair<List<CheckingState>, List<LoopConstraint>> findFairSCC(
            List<CheckingState> SCC,
            List<FairnessConstraint> remainingStrongConstraints) {
        ...
    }

    /**
```

```
     * For finding the admissible sub-SCCs we use the same algorithm
     * as in the initial partition into SCCs with some slight
     * technical modifications, since we cannot reuse the fields of
     * the CheckingState.
     */
    private Pair<List<CheckingState>, List<LoopConstraint>> findSubSCC(
            List<CheckingState> remainingStates,
            List<FairnessConstraint> remainingStrongConstraints) {
        ...
    }
}
```

The methods `findFairSCC` and `findSubSCC` are mutually recursive. The method `findFairSCC` is a simple implementation of Algorithm 5.4 on page 40. The method `findSubSCC`, which is the equivalent of DECOMPOSE_INTO_SCCS, is essentially a copy of `findSCCs` with the exception of using a local set and map to store the `dfsnum` and `current` status of the states. This is because we are within a recursive call and we cannot change the data the calling method is working with, in case it needs to continue looking for more SCCs.

Assuming fairness checking was enabled and it has succeeded in finding a counterexample SCC, it will return a list containing the states of the SCC and a list containing the remaining strong fairness constraints (some strong fairness constraints are automatically satisfied by any loop in the SCC, since it doesn't contain any states in which the respective action is enabled). The final state constraints are added to this list (one for each final set), and then all this is passed to the counterexample finder.

This starts from the root node of the SCC (the one with the lowest `dfsnum`) and does a breadth-first search starting from that node looking for the nearest state which satisfies any of the loop constraints. Then it removes the satisfied constraint from the list and starts the search again until the list of constraints is exhausted. Then it starts one final search to join the path back to the initial node, making it a loop.

# 9 BENCHMARKS

In this chapter we demonstrate by various benchmarks how the RISCAL model checker performs for systems of varying complexity and size when verifying both safety and liveness properties. In every case the performance of RISCAL is compared to the TLC model checker, and for the first benchmark also to Spin.

All benchmarks were performed on a Linux machine with a 3.60Ghz Intel i9-9900K 8 core CPU with 32 GB RAM using the OpenJDK 17.0.3 Java Runtime Environment. RISCAL had access to 25 GB of memory (was ran with the `-Xmx25000m` VM option). The RISCAL invariant checker supports multi-threading, so it was tested using both 1 and 8 worker threads. The RISCAL LTL model checker is single threaded. The TLC LTL checker supports multi-threading so it was also ran with both 1 and 8 threads, unless it made no discernible difference.

Unless otherwise noted, the source code for the benchmarks are from various personal communications by the thesis advisor.

## 9.1 BENCHMARK 1: SIMPLE COUNTER

THE MODEL The benchmark models a system whose state consists of two integer variables $x$ and $y$. Two processes repeatedly increment one of the variables in the interval $[0, N-1]$ for some natural number $N$. Once any of the variables reaches $N-1$, it is reset to 0 instead of being incremented further.

The RISCAL source of this benchmark is given in Appendix A as Program 12, the TLA$^+$ source as Program 13 and the PROMELA source as Program 14. It is taken from the conference paper [SS22] which first presented the model checking extension to RISCAL.

For any given value of the parameter $N$ the system has exactly $N^2$ states.

We verify three properties for this system:

1. the safety property "both x and y are in the interval $[0, N-1]$, expressed in RISCAL as

   ```
   ltl ⟦ 0 ≤ x ∧ x < N ∧ 0 ≤ y ∧ y < N ⟧;
   ```

   which results in an automaton with 3 states.

2. the liveness property "$x$ infinitely often becomes 0", which holds only if we assume weak fairness for the action *incX*. It is expressed in RISCAL naively using the `WeakFairness` LTL operator as

```
ltl WeakFairness incX ⇒ ([] ⟦ x = 0 ⟧);
```

which results in an automaton with 20 states, or using the fairness annotations as

```
ltl[fairness] ([]<> ⟦ x = 0 ⟧);
```

which results in an automaton with 3 states.

3. the liveness property "both $x$ and $y$ infinitely often become 0", which holds only if we assume weak fairness for both the actions *incX* and *incY*. It is expressed in RISCAL naively using the `WeakFairness` LTL operator as

```
ltl WeakFairness incX ∧ WeakFairness incY ⇒
    ([] ⟦ x = 0 ⟧) ∧ ([] ⟦ y = 0 ⟧);
```

which results in an automaton with 160 states, or using the fairness annotations as

```
ltl[fairness] ([] ⟦ x = 0 ⟧) ∧ ([] ⟦ y = 0 ⟧);
```

which results in an automaton with 10 states.

In all five cases the pre-processing and automaton creation took less than 50 micro-seconds, a time which is insignificant compared to the rest of the model checking process.

CHECKING SAFETY The safety property can also be expressed using an invariant which is checked using Algorithm 2.1. We first compare the performance of the invariant check to the performance of the model checker for such simple formulas. As it can be seen in Fig. 9.1, checking using invariants with a single thread is 2 to 3 times faster than even a simple LTL formula. Checking the invariant using 8 threads becomes faster starting at around $N = 1000$ and provides around 30% speedup for the largest tested case.

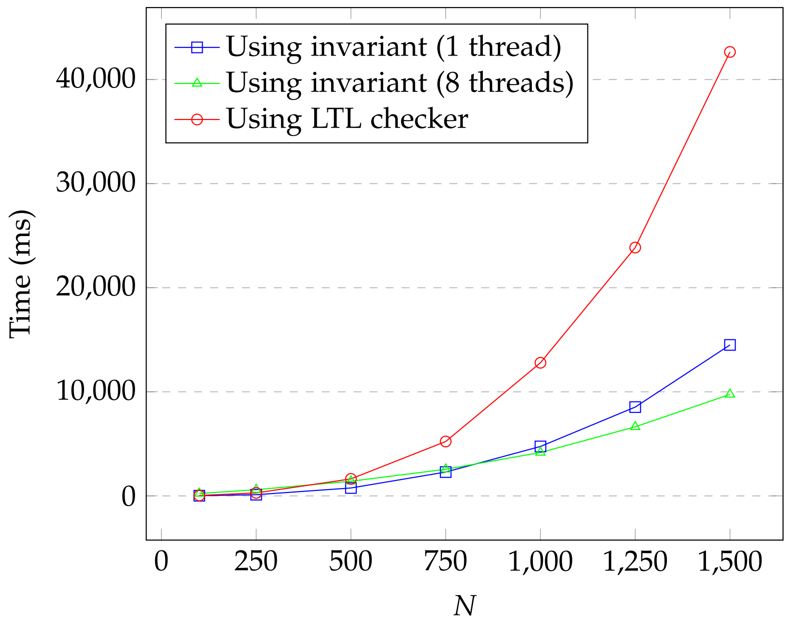


Figure 9.1: Performance of the LTL model checker compared to invariant checking

The primary reason for the difference in performance is that due to the structure of the formula automaton, the LTL model checker traverses $2N^2 + 1$ states in the product automaton compared to the $N^2$ the invariant checking has to inspect. The additional difference

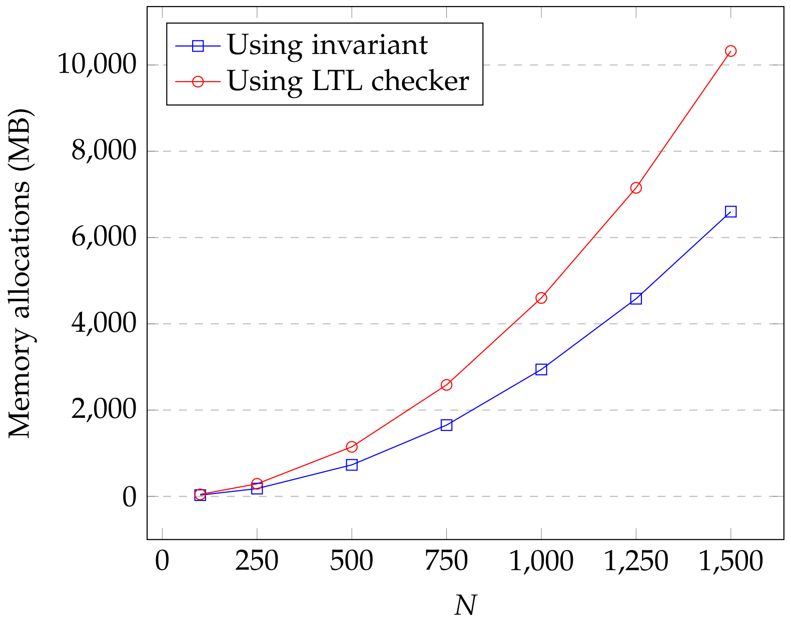


Figure 9.2: Total allocations of the LTL model checker compared to invariant checking

between the time performance can be explained by the increased memory usage of the LTL model checker, see Fig. 9.2. Every product automaton state visited requires the allocation of a new `CheckingState` object which has 4 reference, 2 integer and one Boolean fields, for a total of 40 bytes on a 64 bit machine. However, as the graph shows for such a simple formula automaton the majority of the necessary memory is still taken up by the explicit representation of the system.

CHECKING LIVENESS Next we compare how the two liveness properties perform when using the naive approach (i.e. adding `WeakFairness incX` and `WeakFairness incX` $\wedge$ `WeakFairness incY` respectively as a precondition) versus using the fast fairness checking algorithm (prefixing the formula with `[fairness]` and annotating the actions with `fairness weak`). Fig. 9.3 shows the comparison for both properties.

Unsurprisingly, the naive approach performs significantly worse. In 100 seconds for the first property (one with just a single constraint) we were able to verify a system with $1.5^2 = 2.25$ times as many states. With two constraints the differences in speed are even more significant. In this case, with the same time limit, the naive approach can only verify a system with $N = 500$, compared to $N = 1100$ for the optimized algorithm, meaning a five times larger system.

The final experiment with the *Counter* system concerns how well the fairness checking algorithm performs when there are many constraints. For this, the definition of the system is changed such that the state consists of the variables $x_1$, ..., $x_6$ and for each variable there is an action which increases the variable (`incX_1`, ..., `incX_6`). The formula we check is the same as the first liveness formula from before, i.e.

```
ltl[fairness] ([]<> ⟦ x1 = 0 ⟧);
```

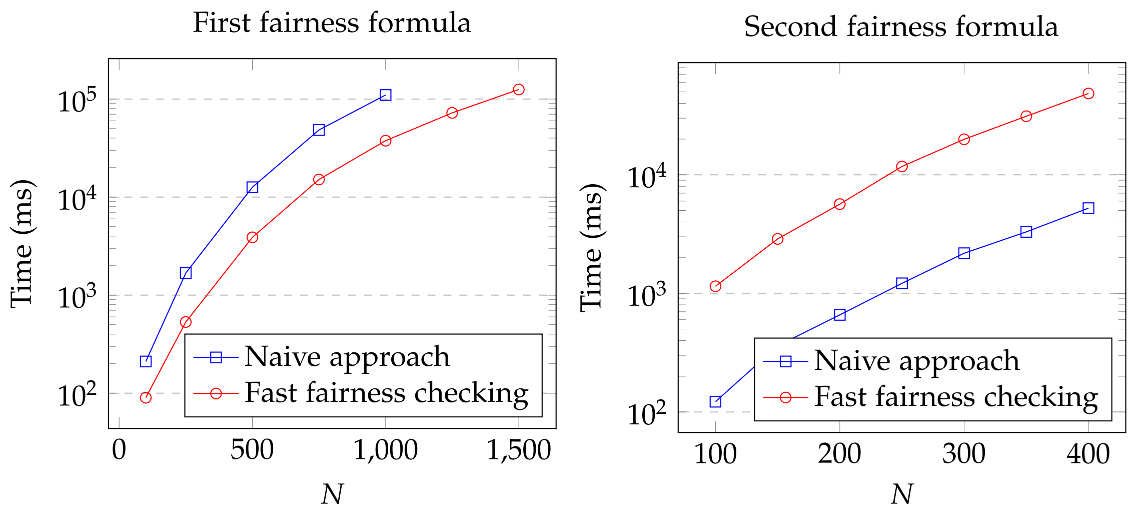


Figure 9.3: Naive fairness checking compared to the fast fairness checking algorithm

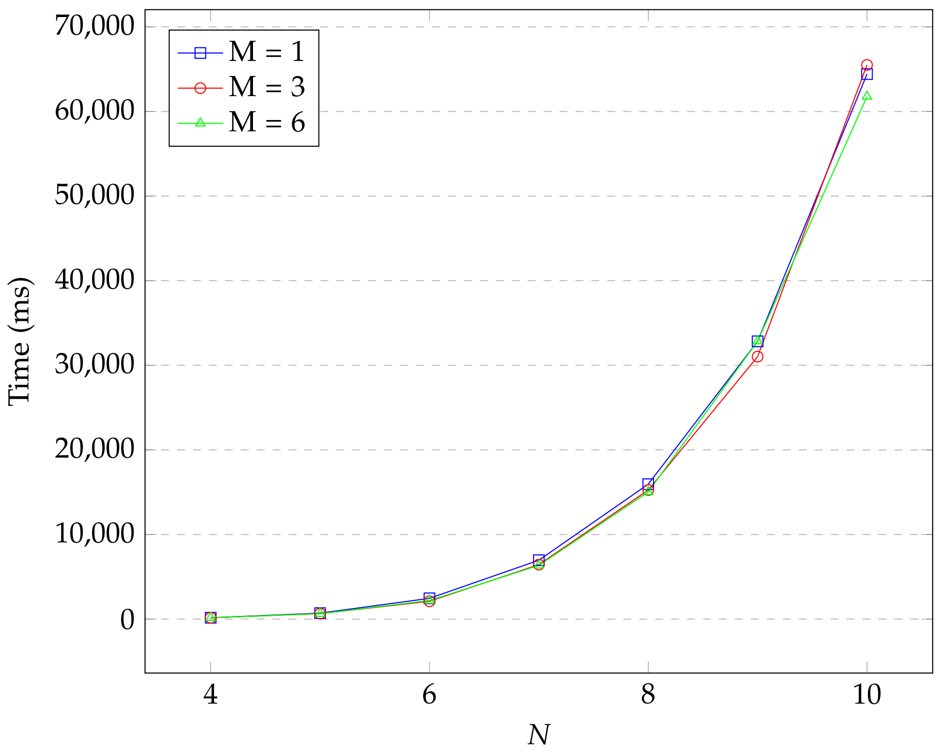


Figure 9.4: Fairness checking algorithm with differing number of weak fairness constraints

A weak fairness constraint on the action `incX_1` is sufficient to verify the property. Fig. 9.4 compares the performance when the actions `incX_1`, ..., `incX_M` are annotated with `fairness weak` for $M = 1$, $M = 3$ and $M = 6$. As the figure shows, there is no observable difference between 1 or 6 weak fairness constraints.

RISCAL VERSUS SPIN Next we compare the performance of the RISCAL model checking extension to the Spin software described in Section 2.2. Spin is a highly optimized tool that translates the model of the system and the properties into C code which is then compiled into machine code. Thus Spin avoids the overhead of interpreting the system by an unoptimized interpreter, and it can also represent system and product automaton states more efficiently. Besides this Spin also supports partial order reduction, an optimization technique which can reduce the number of states that need to be visited by an order of magnitude.

| N | 500 | 1000 | 2500 | 5000 | 10000 |
|---|---|---|---|---|---|
| Safety | 0.09 | 0.335 | 2.315 | 12.5 | 67 |
| Liveness 1 | 0.46 | 1.89 | 13.45 | 55 | |
| Liveness 2 | 0.825 | 3.435 | 23.5 | 103 | |

Table 9.1: Measurements of the counter system using Spin, timings in seconds

As Table 9.1 shows, Spin is capable of handling much larger systems than RISCAL. In the time it took RISCAL to verify the safety property for a system with $2.25 \cdot 10^6$ states (for $N = 1500$), Spin managed to verify it for a system with $10^8$ states (for $N = 10000$), i.e. almost two orders of magnitude more. The difference for the liveness properties was slightly smaller. Compared to being 30 times faster in the verification of the safety property, it verified the first liveness property 20 times faster, while the second one 15 times faster than RISCAL.

Spin can handle formulas with weak fairness constraint (on all actions) well, however it has no built-in support for strong fairness constraints. Thus one can only use the naive approach in this case, which very quickly leads to an explosion in the number of formula automaton states. Spin also has simpler specification language compared to RISCAL, meaning that even though it performs better in the tested cases, it is not a clear winner.

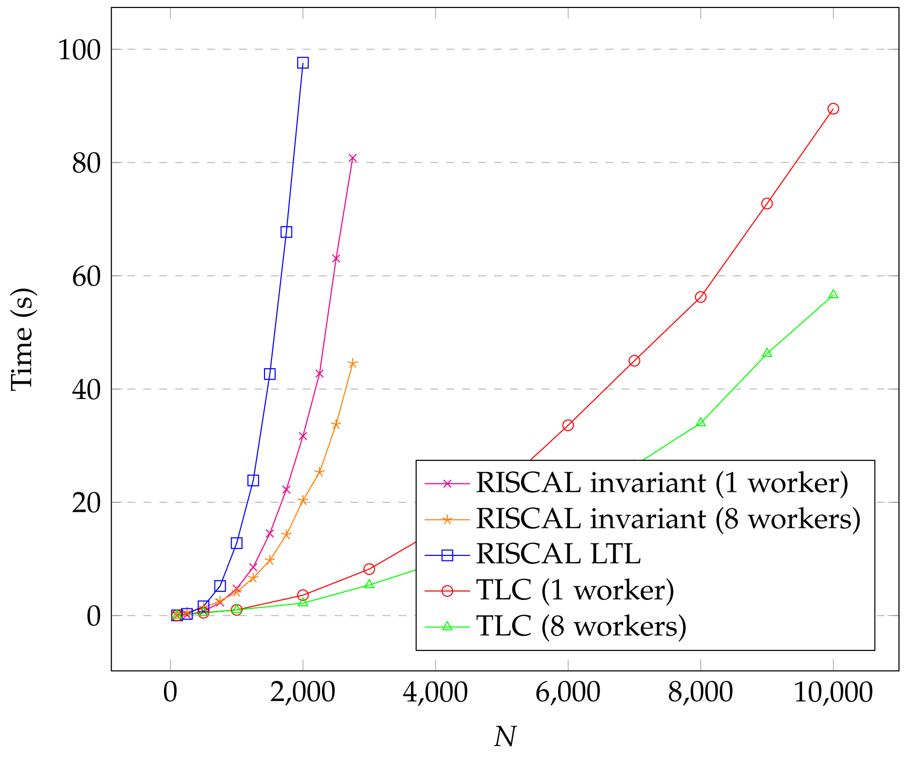


Figure 9.5: Comparison of RISCAL and TLC for checking the safety property

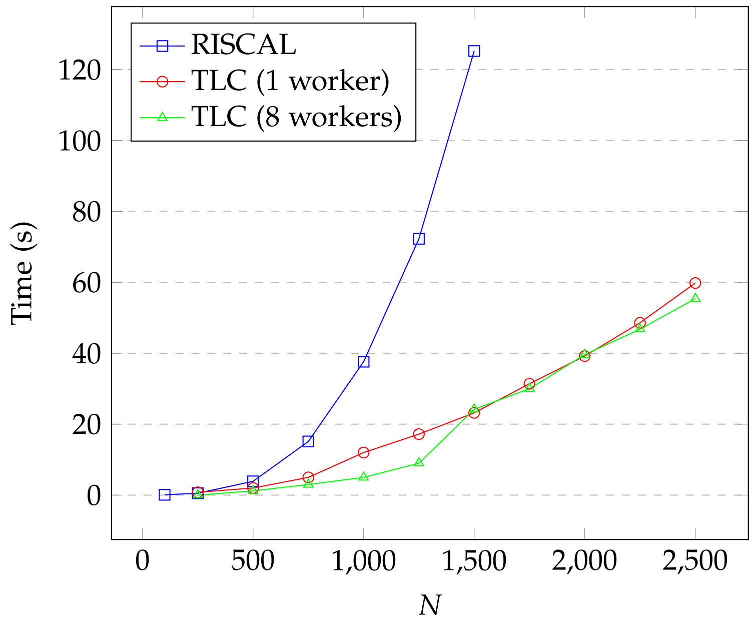


Figure 9.6: Comparison of RISCAL and TLC for checking the first liveness property

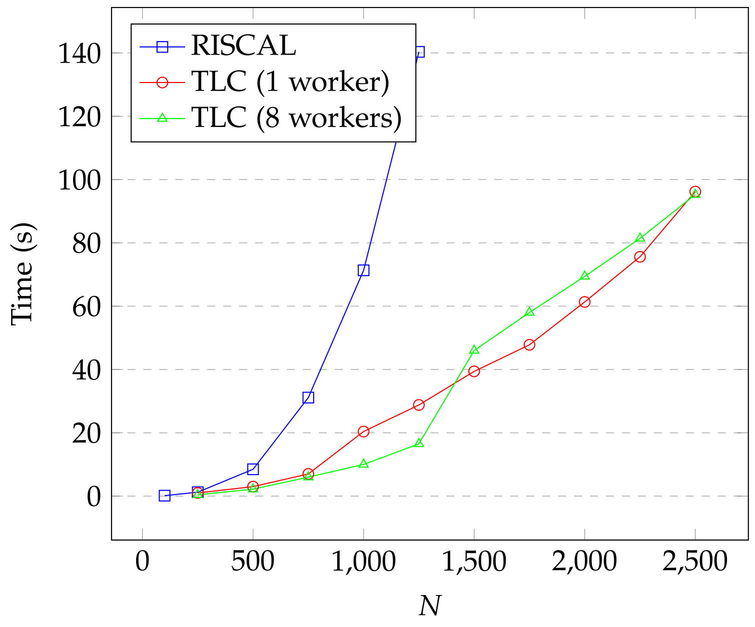


Figure 9.7: Comparison of RISCAL and TLC for checking the second liveness property

RISCAL VERSUS TLC Next we compare RISCAL to TLC, the model checker for specifications written in the TLA$^+$ language, described in Section 2.3. This is a more fair comparison, because unlike Spin, TLC does not compile the definition of the system into machine code, thus it has to deal with the overhead of interpreting the system. Both model checkers are implemented in Java.

TLC also supports invariants, but unlike RISCAL where invariants are checked by a different, more efficient algorithm, TLC simply converts the invariant $f$ to the temporal formula $\mathbf{G}f$ and checks that. Thus it makes no sense to measure the two independently.

Fig. 9.5 compares the performance of RISCAL and TLC with 1 and 8 workers for the safety formula. While all three plots follow roughly quadratic curves, the constant factor for RISCAL is much bigger. Given that TLC uses a "Hash Compact" representation for the states, this is not surprising. The largest system RISCAL was able to verify under 100 seconds was of size $4 \cdot 10^6$ whereas TLC (with only a single worker thread) verified a system with $10^8$ states.

Fig. 9.6 and Fig. 9.7 show that for the liveness properties the situation is very similar. TLC again outperforms RISCAL, though by a slightly smaller margin. Interestingly for these properties using more workers doesn't speed up the verification, at least not in the ranges measured here.

Of note is that for the second liveness property, $\mathbf{GF}(x = 0) \land \mathbf{GF}(y = 0)$, for which RISCAL produces an automaton with 10 states TLC uses an automaton with only three states, just like for the previous two properties. This is probably done by splitting the formula at the conjunction into $\mathbf{GF}(x = 0)$ and $\mathbf{GF}(y = 0)$ which can be verified independently. This is an optimization that RISCAL could also easily implement and it would be especially useful when the outermost operator of the LTL formula is a universal quantifier.

## 9.2 BENCHMARK 2: ALTERNATING BIT PROTOCOL

THE MODEL The alternating bit protocol is an algorithm for transmitting data over a lossy FIFO connection [Tel00].

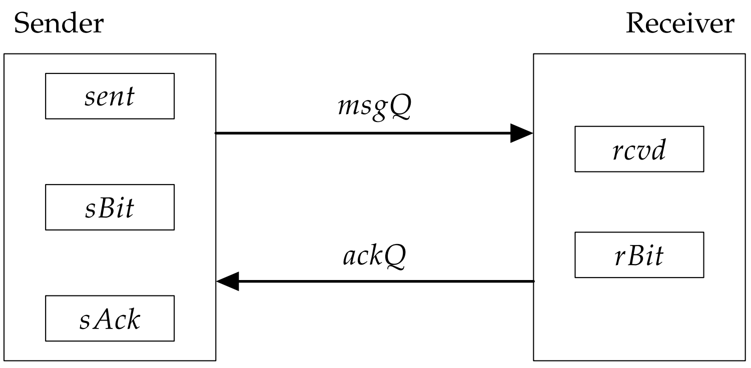


Figure 9.8: Representation of the alternating bit protocol (redrawn from [Lam02])

The protocol consists of two components, a sender and a receiver. The state of the sender consists of the variables *sent*, containing the data being transmitted, and the flags *sBit* and *sAck*, while that of the receiver consists of the variables *rcvd* containing the just received data and the *rBit* flag. The two components are connected by two lossy FIFO message queues with a fixed maximum size: the sender transmits data and additional control information on *msgQ* and the receiver sends acknowledgements on *ackQ*.

Transmission can start when the values of *sBit* and *sAck* are both the same. The sender sets the value of *data* to the value it wants to send, and complements *sBit*. Then it starts pushing packages consisting of *(data, sBit)* into *msgQ*. Any of these packets could be lost, we can only make sure that the receiver eventually receives one of the packets by the assumption of strong fairness on the receiving action. When a package arrives with a different control bit than the one stored in *rBit*, the receiver sets the value of *rcvd* to the received data and *rBit* to the value of the control bit (*sBit*). It then starts sending the value of *rBit* as acknowledgement on the *ackQ* queue. The same assumption of strong fairness must also be made for this queue. Whenever the sender receives an acknowledgement, it sets the value of *sAck* to the acknowledgement bit.

The RISCAL source of this benchmark is given in Appendix A as Algorithm 15, the TLA$^+$ source as Algorithm 16.

The liveness property checked for this benchmark is "every data sent is eventually received", which is formulated in RISCAL as

```
ltl[fairness] ∀m: Msg. □⟦sent = m ∧ sbit ≠ sack⟧ ⇒ ◊⟦rcvd = m⟧;
```

The pre-processing of this formula and the creation of the automaton took approximately 100 microseconds.

As mentioned before, this depends on the strong fairness of the data and acknowledgement receiving actions, thus it has to be prefixed with `[fairness]`. The formula is transformed into an automaton with 10 states. By varying the maximum length of the queue we can benchmark RISCAL and TLC for a large spectrum of system sizes. For a given channel size $M$ the RISCAL model consists of exactly $6M^3 + 20M^2 + 30M + 16$ system states, while the slightly different TLA$^+$ system consists of $6M^3 + 28M^2 + 34M + 12$ states.

RISCAL VERSUS TLC Fig. 9.9 compares the time needed for the verification of the system using RISCAL and TLC for varying maximal channel sizes. In this case RISCAL performs better than TLC using a single worker thread. But apparently the verification of this system is highly parallelizable as using 8 threads improves the speed approximately by a factor of 6 for large enough systems. This example shows the importance of implementing a parallel model checker.

As mentioned before, the number of system states depends on the cube of the channel size, but the total time needed in RISCAL increases noticeably faster than the cube of $M$. Thus the time needed per state is increasing as the systems get larger. Fig. 9.10 shows how much time is needed in RISCAL to verify a single system state: it increases from approximately 0.08 ms for 840 system states to 0.40 ms for 801516 states.

This large increase suggests that the storage/retrieval of the states (i.e. the currently used hash map) could also be improved by various techniques (e.g. choosing a hash function better suited for this application or a different load factor to avoid collisions). Further experimentation is needed to determine whether these could indeed help.

## 9.3 BENCHMARK 3: PETERSON'S MUTUAL EXCLUSION ALGORITHM

THE MODEL Peterson's solution for the mutual exclusion problem was first formulated by Gary L. Peterson in 1981 for two processes [Pet81], and it was later generalized to apply to an arbitrary number of processes. The RISCAL source of this benchmark is given in Appendix A as Algorithm 17, the TLA$^+$ source as Algorithm 18.

The $N$ processes work on two integer arrays, in the code called *room*, of size $N$ and *last*, of size $N - 1$. The elements of *room* take up values up to $N - 1$, each representing a

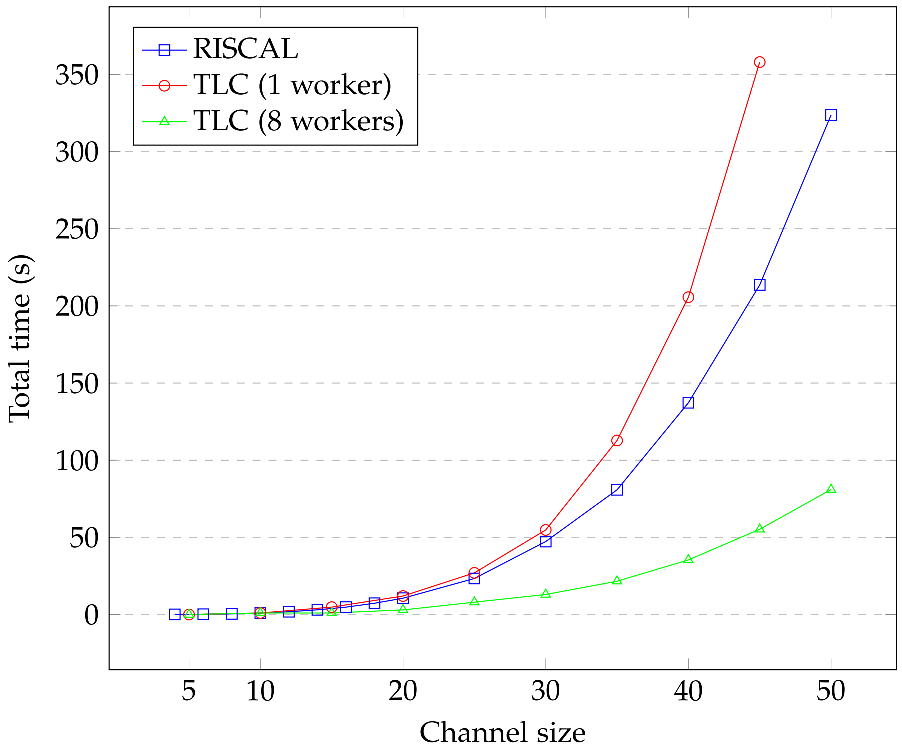


Figure 9.9: Time measurements for the Alternating bit protocol for varying channel sizes

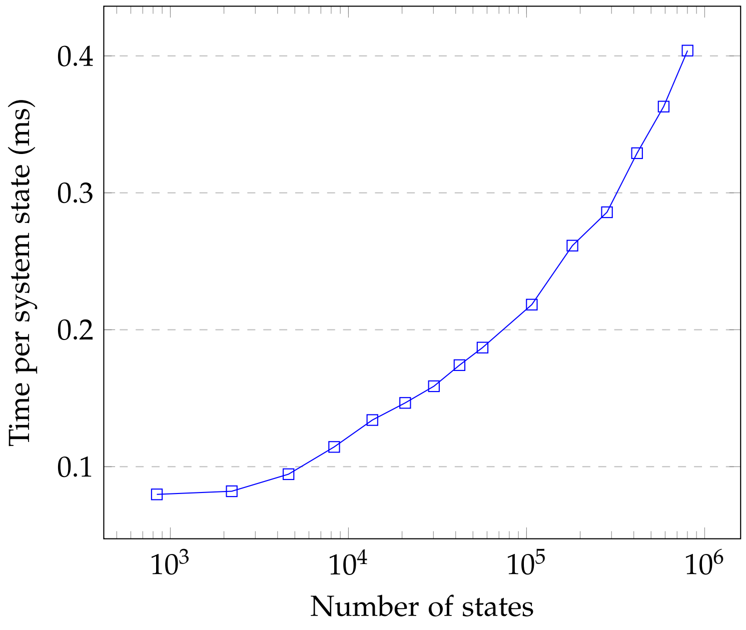


Figure 9.10: Time needed per system state for the Alternating bit protocol in RISCAL

separate "waiting room", while the value of *last* at a position i determines which process entered the given waiting room the last. The processes are initially at level -1, they enter the waiting queue by setting the level to 0 and from then on they are moved to a higher waiting room according certain scheduling rules, until they reach room $N-1$, which is the critical section. They exit the critical section by setting their room number back to -1.

For $N=2$ both models have 280 states, for $N=3$ they have 38069 and for $N=4$ already 8672068 states.

To keep track which process is in which state (ready, waiting, in the critical section etc.), the implementations use a process counter variable *pc*. For us it is important that the value of *pc* is 2 when they start waiting to enter the critical section, and 6 when they have entered it.

The two properties checked for this benchmark are:

1. no two processes are in the critical section at the same time, described in RISCAL as:

   ```
   ltl □ ¬∃i1: ProcN, i2: ProcN with i1 ≠ i2. ⟦ pc[i1] = 6 ∧pc[i2] = 6 ⟧;
   ```

   which results in an automaton with 4 states for 2 processes and 8 states for 3 processes.

2. every process which starts waiting to enter the critical section eventually enters it, described in RISCAL as:

   ```
   ltl[fairness] ∀i: ProcN. □(⟦ pc[i] = 2 ⟧ ⇒◊ ⟦ pc[i] = 6 ⟧);
   ```

   which results in an automaton with 10 states for 2 processes and 15 states for 3 processes.

In all cases the pre-processing of the formula and the creation of the automaton took less than 100 microseconds.

RISCAL VERSUS TLC Table 9.2 shows the time measurements for this algorithms. As the number of system states grows very quickly, it was not possible to verify any of the properties for $N = 5$. TLC was able to verify the safety property for $N = 4$ relatively quickly, while RISCAL ran out of memory. The reverse happened for $N = 3$ for the liveness property: RISCAL succeeded very quickly, while TLC ran out of memory (both had access to 25GB of RAM). For the remaining three cases, where both succeeded, RISCAL outperformed TLC by a wide margin.

| | RISCAL | | | | TLC (8 workers) | |
|---|---|---|---|---|---|---|
| | Invariant (1 worker) | Invariant (8 workers) | Safety | Liveness | Safety | Liveness |
| N = 2 | 0.01 | 0.03 | 0.02 | 0.02 | 1 | 5.75 |
| N = 3 | 0.74 | 0.33 | 0.83 | 1.99 | 7.5 | N/A |
| N = 4 | 365.8 | 204.7 | N/A | N/A | 36.5 | N/A |

Table 9.2: Time measurements for Peterson's algorithm in RISCAL and TLC

Based on this measurement it seems like RISCAL's fairness checker is better than the one implemented in TLC but it struggles with storing and processing large systems even when it comes to simple formulas.

## 9.4 BENCHMARK 4: DISTRIBUTED RESOURCE ALLOCATOR

THE MODEL The next benchmark is the model of a distributed system in which $C$ clients are accessing $R$ different resources. The RISCAL source of this benchmark is given in Appendix A as Algorithm 19, the TLA$^+$ source as Algorithm 20.

The clients randomly request a specific resource from the server by adding a new message to the *network* set. The server regularly takes a message from the set and if the given resource is not yet allocated, it gives it to the client that requested it. At some point the clients give back the resources they have been using after which they can be allocated to another client.

In this benchmark we will check four properties:

- the safety property: no two clients may hold the same resource at the same time, described in RISCAL as:

  ```
  ltl ∀c1:Client,c2:Client with c1 < c2.
      □ ⟦holding[c1] ∩ holding[c2] = ∅[Resource]⟧;
  ```

  which results in a 3 state automaton for two clients and a 13 state one for three.

- the liveness property: if a client has no open requests, eventually it won't hold any resources, described in RISCAL as:

  ```
  ltl[fairness] ∀c:Client. □ (⟦ requests[c] = ∅[Resource] ⟧ ⇒
      ◊⟦ holding[c] = ∅[Resource] ⟧);
  ```

  which results in a 10 state automaton for two clients and a 15 state one for three.

- the liveness property: every client that requests a resource will eventually hold it, described in RISCAL as:

  ```
  ltl[fairness] ∀c:Client,r:Resource. □ (⟦ r ∈ requests[c] ⟧ ⇒
      ◊⟦ r ∈ holding[c] ⟧);
  ```

  which results in automata with sizes between 20 (for two clients and two resources) and 45 (for three clients and three resources).

- the liveness property: every client infinitely often has no open requests, described in RISCAL as:

  ```
  ltl[fairness] ∀c:Client. □◊ ⟦ requests[c] = ∅[Resource] ⟧;
  ```

  which results in a 10 state automaton for 2 clients and a 15 state one for three.

RISCAL VERSUS TLC Table 9.3 shows how many states the model systems consist of for certain values of $C$ and $R$ and the time measurements for the safety property. In this benchmark TLC has consistently outperformed RISCAL: it was able to handle two larger systems that RISCAL could not, and with just a single worker thread it was 5-10 times faster for the larger systems where both succeeded. But again, due to the "Hash Compact" representation of the automaton states in TLC, this is not an entirely fair comparison.

Table 9.4, Table 9.5 and Table 9.6 show the results of the measurements for the liveness properties. Here RISCAL held up much better: for the first two properties it was much better than TLC with one worker thread and it wasn't significantly slower for the third one either. But with 8 workers RISCAL was no match for TLC: these tables again show the importance of a multi-threaded model checker.

| | C = 2, R = 2 | C = 2, R = 3 | C = 2, R = 4 | C = 2, R = 5 |
|---|---|---|---|---|
| System size | 704 | 13426 | 260264 | 5310482 |
| RISCAL LTL | < 0.1 | 1.4 | 47.6 | N/A |
| RISCAL invariant (1 worker) | < 0.1 | 1.0 | 35.0 | N/A |
| RISCAL invariant (8 workers) | < 0.1 | 0.4 | 10.8 | N/A |
| TLC (1 worker) | < 1 | < 1 | 9 | 285 |
| TLC (8 workers) | < 1 | < 1 | 2 | 57.3 |

| | C = 3, R = 2 | C = 3, R = 3 | C = 3, R = 4 |
|---|---|---|---|
| System size | 10204 | 529004 | 26186056 |
| RISCAL LTL | 1.4 | 139.5 | N/A |
| RISCAL invariant (1 worker) | 0.7 | 59.4 | N/A |
| RISCAL invariant (8 workers) | 0.3 | 18.5 | N/A |
| TLC (1 worker) | < 1 | 14.8 | N/A |
| TLC (8 workers) | < 1 | 4 | 217.3 |

Table 9.3: Measurements for the safety property (times in seconds)

| | C = 2, R = 2 | C = 2, R = 3 | C = 2, R = 4 | C = 3, R = 2 | C = 3, R = 3 |
|---|---|---|---|---|---|
| RISCAL | < 0.1 | 1.7 | 61.2 | 1.5 | 140.9 |
| TLC (1 worker) | < 1 | 5.8 | 194 | 5 | 501 |
| TLC (8 workers) | < 1 | 1 | 32.2 | 1 | 77.6 |

Table 9.4: Measurements for the first liveness property (times in seconds)

| | C = 2, R = 2 | C = 2, R = 3 | C = 2, R = 4 | C = 3, R = 2 | C = 3, R = 3 |
|---|---|---|---|---|---|
| RISCAL | < 0.1 | 4.9 | 230.9 | 3.5 | 550.8 |
| TLC (1 worker) | < 1 | 15 | 873 | 9.8 | 1659 |
| TLC (8 workers) | < 1 | 3 | 122 | 2 | 270.6 |

Table 9.5: Measurements for the second liveness property (times in seconds)

| | C = 2, R = 2 | C = 2, R = 3 | C = 2, R = 4 | C = 3, R = 2 | C = 3, R = 3 |
|---|---|---|---|---|---|
| RISCAL | 0.067 | 2.627 | 102.275 | 2.518 | 274.878 |
| TLC (1 worker) | < 1 | 3 | 101.8 | 2 | 179.6 |
| TLC (8 workers) | < 1 | 1 | 18.4 | 1 | 34.6 |

Table 9.6: Measurements for the third liveness property (times in seconds)

## 9.5 BENCHMARK 5: LAMPORT'S BAKERY ALGORITHM

THE MODEL Lamport's bakery algorithm is another solution to the mutual exclusion problem [Lam74]. The RISCAL source of this benchmark is given in Appendix A as Algorithm 21, the TLA$^+$ source as Algorithm 22.

The algorithm is based upon a process commonly used in bakeries, wherein a customer receives a number when entering the store. The holder of the lowest number is the next one served. In this algorithm each process has an identifier from 1 to $N$, and each chooses its own number when starting the wait for entering the critical section. If multiple processes happen to choose the same number, the one with the lowest identifier goes first.

In the implementations the *pc* array stores in which state any given process is at the moment. Its value is 0 before the process requests access to the critical section and 4 when it is in the critical section.

In this benchmark we are interested in three properties:

- the safety property: no two processes are in the critical section at the same time, expressed in RISCAL either as:

  ```
  ltl □ ⟦ ∀i1:Process,i2:Process with i1 ≠ i2. ¬(pc[i1] = 4 ∧ pc[i2] = 4) ⟧;
  ```

  which results in an automaton with 3 states, or as:

  ```
  ltl ∀i1:Process,i2:Process with i1 ≠ i2. □ ⟦¬(pc[i1] = 4 ∧pc[i2] = 4) ⟧;
  ```

  with the quantification at the LTL level, which results in an automaton with 9 states for two processes and 25 states for three processes.

- the liveness property: every process infinitely often exits the critical section, expressed in RISCAL as:

  ```
  ltl[fairness] ∀i:Process. □◊ ⟦ pc[i] = 0 ⟧;
  ```

  which results in an automaton with 15 states for three processes.

- the liveness property: there is a process which infinitely often enters the critical section, expressed in RISCAL as:

  ```
  ltl[fairness] ∃i:Process. □◊ ⟦ pc[i] = 4 ⟧;
  ```

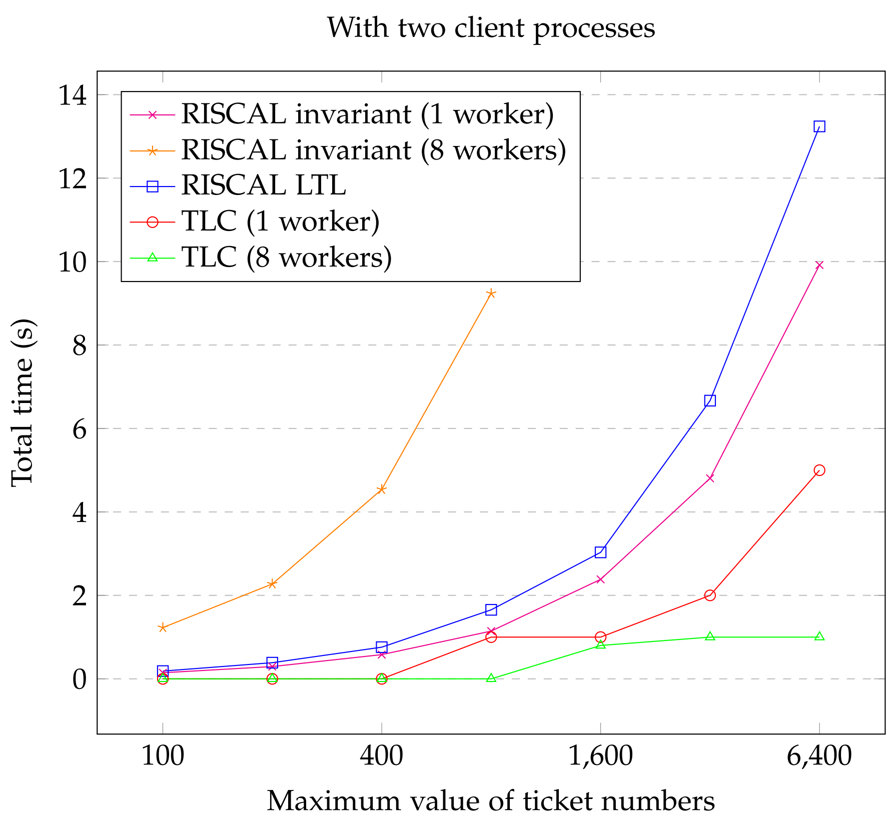
  which results in an automaton with 35 states for three processes.

For all these properties the pre-processing and the automaton creation took less than 100 microseconds.

CHECKING SAFETY Fig. 9.11 compares the runtime for the safety property between RISCAL and TLC for two and three processes for varying maximal ticket numbers. As it has often been the case before, TLC outperforms RISCAL by a wide margin for the safety property, even with just a single worker thread.

In the second formulation of the safery property the quantification over all processes is done at the level of LTL formulas instead of state predicates. In this case the quantifier is expanded to a conjunction of three subformulas, which means a larger automaton will be constructed. Thus whenever possible, it is recommended to move all possible operators and quantifiers to within the semantic brackets instead of having them at the LTL level.

Fig. 9.12 compares the performance between RISCAL and TLC for this formulation for three processes. Interestingly in this case TLC performs significantly worse than RISCAL, even with multi-threading enabled.



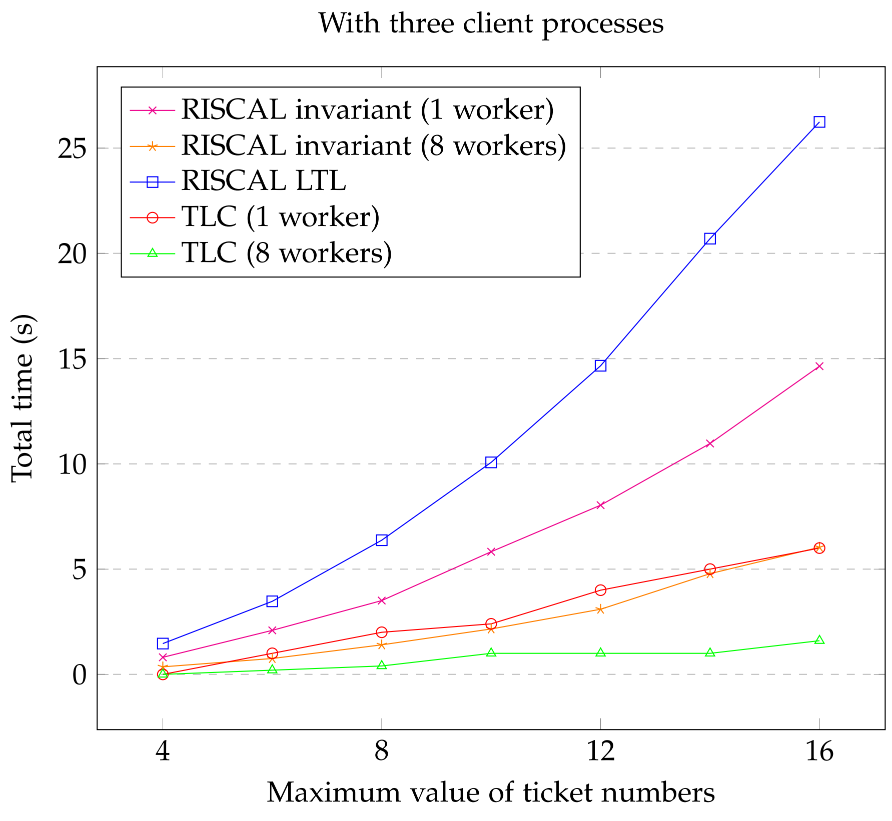


Figure 9.11: Time measurements for the safety property of Lamport's bakery algorithm

CHECKING LIVENESS Fig. 9.13 compares the performance for the first liveness property. In this case RISCAL again outperforms TLC, by about three times. TLC behaves strangely

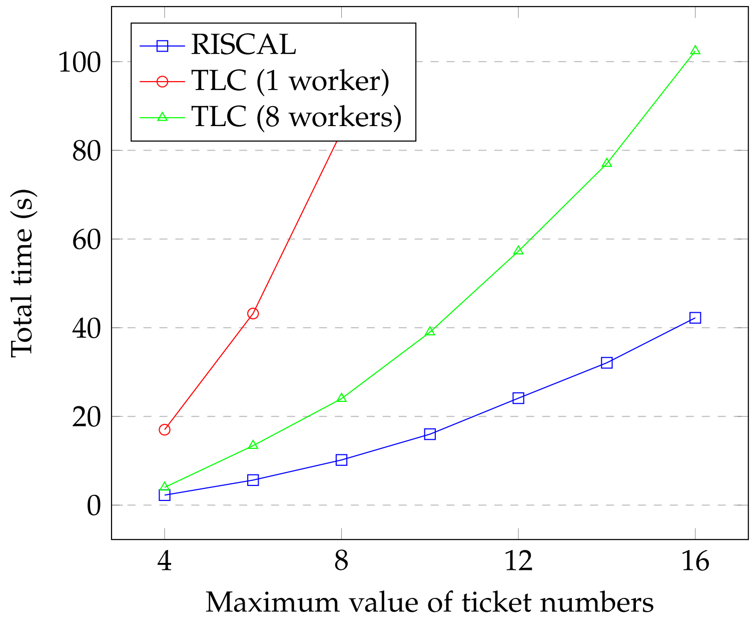


Figure 9.12: Time measurements for the safety property with quantification at the LTL level (with three client processes)

with a 8 worker threads: until $M = 10$ it is faster than using it in the single-threaded mode, but afterwards the use of multiple threads results in a slowdown.

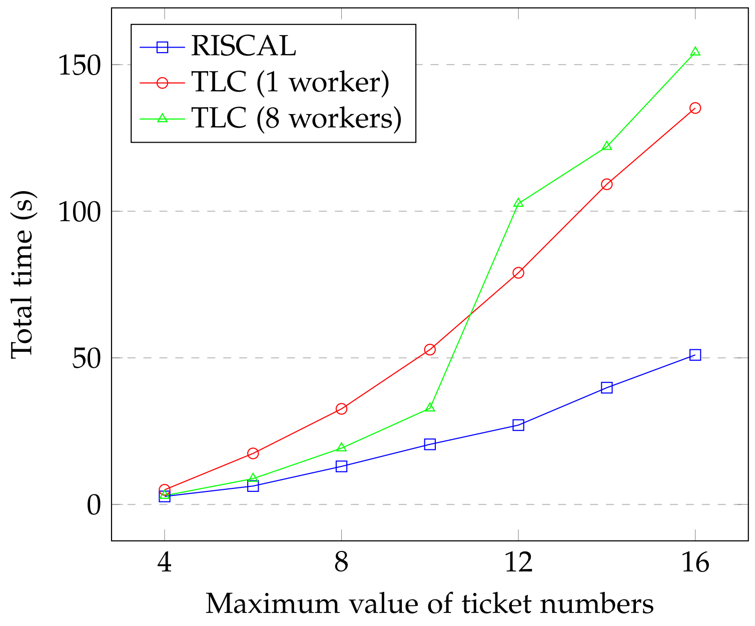


Figure 9.13: Time measurements for the first liveness property (with three client processes)

Finally, Fig. 9.14 compares the total time needed for verifying the second liveness property, one with an existential quantifier at the root. Here TLC is much faster than RISCAL.

This second liveness formula results in an automaton with 35 states, compared to the 15 state automaton for the previous one. Apparently existential quantification (or equivalently disjunction) at the top level is very costly when the full formula is transformed to an automaton. However we could split a formula in which the top level operator is a

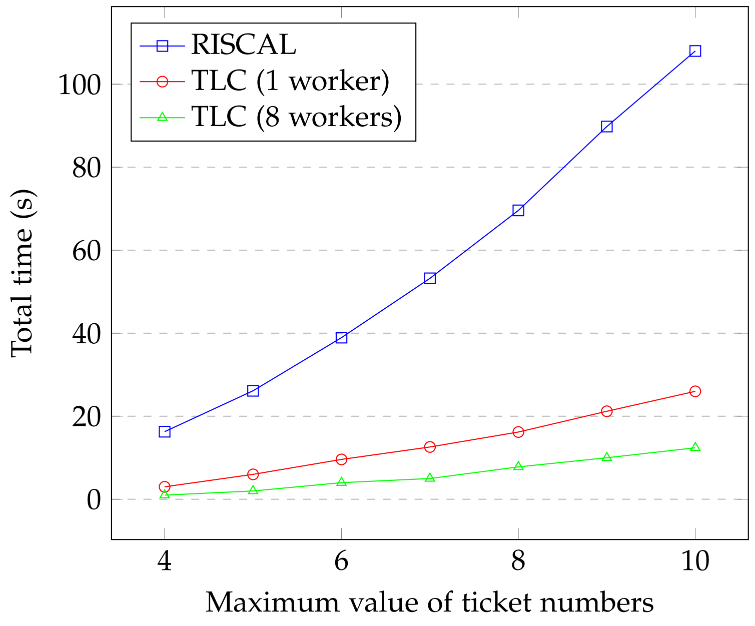


Figure 9.14: Time measurements for the second liveness property (with three client processes)

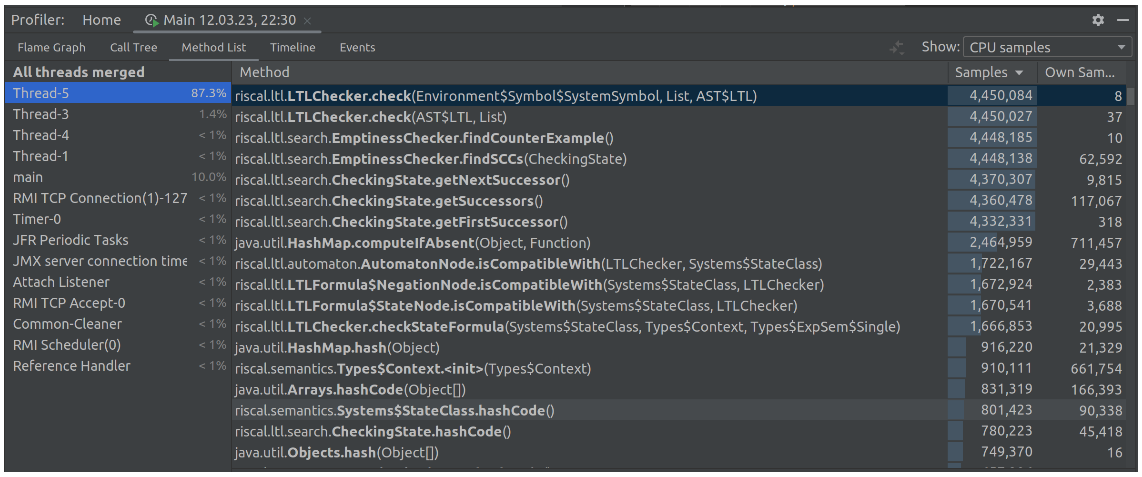
disjunction into a set of subformulas and check them one-by-one. If at least one of them succeeds, the property is satisfied. This would probably yield a much faster algorithm for the decision problem, but it does not provide an easy way for determining a violating execution. However if our goal is only verification, this could be a valuable optimization technique.

## 9.6 PROFILING THE MODEL CHECKER

In the previous sections we have looked at how RISCAL performs compared to TLC and made some observations about improving the algorithm on a higher level (e.g. the importance of a distributed model checker or tricks related to formulas with a conjunction at the root). The goal of this section is to inspect where is the majority of the model checking time spent and based on this suggest improvements to the implementation. To do so we will use the *async-profiler*[1] tool to profile the execution of the model checker at a rate of one sample per 10 microseconds.

First we will profile the model checking of the second liveness property for Lamport's bakery algorithm for $N = 3$ processes and $M = 5$ as the maximal ticket number. Fig. 9.15 shows how the output of the the profiler looks like in IntelliJ IDEA. The column *Samples* denotes how many samples were collected in the given method and in all the methods called from it, while *Own samples* lists only the samples that were collected during the execution of the method. A high value for *Own samples* usually means that this method did a lot of computations in a loop, while a high value for *Samples* can have a lot of causes: there might be redundant calls from it that can be optimized, but it can simply be the entry point of a large part of the system, in which case there is nothing to improve.

1 https://github.com/async-profiler/async-profiler


Profiler: Home Main 12.03.23, 22:30

Flame Graph Call Tree Method List Timeline Events Show: CPU samples

| All threads merged | |
|---|---|
| Thread-5 | 87.3% |
| Thread-3 | 1.4% |
| Thread-4 | < 1% |
| Thread-1 | < 1% |
| main | 10.0% |
| RMI TCP Connection(1)-127 | < 1% |
| Timer-0 | < 1% |
| JFR Periodic Tasks | < 1% |
| JMX server connection time | < 1% |
| Attach Listener | < 1% |
| RMI TCP Accept-0 | < 1% |
| Common-Cleaner | < 1% |
| RMI Scheduler(0) | < 1% |
| Reference Handler | < 1% |

| Method | Samples | Own Sam... |
|---|---|---|
| riscal.ltl.**LTLChecker.check**(Environment$Symbol$SystemSymbol, List, AST$LTL) | 4,450,084 | 8 |
| riscal.ltl.**LTLChecker.check**(AST$LTL, List) | 4,450,027 | 37 |
| riscal.ltl.search.**EmptinessChecker.findCounterExample**() | 4,448,185 | 10 |
| riscal.ltl.search.**EmptinessChecker.findSCCs**(CheckingState) | 4,448,138 | 62,592 |
| riscal.ltl.search.**CheckingState.getNextSuccessor**() | 4,370,307 | 9,815 |
| riscal.ltl.search.**CheckingState.getSuccessors**() | 4,360,478 | 117,067 |
| riscal.ltl.search.**CheckingState.getFirstSuccessor**() | 4,332,331 | 318 |
| java.util.**HashMap.computeIfAbsent**(Object, Function) | 2,464,959 | 711,457 |
| riscal.ltl.automaton.**AutomatonNode.isCompatibleWith**(LTLChecker, Systems$StateClass) | 1,722,167 | 29,443 |
| riscal.ltl.**LTLFormula$NegationNode.isCompatibleWith**(Systems$StateClass, LTLChecker) | 1,672,924 | 2,383 |
| riscal.ltl.**LTLFormula$StateNode.isCompatibleWith**(Systems$StateClass, LTLChecker) | 1,670,541 | 3,688 |
| riscal.ltl.**LTLChecker.checkStateFormula**(Systems$StateClass, Types$Context, Types$ExpSem$Single) | 1,666,853 | 20,995 |
| java.util.**HashMap.hash**(Object) | 916,220 | 21,329 |
| riscal.semantics.**Types$Context.<init>**(Types$Context) | 910,111 | 661,754 |
| java.util.**Arrays.hashCode**(Object[]) | 831,319 | 166,393 |
| riscal.semantics.**Systems$StateClass.hashCode**() | 801,423 | 90,338 |
| riscal.ltl.search.**CheckingState.hashCode**() | 780,223 | 45,418 |
| java.util.**Objects.hash**(Object[]) | 749,370 | 16 |

Figure 9.15: The result of profiling the RISCAL model checker for Lamport's bakery algorithm

From the figure it is immediately obvious that a large majority ($\approx 97\%$) of the model checking time was spent in expanding the product automaton, and not in the subsequent SCC search or fairness checking. The method *findSCCs* only has 62 thousand own samples, while the method *findFairSCC*, the entry point of the fairness checking only has about 5 thousand total samples. Thus we will focus our attention on this part of the model checker.

The expansion of the product automaton consists of three main methods:

- `riscal.ltl.LTLChecker.getSuccessors`, the method for generating the successors for a system state, responsible for about 7.6% of the total time.
- `riscal.ltl.automaton.AutomatonNode.isCompatibleWith`, the method for checking whether a given system state is compatible with an automaton state, by checking if it satisfies certain properties, responsible for about 38.7% of the total time.
- `java.util.HashMap.computeIfAbsent`, the method for checking if a newly computed node in the product automaton has been encountered before, and if not, storing it, responsible for about 47.9% of the total time.

The percentages above vary, with a small automaton more time is spent in `getSuccessors`. In this case a property with a relatively large automaton was chosen.

The method `getSuccessors` depends heavily on the existing system implementation of RISCAL to provide the successor states. The system states are already cached independently of the product automaton states, and based on this data, this is working well, as only a small amount of time is spent here. Further improvements would need significant changes in RISCAL, e.g. in the implementation of the `Seq` class, thus they are out of scope for the current thesis.

The method `isCompatibleWith` spends most of its time checking whether a state formula (i.e. an LTL subformula within semantic brackets) holds for a given state, using the method `checkStateFormula`. `checkStateFormula` works with so called "context objects" which are essentially maps between numerical indices and values of variables. The problem here is that a formula also comes with its own context (which contains, for example, the values of the constants) and this method has to create a new context which combines the contexts of the formula and the state. These operations are very costly and together they amount to 30% of the total model checking time. This could be eliminated by "inlining" the values in the formula context into the formula itself, turning *state.val == N, where N = 5* into *state.val == 5*. However to do this one must consider not only global constants, but also variables that are quantified on the LTL level (for example by using `let` or `forall`) and not only for numerical constants, but for values of an arbitrary type.

The method responsible for most of the model checking time, `computeIfAbsent`, cannot be easily changed, since it is part of the Java HashMap implementation. When looking at where it spends most of its time, we see that it has the most *own samples* out of all the methods called during the model checking. In addition to that, it spends a lot of time computing hashes of automaton nodes and comparing automaton nodes for equality. These together suggest that a lot of time is spent handling hash collisions (when no hash collision occurs, there is no need for equality checks). Thus the best way to improve this method would be to try and find a hash function which results in fewer collisions.

To show how little time is spent in fairness checking, Table 9.7 shows, for the first progress property of the distributed allocator benchmark, how many samples were collected during the execution of the *findCounterExample* method, which is the entry point of the search part of the model checking algorithm, versus how many were collected during the execution of *findFairScc*, which is the entry point for the fairness checking (i.e. the implementation of Algorithm 5.4), and the ratio of the two numbers in the last row.

| | C = 2, R = 2 | C = 2, R = 3 | C = 2, R = 4 | C = 3, R = 2 | C = 3, R = 3 |
|---|---|---|---|---|---|
| findCounterExample | 7618 | 242190 | 8267794 | 188090 | 18768741 |
| findFairScc | 58 | 586 | 8261 | 671 | 22005 |
| | 0.76% | 0.24% | 0.10% | 0.36% | 0.12% |

Table 9.7: Samples collected during the execution of the *findCounterExample* and *findFairScc* methods

Based on the findings in this chapter, the next chapter will conclude the thesis by a short analysis of the achieved results and potential further improvements.

# 10 CONCLUSIONS AND FUTURE WORK

In the first few chapters, this thesis gave a self-contained description of the theory of automaton-based explicit state model checkingm which should be sufficient to build such a software from scratch. During the research we were unable to find a single source with enough detail for this purpose.

The implementation of a model checker involved a lot of different choices and based on the benchmarks given in Chapter 9, we believe that it is possible to build a competitive model checker using the decisions we have made as a starting point. Already the current implementation performs better than TLC is particular examples, especially when it comes to models with many fairness constraints. It is even able to verify some properties where TLC completely fails.

Given the differences in the underlying technology (compiled vs interpreted system), it is unlikely that the RISCAL model checker could ever perform better than Spin, but that was never its goal. RISCAL provides a higher level modelling and specification language and a much better handling of fairness constraints, so it aims to be complementary to Spin, rather than a competitor.

The fact that RISCAL got so close to the performance of TLC, and even outperformed it in specific cases is very encouraging, considering that it uses a deterministic approach, as opposed to the probabilistic approach used by TLC. The RISCAL model checker, being comparatively recent, still lacks many potential optimizations, which could improve it even further. These potential improvements can be roughly categorized into three different groups: code level optimizations (the elimination of unnecessary computations), algorithmic improvements (recognizing certain patterns in the properties and systems that could reduce the number of product automaton states that need to be checked) and parallelization. The RISCAL model checker could also be extended with the option to use the *hash-compact* representation discussed in Section 2.3 for larger systems.

The improvements in the first group were discussed in Section 9.6. The most important would be to reduce the large overhead of the `CheckingStates` that can be observed when comparing the time it takes to check safety properties using LTL model checking compared to using invariants. It might be possible to significantly reduce the number of product automaton states that need to be constructed and the number of times the hash function needs to be called. The second most important optimization, which would eliminate a large amount of unnecessary computations would be to inline the contexts used when evaluating state formulas, but as discussed in the previous section this is not trivial. Furthermore, given that TLC performs better for safety properties than even the invariant checking in RISCAL, it might be necessary to make improvements to the storage and generation of system states if one wants to beat TLC also in those cases.

Some potential improvements in the second group were discussed during the benchmarking. The one most worth trying is the splitting of the formulas to be checked if its negation normal form has a conjunction at the root. This is essentially what is done when checking fairness, but the idea used to check $g \rightarrow f$ when $g$ is a fairness constraint could be extended to other conditions which concern the eventual behaviour of the system and not the finite prefix. A second algorithmic improvement is the so-called partial order reduction, which is a technique that can reduce the number of states to be checked by up to an order of magnitude by exploiting the commutativity of certain transitions. This technique is applied successfully in Spin and RISCAL would also certainly benefit from it.

Thirdly, as the experiments have shown both for TLC and RISCAL invariant checking, parallelization can provide a significant speedup. TLC sometimes got close to the theoretical maximum of an 8 times speedup compared to single-threaded model checking when using 8 worker threads. Replicating this in RISCAL would enable the users to check significantly larger systems in practice.

Implementing the hash-compact representation can be done without completely overhauling the system. We would still have to store the states on the search stack in their entirety (because we have yet to compute some of their successors), but we should only store the fingerprints of the states which have been fully processed. This could be achieved by removing the checking state cache and converting the `dfsnum` and `current` fields of the checking state objects to a map and a set respectively as it is currently done in the fairness checker. Furthermore, whenever we compute a new state we need to store its fingerprint in a `visited` set and when we pop states from the `actives` queue, we have to remove its entry from the `dfsnum` map. The condition currently checking whether `dfsnum` equals zero also has to be updated to check the `visited` set instead.

In conclusion, as part of this thesis we came up with a viable combination of algorithms to implement the automaton-based explicit state model checking approach. Using this strategy, the RISCAL software system was extended by a working model checker that can be used to verify certain systems beyond the capability of commonly used model checking software. Although lacking in certain regards, and in most cases slower than its main competitor TLC, the current implementation could serve as the basis for a competitive model checker, provided that certain further improvements are performed.

# BIBLIOGRAPHY

[Bie+21] Armin Biere et al. *Handbook of Satisfiability, Second edition*. NLD: IOS Press, 2021. ISBN: 978-1-64368-160-3.

[Bie+99] 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. URL: http://fmv.jku.at/papers/BiereCimattiClarkeZhu-TACAS99.pdf.

[Büc60] Julius Richard Büchi. 'On a decision method in restricted second order arithmetic'. In: *Proc. International Congress on Logic, Method, and Philosophy of Science, Stanford: Stanford University Press* (1960), pp. 1–11.

[Bur+90] J.R. Burch et al. 'Symbolic model checking: $10^{20}$ states and beyond'. In: *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*. 1990, pp. 428–439.

[Cla+18] Edmund M. Clarke et al. *Handbook of Model Checking*. Springer, Cham, 2018. ISBN: 978-3-319-10575-8.

[Coo71] Stephen A. Cook. 'The Complexity of Theorem-Proving Procedures'. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. ISBN: 9781450374644.

[Cou+92] Costas Courcoubetis et al. 'Memory-Efficient Algorithms for the Verification of Temporal Properties.' In: *Formal Methods in System Design* 1 (1992), pp. 275–288.

[Cou99] Jean-Michel Couvreur. 'On-the-Fly Verification of Linear Temporal Logic'. In: *FM99: Formal Methods, World Congress on Formal Methods in the Developement of Computing Systems, Toulouse, France, September 20-24, Volume I*. Ed. by Jeanette M. Wing, Jim Woodcock and Jim Davis. Vol. 1708. Lecture Notes in Computer Science (LNCS). Berlin, Germany: Springer, Sept. 1999, pp. 253–271. ISBN: 978-3-540-66587-8.

[CP03] 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.

[DGL16] Stéphane Demri, Valentin Goranko and Martin Lange. *Temporal Logics in Computer Science*. Classical Theory. Cambridge University Press, 2016. ISBN: 978-1-107-02836-4.

[Dij63] Edsger W. Dijkstra. *Over de sequentialiteit van procesbeschrijvingen (EWD-35)*. undated, 1962 or 1963. URL: https://www.cs.utexas.edu/users/EWD/translations/EWD35-English.html.

[Flo67] Robert W. Floyd. 'Assigning Meanings to Programs'. In: *Proceedings of Symposium on Applied Mathematics* 19 (1967), pp. 19–32. ISSN: 0160-7634.

[Ger+96] 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.

[GS09] Andreas Gaiser and Stefan Schwoon. *Comparison of Algorithms for Checking Emptiness on Buechi Automata*. 2009. URL: https://arxiv.org/abs/0910.3766.

[Hoa69] C. A. R. Hoare. 'An axiomatic basis for computer programming'. In: *Communications of the ACM* 12 (1969), pp. 576–580. ISSN: 0001-0782.

[Kar72] Richard Karp. 'Reducibility among combinatorial problems'. In: *Complexity of Computer Computations*. Ed. by R. Miller and J. Thatcher. Plenum Press, 1972, pp. 85–103.

[Kri63] Saul A. Kripke. 'Semantical Considerations on Modal Logic'. In: *Acta Philosophica Fennica* 16 (1963), pp. 83–94.

[Lam] Leslie Lamport. *The TLA+ Home Page*. URL: https://lamport.azurewebsites.net/tla/tla.html.

[Lam02] Leslie Lamport. *Specifying Systems*. The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, 2002. ISBN: 0-321-14306-X. URL: https://lamport.azurewebsites.net/tla/book.html.

[Lam74] Leslie Lamport. 'A New Solution of Dijkstra's Concurrent Programming Problem'. In: *Commun. ACM* 17.8 (Aug. 1974), pp. 453–455. ISSN: 0001-0782.

[LP85] 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.

[McM93] Kenneth L. McMillan. *Symbolic model checking*. Springer, Boston, 1993. ISBN: 978-0-7923-9380-1.

[MF71] A. R. Meyer and M. J. Fischer. ‘Economy of description by automata, grammars, and formal systems’. In: *12th Annual Symposium on Switching and Automata Theory*. 1971, pp. 188–191.

[MP95] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems*. Springer-Verlag New York, 1995. ISBN: 978-0-387-94459-3.

[Pet81] G.L. Peterson. ‘Myths about the mutual exclusion problem’. In: *Information Processing Letters* 12.3 (1981), pp. 115–116. ISSN: 0020-0190. URL: https://www.sciencedirect.com/science/article/pii/002001908190106X.

[Pnu77] Amir Pnueli. ‘The Temporal Logic of Programs’. In: *Proceedings of the 18th IEEE Annual Symposium on the Foundations of Computer Science*. IEEE Computer Society Press, Providence, 1977, pp. 46–57.

[Rei20] Franz-Xaver Reichl. *The Integration of SMT Solvers into the RISCAL Model Checker*. Master Thesis, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria. https://www.risc.jku.at/publications/download/risc_6103/Thesis.pdf. 2020.

[Sch21] Wolfgang Schreiner. *The RISC Algorithm Language (RISCAL)*. https://www.risc.jku.at/research/formal/software/RISCAL/manual/main.pdf. 2021.

[Sch23] Wolfgang Schreiner. *Concrete Abstractions. Formalizing and Analyzing Discrete Theories and Algorithms with the RISCAL Model Checker*. Springer Cham, 2023. ISBN: 978-3-031-24933-4. DOI: https://doi.org/10.1007/978-3-031-24934-1.

[SS22] Wolfgang Schreiner and Ágoston Sütő. ‘A temporal logic extension of the RISCAL model checker’. In: *Informatics 2022, IEEE 16th International Scientific Conference on Informatics*. Ed. by William Steingartner, Štefan Korečko and Anikó Szakál. Poprad, Slovakia, Nov. 2022. ISBN: 979-8-3503-1034-4. DOI: https://doi.org/10.1109/Informatics57926.2022.10083433. URL: https://informatics.kpi.fei.tuke.sk.

[Sti01] Colin Stirling. *Modal and Temporal Properties of Processes*. Springer Science, New York, 2001. ISBN: 0-387-98717-7.

[Tar72] Robert Tarjan. ‘Depth-First Search and Linear Graph Algorithms’. In: *SIAM Journal on Computing* 1.2 (1972), pp. 146–160.

[Tel00] Gerard Tel. *Introduction to distributed algorithms*. Cambridge: Cambridge University Press, 2000. ISBN: 0521794838.

[TJ19] Marcin Tkaczyk and Tomasz Jarmuek. ‘Jerzy Łoś Positional Calculus and the Origin of Temporal Logic’. In: *Logic and Logical Philosophy* 28 (2019), pp. 259–276. ISSN: 1425-3305. URL: https://apcz.umk.pl/LLP/article/view/LLP.2018.013/15481.

[Wol86] Pierre Wolper. ‘Expressing Interesting Properties of Programs in Propositional Temporal Logic.’ In: *Proc. 13th ACM Symp. on Principles of Programming Languages* (Jan. 1986), pp. 184–193.

# A BENCHMARK SOURCES

## A.1 SIMPLE COUNTER SYSTEM

```
val N: ℕ;
type int = ℤ[−1, N];

shared system Counter
{
  var x:int = 0;
  var y:int = 0;

  invariant 0 <= x ∧ x < N ∧ 0 ≤ y ∧ y < N;

  ltl [] ⟦ 0 <= x ∧ x < N ∧ 0 ≤ y ∧ y < N ⟧; // 1
  ltl [fairness] ([]<> ⟦ x = 0 ⟧); // 2
  ltl WeakFairness incX ⇒ ([]<> ⟦ x = 0 ⟧); // 3
  ltl [fairness] ([]<> ⟦ x = 0 ⟧) ∧ ([]<> ⟦ y = 0 ⟧); // 4
  ltl WeakFairness incX ∧ WeakFairness incY ⇒ // 5
    ([]<> ⟦ x = 0 ⟧) ∧ ([]<> ⟦ y = 0 ⟧);

  action incX()
  fairness weak;
  { x = if x < N−1 then x+1 else 0; }

  action incY()
  fairness weak;
  { y = if y < N−1 then y+1 else 0; }
}
```

Program 12: A simple counter algorithm in RISCAL

```
------------------------------ MODULE Counter ------------------------------
EXTENDS Naturals, TLC

CONSTANT N
VARIABLE x,y

(* the initial state condition *)
I == x = 0 /\ y = 0

X == /\ x' = IF x < N−1 THEN x+1 ELSE 0
     /\ y' = y
Y == /\ x' = x
     /\ y' = IF y < N−1 THEN y+1 ELSE 0
R == \/ X         (* increment x or y *)
     \/ Y

var == <<x,y>> (* the system variables *)

(* the whole specification *)
C == I /\ [][R]_var /\ WF_var(X) /\ WF_var(Y)
```

```
(* some properties *)
InRange == 0 <= x /\ x < N /\ 0 <= y /\ y < N
InRangeLTL == [] (0 <= x /\ x < N /\ 0 <= y /\ y < N)
Progress1 == []<> (x = 0)
Progress2 == []<> (x = 0) /\ []<> (y = 0)

==========================================================================
```

Program 13: A simple counter algorithm in TLA$^+$

```
#define B 9
#define N 300

unsigned x : B = 0;
unsigned y : B = 0;

active proctype incX()
{
  do
  ::  atomic { x < N-1 -> x = x+1; }
  ::  atomic { x == N-1 -> x = 0; };
  od;
}

active proctype incY()
{
  do
  :: atomic { y < N-1 -> y = y+1; }
  :: atomic { y == N-1 -> y = 0; }
  od;
}

ltl invariant { [](0 <= x && x < N && 0 <= y && y < N) }
ltl infinity1 { []<>(x == 0) }
ltl infinity2 { ([]<>(x == 0)) && ([]<>(y == 0)) }
```

Program 14: A simple counter algorithm in PROMELA

## A.2 ALTERNATING BIT PROTOCOL

```
type Msg = ℕ[1];
type Bit = Bool;

// arbitrary default message
val Default: Msg = 0;

// the capacity of the package queues
val M: ℕ;
type Package = Record[msg: Msg, bit: Bit];
type Queue = Record[len: ℕ[M], pack: Array[M,Package]];

// the result of sending into queue q a package with message m and bit b
fun add(q:Queue,m:Msg,b:Bit): Queue
  requires q.len < M;
= q with .len = q.len+1 with .pack = q.pack with [q.len] = msg:m,bit:b;

// the result of receiving a package from a queue
fun head(q:Queue):Package = q.pack[0];
proc tail(q:Queue):Queue
{
  var q0:Queue  q;
  q0.len  q0.len-1;
  for var i: ℕ[M] := 0; i < q0.len; i := i+1 do
  {
    q0.pack[i] := q0.pack[i+1];
  }
  // essential to limit size of state space!!!
  q0.pack[q0.len] := ⟨msg: Default,bit: ⊥⟩;
  return q0;
}

// the network version of the protocol
shared system AlternatingBitNetwork
{
  // the messages sent and received (local to each process)
  var sent: Msg = Default;
  var rcvd: Msg = Default;

  // the protocol bits
  var sbit: Bit = ⊥;
  var sack: Bit = ⊥;
  var rbit: Bit = ⊥;

  // the communication channels
  var msgq:Queue = len:0,pack:Array[M,Package](msg: Default,bit: ⊥);
  var ackq:Queue = len:0,pack:Array[M,Package](msg: Default,bit: ⊥);

  // depends on *strong* fairness options for receiving messages and acknowledgements
  // because the network may lose both
  ltl[fairness] ∀m:Msg. [] ⟦ sent = m ∧ sbit ≠ sack ⟧ ⇒<> ⟦rcvd = m ⟧;

  // sender choses a message and sends it
  action sender(m:Msg) with sack = sbit ∧ msgq.len < M;
  {
    sent := m; sbit := ¬sbit; msgq  add(msgq,sent,sbit);
  }

  // sender resends the last message sent (arbitrarily often)
  action resend() with sack ≠ sbit ∧ msgq.len < M;
  fairness weak;
  {
    msgq := add(msgq,sent,sbit);
  }

  // sender receives an acknowledgement
  action receiveAck() with ackq.len > 0;
  fairness strong; // no acknowledgement is permanently lost
  {
    sack := head(ackq).bit; ackq  tail(ackq);
  }

  // receiver receives a message
  action receiver() with msgq.len > 0;
  fairness strong; // no package is permanently lost
  {
    val p = head(msgq);
    msgq := tail(msgq);
    if p.bit ≠ rbit then
    {
      rcvd := p.msg; rbit := p.bit;
    }
  }

  // receiver acknowledges a message (arbitrarily often)
  action sendAck() with ackq.len < M;
  fairness weak;
  {
    ackq := add(ackq,Default,rbit);
  }

  // channels may lose messages and acknowledgemens
  action loseMsg() with msgq.len > 0; { msgq := tail(msgq); }
  action loseAck() with ackq.len > 0; { ackq := tail(ackq); }
}
```

Program 15: Alternating bit protocol in RISCAL

```
--------------------------- MODULE AlternatingBit ---------------------------
EXTENDS Naturals, Sequences
CONSTANTS Data, MaxLen
VARIABLES msgQ, ackQ, sBit, sAck, rBit, sent, rcvd
-----------------------------------------------------------------------------
ABInit == /\ msgQ = << >> /\ ackQ = << >>
          /\ sBit \in {0, 1} /\ sAck = sBit /\ rBit = sBit
          /\ sent \in Data /\ rcvd \in Data

SndNewValue(d) == /\ sAck = sBit /\ sent' = d /\ sBit' = 1 - sBit
                  /\ msgQ' = Append(msgQ, <<sBit', d>>)
                  /\ UNCHANGED <<ackQ, sAck, rBit, rcvd>>
ReSndMsg == /\ sAck # sBit
            /\ msgQ' = Append(msgQ, <<sBit, sent>>)
            /\ UNCHANGED <<ackQ, sBit, sAck, rBit, sent, rcvd>>
RcvMsg == /\ msgQ # <<>> /\ msgQ' = Tail(msgQ) /\ rBit' = Head(msgQ)[1]
          /\ rcvd' = Head(msgQ)[2]
          /\ UNCHANGED <<ackQ, sBit, sAck, sent>>
SndAck == /\ ackQ' = Append(ackQ, rBit)
          /\ UNCHANGED <<msgQ, sBit, sAck, rBit, sent, rcvd>>
RcvAck == /\ ackQ # << >> /\ ackQ' = Tail(ackQ) /\ sAck' = Head(ackQ)
          /\ UNCHANGED <<msgQ, sBit, rBit, sent, rcvd>>
Lose(q) == /\ q # << >>
           /\ \E i \in 1..Len(q) : q' =
                 [j \in 1..(Len(q)-1) |-> IF j < i THEN q[j] ELSE q[j+1] ]
           /\ UNCHANGED <<sBit, sAck, rBit, sent, rcvd>>
LoseMsg == Lose(msgQ) /\ UNCHANGED << ackQ, sBit, sAck, rBit, sent, rcvd>>
LoseAck == Lose(ackQ) /\ UNCHANGED << msgQ, sBit, sAck, rBit, sent, rcvd>>

ABNext == \/ (\E d \in Data : SndNewValue(d))
          \/ ReSndMsg \/ RcvMsg \/ SndAck \/ RcvAck \/ LoseMsg \/ LoseAck

abvars == << msgQ, ackQ, sBit, sAck, rBit, sent, rcvd>>
ABSpec == /\ ABInit /\ [][ABNext]_abvars
          /\ WF_abvars(ReSndMsg) /\ WF_abvars(SndAck)
          /\ SF_abvars(RcvMsg) /\ SF_abvars(RcvAck)
-----------------------------------------------------------------------------
Constraint == Len(msgQ) <= MaxLen /\ Len(ackQ) <= MaxLen
ABTypeInv == /\ msgQ \in Seq({0,1} \X Data) /\ ackQ \in Seq({0,1})
             /\ sBit \in {0, 1} /\ sAck \in {0, 1} /\ rBit \in {0, 1}
             /\ sent \in Data /\ rcvd \in Data
SentLeadsToRcvd == \A d \in Data : (sent = d) /\ (sBit # sAck) ~> (rcvd = d)
=============================================================================
```

Program 16: Alternating bit protocol in TLA$^+$

## A.3 PETERSON'S MUTUAL EXCLUSION ALGORITHM

```
val N: ℕ;
axiom minTwo ⇔ N ≥ 2;
```

```
type ProcN = ℕ[N−1];
type PCN = ℕ[6];

shared system PetersonN
{
  // the program counter of each process
  var pc: Array[N, PCN] = Array[N, PCN](0);

  // room −1..N−1 of each process
  var room: Array[N, ℤ[−1, N−1]] = Array[N, ℤ[−1, N−1]](−1);

  // process that was the last to enter each room 0..N−1
  var last: Array[N−1, ProcN] = Array[N−1, ProcN](0);

  // auxiliary variable for existence check loop
  var k: Array[N, ℕ[N]]   Array[N, ℕ[N]](0);

  // mutual exclusion property
  invariant ∃¬i1:ProcN,i2:ProcN with i1 ≠ i2. pc[i1] = 6 ∧ pc[i2] = 6;

  // mutual exclusion
  ltl ∃ ¬i1:ProcN,i2:ProcN with i1 ≠ i2. ⟦ pc[i1] = 6 ∧ pc[i2] = 6 ⟧;

  // progress
  ltl [fairness] ∀i:ProcN. [] ( ⟦ pc[i] = 2 ⟧ ⇒ <> ⟦ pc[i] = 6 ⟧);

  action start(i:ProcN) with pc[i] = 0;
  fairness weak_all;
  { room[i] = 0; pc[i] = 1; }

  action enter(i:ProcN) with pc[i] = 1 ∧ room[i] = N−1;
  fairness strong_all;
  { pc[i] = 6; }

  action loop(i:ProcN) with pc[i] = 1 ∧ room[i] < N−1;
  fairness strong_all;
  { pc[i] = 2; }

  action last(i:ProcN) with pc[i] = 2;
  fairness weak_all;
  { last[room[i]] = i; pc[i] = 3; }

  action endwait3(i:ProcN) with pc[i] = 3 ∧ last[room[i]] ≠ i;
  fairness strong_all;
  { pc[i] = 5; }

  action wait3(i:ProcN) with pc[i] = 3 ∧ last[room[i]] = i;
  fairness strong_all;
  { k[i] = 0; pc[i] = 4; }

  action endwait4(i:ProcN) with pc[i] = 4 ∧ k[i] = N;
  fairness strong_all;
  { pc[i] = 5; }

  action wait4(i:ProcN) with pc[i] = 4 ∧ k[i] < N ∧
    (k[i] ≠ i ∧ room[k[i]] ≥ room[i]);
  fairness strong_all;
  { pc[i] = 3; }

  action next(i:ProcN) with pc[i] = 4 ∧ k[i] < N ∧
    (k[i] = i ∨ room[k[i]] < room[i]);
  fairness strong_all;
  { k[i] = k[i]+1; }
  action room(i:ProcN) with pc[i] = 5;
  fairness weak_all;
  { room[i] = room[i]+1; pc[i] = 1; }

  action exit(i:ProcN) with pc[i] = 6;
  fairness weak_all;
  { room[i] = -1; pc[i] = 0; }
}
```

Program 17: Peterson's mutual exclusion algorithm in RISCAL

```
------------------------------ MODULE Peterson ------------------------------
EXTENDS Naturals, Integers

CONSTANT N
VARIABLE pc, room, last, k

(* the initial state condition *)
I == pc = [ i \in 0..N-1 |-> 0]
  /\ room = [ i \in 0..N-1 |-> -1 ]
  /\ last = [ i \in 0..N-2 |-> 0 ]
  /\ k = [ i \in 0..N-1 |-> 0]

Start(i) ==
   pc[i] = 0
/\ room' = [ room EXCEPT ![i] = 0 ]
/\ pc' = [ pc EXCEPT ![i] = 1 ]
/\ UNCHANGED <<last,k>>

Enter(i) ==
   pc[i] = 1 /\ room[i] = N-1
/\ pc' = [ pc EXCEPT ![i] = 6 ]
/\ UNCHANGED <<room,last,k>>

Loop(i) ==
   pc[i] = 1 /\ room[i] < N-1
/\ pc' = [ pc EXCEPT ![i] = 2 ]
/\ UNCHANGED <<room,last,k>>

Last(i) ==
   pc[i] = 2
/\ last' = [ last EXCEPT ![room[i]] = i ]
/\ pc' = [ pc EXCEPT ![i] = 3 ]
/\ UNCHANGED <<room,k>>

EndWait3(i) ==
   pc[i] = 3 /\ last[room[i]] # i
/\ pc' = [ pc EXCEPT ![i] = 5 ]
/\ UNCHANGED <<room,last,k>>

Wait3(i) ==
   pc[i] = 3 /\ last[room[i]] = i
/\ k' = [ k EXCEPT ![i] = 0 ]
/\ pc' = [ pc EXCEPT ![i] = 4 ]
/\ UNCHANGED <<room,last>>

EndWait4(i) ==
   pc[i] = 4 /\ k[i] = N
/\ pc' = [ pc EXCEPT ![i] = 5 ]
/\ UNCHANGED <<room,last,k>>

Wait4(i) ==
   pc[i] = 4 /\ k[i] < N /\ (k[i] # i /\ room[k[i]] >= room[i])
```

```
/\ pc' = [ pc EXCEPT ![i] = 3 ]
/\ UNCHANGED <<room,last,k>>

Next(i) ==
   pc[i] = 4 /\ k[i] < N /\ (k[i] = i \/ room[k[i]] < room[i])
/\ k' = [ k EXCEPT ![i] = k[i]+1 ]
/\ UNCHANGED <<pc,room,last>>

Room(i) ==
   pc[i] = 5
/\ room' = [ room EXCEPT ![i] = room[i]+1 ]
/\ pc' = [ pc EXCEPT ![i] = 1 ]
/\ UNCHANGED <<last,k>>

Exit(i) ==
   pc[i] = 6
/\ room' = [ room EXCEPT ![i] = -1 ]
/\ pc' = [ pc EXCEPT ![i] = 0 ]
/\ UNCHANGED <<last,k>>

Step(i) ==
   Start(i) \/ Enter(i) \/ Loop(i)
\/ Last(i) \/ EndWait3(i) \/ Wait3(i)
\/ EndWait4(i) \/ Wait4(i) \/ Next(i)
\/ Room(i) \/ Exit(i)

vars == <<pc,room,last,k>>

Fairness(i) ==
   WF_vars(Start(i)) /\ SF_vars(Enter(i)) /\ SF_vars(Loop(i))
/\ WF_vars(Last(i)) /\ SF_vars(EndWait3(i)) /\ SF_vars(Wait3(i))
/\ SF_vars(EndWait4(i)) /\ SF_vars(Wait4(i)) /\ SF_vars(Next(i))
/\ WF_vars(Room(i)) /\ WF_vars(Exit(i))

(* the whole specification *)
Peterson == I
  /\ [][\E i \in 0..N-1 : Step(i)]_vars
  /\ (\A i \in 0..N-1 : Fairness(i))

(* some properties *)
Mutex == ~(\E i1 \in 0..N-1, i2 \in 0..N-1 : i1 # i2 /\
               pc[i1] = 6 /\ pc[i2] = 6)
Progress == \A i \in 0..N-1 : ((pc[i] = 2) ~> (pc[i] = 6))


============================================================================
```

Program 18: Peterson's mutual exclusion algorithm in TLA$^+$

## A.4 DISTRIBUTED RESOURCE ALLOCATOR

```
// ----------------------------------------------------------------
// a server allocating resources to clients
// ----------------------------------------------------------------
val C: ℕ; // number of clients
val R: ℕ; // number of resources
axiom notNull ⇔ C > 0 ∧ R > 0;

type Client = ℕ[C-1];
type Resource = ℕ[R-1];
type Position = ℕ[C]; // 0=first ... C-1=last, C=none
```

```
// get position of request from client c in queue of requests
fun position(c:Client,pos:Map[Client,Position]):Position =
  let p = pos[c] in
  if p < C then
    p
  else if ∀c0:Client. pos[c0] = C then
    0
  else
    1 + max c0:Client with pos[c0] < C. pos[c0];

// remove client c from queue of requests
proc remove(pos:Map[Client,Position], c:Client):Map[Client,Position]
  ensures ∀c0:Client. result[c0] = if c0 = c then C else
    if pos[c] < pos[c0] ∧ pos[c0] < C then pos[c0]-1 else pos[c0];
{
  var p:Map[Client,Position] = pos;
  for c0:Client with p[c] < p[c0] ∧ p[c0] < C do
    p[c0]   p[c0]-1;
  p[c]   C;
  return p;
}

// server may grant to client c resource set S
pred grant(c:Client,S:Set[Resource],
    unsat:Map[Client,Set[Resource]],
    alloc:Map[Client,Set[Resource]],
    pos:Map[Client,Position])⇔
 pos[c] < C ∧ S ≠ [Resource] ∧
  (∀r∈S. r∈unsat[c] ∧¬∃c0:Client. r∈alloc[c0]) ∧
  (∀c0:Client with pos[c0]<pos[c]. ∀r∈S. r∉unsat[c0]);

// ------------------------------------------------------------
// a shared formulation of the system
// ------------------------------------------------------------

type Tag = ℕ[2];
val request = 0; val allocate = 1; val giveback = 2;
type Message = Record[tag:Tag,client:Client,resources:Set[Resource]];

shared system SharedAllocator
{
  // the local server state (controlled by server)
  // - unsatisfied requests of every client
  // - allocated resources of every client
  // - position of clients with unsatisfied requests (C, if none)
  var unsat:Map[Client,Set[Resource]] =
    Map[Client,Set[Resource]]([Resource]);
  var alloc:Map[Client,Set[Resource]] =
     Map[Client,Set[Resource]]([Resource]);
  var pos:Map[Client,Position] = Map[Client,Position](C);

  // the local client states (controlled by clients)
  var requests: Map[Client,Set[Resource]] =
    Map[Client,Set[Resource]]([Resource]);
  var holding: Map[Client,Set[Resource]] =
    Map[Client,Set[Resource]]([Resource]);

  // the network
  var network: Set[Message] = [Message];

  // the exclusive access property
  invariant ∀c1:Client,c2:Client with c1 < c2.
    holding[c1]   holding[c2] = [Resource];
```

```
ltl ∀c1:Client,c2:Client with c1 < c2. // 1
  [] ⟦holding[c1] ∩ holding[c2] = ∅[Resource]⟧;

ltl[fairness] ∀c:Client. [] (⟦ requests[c] = ∅[Resource] ⟧⇒ // 2
  <>⟦ holding[c] = [Resource] ⟧ );

ltl[fairness] ∀c:Client,r:Resource. [] ⟦( r ∈ requests[c] ⟧⇒ // 3
  <>⟦ r ∈ holding[c] ⟧);

ltl[fairness] ∀c:Client. []<> ⟦ requests[c] = ∅[Resource] ⟧; // 4

// server receives a request message from some client
action serverRequest(m:Message) with
  m ∈ network ∧m.tag = request ∧ alloc[m.client] = ∅[Resource];
fairness weak_all;
{
  network := network\{m}; val c = m.client; val S = m.resources;
  unsat[c] := unsat[c]∪S;
  pos[c] := position(c,pos);
}

// server decides to grant resource set S to client c
action serverAllocate(c:Client,S:Set[Resource]) with
  grant(c,S,unsat,alloc,pos);
fairness weak_all;
{
  alloc[c] := alloc[c]S;
  unsat[c] := unsat[c]\S;
  if unsat[c] = ∅[Resource] then pos := remove (pos,c);
  network := network{tag:allocate,client:c,resources:S};
}

// server receives a giveback message
action serverGiveBack(m:Message) with
  m ∈ network ∧m.tag = giveback;
fairness weak_all;
{
  network := network\{m}; val c = m.client; val S = m.resources;
  alloc[c] := alloc[c]\S;
}

// client c decides to ask for resource set S
action clientRequest(c:Client,S:Set[Resource]) with
  requests[c] = ∅[Resource] ∧holding[c] = ∅[Resource] ∧
  S ≠ ∅[Resource];
fairness weak_all;
{
  requests[c] := S;
  network := network{tag:request,client:c,resources:S};
}

// client c receives an allocation message m from server
action clientAllocate(m:Message) with
  m ∈ network ∧ m.tag = allocate;
fairness weak_all;
{
  network := network\{m}; val c = m.client; val S = m.resources;
  requests[c] := requests[c] \ S;
  holding[c] := holding[c] ∪ S;
}

// client c decides to give back resource set S
```

```
  action clientGiveBack(c:Client,S:Set[Resource]) with
    S ≠ ∅[Resource] ∧S ⊆ holding[c];
  fairness weak_all;
  {
    holding[c] := holding[c]\S;
    network := network{tag:giveback,client:c,resources:S};
  }
}
```

Program 19: Distributed allocator in RISCAL

```
---------------------- MODULE DistributedAllocator -------------------
EXTENDS Naturals, Sequences
CONSTANTS Clients, Resources
VARIABLES unsat, alloc, sched, requests, holding, network
----------------------------------------------------------------------
Messages == [type : {"request", "allocate", "return"},
          clt : Clients, rsrc : SUBSET Resources]
Drop(seq,i) == SubSeq(seq, 1, i-1) \circ SubSeq(seq, i+1, Len(seq))
available == Resources \ (UNION {alloc[c] : c \in Clients})
Range(f) == { f[x] : x \in DOMAIN f }
----------------------------------------------------------------------
Init ==
  /\ unsat = [c \in Clients |-> {}] /\ alloc = [c \in Clients |-> {}]
  /\ requests = [c \in Clients |-> {}] /\ holding = [c \in Clients |-> {}]
  /\ sched = << >> /\ network = {}

RReq(m) ==
  /\ m \in network /\ m.type = "request"
  /\ alloc[m.clt] = {}    \** don't handle request messages prematurely(!)
  /\ unsat' = [unsat EXCEPT ![m.clt] = m.rsrc]
  /\ network' = network \ {m}
  /\ sched' = (* IF m.clt \in Range(sched) THEN sched ELSE *) Append(sched, m.clt)
  /\ UNCHANGED <<alloc,requests,holding>>

RAlloc(m) ==
  /\ m \in network /\ m.type = "allocate"
  /\ holding' = [holding EXCEPT ![m.clt] = @ \cup m.rsrc]
  /\ requests' = [requests EXCEPT ![m.clt] = @ \ m.rsrc]
  /\ network' = network \ {m}
  /\ UNCHANGED <<unsat,alloc,sched>>

RRet(m) ==
  /\ m \in network /\ m.type = "return"
  /\ alloc' = [alloc EXCEPT ![m.clt] = @ \ m.rsrc]
  /\ network' = network \ {m}
  /\ UNCHANGED <<unsat,sched,requests,holding>>

Request(c,S) ==
  /\ requests[c] = {} /\ holding[c] = {}
  /\ S # {} /\ requests' = [requests EXCEPT ![c] = S]
  /\ network' = network \cup {[type |-> "request", clt |-> c, rsrc |-> S]}
  /\ UNCHANGED <<unsat,alloc,sched,holding>>

Allocate(c,S) ==
  /\ S # {} /\ S \subseteq available \cap unsat[c]
  /\ \E i \in DOMAIN sched :
        /\ sched[i] = c
        /\ \A j \in 1..i-1 : unsat[sched[j]] \cap S = {}
        /\ sched' = IF S = unsat[c] THEN Drop(sched,i) ELSE sched
  /\ alloc' = [alloc EXCEPT ![c] = @ \cup S]
  /\ unsat' = [unsat EXCEPT ![c] = @ \ S]
  /\ network' = network \cup {[type |-> "allocate", clt |-> c, rsrc |-> S]}
  /\ UNCHANGED <<requests,holding>>
```

```
Return(c,S) ==
  /\ S # {} /\ S \subseteq holding[c]
  /\ holding' = [holding EXCEPT ![c] = @ \ S]
  /\ network' = network \cup {[type |-> "return", clt |-> c, rsrc |-> S]}
  /\ UNCHANGED <<unsat,alloc,sched,requests>>
----------------------------------------------------------------------------
Next ==
  \/ \E m \in network : RReq(m) \/ RAlloc(m) \/ RRet(m)
  \/ \E c \in Clients, S \in SUBSET Resources :
        Request(c,S) \/ Allocate(c,S) \/ Return(c,S)

vars == <<unsat,alloc,sched,requests,holding,network>>

Liveness ==
  /\ \A c \in Clients : WF_vars(requests[c]={} /\ Return(c,holding[c]))
  /\ \A c \in Clients : WF_vars(\E S \in SUBSET Resources : Allocate(c, S))
  /\ \A m \in Messages : WF_vars(RReq(m)) /\ WF_vars(RAlloc(m)) /\ WF_vars(RRet(m))

Specification == Init /\ [][Next]_vars /\ Liveness
----------------------------------------------------------------------------
TypeInvariant ==
  /\ unsat \in [Clients -> SUBSET Resources]
  /\ alloc \in [Clients -> SUBSET Resources]
  /\ requests \in [Clients -> SUBSET Resources]
  /\ holding \in [Clients -> SUBSET Resources]
  /\ sched \in Seq(Clients) /\ network \in SUBSET Messages
ResourceMutex ==
  \A c1,c2 \in Clients : holding[c1] \cap holding[c2] # {} => c1 = c2
ClientsWillReturn ==
  \A c \in Clients: (requests[c]={} ~> holding[c]={})
ClientsWillObtain ==
  \A c \in Clients, r \in Resources : r \in requests[c] ~> r \in holding[c]
InfOftenSatisfied ==
  \A c \in Clients : []<>(requests[c] = {})
============================================================================
```

Program 20: Distributed allocator in TLA$^+$

## A.5 LAMPORT'S BAKERY ALGORITHM

```
// number of processes
val N: ℕ;
axiom processNumber ⇔ N ≥ 2;
type Process = ℕ[N-1];

// maximum value of ticket numbers
val M: ℕ;

// a finite-state variant of the the bakery algorithm
// (on overflow, processes abort attempts to enter critical region)
shared system Bakery
{
  var e:  Array[N, Bool] = Array[N, Bool](⊥);
  var n:  Array[N, ℕ[M]] = Array[N, ℕ[M]](0);
  var j:  Array[N, ℕ[N]] = Array[N, ℕ[N]](0);
  var m:  Array[N, ℕ[M]] = Array[N, ℕ[M]](0);
  var pc: Array[N, ℕ[4]] = Array[N, ℕ[4]](0);

  // crucial safety property, holds even without fairness
  invariant ∀i1:Process,i2:Process with i1 ≠ i2. ¬(pc[i1] = 4 ∧ pc[i2] = 4);
  // same property in LTL, slower than invariant
  ltl □ ⟦ ∀i1:Process,i2:Process with i1 ≠i2. ¬(pc[i1] = 4 ∧ pc[i2] = 4) ⟧;

  // same property with LTL quantification, slower than previous ltl property
  ltl ∀i1:Process,i2:Process with i1 ≠ i2. □ ⟦¬(pc[i1] = 4 ∧ pc[i2] = 4) ⟧;

  // requires everywhere weak_all and enter1:strong_all
  ltl[fairness] ∀i:Process. □◊ ⟦pc[i] = 0 ⟧;

  // requires everywhere weak_all and enter1:strong_all
  ltl[fairness] ∃i:Process. □◊ ⟦pc[i] = 4 ⟧;

  action start(i:Process) with pc[i] = 0;
  fairness weak_all;
  { e[i] := ⊤; pc[i] := 1; }

  action maximum0(i:Process) with pc[i] = 1 ∧ j[i] = N ∧ m[i] < M;
  fairness weak_all;
  { n[i] := 1+m[i]; j[i] := 0; e[i] := ⊥; pc[i] := 2; }

  action maximum0a(i:Process) with pc[i] = 1 ∧ j[i] = N ∧ m[i] = M;
  fairness weak_all;
  { e[i] := ⊥; pc[i] := 0; }

  action maximum1(i:Process) with pc[i] = 1 ∧ j[i] < N ∧ n[j[i]] ≤ m[i];
  fairness weak_all;
  { j[i] := j[i]+1; }

  action maximum2(i:Process) with pc[i] = 1 ∧ j[i] < N ∧ n[j[i]] > m[i];
  fairness weak_all;
  { m[i] := n[j[i]]; j[i] := j[i]+1; }

  action enter0(i:Process) with pc[i] = 2 ∧ j[i] = N;
  fairness weak_all;
  { j[i] := 0; pc[i] := 4; }

  action enter1(i:Process) with pc[i] = 2 ∧ j[i] < N ∧ ¬e[j[i]];
  fairness strong_all;
  { pc[i] := 3; }

  action enter2(i:Process) with pc[i] = 3 ∧
    let n0 = n[j[i]] in (n0 = 0 ∨ n0 > n[i] ∨ (n0 = n[i] ∧ j[i] ≥ i));
  fairness weak_all;
  { j[i] := j[i]+1; pc[i] := 2; }

  action exit(i:Process) with pc[i] = 4;
  fairness weak_all;
  { n[i] := 0; pc[i] := 0; }
}
```

Program 21: Lamport's bakery algorithm in RISCAL

```
------------------------------- MODULE Bakery -------------------------------
EXTENDS Naturals, Integers
CONSTANTS N, M
VARIABLES e, n, j, m, pc
-----------------------------------------------------------------------------
 BakeryInit ==     /\ e = [ i \in 0..N-1 |-> FALSE]
                   /\ n = [ i \in 0..N-1 |-> 0]
                   /\ j = [ i \in 0..N-1 |-> 0]
                   /\ m = [ i \in 0..N-1 |-> 0]
                   /\ pc = [ i \in 0..N-1 |-> 0]

 Start(i) ==       /\ pc[i] = 0
```

```
                    /\ e' = [e EXCEPT ![i] = TRUE]
                    /\ pc' = [pc EXCEPT ![i] = 1]
                    /\ UNCHANGED <<n, j, m>>

Maximum0(i) ==      /\ pc[i] = 1 /\ j[i] = N /\ m[i] < M
                    /\ n' = [n EXCEPT ![i] = 1 + m[i]]
                    /\ j' = [j EXCEPT ![i] = 0]
                    /\ e' = [e EXCEPT ![i] = FALSE]
                    /\ pc' = [pc EXCEPT ![i] = 2]
                    /\ UNCHANGED <<m>>

Maximum0a(i) ==     /\ pc[i] = 1 /\ j[i] = N /\ m[i] = M
                    /\ e' = [e EXCEPT ![i] = FALSE]
                    /\ pc' = [pc EXCEPT ![i] = 0]
                    /\ UNCHANGED <<n, j, m>>

Maximum1(i) ==      /\ pc[i] = 1 /\ j[i] < N /\ n[j[i]] <= m[i]
                    /\ j' = [j EXCEPT ![i] = j[i] + 1]
                    /\ UNCHANGED <<e, n, m, pc>>

Maximum2(i) ==      /\ pc[i] = 1 /\ j[i] < N /\ n[j[i]] > m[i]
                    /\ m' = [m EXCEPT ![i] = n[j[i]]]
                    /\ j' = [j EXCEPT ![i] = j[i] + 1]
                    /\ UNCHANGED <<e, n, pc>>

Enter0(i) ==        /\ pc[i] = 2 /\ j[i] = N
                    /\ j' = [j EXCEPT ![i] = 0]
                    /\ pc' = [pc EXCEPT ![i] = 4]
                    /\ UNCHANGED <<e, n, m>>

Enter1(i) ==        /\ pc[i] = 2 /\ j[i] < N /\ e[j[i]] = FALSE
                    /\ pc' = [pc EXCEPT ![i] = 3]
                    /\ UNCHANGED <<e, n, j, m>>

Enter2(i) ==        /\ pc[i] = 3 /\ (n[j[i]] = 0 \/ n[j[i]] > n[i]
                        \/ (n[j[i]] = n[i] /\ j[i] >= i))
                    /\ j' = [j EXCEPT ![i] = j[i] + 1]
                    /\ pc' = [pc EXCEPT ![i] = 2]
                    /\ UNCHANGED <<e, n, m>>

Exit(i) ==          /\ pc[i] = 4
                    /\ n' = [n EXCEPT ![i] = 0]
                    /\ pc' = [pc EXCEPT ![i] = 0]
                    /\ UNCHANGED <<e, j, m>>

Step(i) ==          \/ Start(i) \/ Maximum0(i) \/ Maximum0a(i)
                    \/ Maximum1(i) \/ Maximum2(i) \/ Enter0(i)
                    \/ Enter1(i) \/ Enter2(i) \/ Exit(i)

vars ==             <<e, n, j, m, pc>>

Fairness(i) ==      /\ WF_vars(Start(i)) /\ WF_vars(Maximum0(i))
                    /\ WF_vars(Maximum0a(i)) /\ WF_vars(Maximum1(i))
                    /\ WF_vars(Maximum2(i)) /\ WF_vars(Enter0(i))
                    /\ SF_vars(Enter1(i)) /\ WF_vars(Enter2(i))
                    /\ WF_vars(Exit(i))

Bakery ==           /\ BakeryInit
                    /\ [][\E i \in 0..N-1 : Step(i)]_vars
                    /\ (\A i \in 0..N-1 : Fairness(i))

Mutex ==            [](\A i1 \in 0..N-1, i2 \in 0..N-1 : i1 # i2
                        => (pc[i1] # 4 \/ pc[i2] # 4))
```

```
Mutex2 ==        \A i1 \in 0..N-1, i2 \in 0..N-1 : i1 # i2
                     => [](pc[i1] # 4 \/ pc[i2] # 4)
Liveness1 ==     \A i \in 0..N-1 : []<> (pc[i] = 0)
Liveness2 ==     \E i \in 0..N-1 : []<> (pc[i] = 4)

================================================================================
```

Program 22: Lamport's bakery algorithm in TLA$^+$