Applications and Limitations of the SPIN model checker compared to TLC and CBMC

Ksawery Radziwiłowicz[1]

Supervisors: Benedikt Ahrens[1], Anna Lukina[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: Ksawery Radziwiłowicz
Final project course: CSE3000 Research Project
Thesis committee: Benedikt Ahrens, Anna Lukina, Tim Coopmans

Abstract

Model checking is a formal verification method introduced in the 1980s that remains relevant to this day. While each of the many model checking tools created over the past 40 years is well-documented in both scientific literature and technical documents, there is a lack of work comparing different model checkers with one another. In this paper, we take a look at the SPIN model checker and compare it to other popular model checkers - TLC and CBMC. First, we compare the capabilities and ease of use of those model checkers through a literature review. Then, we investigate the performance of the compared tools through synthetic benchmarks. We conclude that SPIN offers better performance and ease of use than TLC when verifying models of concurrent algorithms and that it outperforms CBMC when model checking parallel algorithms, while CBMC remains a better choice for model checking sequential programs.

1 Introduction

Verifying the correctness of (software) systems has always been challenging. While techniques like manual or automated testing are easy to set up and can already help find a wide range of bugs, they are not sufficient for the most critical systems, as they can only show the presence of bugs but not their absence. Luckily, there are more powerful quality assurance techniques, known as formal verification methods. Formal verification methods are techniques that enable formally proving or disproving properties of systems or algorithms using mathematics. As such, with formal verification, it is possible to actually prove whether or not a given system conforms to a given formal specification.

One of the formal verification methods is model checking, in which a (simplified) version of a system is described in a special modelling language alongside desired properties and then verified, either through explicit state space exploration (that is, visiting all reachable states in the state graph), or through symbolic checking with satisfiability solvers. According to Clarke [1, sec. 1], the scientific community started noticing a need for tools like model checkers and developing required theoretical foundations in the 1970s and the early 1980s. Eventually, Clarke and Emerson suggested combining state space exploration and linear temporal logic in their 1982 paper [2], which is the foundation of many model checkers used nowadays. At the same time, Holzmann was developing his Pan verifier, which would be later used as a basis for the SPIN model checker, which would be extended with temporal logic specifications in the late 1980s, shortly before its first full version in 1989 [1, sec. 2.3].

While there are many papers with in-depth descriptions of both capabilities and implementation details of various model checkers, there are surprisingly few academic resources comparing the model checkers with one another. As such, in this paper we focus on the SPIN model checker and answer the following questions to make it easier for people new to the field of model checking to decide whether SPIN is the right choice for them:

1. How do SPIN's capabilities in terms of expressible and verifiable models and properties compare to the capabilities of TLC and CBMC?
2. How do SPIN's performance and memory usage when checking safety and liveness properties compare to the performance of TLC and CBMC?

For those comparisons, we decide to compare SPIN to TLC and CBMC as they are some of the most popular still-maintained model checkers operating respectively on abstract descriptions of systems and directly on C source code.

This paper is structured as follows: in Section 2 we describe the background information necessary to understand the basics of model checking and SPIN. In Section 3 we mention a number of papers showcasing practical usage of model checking, introducing several model checkers, and describing optimisation techniques commonly implemented in model checking software. In Section 4 we describe how we collected the information used in this paper as well as how we set up our experiments. In Section 5 we describe the similarities and differences between SPIN and TLC and between SPIN and CBMC. In Section 6 we describe and analyse the results of our benchmarks. In Section 7 we describe the actions we took to make sure our research was performed ethically and in a reproducible manner. Finally, in Section 8 we summarise our work and suggest future topics to be investigated.

2 Background

In this section we briefly discuss the background necessary to understand our work. We start by introducing linear temporal logic, continue into formally defining model checking, and finally explain what SPIN is and how it works.

2.1 Linear Temporal Logic

To understand the basics of model checking with SPIN, it is necessary to know the basics of linear temporal logic (LTL). LTL is a kind of logic that allows describing statements referring to time. While there are also other kinds of temporal logics (e.g. CTL or CTL* - a common superset of LTL and CTL), we will focus on LTL, as it is simple,

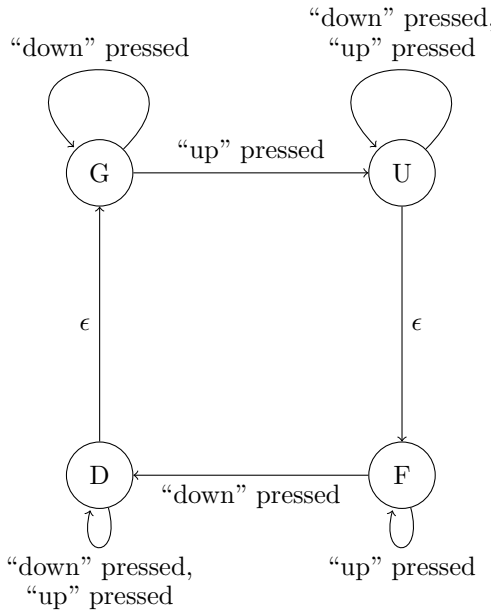

Figure 1: State machine modelling the lift example.

supported in many model checkers (including SPIN), and makes it possible to express a wide range of liveness and safety properties.

LTL formulae consist of propositional variables, atoms $\top$ (true) and $\bot$ (false), propositional logic operators (like $\wedge$, $\vee$, $\neg$, etc.), and temporal modal operators. The most important temporal modal operators include:

- $\bigcirc\varphi$ (next φ) - φ must hold at the next state
- $\Diamond\varphi$ (finally φ) - φ must hold at some future state
- $\Box\varphi$ (globally φ) - φ must hold at all future states
- $\omega\mathcal{U}\varphi$ (ω until φ) - φ must hold at some future state s_i; ω must hold at all states before s_i
- $\omega\mathcal{R}\varphi$ (ω releases φ) - if the first future state at which ω holds is s_i, then φ must hold at all states up to and including s_i; if ω never holds, φ must hold forever

With LTL we can formally express statements such as:

- "When you start learning category theory, you will eventually learn what a monad is."
  $c \implies \Diamond m$, where c means "you start learning category theory" and m means "you learn what a monad is"
- "Every time I sleep, I don't wake up until the alarm inevitably rings."
  $\Box(s \implies \neg w\mathcal{U}a)$, where s means "I'm asleep", w means "I wake up", and a means "the alarm rings"
- "I always eat lunch right after attending the morning lecture."
  $\Box(m \implies \bigcirc l)$, where m means "I attend the morning lecture", and l means "I eat lunch"

2.2 Model Checking

Model checking is a formal verification method that allows checking whether a set of properties P_i holds for a given system S. The idea behind model checking is to model the system as a mathematical structure $\mathcal{K}$ and express desired properties as mathematical formulae φ_i in such a way that property P_i holds for system S if and only if $\mathcal{K} \models \varphi_i$, that is, φ_i evaluates to true with the value assignment from $\mathcal{K}$.

As an example, consider a building with 2 floors and a lift going between them. For simplicity, assume that the lift can only be controlled using "up" and "down" buttons inside the cabin (i.e. it is impossible to call the lift from the outside). Such a system could be modelled by the structure shown in Figure 1.

In this structure, nodes and transitions have the following meanings:

- In state G, the lift is on the ground floor
- In state U, the lift is moving up
- In state F, the lift is on the first floor
- In state D, the lift is moving down
- Transitions annotated with ϵ require no external action to occur

Now, with the help of an appropriate model checker, it is possible to verify e.g. the following properties, expressed in linear temporal logic:

- It is always the case that if the lift is going up, it will eventually reach the first floor - $\square\left(U \rightarrow \Diamond F\right)$
- It is always the case that if the lift is going down, it will not reach the first floor before reaching the ground floor - $\square\left(D \rightarrow \left(G\mathcal{R}\left(\neg F\right)\right)\right)$
- It is always the case that the lift will not start moving down from the ground floor - $\square\left(\neg\left(G \wedge \bigcirc D\right)\right)$

2.3 SPIN

SPIN (Simple Promela INterpreter) is an explicit state model checker created in the 1980s at Bell Labs by Gerard J. Holzmann et al. Its main use case is verifying properties of multithreaded software systems.

The basic primitive in a SPIN model is a "process", representing a thread or a process in a concurrent program or system. Processes are described as lists of instructions that can read and write variables, evaluate arithmetic expressions, perform control flow through conditional statements or loops, and communicate with each other using channels. SPIN performs verification by exhaustive exploration of the state space using depth- or breadth-first search. The state space is the set of all possible program traces with all possible interleavings of instructions of separate processes. Since SPIN explores the state space exhaustively, it can only verify bounded models, that is, models that have a finite number of distinct behaviours. This means that models verified by SPIN cannot have data structures that grow indefinitely (enforced in PROMELA, the modelling language that SPIN uses, where each variable has a fixed size or capacity), nor can they create new processes indefinitely (enforced by SPIN, which limits the number of simultaneously active processes to 255[1]).

While this might sound inefficient both in terms of time and memory required to verify a model, SPIN includes several optimisation techniques that make model checking feasible. Important examples include:

- Partial order reduction (POR), which can merge distinct interleavings of independent instructions into a single transition [3]
- Various techniques for compressing visited states
- Hashcompact [4] and bitstate hashing [5], which allow reliably hashing visited states with negligible probabilities of collisions

SPIN models are described in PROMELA ("PROcess MEta LAnguage") - an imperative language (contrary to most other languages commonly used in model checking, e.g. TLA$^+$ [6] or NuSMV [7], which are declarative). In Figure 2, we show a small example of PROMELA code, which we refer to in this section to present some of the features of the language. While it does not showcase all syntax features of PROMELA, it presents enough for the reader to be able to comfortably understand our paper and the source code of associated experiments. For a more comprehensive overview of PROMELA, we recommend reaching for Holzmann's reference manual for SPIN [8].

The syntax of PROMELA somewhat resembles that of the C programming language combined with the bash shell scripting language.

A description of the process starts with the proctype keyword, followed by a name, parameter list, and a list of instructions enclosed in curly braces, as seen on line 1. A process can be defined to start as active (running) if the active keyword is used in the process definition, optionally with a number in square brackets denoting how many instances of the given process should be created. We show an alternative definition of P starting with 4 active instances in a comment on line 2. A special init process can be created with the init keyword - this process is active by default and is usually used to create other processes. We present how this can be done on line 17. Other processes can be created with a run instruction, as seen on line 19.

A process code consists of a sequence of statements. If a statement can be executed, it is said to be enabled; if a statement is not enabled, it can't be executed and SPIN has to choose an instruction from a different process instance to run next. A statement can be labelled with a label (same as in the C programming language) which can later be referred to with a so-called remote reference. A remote reference of the form procname[pid]@labelname is enabled if and only if an instance of a process procname with PID pid is at the statement labelled with labelname. Other primitive statements that can be disabled include:

[1] https://spinroot.com/spin/Man/run.html

- arithmetic or logic expressions that evaluate to 0
- control flow statements with no enabled branches
- else statements if other branches are enabled
- operations reading from an empty channel or writing to a full channel

PROMELA includes several basic control flow statements, most important being if (line 10), similar to a switch statement in C, and do, which works like an if but is executed in a loop until a break instruction is encountered. Each branch in a control flow statement is enabled if its first statement is enabled; a branch starting with else is enabled if and only if no other branches are enabled. The if or do instruction itself is enabled if at least one of its branches is enabled. An important difference between control flow statements in PROMELA and C is that PROMELA's control flow statements are nondeterministic - if multiple branches of an if or a do statement are enabled, SPIN is allowed to choose either of them (meaning in a simple simulation it will choose one at random but in exhaustive model checking it will explore all of them separately). In our code example, an instance of a process P with process id (PID) 4 can choose to execute either the branch on line 11 or the one on line 12, as 4 is divisible by both 4 and 2.

Furthermore, statements can be wrapped in an atomic block to denote they should not be interleaved with other processes. In our example, we use an atomic block (line 18) to ensure that all 4 instances of process P are active before any of them starts its execution. An atomic block will lose its atomicity if any statement inside of it blocks. There also exists a stricter version of atomic called d_step. d_step behaves similarly but it will cause an error instead of breaking atomicity if a statement inside of it blocks. Furthermore, a sequence of statements in a d_step block is treated by the model checker as a single state, which can further reduce the number of states the model has to store (greatly reducing memory usage). However, as a consequence of that, jumps into and out of a d_step block, such as ones caused by a break instruction, are not allowed.

Desired properties of the system can be described using so-called "never claims", using LTL formulae, or with inline assertions. An inline assertion detects a violation if the expression inside of it evaluates to 0 at the time when the assertion is executed. Never claims can be thought of as processes with no side-effects that cause a failure in liveness checking mode if they terminate or reach an acceptance cycle, that is, reach a cycle containing a label with a name starting with accept, while LTL formulae are internally converted to never claims. In our example code, we use an LTL statement on line 26 to verify a liveness property saying that the instance of process P with PID 1 must eventually reach its exit label if it reaches its enter label. An important difference between a never claim and a process, apart from the additional failure conditions mentioned above, is the fact that at most one never claim can be executed at a time and that it is scheduled separately from the rest of the model. When a never claim is used when model checking, SPIN will always schedule a single step of the never claim followed by a single step of the modelled system in lockstep.

What further sets SPIN apart from other model checkers is the fact that it doesn't verify models directly, but rather generates code in the C programming language that performs this task. This not only provides the user more control over how to perform model checking, but also allows providing inline C code in PROMELA models, leading to a possibility of providing efficient implementations of code fragments requiring heavy calculations [9].

Lastly, SPIN is surrounded by a rich ecosystem of tools making it easier to use. Notable examples include ispin - a graphical user interface automating e.g. generating, compiling, and running the verifier, and Modex [10] - a tool allowing for automatic extraction of PROMELA models from C source code, making it possible to verify programs without manually modelling them first (much like with tools such as CBMC [11] or CPA Checker [12]).

As we are comparing SPIN to CBMC, the latter of which operates directly on C code, Modex is of particular interest to us. Modex can generate a PROMELA model given C source code and a test harness file, which contains configuration required to extract and run the model. Below we show an example of a test harness file containing several of the important commands. While most commands take only a single line, some of them can or must span multiple lines. Multiline commands are terminated with the %% token.

```
1  %H
2  typedef signed char int8_t;
3  %%
4  %F binsearch.c
5  %X -e binsearch
6  %X -a main
7  %L
8  cur=nd_pick(... promela_code_follows_here;
9  needle=nd_needle(... promela_code_follows_here;
10 %%
```

%H defines additional header file information for the model - in our example, we use it to provide a type definition for int8_t. %X defines which C functions should be extracted from the file set with previous invocation of %F. This

```
proctype P() {
// active [4] proctype P() {
    byte id = _pid;
enter:
    assert(id > 0);

    printf("Hello from process %d\n", id);

exit:
    if
    :: id % 2 == 0 -> printf("My PID is even\n");
    :: id % 4 == 0 -> printf("My PID is divisible by 4\n");
    :: else -> printf("My PID is odd\n");
    fi;
}

init {
    atomic {
        run P();
        run P();
        run P();
        run P();
    }
}

ltl liveness_ltl { []((P[1]@enter) -> <>(P[1]@exit)); }
```

Figure 2: Example of PROMELA code

command can also take additional arguments modifying its behaviour - in the case of our example, -a says that the extracted prototype should be converted to an active process, while -e inserts additional instrumentation checks into the extracted process - those most importantly check for out-of-bounds array accesses and null pointer dereferences. %L defines mapping tables that convert specific expressions from the C source code into custom pieces of PROMELA code. In our example, we map statements where the result of calling nd_pick is assigned to the variable cur and statements where the result of calling nd_needle is assigned to the variable needle. Additionally, the harness file can be used for e.g. defining model generation, compilation, and runtime flags, or adding additional C or PROMELA code into the model.

Apart from automatically inserted instrumentation, Modex can translate C assertions into the ones used in PROMELA. Furthermore, never-claims or LTL properties can be defined in the harness file.

3 Related Work

This section describes literature relevant to our work and mentions several existing model checkers.

Since the advent of model checking theory in the 1980s, many model checkers have been developed and used for verification of various systems. Notable examples include the aforementioned SPIN [13], used e.g. to verify Intel's implementation of software transactional memory McRT-STM [14], find side-channel vulnerabilities in Linux and FreeBSD implementations of the TCP protocol [15], and analyse scheduling behaviours in FreeRTOS [16], which also led to bugs being found and reproduced on actual hardware; TLC [17], used e.g. to verify safety properties of Fast Paxos [18] and extensively used for formal verification at Amazon [19]; CBMC [20], used e.g. by Amazon for verifying memory safety in FreeRTOS [21]; and many more.

As we see from the above, model checking has been relevant for the past few decades, with new applications still being explored nowadays, both in academia and industry. The most popular tools, such as the aforementioned ones, are described in great detail in literature, both in terms of theory and formal description of their working, as well as higher-level descriptions of the used modelling language or verification options configurable from the user interface. Examples of such works include, but are not limited to, the overviews of model checkers, such as the ones listed before: SPIN [13], TLC [17], and CBMC [20]; a description of SPIN's bitstate hashing algorithm [5]; a description of the symmetry reduction technique [22], adopted e.g. in TLC and Java Pathfinder checkers; or a tutorial on SPIN and PROMELA [8].

4 Methodology

This section contains a description of our approaches to answer the research questions we set out to answer. Firstly, we discuss how we collected necessary information. Secondly, we explain how we performed the comparison of various tools. Lastly, we describe our setup for running benchmarks.

4.1 Collection of Information

As the literature review constitutes a large portion of this paper, collection of information was a crucial part of the research, serving the purpose of exploring the syntactic capabilities of investigated model checkers, implemented optimisations, as well as real-world examples of the use cases of analysed software. This task was mostly accomplished by querying academic search engines with relevant keywords. Additional searches were performed using general-purpose Internet search engines, such as Google, and by prompting large language models (LLMs). This is because, while remaining the most trustworthy, peer-reviewed scientific articles sometimes lacked relevant implementation details or usage examples of various tools. As such, the sources of information had to be extended to include official forums (e.g. http://spinroot.com/fluxbb/), official homepages related to the software, and online git repositories, all of which were found with the aforementioned methods. However, using general-purpose search engines and LLMs also led to a discovery of several relevant scientific articles that were not found using academic search engines.

4.2 Comparison of Tools

While expressivity of a given modelling language or support for checking a particular kind of property by a given model checker can be objectively measured, this paper also deals with statements that remain subjective.

Firstly, the readability of code in a given language is a subjective metric that might be perceived differently by different people. Whenever a comparison of code readability is performed, we attempt to make it as objective as possible by taking into consideration factors such as domain-specific knowledge needed to understand the code and similarity of the model to the original algorithm or system specification. In particular, in the case of model checking software algorithms: if an algorithm can be verified by directly using its implementation in a programming language, we consider the model checker that performs this verification to be the best in terms of code readability. If an implementation has to be first translated to a modelling language but the code structure is preserved (e.g. the model remains imperative, control flow statements remain in similar places, etc.), we consider it the second-best in terms of readability. If an implementation has to be translated to a modelling language that's not similar to imperative code, but rather one that is declarative and resembles mathematical statements that describe the code abstractly, we consider it to be least readable.

While we acknowledge that a declarative modelling language might be more suitable for describing more abstract systems and could result in code that is, in terms of the modelled system, more understandable than imperative code, our case studies focus on applying model checking to verification of algorithms. As such, we believe the criteria we describe above are reasonable for determining code readability of modelling languages for the purpose of this work.

Secondly, while the performance of two implementations of a given model checked by different model checkers can be measured objectively, the same cannot be said about the performance of the model checkers themselves. In other words, based solely on a limited number of case studies, it is impossible to determine that one model checker will perform better than another one for all (or even most) kinds of problems. To counteract that, specifically in the case of comparing SPIN and CBMC which have distinct areas that serve as their primary use cases, we create both benchmarks that we expect to favour SPIN and ones that we expect to favour CBMC.

4.3 Benchmarking Setup

To answer the second research question, we implement several algorithms and verify them using SPIN, TLC, and CBMC. For the sake of comparing TLC and SPIN, we model a generalisation of Peterson's algorithm with 6 concurrent processes, which we refer to as "filter algorithm/lock" as per Herlihy and Shavit's book [23, pp. 28-31]. Furthermore, we implement a simple system with a client-server infrastructure that provides read/write locks to the clients in TLA$^+$, to demonstrate an example where symmetry reduction - a powerful technique providing significant speedups for TLC in the filter algorithm - backfires and makes model checking slower instead.

When designing the models used to compare TLC and SPIN, we took great care to make them as similar as possible, both in terms of structure and the number of states. Since a SPIN model will usually allow interleaving between any two instructions, leading to a much higher number of states, we extensively used the d_step construct to group instructions together. In PROMELA, d_step executes a block of instructions atomically (that is, it doesn't allow interleaving with other processes within the block) and treats it as a single state.

However, we also take into consideration that this is not how SPIN models would usually be written if they're supposed to closely reflect how the given algorithm would behave when implemented in a normal programming language. As such, we also include tests without artificially introduced d_step blocks, which we believe provide a more accurate insight into the performance of the two model checkers in real-world scenarios. While the former test

is intended to more accurately showcase the raw efficiency of the underlying algorithms, the latter should reflect better on performance in a real-world scenario.

Furthermore, SPIN has many more configuration options than TLC. For SPIN we investigate the following 3 independent parameters:

- storage method - the way visited states are stored in memory. The options are "exhaustive", which uses a default fast compression algorithm; "collapse" which uses a slower but more effective compression method; and 2 hashing methods discussed earlier, hash-compact and bitstate hashing
- partial order reduction - can be enabled or disabled
- state space search algorithm - depth- or breadth-first search (DFS or BFS)

In total, this results in 16 distinct configurations for SPIN. For TLC, the only configuration options we could choose were whether to use symmetry reduction for the filter lock case study or not, and which state space search algorithm to use: default BFS or experimental implementation of iterative deepening DFS. However, as the latter didn't manage to terminate within reasonable time with our benchmarks, we decided to exclude it from the study, leaving us with the symmetry reduction being the only variable between different TLC configurations. When it comes to storage method, TLC uses a method similar to hash-compact. TLC does not support partial order reduction.

To compare liveness checking, we also use the filter algorithm. We check the liveness property $\Box(exit \implies \Diamond enter)$ for one of the processes, which ensures that the process will eventually try to enter the critical section again after leaving it. While we don't check this property for all processes simultaneously, we believe that checking it for a single process is enough, since the filter algorithm is highly symmetric in regard to processes - as such, if any violation of this liveness property could arise, it could arise for any of the processes and as such it would have been found by our model. In these tests, we test fewer configurations - firstly, we don't use BFS with SPIN since SPIN doesn't support using BFS for liveness checking; secondly, we do not perform the TLC test with symmetry reduction, as symmetry reductions might cause a model that checks liveness to yield false positives or false negatives [24].

To compare SPIN to CBMC, we prepare two benchmarks. The first one is a sequential program performing binary search on a sorted array of 5 integers in the range [-10,10]. The second one is checking safety of the aforementioned filter algorithm with 4 concurrent processes. We decide on using those two test cases as the former focuses on checking a sequential algorithm, which is what CBMC was primarily designed for, while the latter focuses on checking a concurrent program, which is SPIN's speciality. We do not run any benchmark checking liveness properties since - as we explain in Section 5.2 - CBMC does not support checking liveness properties.

For both of those tests, appropriate PROMELA models were automatically extracted from C sources using Modex, as we want to replicate CBMC usage experience as closely as possible - that is, building a simple harness around existing code instead of rewriting the program in a modelling language. Yet again, we use the same 16 SPIN configurations we use for verifying safety of the filter algorithm. For CBMC, we disable all the standard checks, so as to limit it to only checking what SPIN also checks, that is, assertions in the code. For the filter algorithm example we use the default memory model - SC - that guarantees sequential consistency, that is, guarantees that the effect of running a program with parallel threads results in the same effects as running all executed instructions in some sequential order (possibly with interleaving processes). We refrain from testing the other two memory models provided by CBMC (TSO and PSO) as using them would result in verifying a program with different properties. Apart from that, CBMC doesn't contain configuration options akin to SPIN.

All benchmarks were executed on a laptop with a 10-core/12-thread Intel i5-1235U CPU and 16GB of memory running Debian 13 as the operating system with version 2.41-12+deb13u2 of the Debian glibc library. The following versions of model checkers were used:

- SPIN - 6.5.2 (with Modex version 2.11)
- TLC - 2026.05.26.235334
- CBMC - 6.8.0

To run TLC, OpenJDK version 25.0.3 distributed by Eclipse Temurin project was used. To provide more accurate results and reduce the effect of other processes running on the machine on the runtime measurements, all experiments were executed 5 times and the runtimes we report in this paper are the results of averaging the runtimes of all runs.

For all the tests, we provide the following resource limits: 2 minutes of wall clock time and 8GiB of memory.

The resource usage was measured in the following manner: for SPIN, the values measured by the verifier were used; for TLC, we used GNU time utility to measure memory usage, since TLC doesn't measure this information itself, but used TLC's reported time as runtime; for CBMC, we also used GNU time to measure memory usage, and CBMC's built-in functionality of reporting timestamps with log messages to calculate the time spent verifying the models. When it comes to runtimes, all methods of measurement measure the same metric - that is, the time required to check the model; in case of SPIN, that does not include the time required to generate and compile the verifier program. When it comes to memory, while GNU time does not report when exactly the highest memory

usage occurs, we can safely assume that model checking itself will use up much more memory than parsing TLA$^+$ or C code. As such, we believe our chosen methods of obtaining measurements are justified.

5 Features of SPIN and Other Model Checkers

In this section we present the results of our theoretical work about comparing other model checkers to SPIN. First, we present the capabilities and limitations of TLC as compared to SPIN and vice versa. Then, we do the same with CBMC.

5.1 TLC

Arguably one of the most popular model checkers used nowadays is TLC, which uses the TLA$^+$ language. Under the hood, both SPIN and TLC are explicit-state, meaning they perform model checking by visiting and verifying all reachable states. As such, both of them suffer from similar limitations, namely susceptibility to state-space explosion and lack of support for models with infinite states.

From a user's perspective, the most important difference lies in the modelling language - SPIN uses imperative PROMELA, while TLC operates on declarative TLA$^+$. Both of those approaches have their own advantages and disadvantages. Being imperative, PROMELA allows the user to stay very close to actual implementation in an imperative language (e.g. C) making it easy to verify that application source code and model represent a system with the same behaviour. This property, along with the possibility of including actual C code in the model [8, ch. 17], is especially useful when checking specific implementations of algorithms instead of general specifications of systems, as mentioned e.g. by Cao et al. in their paper about finding TCP side-channel vulnerabilities in Linux and FreeBSD implementations of the network stack [15, sec. 2].

To visualise the differences between PROMELA and TLA$^+$ for modelling algorithms, we provide 2 simple models of Peterson's algorithm in Appendix A - one written in PROMELA (Figure 3) and one written in TLA$^+$ (Figure 4). While the PROMELA code follows the pseudocode from Peterson's letter [25, fig. 1] closely, adding just slightly more code that initialises variables and ensures mutual exclusion, TLA$^+$ requires adding much more syntax "boilerplate", by forcing the programmer to manually translate the algorithm to a state machine.

It is worth noting that the TLA$^+$ environment does also support an imperative language called PlusCal [26]. While it does provide a syntax more suitable for sequential algorithms, it still gets translated to TLA$^+$ before the model is checked. As such, in case of any errors found in the model specification, the programmer must still debug declarative TLA$^+$ code which might be difficult to understand for larger projects. Furthermore, it also doesn't support embedding C code directly into the specification, making it still less flexible than PROMELA in this regard.

Despite that, TLA$^+$ syntax still has an advantage over PROMELA in certain situations. While PROMELA only supports basic data types (boolean values, integers of various sizes, arrays with predetermined size, channels, and structures aggregating fields of other types), TLA$^+$ offers more complex types, like sets, functions, and bags (multisets). As an example, consider a model of an algorithm that, given a set of edges in a graph, constructs a spanning tree by picking random edges. While in TLA$^+$ it's possible to easily represent the set with a native type and the operation of choosing a random edge as \E e \in Edges, the only way to represent a graph in PROMELA would be to use an adjacency matrix, while the operation of choosing a random edge would consist of iterating over all entries of the matrix and checking whether a given entry exists, and finally choosing one of the existing entries.

TLC also supports a very powerful optimisation technique that SPIN doesn't implement, called symmetry reduction [22]. It allows treating multiple "symmetric" states as a single state, which can drastically reduce the state space size. Two states are considered symmetric if they are identical up to the renaming of model values (abstract values that are equal only to themselves, similar to keywords in LISP). As an example, consider a system with 2 identical processes P1 and P2. The state where P1 is in state S1 and P2 is in state S2 is symmetric to a state where P1 is in state S2 and P2 is in state S1, and as such they can be treated as equivalent if symmetry reduction is used.

It is worth keeping in mind that many concurrent systems are not symmetric and therefore symmetry reduction can't be always applied. A simple example is Lamport's bakery mutual exclusion algorithm [27] - since it relies on unique process IDs to break ties in ticket numbers, it is not symmetric as processes are not treated equally (i.e. processes with lower IDs are given a higher priority).

Lastly, due to the abstract nature of TLA$^+$ and low-level nature of PROMELA, the former is more suitable for verifying specifications described at a higher level. While PROMELA and SPIN are not mentioned directly, Lamport et al. mention in their paper published in 2002: "Hardware engineers routinely use tools based on formal methods to check lower-level designs. They are not accustomed to using tools to check their high-level designs. [...] We have found them receptive to the idea of using TLA+ at the protocol level." [28, sec. 4]. As SPIN had been around for over 10 years in 2002 and was known in the hardware engineering community, as proved e.g. by [29], one could assume that engineers would prefer familiar tools over new TLC and TLA$^+$ if the latter were not more suitable for verifying higher-level specifications.

5.2 CBMC

CBMC is a bounded model checker for verifying C programs. What sets it apart from tools like SPIN and TLC is the fact that CBMC doesn't have a separate modelling language - instead, it performs verification directly on C programs, making sure that all assertions that are present are true. Also in contrast with SPIN and TLC, which are both explicit state model checkers, CBMC is a symbolic model checker - that is, instead of exploring every reachable state in a state graph, it will build a set of constraints C and properties P to verify and try to find a solution to $C \wedge \neg P$ using a SAT solver - if it succeeds, the verification fails and the assignment of variables found by the SAT solver is the counterexample breaking the desired properties [20].

Furthermore, CBMC is a bounded model checker. This means that before performing verification, it must know the maximum number of iterations of all loops and perform unrolling. While in a lot of cases it can do that automatically (e.g. for simple for-loops with constant or easy-to-compute bounds), in some cases it might fail and the programmer must provide the upper bound manually [30].

While the fact that CBMC does not have any custom modelling language makes it easier to use it for programmers not familiar with any model checking software, it also restricts what properties of the system can be verified. Consider that we want to verify the following liveness property in a mutual exclusion algorithm:

$$\Box\,(W \rightarrow \Diamond C)$$

where W means "the process is waiting to acquire the lock" and C means "the process enters the critical section"; as such, the whole property can be read as "it is always the case that if a process starts waiting on a lock, it will eventually enter the critical section". While expressing it in PROMELA and verifying with SPIN is trivial, it is not possible to do the same with CBMC. More generally speaking, liveness properties (i.e. properties specifying that "something good eventually happens") are not verifiable using CBMC.

Additionally, CBMC is not as well-suited for verifying concurrent programs as SPIN. While it does support model checking programs that use the POSIX Threads (pthreads) API for concurrency, the number of formulae that have to be considered by a SAT solver to verify all possible interleavings for larger models usually makes it inefficient. However, there exist tools that convert a concurrent C program into a sequential one that emulates a given scheduler (e.g. a round-robin scheduler) where a context switch bound - that is, an upper limit on the allowed number of context switches - can be imposed. Such a program can be then verified with CBMC. The justification of this approach is that most concurrency-related bugs can already be detected with a relatively low number of context switches, which can greatly reduce the verification time. One example of a tool performing such a conversion is CSeq created by Fischer, Inverso, and Parlato [31]. CSeq has won the concurrency category in the annual Software Verification Competition SV-COMP multiple times, often beating raw CBMC (among other tools) [32, p. 3].

While CBMC lacks some of SPIN's capabilities as described above, it makes up for it in its safety verification features. Similarly to Modex, CBMC supports checking for out-of-bounds accesses to static arrays and null pointer dereferencing. However, its capabilities don't stop there - in addition, CBMC can check for many more kinds of errors, such as out-of-bounds accesses for dynamically allocated memory or integer overflows, which Modex can't check automatically.

6 Experimental Results and Discussion

In this section we present the results of the experiments described in Section 4.3.

6.1 SPIN vs TLC

Firstly, we present the results of case studies for SPIN and TLC.

In Table 1 we show the results of model checking the filter algorithm with TLC and SPIN, the latter of which uses the "simplified" model with d_step blocks that reduce the number of states. We observe that SPIN performs much better than TLC without the symmetry reduction, taking both less time and using less memory to verify the model, despite the fact that the SPIN model results in ∼10-20 times as many states as the TLC model. Configurations using hashing to store visited states (that is, hash-compact or bitstate hashing) perform better than the ones using compression, both in terms of runtime and memory usage.

Configurations using BFS take slightly longer to run than their DFS counterparts and use significantly more memory. The increase in the amount of memory used with BFS as compared to DFS makes sense, as the former must store more states "to be visited" than the latter, which only needs to store the trail of states from the initial state.

Furthermore, we observe the effect of partial order reduction on the time and memory usage. Since POR reduces the number of states that have to be visited, both the time taken to check the model and the memory used were reduced by applying this optimisation. A notable exception is the last checked configuration (POR, bitstate hashing, and BFS), which didn't terminate in under two minutes in any run. We believe this to be a bug in the model checker, since adding the LTL expression from our liveness test makes the checker terminate in a reasonable amount of time,

Table 1: Performance of TLC model and simplified SPIN model on the filter algorithm (safety)

Model Checker	Configuration	Time Taken (s)	Memory Used (MiB)	Unique States Visited
SPIN	N, E, D	3.05	546.99	6045369
	N, E, B	3.34	714.82	6045369
	N, C, D	4.95	506.66	6045369
	N, C, B	5.80	674.58	6045369
	N, HC, D	2.51	408.32	6040556
	N, HC, B	2.90	576.05	6040574
	N, BH, D	2.59	352.05	6045369
	N, BH, B	2.93	519.87	6045369
	P, E, D	1.04	430.61	3156488
	P, E, B	1.22	441.09	2963285
	P, C, D	1.59	363.43	3156488
	P, C, B	2.01	378.39	2963285
	P, HC, D	0.94	358.54	3156843
	P, HC, B	1.22	373.31	2963568
	P, BH, D	1.89	385.30	3156488
	P, BH, B	TO	TO	TO
TLC	N	11.60†	955.53	315222
	S	2.00†	261.57	2809

SPIN configuration: N/P - Partial Order Reduction: disabled/enabled; E/C/HC/BH - storage mode: exhaustive/collapse compression/hash-compact/bitstate hashing; D/B - search algorithm: DFS/BFS
TLC configuration: N/S - Symmetry Reduction: disabled/enabled
TO - timed out
†TLC doesn't report time up to sub-second precision

albeit increased due to the execution of the generated never claim. Even more interestingly, adding simpler never claims (present in the source code of our model) did not fix the issue. We suspect that in this configuration the model checker eventually reaches an infinite loop, which is somehow mitigated by adding a more complex never claim, as those are executed in lockstep with the model's processes. However, we were unable to trace whether this is actually the case.

Interestingly, with POR enabled, configurations using the same storage method visit a different number of states depending on whether DFS or BFS is used. This is due to the fact that DFS and BFS operate in vastly different ways and need different provisos (algorithms) for performing partial order reduction, as described by Bošnački and Holzmann [33]. While in the case of our experiment the proviso for BFS resulted in a smaller number of states than the one for DFS, this is not always the case, as discussed in the aforementioned paper [33, sec. 4].

Additionally, we observe that configurations using hash-compact storage method visit fewer states than the other methods. This, as we suspected and confirmed with Holzmann via email, is likely caused by hash collisions which - although unlikely - can occur when only hashes of visited states are stored, causing some new states to be mistakenly taken as already visited and ignored. This situation could also happen with bitstate hashing, however we do not observe that in this particular experiment.

Finally, we investigate the effect of symmetry reduction on the results. With symmetry reduction enabled, TLC does outperform some configurations of SPIN. As TLC reports, without symmetry reduction 315222 unique states are found on average, while the optimisation can bring that number down to just 2809.

Table 2 contains the results of running the "full" SPIN model - that is, one without artificially inserted d_step blocks - and compares the results against TLC, which used the same model as in Table 1. Here we observe a situation we did not see in the previous test, that is, bitstate hashing suffering from hash collisions, e.g. in the N, BH, D configuration which visits fewer unique states than N, E, D. Additionally, we see that a well-optimised SPIN model with POR can significantly outperform a TLC model without symmetry reduction in terms of runtime even if the former has orders of magnitude more states.

However, as the size of the set over which symmetry is applied grows, so does the cost of performing the reduction. We verify a simple model of a client-server system where the server grants a read/write lock to clients as requested (that is, multiple clients may hold a read lock at the same time but only one write lock can be granted; similarly, a read and a write lock can't be granted simultaneously). With as few as 7 client processes, we observe a significant runtime penalty for the run applying symmetry reduction despite the fact that in the end orders of magnitude fewer states are visited, as shown in Table 3. It is worth noting that in this case symmetry reduction still leads to a lower memory usage.

Table 2: Performance of TLC model and full SPIN model on the filter algorithm (safety)

Model Checker	Configuration	Time Taken (s)	Memory Used (MiB)	Unique States Visited
SPIN	N, E, D	42.38	3490.24	51935282
	N, E, B	47.22	5527.99	51935282
	N, C, D	64.32	3110.56	51935282
	N, C, B	67.54	5148.59	51935282
	N, HC, D	39.06	2289.95	51582935
	N, HC, B	39.90	4313.83	51584759
	N, BH, D	24.06	2139.75	51935268
	N, BH, B	26.48	4177.59	51935270
	P, E, D	8.45	1381.88	18790666
	P, E, B	9.66	2070.48	18870750
	P, C, D	13.18	966.35	18790666
	P, C, B	14.58	1653.00	18870750
	P, HC, D	6.83	948.09	18693228
	P, HC, B	10.10	1720.97	19952279
	P, BH, D	9.95	2186.48	18790666
	P, BH, B	TO	TO	TO
TLC	N	11.60†	955.53	315222
	S	2.00†	261.57	2809

SPIN configuration: N/P - Partial Order Reduction: disabled/enabled; E/C/HC/BH - storage mode: exhaustive/collapse compression/hash-compact/bitstate hashing; D/B - search algorithm: DFS/BFS
TLC configuration: N/S - Symmetry Reduction: disabled/enabled
TO - timed out
†TLC doesn't report time up to sub-second precision

Table 3: Performance of TLC with symmetry reduction on the R/W lock server

Model Checker	Configuration	Time Taken (s)	Memory Used (MiB)	Unique States Visited
TLC	N	4.40	783.63	222417
	S	12.00	330.71	333

TLC configuration: N/S - Symmetry Reduction: disabled/enabled

Table 4: Performance of TLC model and simplified SPIN model on the filter algorithm (liveness)

Model Checker	Configuration	Time Taken (s)	Memory Used (MiB)	Unique States Visited
SPIN	N, E	4.14	659.42	6501420
	N, C	7.26	572.02	6501420
	N, HC	3.85	487.45	6496770
	N, BH	5.70	389.89	6501420
	P, E	4.56	659.42	6501420
	P, C	7.99	572.02	6501420
	P, HC	3.91	487.45	6496770
	P, BH	5.71	390.47	6501420
TLC	-	124.20†	977.23	32062

SPIN configuration: N/P - Partial Order Reduction: disabled/enabled; E/C/HC/BH - storage mode: exhaustive/collapse compression/hash-compact/bitstate hashing
†TLC doesn't report time up to sub-second precision

Table 5: Performance of CBMC and SPIN when model checking binary search algorithm

Model Checker	Configuration	Time Taken (s)	Memory Used (MiB)	Unique States Visited
SPIN	N, E, D	21.86	5278.14	54334024
	N, E, B	MLE	MLE	MLE
	N, C, D	40.44	4563.58	54334024
	N, C, B	80.90	7700.94	64375594
	N, HC, D	8.52	1060.96	27566498
	N, HC, B	18.16	3204.90	35710595
	N, BH, D	9.58	1115.75	54328450
	N, BH, B	19.36	3707.89	64372105
	P, E, D	23.70	5708.63	54334024
	P, E, B	MLE	MLE	MLE
	P, C, D	45.32	3920.44	54334024
	P, C, B	80.72	6948.79	64375594
	P, HC, D	8.31	1299.77	27913197
	P, HC, B	16.20	3486.82	35771486
	P, BH, D	19.86	1146.27	54329389
	P, BH, B	22.76	3731.64	64372369
CBMC	-	0.04	17.04	-

SPIN configuration: N/P - Partial Order Reduction: disabled/enabled; E/C/HC/BH - storage mode: exhaustive/collapse compression/hash-compact/bitstate hashing
MLE - memory limit exceeded

Table 4 contains the results of checking the liveness property $\Box(exit \implies \Diamond enter)$. Firstly, we see that SPIN provides an even greater speedup over TLC than in the safety tests. Secondly, we observe that in this case partial order reduction did not reduce the number of states at all. This might be caused by the fact that in those tests a never claim is present, thus changing the transitions present in the program (as described in Section 2.3).

We decided to not include the liveness tests for the SPIN model without d_step blocks as all of the configurations timed out.

Overall, the general result of TLC being slower than SPIN is in line with what Lamport said about the performance of TLC in several of his papers [28, sec. 2.2] [34].

6.2 SPIN vs CBMC

Table 5 presents the results of verifying a sequential program implementing the binary search algorithm and running it on a sorted array of 5 arbitrary numbers in the range [-10,10]. Firstly, by looking at the number of states visited by SPIN, we yet again observe that configurations using hashing methods visited fewer states than the ones storing compressed states. We attribute this once more to hash collisions. Secondly, we observe no improvements in the numbers of states when POR is enabled. This is expected as we are verifying a sequential program. As such, we don't ever encounter an interleaving of independent instructions from different processes that could be reordered without changing the model's behaviour and POR has no effect on the final number of visited states.

Looking at CBMC results, it is clear that it beats SPIN in this test, both when it comes to runtime and memory consumption. This is caused by the fact that SPIN, as an explicit state model checker, must separately verify the algorithm for each set of values in the array and for each needle value, resulting in many distinct configurations, each consisting of multiple states. Meanwhile, CBMC with its symbolic approach can verify multiple of those configurations at once. Consider the verifier entering the first iteration of the binary search loop. In this scenario there are 3 options:

- The middle element of the array is equal to the needle - the loop breaks and the verifier never has to consider the values of other elements of the array.
- The middle element is lesser than the needle - the algorithm never uses the numbers in the lower half of the array and as such the verifier never has to consider their values.
- The middle element is greater than the needle - the algorithm never uses the numbers in the upper half of the array and as such the verifier never has to consider their values.

This process continues recursively until the index of the needle is found or it is determined that the needle doesn't occur in the array. At each step at least half of the remaining search space is considered, and as such the verifier will only have to consider the values of approximately $\log n$ variables where n is the length of the array. Comparing that

Table 6: Performance of CBMC and SPIN when model checking filter algorithm

Model Checker	Configuration	Time Taken (s)	Memory Used (MiB)	Unique States Visited
SPIN	N, E, D	0.02	133.86	47578
	N, E, B	0.02	135.91	47578
	N, C, D	0.03	130.44	47578
	N, C, B	0.04	132.49	47578
	N, HC, D	0.02	129.66	47577
	N, HC, B	0.02	131.71	47577
	N, BH, D	0.01	2.19	47578
	N, BH, B	0.01	4.25	47578
	P, E, D	0.01	131.03	22586
	P, E, B	0.01	132.10	22586
	P, C, D	0.02	129.07	22586
	P, C, B	0.02	130.15	22586
	P, HC, D	0.01	128.98	22585
	P, HC, B	0.01	130.05	22578
	P, BH, D	0.01	2.19	22586
	P, BH, B	0.02	3.17	22586
CBMC	-	43.72	90.24	-

SPIN configuration: N/P - Partial Order Reduction: disabled/enabled; E/C/HC/BH - storage mode: exhaustive/collapse compression/hash-compact/bitstate hashing

to the growth of the computation size that SPIN has to perform, which is exponential in the length of the array, it is clear why CBMC performs so much better here.

Lastly, we compare the performance of SPIN and CBMC when verifying the filter algorithm with 4 processes. As expected, SPIN finishes very quickly - we already saw it perform well with the same algorithm with more processes and the runtime growth is exponential in the number of processes. We also observe a huge reduction in memory usage when choosing bitstate hashing over other storage methods.

CBMC does not perform nearly as well. While it does use less memory than SPIN with configurations not using bitstate hashing, the runtime is far greater than that of any of the SPIN runs. This result confirms the information we found about CBMC not performing well when verifying multithreaded programs due to the number of formulae required to be given to the SAT solver to consider all possible paths, a fact which also prompted the creation of tools such as CSeq.

7 Responsible Research

This section covers the actions we take to make sure our research is performed responsibly.

Firstly, we take great care to not misinterpret capabilities of the tools discussed in this paper. For this, we carefully review available technical documentation and, where possible, provide examples of how the features we mention can be used or observed in practice.

Secondly, in the course of this research, we perform experiments that analyse the performance of various model checkers. During the literature review, we have noticed that many papers in the field, including those that describe novel extensions to existing tools or include a performance comparison between various solutions, do not publicly share related code. To ensure that our results stay reproducible and verifiable, we publish all relevant code in the following Git repository: https://gitlab.com/ksaweryr/cse3000-research-project-code.

Lastly, in recent years, we have observed a massive increase in the use of Artificial Intelligence (AI)-based tools and Large Language Models (LLMs) in both software development and research. While AI and LLMs are a useful tool for various tasks, they are not flawless and their output should not be trusted blindly. As such, the usage of AI in this project is limited to the following:

1. Finding more examples of relevant literature that wasn't found through other means.
2. Generating simple code examples for the purpose of getting an insight into new languages.
3. Checking the spelling and grammar in the paper.

It is important to note that we have never cited a paper suggested by AI without first verifying that it actually exists and contains content relevant to our research. Additionally, any code we generated using AI was only used for learning purposes for us; no code or text generated by AI made it into the final version of the deliverables of this project.

8 Conclusions and Future Work

In this paper, we compare the SPIN model checker to 2 other popular tools: TLC and CBMC. We analyse the capabilities of SPIN, TLC, and CBMC, and compare the performance thereof on synthetic benchmarks we construct.

When comparing SPIN and TLC, we conclude that each has a respective use case where it is more suitable than the other. In the case of SPIN, which uses the PROMELA modelling language, it is verifying imperative algorithm descriptions, as PROMELA is imperative itself, and verifying specific implementations, as PROMELA makes it possible to embed C code directly into a model. TLC, which uses TLA^+ as its modelling language, is superior for verifying higher-level descriptions of systems due to the declarative nature of the language, which is suitable for describing abstract specifications, and a richer selection of available types and operations (i.e. sets, functions, picking an arbitrary element) than SPIN.

When comparing SPIN and CBMC, we notice that the two are suitable for different goals - while the former is intended to verify safety and liveness properties of concurrent systems, the latter was designed to check correctness (e.g. memory safety, lack of undefined behaviour) of sequential programs and works directly on C source code, without the need to rewrite it to a modelling language first. While CBMC itself isn't particularly efficient when verifying larger multithreaded programs, external tools or extensions can be used to simplify the model while not sacrificing much accuracy; similarly, while SPIN can't verify C code directly, its accompanying tool Modex can be used to convert it to a PROMELA model.

Our synthetic benchmarks show that SPIN usually outperforms TLC both in terms of runtime and memory, even if the checked models (describing the same system) are different enough to result in a larger number of states for SPIN. In certain cases, TLC can be made to outperform SPIN if symmetry reduction is used. However, this method is not a silver bullet, since firstly, it is only applicable to some problems (in particular, ones where concurrent processes are indistinguishable in terms of behaviour), and secondly, in larger models, the time it takes to detect symmetries might end up larger than the time saved by applying the reduction.

When benchmarking SPIN and CBMC, we clearly see the advantage of the former on parallel programs and the advantage of the latter on sequential programs.

For future work, we suggest the following: firstly, the comparison of capabilities and performance should be extended to more model checkers. Interesting examples include but are not limited to: nuXmv, which allows specifying properties in CTL in addition to LTL, as well as verifying infinite state systems; UPPAAL, which supports verifying real-time systems; or Murphi, which, although not actively maintained anymore, made some design decisions similar to SPIN. Rerunning the SPIN-versus-CBMC benchmark but this time feeding CBMC a version of the program sequentialised by CSeq would also be a worthy experiment.

Secondly, we believe investigating the performance of model checking with multiple threads/workers could be interesting. While we focused on single-core performance of the presented model checkers, their capabilities to scale on multithreaded systems are worth comparing with each other.

Lastly, we recommend extending the set of our benchmarks with more examples. Adding a model of some real abstract description of a system, such as a consensus protocol like Paxos or Raft, could be interesting and perhaps could better show SPIN's weaknesses.

References

[1] E. M. Clarke, "The birth of model checking," in 25 Years of Model Checking: History, Achievements, Perspectives, O. Grumberg and H. Veith, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–26, ISBN: 978-3-540-69850-0. DOI: 10.1007/978-3-540-69850-0_1 [Online]. Available: https://doi.org/10.1007/978-3-540-69850-0_1

[2] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in Logics of Programs, D. Kozen, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 52–71, ISBN: 978-3-540-39047-3.

[3] P. Godefroid, Partial-Order Methods for the Verification of Concurrent Systems. Springer Berlin Heidelberg, 1996, vol. 1032, ISBN: 978-3-540-60761-8. DOI: 10.1007/3-540-60761-7

[4] P. Wolper and D. Leroy, "Reliable hashing without collision detection," in Computer Aided Verification, C. Courcoubetis, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 59–70, ISBN: 978-3-540-47787-7.

[5] G. J. Holzmann, "An analysis of bitstate hashing," 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, P. Dembiński and M. Średniawa, Eds. Boston, MA: Springer US, 1996, pp. 301–314, ISBN: 978-0-387-34892-6. DOI: 10.1007/978-0-387-34892-6_19 [Online]. Available: https://doi.org/10.1007/978-0-387-34892-6_19

[6] L. Lamport, Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. USA: Addison-Wesley Longman Publishing Co., Inc., 2002, ISBN: 032114306X.

[7] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri, "Nusmv: A new symbolic model verifier," in Proceedings of the 11th International Conference on Computer Aided Verification, ser. CAV '99, Berlin, Heidelberg: Springer-Verlag, 1999, pp. 495–499, ISBN: 3540662022.

[8] G. Holzmann, The SPIN Model Checker: Primer and Reference Manual, 1st. Addison-Wesley Professional, 2011, ISBN: 0321773713.

[9] G. J. Holzmann, "Logic verification of ansi-c code with spin," in SPIN Model Checking and Software Verification, K. Havelund, J. Penix, and W. Visser, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 131–147, ISBN: 978-3-540-45297-3.

[10] G. J. Holzmann, Accessed on 02/06/2026. [Online]. Available: https://spinroot.com/modex/MANUAL.html

[11] D. Kroening, P. Schrammel, and M. Tautschnig, Cbmc: The c bounded model checker, 2023. arXiv: 2302.02384 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2302.02384

[12] D. Beyer and M. E. Keremoglu, Cpachecker: A tool for configurable software verification, 2009. arXiv: 0902.0019 [cs.PL]. [Online]. Available: https://arxiv.org/abs/0902.0019

[13] G. Holzmann, "The model checker spin," IEEE Transactions on Software Engineering, vol. 23, pp. 279–295, May 1997.

[14] J. O'Leary, B. Saha, and M. Tuttle, "Model checking transactional memory with spin," in 2009 29th IEEE International Conference on Distributed Computing Systems, IEEE, Jun. 2009, pp. 335–342. DOI: 10.1109/ICDCS.2009.72

[15] Y. Cao, Z. Wang, Z. Qian, C. Song, S. V. Krishnamurthy, and P. Yu, "Principled unearthing of tcp side channel vulnerabilities," in Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, ser. CCS '19, London, United Kingdom: Association for Computing Machinery, 2019, pp. 211–224, ISBN: 9781450367479. DOI: 10.1145/3319535.3354250

[16] C.-K. Lin and B.-Y. Wang, Analyzing freertos scheduling behaviors with the spin model checker, 2022. arXiv: 2205.07480 [cs.FL]. [Online]. Available: https://arxiv.org/abs/2205.07480

[17] Y. Yu, P. Manolios, and L. Lamport, "Model checking tla+ specifications," in Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods, ser. CHARME '99, Berlin, Heidelberg: Springer-Verlag, 1999, pp. 54–66, ISBN: 3540665595.

[18] L. Lamport, "Fast paxos," Distrib. Comput., vol. 19, no. 2, pp. 79–103, Oct. 2006, ISSN: 0178-2770. DOI: 10.1007/s00446-006-0005-x [Online]. Available: https://doi.org/10.1007/s00446-006-0005-x

[19] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How amazon web services uses formal methods," Communications of the ACM, 2015. [Online]. Available: https://www.amazon.science/publications/how-amazon-web-services-uses-formal-methods

[20] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004), K. Jensen and A. Podelski, Eds., ser. Lecture Notes in Computer Science, vol. 2988, Springer, 2004, pp. 168–176, ISBN: 3-540-21299-X.

[21] R. Barry and A. W. Services, Accessed on 02/06/2026. [Online]. Available: https://github.com/FreeRTOS/FreeRTOS/tree/b72aef2d316c0571eaf6a9e528e759b8e2ba107b/FreeRTOS/Test/CBMC

[22] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla, "Symmetry reductions in model checking," in Computer Aided Verification, A. J. Hu and M. Y. Vardi, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 147–158, ISBN: 978-3-540-69339-0.

[23] M. Herlihy and N. Shavit, The Art of Multiprocessor Programming, Revised Reprint, 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012, ISBN: 9780123973375.

[24] TLA+ Toolbox Developers, Accessed on 02/06/2026. [Online]. Available: https://tla.msr-inria.inria.fr/tlatoolbox/doc/model/model-values.html

[25] G. L. Peterson, "Myths about the mutual exclusion problem," Inf. Process. Lett., vol. 12, pp. 115–116, 1981. [Online]. Available: https://api.semanticscholar.org/CorpusID:45492619

[26] L. Lamport, "The pluscal algorithm language," Theoretical Aspects of Computing-ICTAC 2009, Martin Leucker and Carroll Morgan editors. Lecture Notes in Computer Science, number 5684, 36-60., Jan. 2009. [Online]. Available: https://www.microsoft.com/en-us/research/publication/pluscal-algorithm-language/

[27] L. Lamport, "A new solution of dijkstra's concurrent programming problem," Commun. ACM, vol. 17, no. 8, pp. 453–455, Aug. 1974, ISSN: 0001-0782. DOI: 10.1145/361082.361093 [Online]. Available: https://doi.org/10.1145/361082.361093

[28] L. Lamport, J. Matthews, M. Tuttle, and Y. Yu, "Specifying and verifying systems with tla+," in Proceedings of the Tenth ACM SIGOPS European Workshop, INRIA (Institut National de Recherche en Informatique et en Automatique), Association for Computing Machinery, Inc., Sep. 2002, pp. 45–48. [Online]. Available: https://www.microsoft.com/en-us/research/publication/specifying-and-verifying-systems-with-tla/

[29] B. Rahardjo, "Spin as a hardware design tool," Nov. 1995.

[30] U. of Oxford Systems Verification Group, Accessed on 02/06/2026. [Online]. Available: http://www.cprover.org/cprover-manual/

[31] B. Fischer, O. Inverso, and G. Parlato, "Cseq: A concurrency pre-processor for sequential c verification tools," in 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2013, pp. 710–713. DOI: 10.1109/ASE.2013.6693139

[32] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato, "Bounded verification of multi-threaded programs via lazy sequentialization," ACM Trans. Program. Lang. Syst., vol. 44, no. 1, Dec. 2021, ISSN: 0164-0925. DOI: 10.1145/3478536 [Online]. Available: https://doi.org/10.1145/3478536

[33] D. Bošnački and G. J. Holzmann, "Improving spin's partial-order reduction for breadth-first search," in Model Checking Software, P. Godefroid, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 91–105, ISBN: 978-3-540-31899-6.

[34] L. Lamport, "Real-time model checking is really simple," in Correct Hardware Design and Verification Methods, D. Borrione and W. Paul, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 162–175, ISBN: 978-3-540-32030-2.

A Peterson's algorithm in PROMELA and TLA$^+$

```
bool flag[2] = { false, false };
bit turn = 0;
byte mutex = 0;

active [2] proctype P() {
    bit i = _pid % 2;

    flag[i] = true;
    turn = 1 - i;

    !flag[1 - i] || turn != 1 - i;

    mutex = mutex + 1;
    CRIT:
    assert mutex == 1;
    mutex = mutex - 1;

    flag[i] = false;
}
```

Figure 3: Implementation of Peterson's algorithm in PROMELA

```
---- MODULE Peterson ----

EXTENDS Naturals, TLC

VARIABLES flag, turn, pc
vars == <<flag, turn, pc>>

ProcSet == {0, 1}

Init ==
    /\ flag = [i \in ProcSet |-> FALSE]
    /\ turn = 0
    /\ pc = [i \in ProcSet |-> "start"]

Entry(i) ==
    /\ pc[i] = "start"
    /\ flag' = [flag EXCEPT ![i] = TRUE]
    /\ turn' = 1 - i
    /\ pc' = [pc EXCEPT ![i] = "gate"]

Gate(i) ==
    /\ pc[i] = "gate"
    /\ (~flag[1 - i] \/ turn /= 1 - i)
    /\ pc' = [pc EXCEPT ![i] = "crit"]
    /\ UNCHANGED <<flag, turn>>

Exit(i) ==
    /\ pc[i] = "crit"
    /\ flag' = [flag EXCEPT ![i] = FALSE]
    /\ pc' = [pc EXCEPT ![i] = "terminated"]
    /\ UNCHANGED turn

Proc(i) ==
    \/ Entry(i)
    \/ Gate(i)
    \/ Exit(i)

AllDone ==
    /\ \A i \in ProcSet: pc[i] = "terminated"
    /\ UNCHANGED vars

Termination ==
    <> (\A i \in ProcSet: pc[i] = "terminated")

Next ==
    \/ \E i \in ProcSet: Proc(i)
    \/ AllDone

Exclusivity == ~(pc[0] = "crit" /\ pc[1] = "crit")

Spec ==
    /\ Init
    /\ [][Next]_vars
    /\ \A i \in ProcSet: WF_vars(Proc(i))

====
```

Figure 4: Implementation of Peterson's algorithm in TLA$^+$