Performance Evaluation of Specification Types in NuSMV

Oana-Teodora Mirea[1]

Supervisor(s): Anna Lukina[1]**, Benedikt Ahrens**[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: Oana-Teodora Mirea
Final project course: CSE3000 Research Project
Thesis committee: Anna Lukina, Benedikt Ahrens, Tim Coopmans

Abstract

Systems are becoming increasingly complex and traditional software testing and manual proofs cannot keep up to ensure correctness. Model checking is a method of performing formal verification of a model by exhaustively checking the state-space. NuSMV is a widely used symbolic verifier and supports three different specification languages: computational tree logic (CTL), linear temporal logic (LTL), and property specification language (PSL). While previous benchmarks on NuSMV exist, they do not detail impacts of the specification language on performance. This paper investigates when each type of logic is most suitable. We evaluate three use cases and develop specifications for them, equivalent in all languages. We analyze runtime, memory and size of the internal representation. Our results show CTL is often faster than LTL and PSL, but memory usage varies by model. CTL performs better on simple global properties or those containing 'next' or 'until' operators. In contrast, LTL and PSL perform better on nested 'global' and 'eventually' operators. LTL and PSL perform nearly identically in all metrics. Bounded Model Checking (BMC) generally outperforms Binary Decision Diagrams-based verification, however, due to the bound, BMC does not guarantee global safety.

1 Introduction

Modern computers have to deal with complex systems composed of many concurrent processes which need to communicate and cooperate with one another. Traditional testing is often insufficient to keep up with the increasing complexity of such systems and to assert with certainty a model is correct [1]. Model checking addresses this issue by offering a method for formal verification, which exhaustively checks a system's state space to assert if a property holds. If the property does not hold, then the model produces a counterexample [2, p. 9].

NuSMV is a well-established symbolic model checker which is used in both research and industrial settings [3; 4]. The most recent version (2.7.1) supports specifying properties in three temporal logics: Linear Temporal Logic (LTL), Computational Tree Logic (CTL), and Property Specification Language (PSL). However, most published works focus on the tool's performance in comparison to other tools [5] or simply describe their experience of using the tool without providing details about the impact of the specification language [6]. To our knowledge, there is no study which systematically shows how the choice of specification type affects the verification efficiency within NuSMV.

We aim to answer the following question: as systems scale, which specification type performs best with respect to runtime and memory consumption? Our hypothesis is that PSL and LTL specifications perform similarly to one another, but they will experience a steeper increase in runtime and peak memory compared to equivalent CTL specifications. Within NuSMV, LTL is first converted to an automaton which is then reduced to CTL model checking [7, sec. 3]. PSL represents an extension of LTL with a few added operators, but it is solved via the same algorithm [1, sec. 1]. To answer the question we analyze three classical verification problems selected from NuSMV standard examples suite: a pipeline scheduling problem, a cache coherence protocol and a master bus arbitration protocol, each representative of real-life concurrent systems. The initial models were devised with a small number of agents, so we have written custom scripts to scale the problems accordingly, while still preserving underlying behavior. For each model, we express relevant properties, equivalent in all three specification languages and then measure runtime, peak memory, and the size of the underlying data structures.

This study provides an empirical perspective on the selection of specification types in NuSMV. Although the investigated use cases are not of industrial scale, which limits the generalization of our findings, the results provide insight into which specification is better suited for scalable verification tasks when expressing the same property.

The outline of the paper is as follows. Section 2 introduces the key concepts needed to understand the paper. In Section 3 we present previous research on temporal logics and the usage of NuSMV. Sections 4 and 5 describe the methodology and experiments. Section 6 presents the discussion of the results. We address reproducibility and AI usage in Section 7. Finally, Section 8 presents conclusions and suggestions for future work.

2 Preliminaries

This section aims to familiarize the reader with the main concepts and notations used in the paper, which form the basis for a comprehensive understanding of model checking. We begin by presenting model checking as a general concept, then go into the formal specifics of the tools (temporal logics and state-space representations) which are needed to understand the rest of the paper. Finally, we describe NuSMV, the tool we use throughout.

2.1 Model Checking

Model checking provides an automated and exhaustive approach to verification. It is a formal verification technique that systematically determines whether a finite-state program satisfies a specified property. Given a labeled state-transition graph modeling a system, the model checker exhaustively explores the state space to confirm or deny the property. If the specification is violated, the model checker produces a counterexample as an execution path from the initial state to a violating state [2, p. 9].

The properties model checkers are used to verify are those such as **safety** properties (nothing bad ever happens), **mutual exclusion** (two processes do not occupy a critical section simultaneously), **liveness** properties, including reachability (a desired state is eventually reached) and **bounded response** (a response occurs within a defined number of steps). **Fairness** and **precedence constraints** can also be encoded, ensuring all enabled processes eventually execute and events occur in the desired order [2, p. 12].

These properties are expressed as temporal logic formulas, which provide an intuitive language for describing the order of events over time. The following subsections introduce the three temporal logics supported by NuSMV (LTL, CTL, PSL) as well as how model checkers represent the states of a system.

2.2 Linear Temporal Logic

In linear temporal logic (LTL), each state has a unique possible successor. Therefore, LTL formulas are evaluated based on a single execution path of the system. It is especially suitable to express properties of sequential behavior, such as safety and liveness properties [2, p. 231].

The LTL formulas are composed from the set of atomic propositions using the usual Boolean connectives ($\wedge$, $\vee$, $\neg$, $\rightarrow$) as well as combinations of the temporal connectives shown in Table 1.

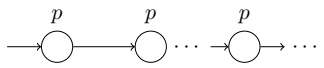
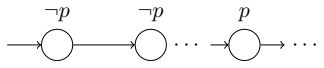
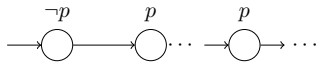
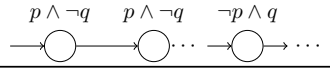
Temporal connectives	Formula	Trace
G - globally	Gp	
F - eventually	Fp	
X - next	Xp	
U - until	pUq	

Table 1: LTL Operators. Each node in a trace represents a state of the machine and the labels indicate which condition(s) hold in that state.

Some additional examples for LTL formulas are the following:

- $\mathbf{GF}p$ — p holds infinitely often; the system will always eventually return to a state where p is true.
- $\mathbf{F}(p\,\mathbf{U}\,q)$ — at some point in the future, p will hold continuously until q becomes true.
- **XF** ($\neg$p $\vee$ q) — eventually in the future, in the next state either p will not hold or q will hold

2.3 Computational Tree Logic

Unlike in a linear time perspective, in a branching-time view, each moment in time has multiple following possibilities simultaneously. CTL is useful when a property might happen across different execution branches, such as an error that is only reached in one specific trace. The CTL formulas allow for describing properties which consider the non-deterministic and branching evolution of a system [8, p. 13]. They extend LTL formulas with path quantifiers A and E as shown in Table 2.

However, when it comes to counterexamples, it is known that there are CTL formulas (ex: AFAXp), whose failure cannot be witnessed by a linear counterexample [9, sec. 2.4].

2.4 Property Specification Language

The third temporal logic is Property Specification Language (PSL). It is an extension of LTL with Sequential Extended Regular Expressions (SEREs), enabling concise expression of complex multi-cycle sequential behaviors, that would require deeply nested operators in LTL or CTL [10, p. 2].

Consider the following example: a request must be acknowledged within exactly three clock cycles. In LTL, this requires nesting next-state operators: $\mathbf{G}(req \rightarrow \mathbf{X}(\mathbf{X}(\mathbf{X}\,ack)))$.

In PSL, the same property is expressed concisely using a SERE: `always {req; true; true; ack}`.

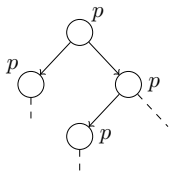
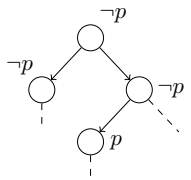
Temporal connectives	Formula	Trace
A - on all paths	AGp	
E - there exists a path	EFp	

Table 2: CTL operators

where the semicolons denote consecutive clock cycles, and the sequence reads directly as: `req` happens, then two arbitrary cycles, then the `ack` arrives. This is what is meant by a multi-cycle sequential behavior, where the truth of a property depends on the relationship between signals across multiple discrete time steps, expressed as a timeline.

2.5 Representing State Space

After we have established the temporal logics, the next questions is how the system's state space is encoded during the verification process. There are two primary techniques used for this, both for hardware and software designs:

Binary Decision Diagrams (BDD) is a graph-based data structure used to represent Boolean functions compactly. Instead of explicitly listing all possible states of a system, BDD represent large sets of states and transitions [11, sec. 2].

Boolean Satisfiability (SAT) represents a Boolean formula, usually in Conjunctive Normal Form, and is used to determine whether or not the formula can be made true. A common technique is via Bounded Model Checking (BMC) which checks whether or not the formula can be satisfied within a number k of steps [7, sec. 3].

Example: Consider the formula (p $\wedge$ q) $\vee$ $\neg$t

BDD: Represents the formula as the graph seen in Figure 1.

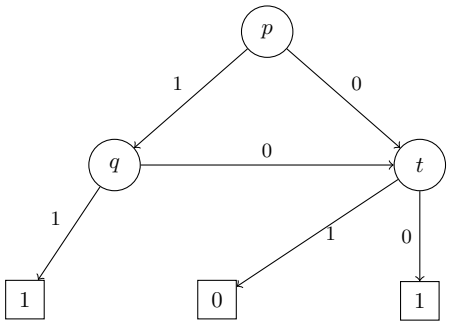


Figure 1: BDD for $f = (p \wedge q) \vee \neg t$, variable order $p \rightarrow q \rightarrow t$.

SAT: Transforms the formula to a CNF (p $\vee\neg$ t) $\wedge$ (q $\vee\neg$ t).

2.6 NuSMV

With the relevant logics and underlying representations defined, we can now properly introduce NuSMV. This tool is a re-implementation and extension of SMV, which is the first BDD-based symbolic model checker [7, p. 1]. It supports CTL, LTL, and PSL specifications, and has been designed as an open, extensible architecture suitable for both industrial verification and research [7, p. 1-4].

Symbolic model checking was introduced to combat the state-explosion issue which often arises in explicit-state model checking [12, p. 2-3]. Rather than enumerating all states, it encodes them as Boolean formulas, either as BDDs or SAT formulas, allowing for verification of significantly larger designs.

This paper analyses NuSMV 2.7.1. This version, however, supports PSL only partially, meaning that many of the features which give expressive advantage to PSL are not available [13, p. 41-45]. Despite this, since PSL is part of the tool, assessing performance relative to the other languages is still valuable.

3 Related Work

The question of whether linear-time or branching-time logic is the most appropriate framework for system verification has sparked a long-standing debate within the formal methods community. This chapter reviews the theoretical complexity of these paradigms, details the expressive power trade-offs between them, and discusses practical implementation and usage scenarios within symbolic model checker NuSMV.

3.1 Theoretical Analysis of Branching vs. Linear Time

The foundational complexity of model checking for linear and branching temporal logics is well established. If we are given a transition system of size n and a temporal logic formula of size m, model checking for the branching-time logic CTL can be computed efficiently in polynomial time specifically $\mathcal{O}(nm)$ [9, p. 3]. Similarly, evaluating the linear-time logic LTL is inherently more computationally expensive, resulting in a complexity bound of $n \cdot 2^{\mathcal{O}(m)}$. Because LTL model checking is a PSPACE-complete problem, this exponential bound cannot be lowered [9, p. 2-3].

This mathematical gap led early researchers to consider CTL superior on the basis that “while specifying is easier in LTL, verification is easier for CTL” [9, p. 1]. However, as argued by Vardi in his paper “Linear Time vs. Branching Time: The Showdown” [9], selecting a specification language does not only lie in the algorithmic optimization aspect. Although the restricted syntax of CTL guarantees linear-time model checking, it may heavily compromise expressive power, making it entirely incapable of specifying certain concurrent behaviors such as strong fairness constraints [9, p. 1]. In contrast, LTL offers an intuitive syntax that naturally supports formal verification, and a better integration for explicit and symbolic search systems.

PSL was designed as an extension of LTL, sharing its core theoretical complexity profile while adding SEREs and hardware-centric operators [10, p. 5]. Although PSL was designed to facilitate nested or multi-cycle requirements, benchmarking it against LTL and CTL is essential to determine whether it causes the model checker engine to run slower or consume more memory as the system scales.

3.2 Expressive Incomparability

Theoretically, LTL and CTL cannot always express the same requirements. Branching-time logic cannot capture linear fairness properties like $\mathbf{A}(\mathbf{FG}p)$, while linear-time logic cannot evaluate existential path properties like $\mathbf{EF}p$ [14, sec. 1]. However, most industrial properties are functionally linear. Nontrivial CTL formulas can be error-prone and difficult for engineers to formulate, so often time they are not used [9, sec. 2.1].

To make hardware verification more intuitive, organizations like IBM historically tried to "linearize" CTL syntax [9, sec. 2.1], which directly inspired Sugar, the precursor to PSL. The latter was introduced as hardware engineers naturally think in linear execution traces and clock cycles rather than branching trees [15].

3.3 NuSMV in Industrial Case Studies

The algorithmic implementation of these languages within the model checking tools heavily influences their performance profiles [5]. To execute an LTL or PSL property, NuSMV delegates the formula to an external module that translates the linear-time specification into a symbolic tableau or an infinite automaton. For PSL, the model checker first translates the expressions and multi-cycle conditions partially into an LTL and partially as symbolic infinite-automata [1, p. 3]. This automaton is then merged directly with the system's original variables, turning the property into an equivalent CTL check over an expanded state space [7; 16].

In broad verification benchmarks, NuSMV is a widely recognized tool used to verify critical, large-scale systems like railway signaling networks [4] and robotic material handlers [17]. While NuSMV is sometimes outperformed by other tools in highly concurrent or deeply unrolled scenarios, it has won a few competitions in the Hardware Model Checking Competition, but it has not participated since 2013, being replaced by its successor NuXMV [18].

Additionally, a significant amount of existing literature focuses on system architectures and compares performance across symbolic and explicit model checkers [4; 17; 5]. Very few studies systematically look at how the specification logic type itself impacts the efficiency of the model. For example, while in the paper "Experiences from Large-Scale Model Checking" [6, sec. 5] it is mentioned the team switched their specification language from CTL to LTL, it does not explain the reasoning or the performance impact behind that choice.

4 Methodology

We set to answer the question of which temporal logic performs best within the NuSMV 2.7.1 environment. This chapter details the evaluation framework and empirical approach used to analyze the runtime and memory complexities of CTL, LTL, PSL specifications. The primary objective is to evaluate how branching-time and linear-time temporal logics scale under increasing state-space dimensions.

4.1 Benchmark Selection and Scaling

To evaluate the specification languages across diverse configurations, we chose three classical verification problems from the standard NuSMV example suite:

1. **Pipeline Scheduling Model:** A real-time scheduling system analyzing priority-based preemption and data-driven pipelines.
2. **Cache Coherence Protocol:** A concurrent multi-agent system analyzing data consistency and race conditions.
3. **Master Bus Arriving Protocol:** A shared-resource hardware arbitration model analyzing handshake mechanics.

To simulate real-world usage, we parametrized the initial models and we scaled them using custom Python generator scripts (more about this in Section 7). The scaling process increases the number of agents (ex: the number of concurrent pipelines or bus agents) while preserving the core underlying behavior of the initial model. This enables a controlled analysis of how model checking backends respond as the state-space density increases.

4.2 Formal Property Specification

For each case study, we formally specified system requirements across all three temporal logics (CTL, LTL, and PSL). Properties are split into distinct behavioral classes, such as safety invariants, deadlock freedom, non-starvation (bounded liveness), and existential reachability.

We kept specifications equivalent to ensure a direct baseline comparison between branching-time and linear-time properties. Furthermore, we were wary of vacuously true statements (ex: an implication is true because the premise never happens) [19]. We added fairness constraints to ensure only paths where the relevant conditions hold are considered.

4.3 Evaluation Metrics

To measure performance, we used four base metrics, following standards from the Hardware Model Checking Competition [18]:

- **Runtime (Seconds):** The wall time dedicated to solving the target specification.
- **Peak Memory (MB):** The maximum RAM recorded during the verification life cycles.
- **BDD Size (Total Nodes):** Evaluated during symbolic runs, tracking the exact number of allocated nodes required to represent the transition relations and state formulas.
- **SAT Formula Complexity (Clauses):** Tracked during BMC routines for LTL and PSL, measuring the size of the boolean propositional formula generated when unrolling the execution loops into logic circuits.

There are a few optimization choices we made. For all problems, we ran them both without any flags and with flag `-coi`. The flag activates Cone of Influence reduction which is a technique to remove irrelevant variables and logic from the system before model checking a specification. We did not obtain better results from other flags. We only reported on the configuration with best results (least time and memory used), as flags do not always improve performance [6, sec. 6]. For all problems we disclose under which configurations they were run.

For BMC, we initially set the unrolling bound to size k=20 for all models. The default size value in NuSMV is 10, but a bound of 20 has been shown that it is often enough to conclude a lack of counterexamples for models of this size [12, p. 24-25]. However, for the Cache Protocol problem k was decreased to a value of 7 because the model ran early into the state explosion problem and the computation would not finish. However, even for the value of 7 patterns in performance were still visible. We keep the value of k consistent across each use case.

We run the model checker five times on each specification and report the median result in order to mitigate the effect of outliers.

4.4 Experimental Setup

We executed all experimental benchmarks in an isolated environment to ensure consistent CPU performance. The hardware and software specifications of the evaluation platform are detailed below:

- **Model Checker:** NuSMV (Version 2.7.1)
- **Operating System:** Windows 11 Home
- **Processor:** 13th Gen Intel Core i7-13700H @ 2.40GHz 14 Cores, 20 Logical Processors
- **Memory:** 16GB
- **Solvers Enabled:** CUDD (BDD Backend) / MiniSat (SAT BMC Backend)

5 Experiments

This chapter presents the experimental evaluation of the model checking backends using our parameterized suite. The goal of these experiments is to systematically compare the performance of CTL, LTL, and PSL, when verifying identical behavioral requirements under increasing state-space dimensions. Each benchmark model, its respective generation constraints, and the verified property suites are detailed below. We also provide tables with the results and highlight those which we discuss later in Section 6.

5.1 Pipeline model

The original pipeline model is taken from the NuSMV example suite[1], authored by Sergio Campos (1994). It models a real-time scheduling problem in which three data-driven pipelines share a single processor where pipelines with higher-priority can interrupt lower-priority ones in the middle of their execution. Each pipeline consists of three sequential stages, with periods of various time. A stage advances its internal state by one unit only when it has access to the processor, otherwise keeping its value to model preemption. The pipeline is data-driven which indicates that each stage is triggered by the completion of the previous one, rather than by an independent periodic timeout. The `error` is defined as a timeout going off while a stage is still being processed. The central property verified is `AG !error`.

We wrote an algorithm in order to scale the model for experimental evaluation, which is a Python generator script that

[1] https://nusmv.fbk.eu/examples.html

produces a NuSMV model with N pipelines, each still composed of three stages. The scaling complies to the following rules:

Periods grow with the pipeline index, grouped in sets of three. The first group uses periods of 20ms, 30ms, and 60ms; the second group doubles these and so on, with the multiplier increasing by one every three pipelines. This ensures that the lowest common multiple of all periods, which determines the length of the global timer and therefore the state space, grows in a controlled but increasingly demanding manner.

Execution times per stage are kept low (1-2ms per stage) and follow a rotating pattern across the three positions within each group, to avoid deadline misses that would make `AG !error` trivially false as the model scales, while still producing a non-trivial scheduling load.

The scheduler remains rate monotonic: pipelines with shorter periods are given higher priority in the `processor_granted` block, preserving preemptive behaviour.

Based on the criteria needed to guarantee reliability, we defined several specifications to evaluate the system. For a model with N pipelines, numbered 1 from N, these are the specifications: [2]

Safety: verifies no timing deadline is violated across the execution path:

```
SPEC AG !error;
```

```
LTLSPEC G !error;
```

```
PSLSPEC always (!error);
```

Deadlock freedom: ensures the system does not freeze, checking that the highest priority task continues to request and receive processor cycles infinitely often:

```
SPEC AG AF (P_1_1.state > 0 | !P_1_1.request);
```

```
LTLSPEC G F (P_1_1.state > 0 | !P_1_1.request);
```

```
PSLSPEC always eventually! (P_1_1.state > 0 | !P_1_1.
request);
```

Non-starvation (Bounded Liveness): checks that whenever the lowest-priority pipeline makes a processor request, it is guaranteed to eventually execute, pass data downstream, and successfully cycle back to an idle state:

```
SPEC AG (P_N_1.request -> AF (P_N_3.finish & AF (P_N_3.
state = 0)));
```

```
LTLSPEC G (P_N_1.request -> F (P_N_3.finish & F (P_N_3.
state = 0)));
```

```
PSLSPEC always (P_N_1.request -> eventually! (P_N_3.
finish & eventually! (P_N_3.state = 0)));
```

Cross-preemption:verifies Rate Monotonic priority: whenever a high-priority process requests, the CPU is not held by a low-priority process:

```
SPEC AG (P_1_1.request -> processor_granted != p_4_1 &
processor_granted != p_4_2 & processor_granted != p_4_3);
```

```
LTLSPEC G (P_1_1.request -> processor_granted != p_4_1 &
processor_granted != p_4_2 & processor_granted != p_4_3);
```

[2]P_a_b denotes stage b of pipeline a.

```
PSLSPEC always (P_1_1.request -> processor_granted !=
p_4_1 & processor_granted != p_4_2 & processor_granted !=
 p_4_3);
```

We present Tables 3 and 4 here with some significant results, while the others are added to Appendix A for the sake of brevity. We did not enable flag `-coi` and used k=20 for BMC.

Scale (n)	Spec Type	Runtime (s)	Peak memory (mb)	BDD size (nodes)
Pipelines-10	CTL	0.27	15.17	198 268
	LTL	0.56	30.63	421 064
	PSL	0.56	30.63	421 064
Pipelines-15	CTL	**1.55**	**20.33**	348 502
	LTL	**2.18**	**35.57**	811 468
	PSL	**2.25**	**35.57**	811 468
Pipelines-18	CTL	**14.00**	**52.86**	1 080 254
	LTL	**3.63**	**44.66**	1 054 704
	PSL	**3.74**	**44.66**	1 054 704

Table 3: Non-starvation properties - BDD

Scale (n)	Spec Type	Runtime (s)	Peak Memory (MB)	SAT Clauses
Pipelines-10	LTL	**1.02**	39.78	26 115
	PSL	**0.62**	39.41	26 115
Pipelines-15	LTL	**2.65**	54.80	51 652
	PSL	**1.90**	54.58	51 652
Pipelines-18	LTL	**2.90**	62.11	62 842
	PSL	**2.28**	60.75	62 842

Table 4: Non-starvation properties - BMC

5.2 Cache Coherence Protocol

The original model is taken from NuSMV's collection of standard examples, but the author is unknown. This Cache Coherence Protocol represents a version with three processors and one memory module, which coordinate access to the data via a shared bus. Each cache can be in one of three states: `invalid` (data is stale and cannot be used), `shared` (data is readable by multiple processors) or `owned` (only one processor has write access). The protocol ensures consistency by distributing access rights and different signals such that the processors can `snoop` to know if they should update or discard their version of the data. At most one agent can become bus master at a time, for example: `p0` may become master freely, while each subsequent processor can only do so if none of the previous ones is currently master.

When writing the script to scale the model to N processors, we had to ensure it remains correct and consistent. In order to do this, we abided by the following rules:

Processors' behavior remains the same as in the original model. Instances are renamed `p0` to `p(N-1)`.

Assigning Master is achieved by a longer cascade of case statements, where each processor checks if any of the precedent processors is a master (`pN` checks for `p0` to `p(N-1)`). If not, then it might become a master itself.

Global Bus Signals (`OWNED`, `WAITING`, `STALL`) are computed with the OR operation across all N processors to check if any signal was sent, so the bus can always reliably transmit the same information to all processor.

We wrote the following specifications to verify that processors will never use outdated data:

Bounded Response: when the command `invalidated` is broadcast, then in the next step all processor caches must know they own a stale version copy:

```
SPEC AG (CMD = invalidate -> AX (p0.state = invalid & ...
 p(N-1).state =invalid))
```

```
LTLSPEC G (CMD = invalidate -> X (p0.state = invalid & ...
 p(N-1).state =invalid))
```

```
PSLSPEC always (CMD = invalidate -> next! (p0.state =
invalid & ... p(N-1).state =invalid));
```

Mutual Exclusion: for every processor pair, if one of them has write access then no other one can have write access at the same time:

```
SPEC AG !(pi.writable & (p(i+1).writable | p(i+2).
writable | ... | p(N-1).writable))
```

```
LTLSPEC G !(pi.writable & (p(i+1).writable | p(i+2).
writable | p(i+1).writable | ... | p(N-1).writable))
```

```
PSLSPEC always !(pi.writable & (p(i+1).writable | p(i+2).
writable | ... | p(N-1).writable))
```

Safety Invariant: when the signal `write-resp-invalid` is on the bus, which indicates a processor wants to write, any processor that owns the `state` must be the current bus master:

```
SPEC AG (CMD = write-resp-invalid -> (p(N-1).state =
owned -> p(N-1).master))
```

```
LTLSPEC G (CMD = write-resp-invalid -> (p(N-1).state =
owned -> p(N-1).master))
```

```
PSLSPEC always (CMD = write-resp-invalid -> (p(N-1).state
 = owned -> p(N-1).master));
```

Tables 5 and 6 are shown here for a brief indication of results, while all the others have been included in the Appendix A. We did not enable flag `-coi` and used k=7 for BMC.

Scale (n)	Spec Type	Runtime (s)	Peak Memory (MB)	BDD size (nodes)
Cache-10	CTL	0.32	42.14	764 456
	LTL	0.48	48.36	954 548
	PSL	0.50	48.36	954 548
Cache-15	CTL	**1.17**	58.00	1 206 982
	LTL	**2.68**	57.72	1 223 334
	PSL	**2.70**	57.72	1 223 334
Cache-20	CTL	**4.38**	**107.73**	2 717 498
	LTL	**13.27**	**80.28**	1 891 722
	PSL	**15.63**	**80.28**	1 891 722

Table 5: Mutual Exclusion - BDD

Scale (n)	Spec Type	Runtime (s)	Peak Memory (MB)	SAT Clauses
Cache-10	LTL	**10.07**	32.94	11 020
	PSL	**4.24**	34.21	11 020
Cache-15	LTL	**27.07**	40.92	16 530
	PSL	**14.34**	43.71	16 530
Cache-20	LTL	**87.05**	53.20	22 240
	PSL	**25.99**	57.39	22 240

Table 6: Mutual Exclusion - BMC

5.3 PCI Bus Protocol

The initial PCI model similarly comes from the NuSMV curated examples, written by Sergio Campos (1995). It models a hardware-level PCI Bus Protocol about transactions and permissions to the bus, to analyze for latency, fairness and safety properties of the protocol. It is split into two main components: a `hierarchical arbiter` and a set of `bus masters`. The first one controls which devices get access to the shared bus. It is organized into two levels: three leaf banks (`bank0`, `bank1`, `bank3`), and a central bank (`bank2`). Each leaf bank arbitrates between a pair of masters using either Fixed Priority (FP) or Round Robin (RR) policy and the central banks issues the final global grant given. The `bus_masters` act as individual agents, each with three possible states: `idle`, `address`, `data`. There are also two passive masters (namely `bus_master_null`) which do not make requests for the bus. The shared bus signals (`b_frame`, `b_irdy`, and `b_trdy`) are calculated via OR-ing the corresponding signals across all masters to indicate if a transaction is in progress, if the initiating master is ready to transfer data, or if the target is ready to receive it.

We designed an algorithm to scale the model to an arbitrary number N of bus masters, each still having three states. The scaling was done following these rules:

Arbiter maintains the split into two tiers. Masters are paired into leaf banks of two, and the central bank has to choose between the winners of all leaf banks. This preserves the original Fixed Priority and Round Robin policies. For odd N, the last leaf bank handles a single master, so in order to avoid this all use cases will have an even number of masters.

Bus masters are kept like the original `bus_master` and `bus_master_null` as they are already parametrized with the master id. However, the masters have been renamed to `master0`, ... `master(N-1)`. Master indices 3 and 5 are preserved as being null, and we added one extra null master for each scale to avoid state space explosion.

Transaction frames were decreased from 15 to 3 to prevent tremendous state explosion as the number of masters increases, while still preserving the abort behavior of the initial model.

The first two specifications we wrote for this model have been inspired by some of the specifications suggested through comments in the original document, whereas the other two are new:

Safety (Invariant): verifies that whenever a rising edge on `b_frame` is detected by `b_frame_switch`, the frame signal itself is indeed active. This ensures the the bus state is consistent with the detection logic.

```
SPEC AG(b_frame_switch -> b_frame)
```

```
LTLSPEC G(b_frame_switch -> b_frame)
```

```
PSLSPEC always (b_frame_switch -> b_frame);
```

Liveness (Eventual Service): checks that under a Round Robin policy, a master which raises a requests will eventually get access to the bus. This ensures the non-starvation policy of RR arbitration:

```
SPEC  AG((master(N-1).req & arb.policy_central=RR) -> A[
master(N-1).req U arb.grant=N-1])
```

```
LTLSPEC  G((master(N-1).req & arb.policy_central=RR) -> (
master(N-1).req U arb.grant=N-1))
```

```
PSLSPEC always ((master(N-1).req & arb.policy_central =
RR) -> (master(N-1).req until! arb.grant = N-1));
```

Safety (Priority Precedence): verifies that when a master raises a request, it will remain active until it is either granted the bus or a master with higher priority claims the request:

```
SPEC AG(master(N-1).req -> A[master(N-1).req U (grant=(N
-1) | master(N-2).req | ... | master0.req)])
```

```
LTLSPEC G(master(N-1).req -> (master(N-1).req U (grant=N
-1 | master(N-2).req | ... | master0.req)))
```

```
PSLSPEC always (master(N-1).req -> (master(N-1).req until
! (grant=N-1 | master(N-2).req | ... | master0.req)));
```

Mutual exclusion: affirms that no two masters obtained the `frame` signal at the same time, granting them access of the bus simultaneously. This is checked for all pairs $i \neq j, 0 \leq i, j \leq N$:

```
SPEC AG !(master_i.frame & master_j.frame)
```

```
LTLSPEC G!(master_i.frame & master_j.frame)
```

```
PSLSPEC always !(master_i.frame & master_j.frame);
```

Some results have been provided below in Tables 7 and 8 while the rest can be found in Appendix A. We enabled flag `-coi` and used k=20 for BMC.

Scale (n)	Spec Type	Runtime (s)	Peak Memory (MB)	BDD size (nodes)
Masters-6	CTL	0.14	15.94	221 774
	LTL	0.10	11.72	94 029
	PSL	0.09	11.72	94 029
Masters-8	CTL	0.62	50.95	1 031 198
	LTL	6.09	57.61	1 229 466
	PSL	5.88	57.61	1 229 466
Masters-10	CTL	**23.78**	**67.79**	1 530 956
	LTL	**119.69**	**63.13**	1 385 832
	PSL	**138.86**	**63.10**	1 386 854

Table 7: Liveness - BDD

Scale (n)	Spec Type	Runtime (s)	Peak Memory (MB)	SAT Clauses
Masters-6	LTL	**0.10**	20.28	1
	PSL	**0.07**	19.25	1
Masters-8	LTL	**18.52**	48.11	15 067
	PSL	**10.07**	48.69	15 081
Masters-10	LTL	**13.14**	54.52	19 025
	PSL	**6.94**	53.55	19 018

Table 8: Liveness - BMC

6 Discussions

We will now analyze the data we have collected from our experiments. The results across all three models broadly support the hypothesis that CTL outperforms LTL and PSL in terms of both runtime and memory, and also that LTL and PSL behave similarly. However, the results reveal that this is not a universal pattern, and the relation between performance and specification type is dependent on the structural characteristics of the model which is being checked. We will detail below the most relevant findings in these use cases.

6.1 CTL vs. PSL and LTL

First we present some general conclusions, then we showcase some examples where CTL outperformed LTL and PSL, followed by a few cases where the latter did best. CTL consistently achieves lower runtime and less memory usage than LTL and PSL in most conditions, however the scale by which it does so can vary. We observe throughout our models that CTL does better when the property which is being verified contains only one global temporal operator ("G"). These differences can probably be attributed to structural issues. CTL operates directly on the state-space via computations on the BDD, whereas NuSMV first transforms LTL into an infinite automaton, which can often introduce overhead. Additionally, when the specification includes temporal operators such as the "X" (next) and "U" (until) CTL stands out for its fast execution time.

Take for example the Cache Model with 20 processors for Mutual Exclusion (Table 5): CTL requires ∼4s while LTL/PSL take ∼13/∼15s. The property checks that no two processors are writable simultaneously, and must hold throughout the model and lacks temporal depth (does not depend on previous or following states). However, the memory and BDD size for CTL is significantly larger than that for LTL/PSL, 107MB and 2.7m nodes compared to 80MB and 1.8m nodes. Possibly the BDD for CTL creates cross-product states (for each group of processors) that the automaton must explicitly track. However, LTL first builds an automaton during preprocessing, which might lead to a more compressed BDD, but this adds a lot of overhead to the computation. A similar pattern appears for the Bounded Response property as well (Table 17), where CTL is almost twice as fast but consumes more memory. This property has more temporal depth because it involves the "X" operator, requiring the tool to track all of the states as they must become "invalid" upon receiving a signal. Two other cases stand out involving the "U" operator. In both Liveness and Priority Precedence for the Bus Arbitration protocol (Tables 7, 27), CTL does significantly better (ex: 1.4s/180s and almost uses half less memory: 58MB/92MB,

1.2m/2.2m nodes). This is probably because for "U" formulas, the generated automata must simultaneously track whether the left-hand condition has been holding before the right-hand condition is reached across all interleavings which leads to a complex structure. When presented with the 10 masters model, it produces a far larger state space for Priority Precedence.

Conversely, there are a few models where LTL/PSL perform better than CTL, though the differences are smaller. In the Bus Arbitration Safety invariant (Table 23), CTL takes ∼1.9s and 60MB at Masters-10 versus ∼0.25s and 25MB for LTL/PSL, since the invariant (`b_frame_switch -> b_frame`) seems to produce a compressed and simple automaton with little overhead. The most significant gap appears in the Pipelines model for Non-starvation property (∼14s vs. ∼3.6s) (Table 3), where the property contains two nested "F" (eventually) operators. We hypothesize that the automaton built only needs to track whether each operation has been fulfilled, whereas CTL must perform nested computations over the state-space of the entire BDD. Thus, LTL prevents the unnecessary repetitions.

6.2 PSL vs. LTL

LTL and PSL produce almost identical BDD sizes and SAT clauses, due to NuSMV's similar internal representation of the languages. Consequently, their runtimes and memory usages are also very close in most cases (less than 1-2s difference). However, if there is a discrepancy, we notice PSL to be faster usually.

Such cases are usually via BMC (ex: Tables 4, 16, 22). For Cache Protocol, Mutual Exclusion (Table 6) PSL is checked in less than half the time needed for LTL across all scalings, with the last scaling even providing ∼26s for PSL and ∼87s for LTL. Similar runtime differences can be observed for Liveness (Table 8) and Priority Precedence (Table 28) for Bus Arbitration. We believe that the module which transforms PSL into LTL formulas performs certain optimization steps. However, we cannot find a clear pattern between the specifications and to the extend of our knowledge there is no detailed description of NuSMV's internals to assert for sure why BMC is more advantageous for PSL.

In a few cases, LTL is observed to perform better. The two most noticeable cases are Liveness (Table 7) and Priority Precedence (Table 27) for BDD within the Bus Arbiter, both properties containing the "U" operator. For masters-10, we obtained: ∼119s versus ∼138s for Liveness and ∼184s versus ∼224s for Priority Precedence. This might be due to PSL's syntax ("until!") which might add overhead to constructing the automata when using BDDs. LTL formulas go through intermediate normalization steps (ex: rewriting into negation normal form, handling implications) [2] which could simplify the automata, as the LTL specifications also use fewer nodes.

6.3 BMC vs. BDD

Bounded Model Checking unrolls the relations up to a fixed bound k and checks satisfiability via a SAT solver, making it faster and more efficient memory wise than BDD checking. However, it does not provide guarantees beyond that bound, so the value of k must be chosen large enough that the system can be considered correct. We have chosen k of size 20 for use cases 1 and 3 as it is a large enough value for our models and we narrowed down to 7 for Cache Protocol as we ran into the state explosion problem early on with larger values. Overall, performance under BMC is faster, but this comes at a trade-off of guaranteed safety.

In the Pipelines model, BDD and BMC results are mostly similar, with differences up to 2 seconds, but BMC uses more memory on average (ex: Tables 3 vs. 4, 11 vs. 12). Within the Cache Coherence Protocol for Mutual Exclusion, LTL with BMC completes in ∼87s at Cache-20 against ∼13s under BDD, while using less memory (53MB vs 80MB) (Tables 5, 6). Similar results can be found across all scalings. Likewise, BMC is slower for the Mutual Exclusion property for Bus Arbitration (Tables 29, 30). Performance can probably be attributed to the difference in processing of the pairwise comparisons that the formulas require. However, there are some cases in which BMC is faster and uses less memory, such as Bounded Response property within Cache Protocol (Tables 17, 18). This could be due to the usage of the "X" operator; the SAT instance that it is unrolled into will be of a more manageable size compared to the representation of the BDD. A greater difference shows up once again in the Bus Arbitration protocol for Liveness and Priority Precedence, involving the "U" operator. BDD times reach ∼180-220s at Masters-10, while BMC completes in ∼11s and ∼5s, although at almost double the memory cost for Priority Precedence (55MB vs 92MB). This reinforces the idea that "U" formulas are expensive for BDD computation but they are more manageable time-wise when unrolled into SAT instances.

6.4 Limitations

The models we investigated are derived from diverse academic benchmarks and, while representative of standard system patterns, do not reach industrial scales. Additionally, the machine we used has limited capabilities, so we needed to bound the models in order to avoid state explosion. The upper bounds were selected to be large enough to reveal meaningful performance trends, while keeping runtimes within a practical limit.

Moreover, the generator scripts that produce the scaled models have certain design choices (ex: period groupings, arbiter topology) which can impact the structure of the state space and may unintentionally favor one verification approach over another. Similarly, we chose the specifications to be some of the most relevant for each use case, but undoubtedly we could have added many others, which might have impacted the performances in unexpected ways.

7 Responsible Research

Throughout the entire project we pursued to maintain proper responsible research standards. We will present our considerations in this section: steps taken to allow for reproducing our results (Section 7.1) and how AI was used in the project (Section 7.2).

7.1 Reproducibility

To ensure our study is reproducible, we have provided all details for the experiments in Section 4 (Methodology) and 5 (Experimental Setup). These include the details of our machine, the initial use cases chosen, the steps taken for scaling them and the added specifications. Additionally, the code used to scale the original models have been made available in a public GitHub

repository [3]. Inside, there is also a README file detailing what commands were used for running NuSMV and obtaining all the metrics.

7.2 Use of AI in The Project

Within this project, AI was used in the initial phases of the project to explain SMV code and to help familiarise with some of the concepts which were introduced. Additionally, it was used to provide packages for LaTeX and help fix some of the formatting issues encountered.

8 Conclusions and Future Work

8.1 Main Conclusions

We set out to analyze the performance differences between CTL, PSL and LTL within the NuSMV tool. We have chosen three models based on standard verification benchmarks that are representative of real-life systems: pipeline modeling, a cache protocol, and a bus arbiter. We scaled each of them, while preserving the underlying logic, and wrote a set of relevant specifications in all the supported languages, which the model should comply to. Afterwards, we ran them inside the NuSMV tool and noticed several patterns emerging within our use cases. A general pattern is that CTL outperformed both in time and memory, especially on simple properties that just had a global ("G") quantifier. The addition of "X" and "U" operators also created a lot of overhead for LTL/PSL. In general, LTL and PSL have the same performance due to NuSMV internals treating PSL as an LTL formula [13, p. 45]. Still, for BMC PSL is observed to perform almost twice as fast as LTL on several cases. In BDD variations, LTL and PSL mostly do not have significant differences between them, so choosing one or the other is strictly based on preference of expressive capabilities, at least in this version of NuSMV (2.7.1) with limited PSL functionality. BMC also is generally faster than BDD, but the depth of the unrolling influences how much trust you can add into the model being correct.

8.2 Future work

To extend the research done in this paper, similar studies should be reproduced on industrial scale systems, where discrepancies between the performance of the three specification languages would become more apparent. It would also be valuable if authors disclosed all details on how NuSMV was used when publishing papers which discuss the incorporation of the tool.

Moreover, future studies could look at classes of specifications (such as safety or invariants) and compare CTL, LTL, and PSL over those categories. Such analysis may reveal whether certain specification languages are better suited for particular classes of properties.

References

[1] T. Launiainen, "Model Checking PSL Safety Properties," *978-952-248-041-5*, 01 2009.

[2] C. Baier and J.-P. Katoen, *Principles of model checking*, ser. The MIT Press Ser. Cambridge, Massachusetts London, England: The MIT Press, 2008.

[3] A. Classen, "CTL model checking for software product lines in NuSMV," Nov 2012. [Online]. Available: https://researchportal.unamur.be/en/publications/ctl-model-checking-for-software-product-lines-in-nusmv

[4] A. Ferrari and M. H. T. Beek, "Formal Methods in Railways: A Systematic Mapping Study," *ACM Computing Surveys*, vol. 55, no. 4, pp. 1–37, Apr. 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3520480

[5] M. Frappier, B. Fraikin, R. Chossart, R. Chane-Yack-Fa, and M. Ouenzar, "Comparison of Model Checking Tools for Information Systems," in *Formal Methods and Software Engineering*, J. S. Dong and H. Zhu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, vol. 6447, pp. 581–596, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-642-16901-4_38

[6] J. Fritzsch, T. Schmid, and S. Wagner, "Experiences from Large-Scale Model Checking: Verification of a Vehicle Control System."

[7] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking," in *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, ser. LNCS, vol. 2404. Copenhagen, Denmark: Springer, Jul. 2002.

[8] R. Cavada, A. Cimatti, G. Keighren, E. Olivetti, M. Pistore, and M. Roveri, *NuSMV 2.7 Tutorial*. [Online]. Available: https://nusmv.fbk.eu/tutorial/v27/tutorial.pdf

[9] M. Y. Vardi, "Branching vs. Linear Time: Final Showdown," in *Tools and Algorithms for the Construction and Analysis of Systems*, G. Goos, J. Hartmanis, J. Van Leeuwen, T. Margaria, and W. Yi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, vol. 2031, pp. 1–22, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/3-540-45319-9_1

[10] "IEEE standard for property specification language (PSL)," *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)*, pp. 1–182, 2010.

[11] E. M. Clarke, O. Grumberg, and K. Hamaguchi, "Another Look at LTL Model Checking," 1 1994. [Online]. Available: https://kilthub.cmu.edu/articles/journal_contribution/Another_Look_at_LTL_Model_Checking/6603527

[12] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded Model Checking," *Advances in Computers*, vol. 58, pp. 117–148, 2003.

[13] R. Cavada and A. Cimatti, "Nusmv 2.7.0 user manual." [Online]. Available: https://nusmv.fbk.eu/userman/v27/nusmv.pdf

[14] E. M. Clarke and I. A. Draghicescu, "Expressibility Results for Linear-time and Branching-time Logics," 1 1989. [Online]. Available: https://kilthub.cmu.edu/articles/journal_contribution/Expressibility_Results_for_Linear-time_and_Branching-time_Logics/6605483

[3] https://github.com/omirea/Scaled-models-for-NuSMV

[15] "The Development of PSL/Sugar — doulos.com," https://www.doulos.com/knowhow/psl/the-development-of-pslsugar/, [Accessed 02-06-2026].

[16] K. Rozier and M. Vardi, "LTL Satisfiability Checking," *STTT*, vol. 12, pp. 123–137, 01 2010.

[17] G. Lilli, M. Xavier, E. Le Priol, V. Perret, T. Liakh, R. Oboe, and V. Vyatkin, "Formal Verification of the Control Software of a Radioactive Material Remote Handling System, Based on IEC 61499," *IEEE Open Journal of the Industrial Electronics Society*, vol. 4, pp. 417–431, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10268612/

[18] "Hardware Model Checking Competition 2015 — fmv.jku.at," https://fmv.jku.at/hwmcc15/, [Accessed 02-06-2026].

[19] O. Kupferman and M. Vardi, *Vacuity Detection in Temporal Model Checking*. Springer-Verlag, 12 2000, vol. 4.

A Tables with Experiment Results

A.1 Data-driven Pipeline Model

Scale (n)	Spec Type	Runtime (s)	Peak memory (mb)	BDD size (nodes)
Pipelines-10	CTL	0.09	13.04	132 860
	LTL	0.20	14.85	188 048
	PSL	0.19	14.85	188 048
Pipelines-15	CTL	**0.92**	**19.93**	**341 348**
	LTL	**2.46**	**28.53**	**598 892**
	PSL	**3.17**	**28.68**	**598 892**
Pipelines-18	CTL	**1.20**	**42.9**	**1 028 132**
	LTL	**4.72**	**30.26**	**649 992**
	PSL	**4.02**	**30.26**	**649 992**

Table 9: Safety properties - BDD

Scale (n)	Spec Type	Runtime (s)	Peak Memory (MB)	SAT Clauses
Pipelines -10	LTL	1.01	39.71	26 837
	PSL	0.59	37.80	26 837
Pipelines -15	LTL	3.20	67.88	59 061
	PSL	2.22	66.59	59 061
Pipelines -18	LTL	4.01	75.12	70 203
	PSL	2.30	72.20	70 203

Table 10: Safety properties - BMC

Scale (n)	Spec Type	Runtime (s)	Peak memory (MB)	BDD size (nodes)
Pipelines-10	CTL	0.10	12.89	128 772
	LTL	0.13	10.91	69 496
	PSL	0.14	10.91	69 496
Pipelines-15	CTL	1.00	19.51	328 062
	LTL	1.98	20.31	351 568
	PSL	1.79	20.31	351 568
Pipelines-18	CTL	1.23	42.27	1 010 758
	LTL	2.50	22.38	419 020
	PSL	2.56	22.38	419 020

Table 11: Deadlock freedom properties - BDD

Scale (n)	Spec Type	Runtime (s)	Peak Memory (MB)	SAT Clauses
Pipelines -10	LTL	0.14	24.96	1
	PSL	0.13	25.63	1
Pipelines -15	LTL	0.97	35.70	1
	PSL	0.96	30.59	1
Pipelines -18	LTL	1.01	39.32	1
	PSL	0.99	38.81	1

Table 12: Deadlock freedom properties - BMC

Scale (n)	Spec Type	Runtime (s)	Peak memory (mb)	BDD size (nodes)
Pipelines-10	CTL	0.27	15.17	198 268
	LTL	0.56	30.63	421 064
	PSL	0.56	30.63	421 064
Pipelines-15	CTL	**1.55**	**20.33**	348 502
	LTL	**2.18**	**35.57**	811 468
	PSL	**2.25**	**35.57**	811 468
Pipelines-18	CTL	**14.00**	**52.86**	1 080 254
	LTL	**3.63**	**44.66**	1 054 704
	PSL	**3.74**	**44.66**	1 054 704

Table 13: Non-starvation properties - BDD

Scale (n)	Spec Type	Runtime (s)	Peak Memory (MB)	SAT Clauses
Pipelines-10	LTL	**1.02**	39.78	26 115
	PSL	**0.62**	39.41	26 115
Pipelines-15	LTL	**2.65**	54.80	51 652
	PSL	**1.90**	54.58	51 652
Pipelines-18	LTL	**2.90**	62.11	62 842
	PSL	**2.28**	60.75	62 842

Table 14: Non-starvation properties - BMC

Scale (n)	Spec Type	Runtime (s)	Peak memory (mb)	BDD size (nodes)
Pipelines-10	CTL	0.10	12.89	128 772
	LTL	0.14	10.88	68 474
	PSL	0.13	10.88	68 474
Pipelines-15	CTL	0.98	19.51	328 062
	LTL	1.81	20.29	351 568
	PSL	1.77	20.29	351 568
Pipelines-18	CTL	1.23	42.27	1 010 758
	LTL	2.06	22.38	419 020
	PSL	2.10	22.38	419 020

Table 15: Cross-preemption properties - BDD

Scale (n)	Spec Type	Runtime (s)	Peak Memory (MB)	SAT Clauses
Pipelines -10	LTL	0.90	32.24	22 627
	PSL	0.50	31.90	22 627
Pipelines -15	LTL	2.41	46.61	47 329
	PSL	1.81	46.42	47 329
Pipelines -18	LTL	2.60	52.84	58 051
	PSL	2.00	51.73	58 051

Table 16: Cross-preemption properties - BMC

A.2 Cache Coherence Protocol

Scale (n)	Spec Type	Runtime (s)	Peak Memory (MB)	BDD size (nodes)
Cache-10	CTL	0.38	46.53	897 316
	LTL	0.41	47.07	913 668
	PSL	0.44	47.07	913 668
Cache-15	CTL	**1.75**	**72.41**	1 642 354
	LTL	**2.68**	**57.75**	1 225 378
	PSL	**2.87**	**57.75**	1 225 378
Cache-20	CTL	**5.90**	**112.78**	2 873 864
	LTL	**10.85**	**76.05**	1 759 884
	PSL	**10.79**	**76.05**	1 759 884

Table 17: Bounded Response - BDD

Scale (n)	Spec Type	Runtime (s)	Peak Memory (MB)	SAT Clauses
Cache-10	LTL	0.99	31.52	12 525
	PSL	0.41	32.10	12 525
Cache-15	LTL	1.57	38.12	18 680
	PSL	0.88	40.06	18 680
Cache-20	LTL	2.33	48.58	25 035
	PSL	1.17	49.10	25 035

Table 18: Bounded Response - BMC

Scale (n)	Spec Type	Runtime (s)	Peak Memory (MB)	BDD size (nodes)
Cache-10	CTL	0.32	42.14	764 456
	LTL	0.48	48.36	954 548
	PSL	0.50	48.36	954 548
Cache-15	CTL	**1.17**	58.00	1 206 982
	LTL	**2.68**	57.72	1 223 334
	PSL	**2.70**	57.72	1 223 334
Cache-20	CTL	**4.38**	**107.73**	2 717 498
	LTL	**13.27**	**80.28**	1 891 722
	PSL	**15.63**	**80.28**	1 891 722

Table 19: Mutual Exclusion - BDD

Scale (n)	Spec Type	Runtime (s)	Peak Memory (MB)	SAT Clauses
Cache-10	LTL	**10.07**	32.94	11 020
	PSL	**4.24**	34.21	11 020
Cache-15	LTL	**27.07**	40.92	16 530
	PSL	**14.34**	43.71	16 530
Cache-20	LTL	**87.05**	53.20	22 240
	PSL	**25.99**	57.39	22 240

Table 20: Mutual Exclusion - BMC

Scale (n)	Spec Type	Runtime (s)	Peak Memory (MB)	BDD size (nodes)
Cache-10	CTL	0.28	42.11	763 434
	LTL	0.32	43.11	797 160
	PSL	0.39	43.11	797 160
Cache-15	CTL	1.08	57.97	1 205 960
	LTL	1.48	57.65	1 223 334
	PSL	1.63	57.65	1 223 334
Cache-20	CTL	5.04	107.99	2 726 696
	LTL	12.26	70.58	1 593 298
	PSL	8.15	70.58	1 593 298

Table 21: Safety Invariant - BDD

Scale (n)	Spec Type	Runtime (s)	Peak Memory (MB)	SAT Clauses
Cache-10	LTL	0.86	29.38	10 768
	PSL	0.44	30.45	10 768
Cache-15	LTL	1.82	35.57	16 133
	PSL	1.32	37.20	16 133
Cache-20	LTL	2.99	43.63	21 698
	PSL	1.51	46.95	21 698

Table 22: Safety Invariant - BMC

A.3 Master Bus Arbitration Protocol

Scale (n)	Spec Type	Runtime (s)	Peak Memory (MB)	BDD size (nodes)
Masters-6	CTL	0.13	16.51	216 664
	LTL	0.08	11.55	88 914
	PSL	0.09	11.55	88 914
Masters-8	CTL	0.26	18.39	295 358
	LTL	0.14	12.00	102 200
	PSL	0.14	12.00	102 200
Masters-10	CTL	1.85	60.87	1 271 368
	LTL	0.24	25.09	496 692
	PSL	0.26	25.09	496 692

Table 23: Safety Invariant - BDD

Scale (n)	Spec Type	Runtime (s)	Peak Memory (MB)	SAT Clauses
Masters-6	LTL	0.17	21.70	1
	PSL	0.08	19.36	1
Masters-8	LTL	0.22	21.73	1
	PSL	0.14	21.34	1
Masters-10	LTL	0.10	23.86	1
	PSL	0.17	21.28	1

Table 24: Safety Invariant - BMC

Scale (n)	Spec Type	Runtime (s)	Peak Memory (MB)	BDD size (nodes)
Masters-6	CTL	0.14	15.94	221 774
	LTL	0.10	11.72	94 029
	PSL	0.09	11.72	94 029
Masters-8	CTL	0.62	50.95	1 031 198
	LTL	6.09	57.61	1 229 466
	PSL	5.88	57.61	1 229 466
Masters-10	CTL	**23.78**	**67.79**	1 530 956
	LTL	**119.69**	**63.13**	1 385 832
	PSL	**138.86**	**63.10**	1 386 854

Table 25: Liveness - BDD

Scale (n)	Spec Type	Runtime (s)	Peak Memory (MB)	SAT Clauses
Masters-6	LTL	**0.10**	20.28	1
	PSL	**0.07**	19.25	1
Masters-8	LTL	**18.52**	48.11	15 067
	PSL	**10.07**	48.69	15 081
Masters-10	LTL	**13.14**	54.52	19 025
	PSL	**6.94**	53.55	19 018

Table 26: Liveness - BMC

Scale (n)	Spec Type	Runtime (s)	Peak Memory (MB)	BDD size (nodes)
Masters-6	CTL	**0.14**	**17.35**	264 698
	LTL	**0.80**	**11.59**	89 936
	PSL	**0.82**	**11.59**	89 936
Masters-8	CTL	**0.38**	**31.09**	436 394
	LTL	**14.47**	**57.71**	1 238 664
	PSL	**14.34**	**57.71**	1 238 664
Masters-10	CTL	**1.41**	**58.09**	**1 249 906**
	LTL	**184.03**	**92.42**	**2 231 026**
	PSL	**224.56**	**92.39**	**2 232 048**

Table 27: Priority Precedence - BDD

Scale (n)	Spec Type	Runtime (s)	Peak Memory (MB)	SAT Clauses
Masters-6	LTL	0.09	20.30	1
	PSL	0.06	19.34	1
Masters-8	LTL	**9.87**	48.97	15 166
	PSL	**4.95**	50.72	15 134
Masters-10	LTL	**11.82**	55.86	19 072
	PSL	**5.72**	55.72	19 076

Table 28: Priority Precedence - BMC

Scale (n)	Spec Type	Runtime (s)	Peak Memory (MB)	BDD size (nodes)
Masters-6	CTL	0.14	16.54	240 170
	LTL	0.09	18.88	106 288
	PSL	0.10	18.88	106 288
Masters-8	CTL	0.27	18.39	295 358
	LTL	0.17	13.31	141 036
	PSL	0.21	13.31	141 036
Masters-10	CTL	**2.28**	**60.65**	**1 276 478**
	LTL	**0.55**	**44.50**	**835 996**
	PSL	**0.54**	**44.50**	**835 996**

Table 29: Mutual Exclusion - BDD

Scale (n)	Spec Type	Runtime (s)	Peak Memory (MB)	SAT Clauses
Masters-6	LTL	0.93	31.20	10 517
	PSL	0.52	33.13	10 517
Masters-8	LTL	1.10	31.18	11 191
	PSL	1.22	30.96	11 191
Masters-10	LTL	2.30	39.47	17 015
	PSL	1.41	38.87	17 015

Table 30: Mutual exclusion - BMC