

Designing a New Formal Methods Language and Verification Tools

by

Maxim Tsoy

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

NAZARBAYEV UNIVERSITY

May 2026

© Nazarbayev University 2026. All rights reserved.

Author
Department of Computer Science
May 2026

Certified by
Benjamin Tyler
Associate Professor
Thesis Supervisor

Accepted by
Elizabeth Arkhangelsky
Dean, School of Engineering and Digital Sciences

Designing a New Formal Methods Language and Verification Tools

by

Maxim Tsoy

Submitted to the Department of Computer Science
on May 2026, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

Verification is an important technique in software and hardware development to ensure the correctness of systems, especially in domains where safety and reliability are critical. Such approaches often incorporate the use of specialized languages for system modeling, along with associated tools which can be used to partially or fully verify system correctness with respect to their formal specifications. However, existing common solutions and tools can be difficult to use for non-specialists, as they often incorporate complex syntax for the specification and design of system models. This can hinder the creation of models that accurately capture the important features of systems, as well as diminish their clarity and accessibility, ultimately limiting their overall usefulness.

This thesis outlines the design of a new systems modeling language and associated tools with a focus on simplicity, strong typing, and overall expressiveness. In particular, the language syntax was designed to support enumerations, typed module parameters, and exhaustive case branches. In addition, a tool-set for the language was developed, including a model checker for the verification of temporal properties. To help evaluate the tools, a framework was developed to run and verify the models, and compare the results against existing tools, namely SPIN and NuSMV. The final version of the model checker was implemented using symbolic model checking techniques with binary decision diagrams. Finally, several case study examples were used to compare the syntactic differences and overall efficiency of the verification process against existing tools. While the proposed language and tools do not aim to outperform the current state of the art, they do demonstrate a proof of concept for the flexibility of the language and its sufficiency to express real systems, as well as the feasibility of applying formal methods to such models.

Thesis Supervisor: Benjamin Tyler
Title: Associate Professor

Acknowledgments

I would like to thank my advisor, Professor Benjamin Tyler, for his support, guidance, and feedback, which helped me a lot during the completion of this thesis. Also, I am grateful to the thesis committee members for their feedback and comments, and to the Department of Computer Science for giving me the opportunity to complete this thesis. Lastly, I would like to thank my family and friends for their support.

Contents

1	Introduction	15
1.1	Motivation and General Background	15
1.2	Traditional Approaches and Research Objectives	16
1.3	Thesis Outline	17
2	Related Work	19
2.1	Formal Specification Languages	19
2.2	Limitations of Existing Approaches	21
2.3	Software and Hardware Development	23
2.4	Formal Verification Tools	23
2.4.1	NuSMV	23
2.4.2	NuSMV and nuXmv	24
2.4.3	SPIN	24
2.4.4	Other Model Checkers	25
2.4.5	Interactive Theorem Provers	25
2.5	Type Systems in Formal Specification Languages	26
2.5.1	Finite-Domain Type Systems	26
2.5.2	Strong and Weak Typing in Specification Languages	27
2.6	Syntax, Type System, and Expressiveness Analysis	27
2.6.1	Analysis of nuXmv	27
2.6.2	SPIN	28
2.7	Temporal Logic Background	28
2.7.1	Computation Tree Logic	30

2.7.2	Linear Temporal Logic	32
3	Methodology	35
3.1	Model Checker Overview	35
3.2	Syntax and Language Design	36
3.2.1	Model Syntax Overview	36
3.2.2	Model Structure Syntax	37
3.2.3	Type System	40
3.2.4	Logical and Temporal Operations	40
3.2.5	Selected Syntax and Type-System Features	41
3.3	Model Checker Architecture	43
3.3.1	Lexical Analysis	43
3.3.2	Parsing	44
3.3.3	Flattening	45
3.3.4	Type Checking	46
3.3.5	Property Verification	46
3.4	Explicit-State Model Checking	46
3.4.1	State Space Generation	46
3.4.2	Evaluating Properties	48
3.5	Symbolic Model Checking	50
3.5.1	Motivation	50
3.5.2	Encoding Flattened Variables into Bits	51
3.5.3	BDD Structure	52
3.5.4	Reachability Analysis	54
3.5.5	Assert Formula Evaluation	55
3.5.6	Assessing CTL Formulas	56
3.5.7	Assessing LTL Formulas	59
4	Results	61
4.1	Explicit vs. Symbolic Model Checking	61
4.2	Case Study: NuSMV Models	62

4.2.1	Model 0: A Simple Model for LTL and CTL Operators	62
4.2.2	Model 1: Priority-Scheduled Multi-Pipeline Real-Time Process- ing Model	64
4.2.3	Model 2: A Shared-Bus Cache Coherence Protocol for Three Processors and One Memory Module	67
4.3	Case Study: SPIN Models	70
4.3.1	Model 3: Bounded Petri Net Model	70
4.3.2	Model 4: ALICE Data Point Processing Framework	72
5	Conclusion	79
5.1	Summary	79
5.2	Future Work	80

List of Figures

4-1	The diagram of the model.	62
4-2	The diagram of model 1	64
4-3	The diagram of model 2	69
4-4	The diagram of model 3	71
4-5	The diagram of model 4	74

List of Tables

2.1	Comparison of Major Formal Specification Language Groups	20
2.2	Differences between NuSMV and Promela	25
2.3	Potential nuXmv type-system concerns with illustrative syntax	29
2.4	Potential SPIN / Promela type concerns	30
2.5	CTL operator equivalences using a minimal operator set	32
2.6	Derived LTL operators	33
3.1	General Pipeline	36
3.2	Summary of Language Constructs	38
3.3	Supported finite-state variable types	40
3.4	Supported assertion operators	41
3.5	Summary of Lexer Token Categories	44
3.6	Set-theoretic semantics of Boolean state formulas	56
4.1	Comparison of verification results between the Model Checker and NuSMV.	63
4.2	Overview of the periodic pipeline scheduling model	65
4.3	Comparison of the original NuSMV model and the revised implemen- tation	66
4.4	Results of the original NuSMV model and the revised implementation	66
4.5	Brief summary of the cache-coherence model	67
4.6	Comparison of the original NuSMV model, the proposed-language im- plementation, and the generated NuSMV translation	68

4.7	Model 3 property results across SPIN, the proposed language, and the translated NuSMV model	72
4.8	Model 3 property verification times	72
4.9	Model 4 property verification results	75
4.10	Model 4 properties that do fully map from SPIN to the proposed language	76
4.11	Comparison of SPIN and proposed-language execution time for Model 4	77

Chapter 1

Introduction

1.1 Motivation and General Background

Nowadays, software and hardware systems have become very complex, requiring a reliable and consistent way to ensure their correctness and expected behavior. Domains like finance, healthcare, transport, etc. require correct execution, as mistakes and errors may lead to severe consequences, such as large financial, data, or human life losses [1]. While classical quality assurance methods such as testing can help prevent major critical bugs, absolute correctness is needed in safety-critical systems. Ensuring correct flow and execution for the critical parts of such systems is essential, which modern quality assurance techniques cannot replace [2].

Formal methods tools give a reliable and mathematically correct way to prove and ensure the correctness of systems [3]. These tools rely on rigorous techniques for mathematically specifying the behavior of the model and the corresponding expected behavior. For example, ensuring liveness and safety properties of the system over time flow is essential for verifying if a specific desired property will always or eventually hold [4]. Historically, formal methods tools found their application in hardware, safety-critical software systems, or protocols [5], but nowadays they are still not widely adopted in other fields like software engineering. For example, as was mentioned in [1], the importance of formal methods in software engineering is growing, with a focus on aerospace, medical devices, and railway systems, where they are mainly used to

prevent errors and improve reliability. Additionally, as was discussed in [6], formal methods become more valuable if they are developer-friendly and can support the implementation of modern systems, where the idea of lightweight formal methods that apply semantics and syntax for common software programs is proposed.

1.2 Traditional Approaches and Research Objectives

Existing methods for verifying systems include applying mathematical tools and techniques, for example, formal logic [7] and automata theory [8], to precisely describe the model and expected result. A common approach is state exploration [9], where the program ensures that the system works in all possible states and behaves according to its expected logical properties.

Generally, these approaches require developers to mathematically model the system, which is initially represented using a general programming language or even a technical document with a list of requirements. Common solutions include the use of Model Checking (exhaustive state exploration over logic properties, using tools such as NuSMV [10] or SPIN [11]), Theorem Proving (construction of proofs for verifying properties of a program, using tools such as Coq [12] or Isabelle [13]), and Static Analysis (code error detection) [14].

This work aims to investigate and analyze common formal methods tools and address the issue of usability and expressiveness of model descriptions over syntax and type systems. As part of this thesis, language syntax for formal system descriptions is investigated, and a comparison of the implementation approaches with common existing formal methods tools is performed.

The primary objective of this paper, however, is to design a new formal specification language as well as model checking capabilities for that language. Despite the existence of state-of-the-art solutions with both generic and niche implementations, this paper proposes a more simple and expressive language to address potential gaps in syntax design and choices [3]. Additionally, the focus will be on providing an improved type system with modern programming language design considerations

[15].

A comparison and analysis using case studies and examples of models commonly used with other model checking tools will be performed to evaluate the language and tools. This part focuses on the expressiveness and feasibility of the solution to represent and verify the given example systems. This will include results comparing model implementations, verification performance, and applicability.

The formal methods field is constantly developing, with new solutions appearing and improving over time. While many areas can be addressed with the targeted use of ready-to-use tools, this work aims to analyze existing solutions and compare them against the proposed new formal methods language and tools.

1.3 Thesis Outline

The remainder of the thesis is organized as follows:

In Chapter 2, the literature review is presented.

In Chapter 3, the methodology is presented, where a detailed implementation description is provided.

In Chapter 4, the case studies are presented and the comparative results are shown.

In Chapter 5, conclusions and future work are provided.

Chapter 2

Related Work

2.1 Formal Specification Languages

A Formal Specification Language is a mathematically based representation of software, hardware, and other systems that is used to rigorously describe and verify their structure and correctness [16]. Formal Specification Languages can be divided into several groups depending on their purpose and area of application: Algebraic Specification Languages, Model-Based Specification Languages, Process-Oriented Specification Languages, and Temporal Logic-Based Specification Languages. Overall, Formal Specification Languages are based on logic, set theory, algebra, and other mathematical constructs for the purpose of proving and verification [3].

Algebraic specification languages represent a group of formal specification languages based on algebraic structures and are used to define the behavior between them. Usually, they are based on operations relying on axioms and equations. This group of languages can be used for verifying complex systems based on data structures with custom relationships between them [17]. Examples of these languages are: CASL, OBJ, as well as CASL expressions such as HasCASL, CoCASL, and CspCASL [17].

Model-Based Specification Languages are characterized by properties of explicit state modeling and transformations between system states [18]. Usually, these languages are based on and constructed using sets and functions as main mathematical

Table 2.1: Comparison of Major Formal Specification Language Groups

Category	Main Features	Use Cases / Verification Role	Examples
Algebraic Specification Languages	Algebraic structures, equations, and axioms describing relationships between operations and data types are used for system description.	Abstract data types, mathematical modeling of software components.	CASL, OBJ, Has-CASL, CoCASL, CspCASL
Model-Based and State-Based Specification Languages	Collections of states and transitions, usually based on sets and functions.	Safety-critical system design, system-level specification, state transition validation, control system modeling.	VDM, Z, Event-B, B Method
Process-Oriented Specification Languages	Interacting processes communicating via synchronization or message passing.	Distributed systems, communication protocols, concurrent system verification, synchronization correctness.	CSP, CCS, LOTOS, ACP
Temporal Logic-Based and Model Checking Approaches	Temporal logic formulas such as safety and liveness properties for specifying system properties.	Model checking, verification of reactive systems, concurrent and real-time system validation. Mainly finds errors in finite models; absence of errors does not prove full program correctness.	LTL/CTL specifications, Promela + SPIN, NuSMV, nuXmv
Theorem Proving Approaches	Formal proofs are constructed in a logical system to prove correctness under stated assumptions.	Proof-oriented verification. Can prove correctness; Rocq and Lean can also output verified functional programs.	Rocq, Lean, Isabelle

constructs. Examples of such languages include VDM and Z [19]. They can also be used in various systems, for example, in safety-critical systems where further customizations or complex rules could be used to verify and express system behavior.

Process-Oriented Specification Languages represent a group of languages focusing on concurrent behavior and relationships between processes. Overall, these languages can be used for verifying systems composed of independent distributed components and for solving problems related to synchronization or communication [20]. Examples of these languages include CSP and LOTOS.

Temporal Logic-Based Specification Languages focus on verifying the properties of a system through execution over time. These languages could be used to formally check safety and liveness properties of a system. Certain frameworks and model checkers implement the above-mentioned language properties, usually combining modeling languages with specification logic in their implementation. Popular model checkers include NuSMV and SPIN, which search for violations in a finite model of the system. [10]. Theorem provers such as Rocq, Lean, and Isabelle are used to prove correctness under formal assumptions.

Overall, the general comparative table with key features and examples is shown in Table 2.1.

2.2 Limitations of Existing Approaches

Formal methods is a mature research area with several well-known problems, including state-space explosion [9] and steep learning curves [21]. Nevertheless, new tools and languages are still being developed to improve efficiency, expressiveness, and to address usability concerns.

Additionally, the complexity and expressiveness of the language syntax can cause modeling and specification issues due to hard-to-understand or limited language constructs and complex requirements. For example, as was discussed in [22], formal requirements and specifications could be hard to use and express, which could be the reason why formal methods fail as a common solution for regular real-life problems.

Moreover, the authors discuss whether informal English language descriptions instead of formal commands could be used for system verification, but still note that it is too vague and not suitable on its own. As a result, the authors propose a bridge between formal instructions and informal language, more natural grammar, mathematical expressions, and formulas, all which drastically increase readability and reduce the learning curve. For example, as was discussed in [23], formal methods could be applied in business compliance with mature tools theoretically, though they are not widely adopted in practice because of usability and business life-cycle issues. Nevertheless, one could argue that developers should know the basics of Formal Methods, sufficient enough to apply tools for any real-life problem. However, as was argued in [24], formal methods are foundational knowledge which improve reasoning and programming skills, though the tools are niche and complex which require professional expertise to apply them in complex models.

Another potential limitation of existing tools may be their fragmentation and the tooling ecosystem. Due to the academic focus and design of formal methods solutions, they may lack modern and detailed examples, tutorials, and guidelines on creating real-world industrial or business models. Additionally, they may require an extra step for developers or engineers when they want to model a system beyond regular quality assurance, potentially redesigning and reimplementing the system using formal methods tools. Additionally, most formal specification languages are not easily integrable into the regular flow of system development, with a lack of support for modern ecosystems. Furthermore, scalability issues may arise, as a model may not be easily modified due to requirements or minor system changes. Nevertheless, the emphasis on the importance and improvement of adopting formal methods can be seen in hardware development. For example, as was mentioned in [25], the authors argue that formal methods accelerate the development process and help prevent bugs, but they still note that adoption and tool support are very important for the aforementioned benefits. Nevertheless, traditional approaches are not designed for frequently changing modern modeling or development processes with evolving and restated business technical requirements, as they tend to assume stable and non-changing models. However, not

all systems need formal methods. For example, as was mentioned in [26], high-risk, high-value critical systems require formal verification, but general-purpose software does not.

2.3 Software and Hardware Development

In software and hardware development, formal methods could be applied for requirements, design, or specification creation. Usually, they are applied in safety-critical systems to ensure execution without mistakes and prove correctness of execution. For example, they could be used in protocols, cryptographic algorithms, and safety-critical system verification, which as a result created certified standards, for example, DO-178C [26]. They could be applied both for functional programming—where mathematical properties of a system could be transformed into a set of test cases, close to rigorous verification—and OOP—where relationships and properties of objects could be converted into a verification set of test cases. Furthermore, for concurrent or distributed systems, Petri nets [27] can be used to specify model flow and concurrency, whereas the process calculus [28] can specify such systems as a set of processes and states.

In hardware development, the usage of formal methods is primarily motivated by the high cost of fixing potential errors and the inability to easily patch a bug. As was mentioned in [29] and [5], in hardware development, model checking, theorem proving, and equivalence checking could be used as main solutions for verification.

2.4 Formal Verification Tools

2.4.1 NuSMV

NuSMV is a symbolic model checker which serves as a tool for verification of complex models and a utility for other model checkers [10]. It is based on binary decision diagrams, a data structure used for representing boolean functions [30]. It supports CTL and LTL property statements, with state traces of state sequences that led

to the violating state if the checked statement was false. It also supports modular program structures, allowing for the design and description of complex models with subcomponents.

Regarding the syntax and type system, NuSMV is a declarative module language following formal specification language paradigms with a static, explicit type system. NuSMV supports the following types: boolean, finite integer ranges, enumerated types, finite arrays, and module types without support for interfaces.

2.4.2 NuSMV and nuXmv

In addition, nuXmv is an extension of NuSMV used for verification of finite and infinite-state systems [31]. It supports CTL, LTL similarly to NuSMV, also, supports richer data types, infinite-state reasoning and integrates SMT-based model checking [32].

Regarding the type system nuXmv is based on the typed SMV input language and extends it with richer data types for both finite- and infinite-state modeling. It supports unbounded integer and real types for SMT-based verification. It also supports typed function declarations, although the function argument and return types are currently restricted to only built-in types.

2.4.3 SPIN

SPIN is a verification tool for concurrent models for systems described in Promela [11]. It supports LTL operators, with state path traces over the execution paths.

Regarding the syntax and type system, Promela is an imperative modeling language supporting regular syntax constructs: processes, statements, loops, and conditions. However, in contrast with most imperative programming languages, execution is inherently non-deterministic and concurrent for multiple processes. Being a non-functional or OOP language, Promela is conceptually closest to Communicating Sequential Processes. The type system is relatively simple, being static and strong, yet with some implicit conversions. Similarly to NuSMV, all types are finite; for ex-

ample, for integer-like types, SPIN provides bit, bool, byte, short, int, and unsigned, all of which are bounded and not truly infinite. In addition, SPIN provides enumerated constants using the keyword "mtype", communication channels, and finite-size arrays.

The relative comparison between NuSMV and SPIN (Promela) is shown in Table 2.2.

Feature	NuSMV	Promela
Style	Declarative	Imperative
Execution	State transition relation	Process execution with interleavings
Concurrency	Modeled declaratively	Explicit processes
Focus	Logical specification	Behavioral modeling
Type system	Strict finite symbolic	C-like type system with fixed range

Table 2.2: Differences between NuSMV and Promela

2.4.4 Other Model Checkers

Java Pathfinder (JPF) is a model checker used for Java bytecode verification. It is used to verify all paths of an executable program and prevent potential deadlocks or invalid program execution flows [33].

Another commonly used model checking tool is UPPAAL, which is used for simulation and verification of real-time systems. It is focused on explicit definitions of time and concurrency constraints and therefore could be used in communication protocols, timed automata, and other time-critical models [34].

2.4.5 Interactive Theorem Provers

Rocq (formerly known as Coq) is a proof assistant for mathematical proofs and software correctness [12]. It is based on the theory of the calculus of inductive constructions [35], which is represented as a higher-order typed lambda calculus. Rocq uses Gallina [36] as the main specification language, which is a strongly typed functional language. To implement proofs, Rocq uses tactic procedures that transform goals into

proven statements. This syntax supports various data structures, recursive functions, and theorem declarations.

Another language representative of theorem provers is PVS (Prototype Verification System), which is a higher-order logic theorem prover [37] with strong automation but still proof-oriented, which is used in aerospace and safety-critical verification where rigorous proofs are needed.

ACL2 (A Computational Logic for Applicative Common Lisp) is an automated theorem prover based on the Lisp programming language. It is based on term rewriting and used in software or hardware verification [38]. Lastly, another commonly used theorem prover is Isabelle, which uses Standard ML and Scala as programming languages [13]. It supports multiple logics (HOL, FOL, set theory), also combining interactive proofs with automated tools like SMT solvers.

2.5 Type Systems in Formal Specification Languages

2.5.1 Finite-Domain Type Systems

Type systems in formal verification languages are used for the purpose of limiting and defining the state space, adding abstraction (modules, subprograms), and specifying the functionality support of a verification tool. The syntax and type choices affect the scalability, expressiveness, and readability of the models. On the other hand, as was mentioned in [39], too strict type systems may not be needed for a formal verification language, and untyped languages are more expressive and flexible, while type systems provide safety and compile-time checks to prevent mistakes. Also, as was discussed in [16], Higher-Level Formal Languages improve software system specification with the benefits of structured abstraction and semantic clarity.

The primary purpose of finite-domain type systems is to restrict the state space of verifications. As a result, it becomes possible to analyze every state and formally verify all execution paths. This type system is especially useful in hardware verification, protocol analysis, and safety-critical systems with bounded ranges of variables.

As a potential drawback of this type system, infinite ranges, for example, a set of natural numbers, are a potential drawback and are impossible to be expressed.

2.5.2 Strong and Weak Typing in Specification Languages

Strongly typed languages (Rocq or Isabelle) prevent implicit type conversions and ask for explicit type coercions in every expression. It could be used to prevent potential errors at the specification level. In contrast, weakly typed languages allow implicit coercions or treat certain domains uniformly. For example, truth values that are not boolean may be converted and be equal to the boolean true value. But they let the language be more expressive and may decrease the boilerplate code. Moreover, as was mentioned in [40], type systems enhance reliability and expressiveness of a programming language.

2.6 Syntax, Type System, and Expressiveness Analysis

2.6.1 Analysis of nuXmv

It is worth mentioning that nuXmv is designed as a modular, declarative system. Sub-modules in nuXmv are designed hierarchically, providing reusable blocks of code throughout the system. Furthermore, variables are declared with explicit types, also separating between state variables (**VAR**), input variables (**IVAR**), initial conditions (**INIT**), and transition relations (**NEXT**, **TRANS**). For program flow, nuXmv provides **case** and **esac**, combined with the **next** statement. The overall syntax is designed as a formal, clear, and transitional structure, which is ideal for modeling transitional systems.

Regarding the expressiveness of the language, nuXmv provides flexible structures such as arrays, sets, and **enumeration** types. However, as was stated in the manual, “expressions cannot be of actual **enumeration** types, but of their simplified and generalized versions only” [41], which abstracts away **enumeration** at the expression level,

simplifying compilation and verification while also providing flexibility of assigning different types to an `enumeration`. Also, the set structure, which could be used only in `case` or in operators, limits the expressiveness, essentially keeping them read-only, static, and expression-level. Nevertheless, theoretically, set operations could be replaced with plain equations.

2.6.2 SPIN

Overall, SPIN (Promela) programs consist of processes, message channels, and variables, which create a clear separation between behavior (processes), communication structure (channels), and state (variables). As was mentioned before, the syntax resembles C; however, it is fundamentally different from it due to the use of guarded commands and their impact on execution flow. Here, the execution of a statement may be conditional on the current truth value of its guard, and the choice between executable guarded commands in a given structure is made non-deterministically [42].

Moreover, SPIN provides a fixed, statically declared type system consisting of primitive numeric types (bit, bool, byte, short, int) and channel types [42]. There is no concept of immutability or constants to support general-purpose computations. It is worth mentioning channel types as a typed structure for communication between processes, which are essentially first-class typed entities whose buffer capacity [42] and message types define the data sent through channels. However, the type system is intentionally not too expressive or flexible; for example, there are no user-defined algebraic data types and no parametric polymorphism. Additionally, arrays have a fixed size, with no dynamic memory allocation, which provides a way to limit the model for the purpose of formal verification.

2.7 Temporal Logic Background

A Temporal logic [43] is a formal logic used to specify the behavior of a system during its execution over time. Temporal logics are commonly used in conjunction with model checking, and are used to specify what must be true in future states

Type feature	Syntax example
Enumeration values are not typed	<pre> VAR s : {s1, s2}; VAR t : {t1, t2}; ASSIGN next(t) := s; -- </pre>
Module parameters have no declared or constrained types	<pre> MODULE M(x) DEFINE y := x + 1; MODULE main VAR b : boolean; m : M(b); -- type mismatch appears only -- when the module is instantiated </pre>
Array bounds not captured at the type level	<pre> VAR a : array 0..2 of boolean; i : integer; ASSIGN next(a[i]) := TRUE; -- if i is outside 0..2 the index is invalid -- but the type system does not prevent it </pre>
Word types are isolated from numeric hierarchy	<pre> VAR w : unsigned word[8]; i : integer; INVAR w = i; -- type error: word and integer -- types are not compatible </pre>
No refinement or range-constrained types	<pre> VAR idx : integer; INVAR idx >= 0 & idx <= 5; -- value restriction must be expressed -- separately from the variable type </pre>

Table 2.3: Potential nuXmv type-system concerns with illustrative syntax

SPIN type concern	Syntax example
Implicit narrowing and truncation	<code>byte b = 300; /* overflow */</code>
<code>bool</code> and <code>bit</code> accept non-boolean values	<code>bool f = 5; /* treated as true */</code>
Array index out of bounds is undefined behavior	<code>byte a[3]; a[5] = 1;</code>
Single global <code>mtype</code> namespace	<code>mtype = { idle, busy, req, ack };</code>
Uninitialized <code>mtype</code> has unnamed value 0	<code>mtype x; /* x == 0 */</code>

Table 2.4: Potential SPIN / Promela type concerns

or along possible execution paths. Two commonly used Temporal Logics in model checking are Computation Tree Logic, and Linear Temporal Logic.

Let AP be a set of atomic propositions, then a transition system is written as

$$TS = (S, R, L),$$

where S is a set of states, $R \subseteq S \times S$ is a transition relation, and

$$L : S \rightarrow 2^{AP}$$

is a labelling function. Here, 2^{AP} denotes the powerset of AP , namely the set of all subsets of AP . Therefore, for each state $s \in S$, $L(s)$ gives the set of atomic propositions that are true in s .

Afterwards, a path is an infinite sequence of states

$$\pi = s_0 s_1 s_2 \dots$$

such that $(s_i, s_{i+1}) \in R$ for every $i \geq 0$.

2.7.1 Computation Tree Logic

Computation Tree Logic (CTL) [44] is a branching-time temporal logic used to describe the properties of a system over a tree of possible future executions starting

from a given state.

The two CTL path quantifiers are:

$$A \text{ “for all paths”,} \quad E \text{ “there exists a path”}.$$

which must be paired with one of the pair temporal operators:

$$X \text{ next,} \quad F \text{ eventually,} \quad G \text{ globally,} \quad U \text{ until.}$$

when quantifying a CTL formula.

The meaning of each of the CTL operators are as follows:

- $AX \phi$: on all paths, ϕ holds in the next state;
- $EX \phi$: there exists a path where ϕ holds in the next state;
- $AF \phi$: on all paths, ϕ eventually holds;
- $EF \phi$: there exists a path where ϕ eventually holds;
- $AG \phi$: on all paths, ϕ always holds;
- $EG \phi$: there exists a path where ϕ always holds;
- $A(\phi U \psi)$: on all paths, ϕ holds until ψ holds;
- $E(\phi U \psi)$: there exists a path where ϕ holds until ψ holds.

In Computation Tree Logic (CTL) there exist minimal sets of operators that are sufficient to express all the remaining CTL operators. One commonly used minimal set of CTL operators is [45]:

$$\{\text{true}, \vee, \neg, EG, EU, EX\}.$$

The operators supported in the tool and their derivations are presented in Table 2.5, which are discussed and derived from [46].

Operator / Formula	Equivalent using minimal CTL operators
$AX \phi$	$\neg EX(\neg\phi)$
$AG \phi$	$\neg EF(\neg\phi)$
$AF \phi$	$\neg EG(\neg\phi)$
$EF \phi$	$E(\text{true } U \phi)$
$A(\phi U \psi)$	$\neg (E(\neg\psi U (\neg\phi \wedge \neg\psi)) \vee EG(\neg\psi))$

Table 2.5: CTL operator equivalences using a minimal operator set

2.7.2 Linear Temporal Logic

In contrast to CTL, Linear Temporal Logic (LTL) [47] describes properties over all possible execution paths rather than over a branching tree of possible futures. In this way, LTL verification involves checking logical properties across possible paths from a given state, in contrast to CTL.

The primary temporal operators of LTL are:

X next, F eventually, G globally, U until.

Their meanings are as follows:

- $X \phi$: ϕ holds in the next state of the path;
- $F \phi$: ϕ eventually holds at some future point on the path;
- $G \phi$: ϕ holds globally at every point on the path;
- $\phi U \psi$: ϕ holds until ψ holds.

One minimal adequate basis for LTL [48] consists of the Boolean operators $\neg$ and $\vee$ together with the temporal operators X and U :

$$\{\neg, \vee, X, U\}.$$

The equivalencies for the remaining derived operators are shown in Table 2.6.

Derived Operator	Equivalent Using Minimal LTL Operators
$\phi \wedge \psi$	$\neg(\neg\phi \vee \neg\psi)$
$F \phi$	$true \ U \ \phi$
$G \phi$	$\neg(true \ U \ \neg\phi)$

Table 2.6: Derived LTL operators

Chapter 3

Methodology

3.1 Model Checker Overview

The implementation aims to design a hybrid model checker that includes language features of both NuSMV and SPIN. The overall syntax and functionality start with NuSMV as a basis, but with a focus on more concise and type-safe syntax design. While optimizing model verification is of practical importance, it is not the primary goal of this work.

The main parts of the underlying software include a lexer and parser for initial processing, as well as components for the Abstract Syntax Tree (AST), Intermediate Representation (IR), State Graph, and Binary Decision Diagram [49] for the internal representation of the model.

The main tools that were used for generating the parser and lexer were the Java-based tools Java CUP [50] and JFlex [51], respectively. In addition, all remaining parts of the model checker have been implemented in Java. Similarly to regular model checkers, the system expects the user to provide an input consisting of a formally and syntactically correct model description, as well as logical statements to be verified written as LTL and CTL formulas. The tool computes all possible states and apply established algorithms to verify the logical properties. The general pipeline of the program is shown on Table 3.1.

Step	Tools used	Description
Tokenizing	JFlex [51]	Implemented lexer for the programming language
Parsing	Java CUP [50]	Parser and constructing the program structure
Flattening	Java	Flattening modules and converting a program into a single main module model
State space generation	Java	Constructing a graph of states
CTL and LTL verifications	Java	Verifying CTL and LTL properties on the graph

Table 3.1: General Pipeline

3.2 Syntax and Language Design

3.2.1 Model Syntax Overview

The overall structure of the models to be processed by the tool consists of a main module and submodules defined by the user. The main module may include the following components: a declaration list of all variables used in the model, assign blocks specifying the initial assignment of variables, and next blocks to describe the transitions in the model. Additionally, there are user-defined module blocks that also contain declare, assign, and next blocks; however, they additionally include a define block to support user-defined types or complex variables. Internally, the entire model is flattened during processing, and all user-defined modules are treated as additional variables within the model design.

Regarding types, the user may define types manually, which are stored as ranges of possible values. These ranges are then used to limit the model and the total number of possible states. Additionally, model blocks are treated as types so that the system may auto-infer ranges based on the user-defined blocks.

To address non-determinism, a special "choose" statement is used. It accepts a list of expressions that the system may non-deterministically iterate through, allowing exploration of all possible states. The next rules and values with respect to the current state are computed entirely based on the current state values.

3.2.2 Model Structure Syntax

A source file contains exactly one **main** block followed by zero or more **module** blocks, as shown below:

Listing 3.1: Model structure

```
model ::= main_block module_block*

main_block ::= main {
    type?
    declare
    initialize
    next
    define?
    assert
    compute?
}

module_block ::= module Name(parameters) {
    type?
    declare
    initialize
    next
    define?
}
```

The designed syntax introduces a modular structure of a model using the **main** keyword to denote the starting module. Each module is divided into the following sections: **declare** used to denote state variables, **assign** defines initial values of variables, **next** specifies the transition relations between states, while **define** introduces reusable expressions for readability, and **assert** is used to denote LTL and CTL statements. This separation was made to clearly separate the module structure and

enhance readability.

The transition relation between variable states is expressed declaratively using the **case** construct. It provides explicit and deterministic control-flow decisions. Nondeterminism is introduced via the **choose** construct, which lists all possible transitions of a variable. This allows depicting models with nondeterministic flows.

Modularity is supported through parameterized **module** definitions. They are used to encapsulate local state and behavior and interact with the remaining system using declared parameters.

A summary of the different syntactic constructs of the language is provided below:

Table 3.2: Summary of Language Constructs

Operator	Meaning	Example
<code>declare { ... }</code>	Declares state variables and their finite domains.	<code>declare { count : 0..3; mode : Mode; }</code>
<code>initialize { ... }</code>	Defines the set of allowed initial states.	<code>initialize { count := 0; mode := idle; }</code>
<code>next { ... }</code>	Defines the transition relation by setting up next-state values.	<code>next { count' := count + 1; }</code>
<code>define { ... }</code>	Introduces named derived expressions that do not create state storage.	<code>define { active := mode == run; }</code>
<code>assert { ... }</code>	Declares named properties over the reachable state space.	<code>assert { safe AG(count <= 3); }</code>
<code>fairness { ... }</code>	Declares fairness conditions using assertion-expression syntax.	<code>fairness { scheduler_runs; }</code>
<code>compute { ... }</code>	Declares graph-distance queries over the reachable state graph.	<code>compute { MIN[x == 0, done]; }</code>

Operator	Meaning	Example
<code>x := choose { ... };</code>	Creates multiple possible initial values, therefore multiple initial states.	<code>initialize { mode := choose { idle, run }; }</code>
<code>assign(x) := expr;</code>	Constrains a variable by filtering its domain rather than assigning a single explicit value.	<code>assign(x) := case { enabled => 1; default => 0; };</code>
<code>x' := choose { ... };</code>	Defines a multi-valued next-state rule, creating nondeterministic branching.	<code>mode' := choose { idle, run, stop };</code>
<code>case { ... }</code>	Branches by condition from top to bottom. The first matching branch wins.	<code>case { x == 0 => 1; default => 0; }</code>
<code>default => expr;</code>	Adds the explicit fallback branch inside <code>case</code> or <code>case*</code> .	<code>default => x;</code>
<code>case* { ... }</code>	Branches on the assignment target itself. This is the exhaustive-case form used for domain coverage checking.	<code>state' := case* { 0 => 1; 1 => 2; default => 0; };</code>
<code>name formula;</code>	Declares a default-CTL assertion without writing the CTL keyword explicitly.	<code>reachable EF(done);</code>
<code>CTL name formula;</code>	Declares a CTL property explicitly.	<code>CTL safe AG(counter <= 3);</code>
<code>LTL name formula;</code>	Declares an LTL property.	<code>LTL progress [] (request -> <>(grant));</code>
<code>INVAR name formula;</code>	Declares an invariant-style property.	<code>INVAR bounded counter in set { 0, 1, 2, 3 };</code>

Operator	Meaning	Example
MIN[from, to];	Requests the shortest path length from any state satisfying from to any state satisfying to.	MIN[counter == 0, mode == stop];
MAX[from, to];	Requests the longest finite path length from from to to, or infinity if the target can be postponed indefinitely.	MAX[counter == 0, workers[1].ready];

3.2.3 Type System

The proposed language uses a finite-domain type system with support for explicit-state model checker support. The syntax, while being based on NuSMV and SPIN/Promela, is simplified and has reduced boilerplate code. It also supports module typing, letting the system define transition systems. Infinite types such as unbounded integer ranges were excluded to make complete state exploration possible. The types used are shown in the following table:

Type	Example
Range	0..10
Boolean	boolean
Set	set { idle, run, stop }
Alias	Mode
Enum	enum Mode { ... }
Module	Worker(limit)
Array	array[4] of boolean

Table 3.3: Supported finite-state variable types

3.2.4 Logical and Temporal Operations

The logical and temporal operations that can be used to express the properties to be verified are shown in Table 3.4. The basic C-inspired logical operators can be used

with most mathematical expressions to construct boolean expressions, which can then be used with the temporal operators of CTL and LTL.

Table 3.4: Supported assertion operators

Operator	Meaning	Example
!	Negation	<code>!error</code>
<code>&&, &</code>	Conjunction	<code>safe && ready</code>
<code> , </code>	Disjunction	<code>done failed</code>
<code>-></code>	Implication	<code>request -> AF(grant)</code>
<code><-></code>	Logical equivalence	<code>a <-> b</code>
<code>[]</code>	Always (LTL)	<code>[]safe</code>
<code><></code>	Eventually (LTL)	<code><>(done)</code>
<code>X(...)</code>	Next (LTL)	<code>X(done)</code>
<code>AX(...)</code>	All-next (CTL)	<code>AX(ready)</code>
<code>EX(...)</code>	Exists-next (CTL)	<code>EX(ready)</code>
<code>AF(...)</code>	All-eventually (CTL)	<code>AF(done)</code>
<code>EF(...)</code>	Exists-eventually (CTL)	<code>EF(done)</code>
<code>AG(...)</code>	All-globally (CTL)	<code>AG(safe)</code>
<code>EG(...)</code>	Exists-globally (CTL)	<code>EG(running)</code>
<code>U(...)</code>	Until	<code>U(active, done)</code>
<code>AU(...)</code>	All-until (CTL)	<code>AU(active, done)</code>
<code>EU(...)</code>	Exists-until (CTL)	<code>EU(active, done)</code>

3.2.5 Selected Syntax and Type-System Features

This work attempts to address potential limitations in nuXmv and SPIN syntax design using new expressions and type constructs as part of the proposed language. Such additions are meant to improve expressiveness and readability, and to prevent potential runtime bugs and errors.

Named and reusable enumeration types

In existing tools, types must often be redeclared and cannot be reused. In an attempt to improve flexibility, readability, and reusability, the language introduces named enumeration types as first-class entities, which could be declared once and easily be reused. For example, we can define the following, which can then be used (and reused) in further declarations:

```
type CacheState = set { owned, shared, invalid };
type Cmd = set { idle, read_shared, read_owned, response };

declare {
  p0_state : CacheState;
  CMD      : Cmd;
}
```

Note that a type checker is included as part of the tools to prevent invalid assignments using values outside of the enumeration set ranges.

Typed Module Parameters

In nuXmv-style module systems, parameters are untyped, while in SPIN the type choice is limited to built-in primitives. Although it was intentionally designed to not restrict users when they pass values to modules, optional type restrictions for parameters could optimize the model checker and prevent potential errors. For example, we can do the following in the new language:

```
type Grant = enum { idle, p0, p1 };

module pipe(timeout: boolean, granted: Grant) {
  declare { state : 0..3; }
}
```

Exhaustive Case Analysis on Enums

As a potential error source, partial case checks over enums could omit states, leading to underspecified behavior. Therefore, a total case construct (**case***) was introduced to ensure exhaustive comparison against all enumeration values.

```
type state = enum { invalid, shared, owned };

p : state;

p' := case* {
  invalid => shared;
  shared  => owned;
  owned   => owned;
};
```

The type checker ensures that all values are checked, and throws an error otherwise.

3.3 Model Checker Architecture

3.3.1 Lexical Analysis

Lexical analysis is the first step in the toolchain. It's main purpose as for any tokenizer is to accept raw text input, and transform it into a series of strings that represent atomic parts of the language, which can then be processed by a parser. The Java-based lexer generator, JFlex, was used to create the tokenizer for the tool. The reason for this choice was mainly simplicity, flexibility, and direct integration into the Java ecosystem. The tokens used in the language are shown in Table 3.5.

Table 3.5: Summary of Lexer Token Categories

Category	Lexical Elements
Keywords	main, module, define, function, assert, compute, ctl, ltl, invar, min, max, var, let, const, boolean, type, declare, initialize, assign, next, choose, return, case, case*, set, default
CTL Operators	AX, EX, AU, EU, AF, EF, AG, EG
LTL Operators	[], <>, U, X
Comparison and Equivalence Operators	>=, <=, ==, !=, >, <, =, <->
Assignment and Transition Operators	:=, ->, =>
Arithmetic Operators	+, -, *, /, %, **
Logical Operators	!, &&,
Set Operators	(union), & (intersection)
Delimiters	(), { }, [], ;, ,, ., :, '
Literals	Integer literals (0, 1, ...), Boolean literals (true , false , case-insensitive), String literals ("...")
Identifiers	[A-Za-z_] [A-Za-z_0-9]*
Comments	Single-line: // ..., - ... Multi-line: /* ... */
Whitespace	Spaces, tabs, form-feed characters, and line terminators
Error Handling	Illegal characters produce a lexical error

3.3.2 Parsing

A parser is used to generate the initial structured representation of the model from the sequence of tokens obtained from the lexer. For this stage, the Java CUP parser generator was used to create the parser for the model checker, as it can be directly integrated into the Java ecosystem. As input, Java CUP takes the syntactic specification of the language in the form of a context-free grammar, and produces as output a Java-based parser for the language.

Given a valid model description, the parser constructs the Abstract Syntax Tree (AST) rooted in a **Program** object instance, which contains a **MainModule** object and an optional list of **UserModules**.

Inside each module the following AST elements are produced:

- **DeclarationStatement** object which defines the variable and its type

- `AssignmentStatement` object which determines the initial values of the variable
- `NextStatement` defines transition rules and relations between current and next state of the system.
- `DefineStatement` is used to add typed macros
- `ComputeStatement` is used for quantitative reachability queries, such as the minimum or maximum number of transitions required to reach a target state set from a source state set

Assertion constructs are parsed into a dedicated assertion block where CTL operators (`AX`, `EX`, `AG`, `EG`, `AF`, `EF`, `AU`, `EU`) and LTL operators (`X`, `U`) are supported. Each logical property has an identifier name and a corresponding boolean expression. To explicitly specify that the assertion must be LTL, the keyword `LTL` can be written before the expression, for example, `LTL progress [] (request -> <>(grant))`, while CTL properties may similarly be introduced with the `CTL` keyword. However, by default, any property is considered CTL, so this prefix is optional.

3.3.3 Flattening

In modular formal methods tools, models are usually described using several modules, each of which can contain variables, as well as sub-modules. Such an approach improves readability of the system, makes the code more scalable, and facilitates reuse.

The flattening of the model is the process of converting the list of modules of a system model into a single module. This is accomplished through variable renaming and collecting the statements of the individual modules into the single module. The flattened model simplifies the later stages involved in generating the state space representation used for model checking.

3.3.4 Type Checking

Type checking is implemented in the following way. The tool first resolves named type aliases into concrete finite-domain types, for example, ranges, booleans, explicit sets, enumerations, arrays, and module instance types. All declared state variables are associated with a finite domain, which is later used by the state-space generator and model checker. During this phase, the tool also ensures that type-dependent constructs indeed satisfy their type definitions and ensure type consistency; for example, the `case*` expression checks that all types are covered. If any expression violates its type definition or provides a value out of range, a warning is reported to the user.

3.3.5 Property Verification

In the final stage, the tool converted the flattened model into a state representation that could be used in verifying the logical properties specified in the assertion statements.

For the first iteration of the model checking tool, an explicit-state approach was used, which essentially generated the state graph of the model. However, due to performance issues described in the next chapter, a symbolic model checking approach was implemented to replace the former approach. Both of these are described in detail in the following two sections.

3.4 Explicit-State Model Checking

3.4.1 State Space Generation

For explicit state space generation, the finite directed graph of states is constructed. Each state is represented by the record:

`<id, parentId, childrenIds, values, nextStatements, isRootState>`

where `id` is used for unique identification, `childrenIds` denote outgoing transitions, `parentIds` encodes incoming transitions (it may seem redundant, however, was used for implementation purposes), and `values` stores the values of a given state.

As a result of parsing and flattening the initial values of a model are constructed. They are represented by a set of values where each variable is assigned to a default value. If during initialization users applied the non-deterministic `choose` operator to a variable, there could be a set of initial states.

The state graph generated through space exploration is built using the following algorithm:

Algorithm 1 Explicit State Space Generation

```

 $S \leftarrow$  initial states
 $W \leftarrow$  worklist initialized with  $S$ 
while  $W$  not empty do
     $s \leftarrow$  remove state from  $W$ 
    for all next statements from  $s$  do
        execute and evaluate next state  $s'$ 
        if  $s'$  not in  $S$  then
            add  $s'$  to  $S$ 
            add  $s'$  to  $W$ 
        store transition  $(s, s')$ 

```

The resulting system essentially becomes a finite directed graph of states, resembling a Kripke structure [52].

This algorithm for explicit state space generation, when compared to those in existing tools like SPIN [42] or Java Pathfinder (JPF) [33], has issues regarding memory efficiency, transition reduction, and state explosion. Nevertheless, this approach was initially used in our model checker due to its conceptual simplicity. Such an approach deterministically constructs the whole state space graph which can be used for directly evaluating logical properties written in CTL and LTL, and is useful for finding counterexamples (and for debugging the tool itself.)

3.4.2 Evaluating Properties

The implementation of the $EX(\varphi)$ algorithm relies on identifying all states that satisfy the formula φ , and then finding the states that have a transition to at least one such state. If a state s has a successor state s' that satisfies φ , then s satisfies $EX(\varphi)$. The algorithm for computing $EX(\varphi)$ is shown below.

Algorithm 2 Evaluation of $EX(\varphi)$

```

1:  $Sat(\varphi) \leftarrow \{s \in S \mid s \models \varphi\}$ 
2:  $W \leftarrow \emptyset$ 
3: for all  $s \in S$  do
4:   if  $\exists s'$  such that  $(s, s') \in R$  and  $s' \in Sat(\varphi)$  then
5:      $W \leftarrow W \cup \{s\}$ 
6: return  $W$ 

```

The implementation of the $EU(\varphi_1, \varphi_2)$ algorithm is based on first identifying all states that satisfy φ_2 , and then iteratively adding states that satisfy φ_1 and have at least one successor already marked as satisfying $EU(\varphi_1, \varphi_2)$. The pseudocode algorithm is shown below.

Algorithm 3 Evaluation of $E(\varphi_1 U \varphi_2)$

```

1:  $W \leftarrow \{s \in S \mid s \models \varphi_2\}$ 
2: repeat
3:   for all  $s \in S$  do
4:     if  $s \models \varphi_1$  and  $\exists s'$  such that  $(s, s') \in R$  and  $s' \in W$  then
5:        $W \leftarrow W \cup \{s\}$ 
6: until no new states are added
7: return  $W$ 

```

Lastly, to implement the $EG(\varphi)$ algorithm, we first label all states that satisfy φ as candidates for $EG(\varphi)$. Then we repeatedly remove the label $EG(\varphi)$ from any state whose successors do not contain a state already labeled with $EG(\varphi)$. This process continues until no further changes occur.

Algorithm 4 Evaluation of $EG(\varphi)$

```
1:  $W \leftarrow \{s \in S \mid s \models \varphi\}$ 
2: repeat
3:    $Changed \leftarrow false$ 
4:   for all  $s \in W$  do
5:     if  $\nexists s'$  such that  $(s, s') \in R$  and  $s' \in W$  then
6:        $W \leftarrow W \setminus \{s\}$ 
7:        $Changed \leftarrow true$ 
8: until  $Changed = false$ 
9: return  $W$ 
```

These CTL operators form a minimal existential set that can be used to express all remaining CTL operators.

The algorithm for the next operator $X(\varphi)$ can be described as identifying all states that satisfy the formula φ , and then testing whether a preceding state s exists. In other words, if a state s has a successor state s' that satisfies φ , then s satisfies $X(\varphi)$. The pseudocode algorithm is shown below.

Algorithm 5 Evaluation of $X(\varphi)$

```
1:  $W \leftarrow \emptyset$ 
2: for all  $s \in S$  do
3:   if  $\exists s'$  such that  $(s, s') \in R$  and  $s' \models \varphi$  then
4:      $W \leftarrow W \cup \{s\}$ 
5: return  $W$ 
```

The LTL $U(\varphi_1, \varphi_2)$ operator is implemented by first finding all states that satisfy φ_2 , and afterwards checking whether there exists a path consisting only of states that satisfy φ_1 until a state satisfying φ_2 is eventually reached. In other words, the algorithm checks whether φ_1 holds in every state until φ_2 becomes satisfied. The pseudocode algorithm is shown below.

Algorithm 6 Evaluation of $\varphi_1 U \varphi_2$

```
1:  $W \leftarrow \{s \in S \mid s \models \varphi_2\}$ 
2: repeat
3:   for all  $s \in S$  do
4:     if  $s \models \varphi_1$  and  $\exists s'$  such that  $(s, s') \in R$  and  $s' \in W$  then
5:        $W \leftarrow W \cup \{s\}$ 
6: until no new states are added
7: return  $W$ 
```

3.5 Symbolic Model Checking

3.5.1 Motivation

If a model contains the finite-domain state variables:

$$var_1, var_2, \dots, var_n$$

with domain sizes

$$|Domain(var_1)|, |Domain(var_2)|, \dots, |Domain(var_n)|$$

then the total concrete state space has size:

$$\prod_{i=1}^n |Domain(var_i)|.$$

This product could grow very fast even for small models, making it impractical to store the states explicitly.

As was mentioned, the explicit-state approach was not efficient in the first implementation of the model checker. To avoid this issue, a symbolic model checking capability was implemented for the language. Here, Binary Decision Diagrams (BDDs) [49] were used, which avoids the explicit enumeration of all model states and

represents the states inside the system using Boolean functions.

The model checker accepts as input a flattened program instance with checked types. In this way the BDD checker always works on a fixed clear set of variables and statements.

3.5.2 Encoding Flattened Variables into Bits

The symbolic model checker accepts a flattened instance of a program with clear ranges and variable set. Initially, it allocates bits for each variable depending on the variable domain. For example, suppose we have two variables, x and y , where x has 11 possible values while y has only 5. A variable with 11 possible values needs 4 bits, because $2^3 = 8$ is not enough to store all possible values, while $2^4 = 16$ is sufficient. Similarly, a variable with 5 possible values needs only 3 bits. As a result, storing these two variables requires a binary vector of length 7. Conceptually, the checker works with this structure:

$$x_0, x_1, x_2, x_3, y_0, y_1, y_2$$

More formally, each declared program variable v_i has a finite domain

$$Domain_i = \{d_{i,0}, d_{i,1}, \dots, d_{i,m_i-1}\}. \quad (3.1)$$

Here, $d_{i,j}$ denotes the j -th possible value of v_i . To encode v_i symbolically, the implementation assigns a value for each bit leading the required number of bits to be:

$$b_i = \lceil \log_2(|Domain_i|) \rceil. \quad (3.2)$$

Initially, the variables were parsed and handled as integers in the Java code, but for encoding, they are stored based on the variable's declared finite domain and corresponding minimum number of bits.

3.5.3 BDD Structure

First, the initial state is constructed which is derived from initial assignments of variables and the variable range. Formally, a concrete initial state $s \in S_0$ assigns a value $s(v_i) \in \text{Domain}_i$ to each variable v_i , and its Boolean encoding is represented by a conjunctive Boolean assignment

$$C_s(\mathbf{x}) = \bigwedge_i \text{equals}(v_i, d_i), \quad (3.3)$$

where $\text{equals}(v_i, d_i)$ is the Boolean formula stating that the bit-vector of v_i equals the binary code of d_i .

The predicate $\text{equals}(v_i, d)$ checks whether the bit-vector assigned to variable v_i encodes the value d . Suppose v_i is represented by bits

$$x_{i,0}, x_{i,1}, \dots, x_{i,b_i-1},$$

and the binary encoding of d is

$$c_0, c_1, \dots, c_{b_i-1},$$

where each $c_j \in \{0, 1\}$. Then

$$\text{equals}(v_i, d) = \bigwedge_{j=0}^{b_i-1} \begin{cases} x_{i,j}, & \text{if } c_j = 1, \\ \neg x_{i,j}, & \text{if } c_j = 0. \end{cases}$$

Thus, $\text{equals}(v_i, d)$ is true exactly when the Boolean bits of v_i represent d . The formula $\text{equals}'(v_i, d)$ is defined similarly, but uses the next-state bits $x'_{i,j}$. To handle the non-deterministic initial variable assignment, the initial state set is defined which contains all possible values for the initial state.

$$\mathcal{I}(\mathbf{x}) = \bigvee_{s \in S_0} C_s(\mathbf{x}). \quad (3.4)$$

Afterwards, the transition is defined as a Boolean function over current and next variable values:

$$\mathcal{R}(\mathbf{x}, \mathbf{x}'). \quad (3.5)$$

To fully describe the transition per state, the transition formula is represented as a simultaneous combination of all variable transitions

$$\mathcal{R}(\mathbf{x}, \mathbf{x}') = \bigwedge_{i=1}^n \mathcal{R}_i(\mathbf{x}, \mathbf{x}'_i), \quad (3.6)$$

where $\mathcal{R}_i$ constrains the next value of variable v_i . To handle logic branches, a boolean function is defined for a current state that checks if the given branch transition is allowed:

$$E_{i,d}(\mathbf{x}) \quad (3.7)$$

where a domain of a variable $d \in D_i$. Additionally, the function also takes into account the range of variables, and throws a Java Exception if an assignment leads to a value that is out of range. Finally, the transition relation becomes

$$\mathcal{R}_i(\mathbf{x}, \mathbf{x}'_i) = \bigvee_{d \in D_i} (E_{i,d}(\mathbf{x}) \wedge \text{equals}'(v_i, d)), \quad (3.8)$$

where $\text{equals}'(v_i, d)$ constrains the next-state bit-vector $\mathbf{x}'_i$ to encode d . Moreover, to symbolically evaluate an expression, a set that computes the symbolic mapping of variables to a BDD condition representation is defined, which is also represented as (3.7). The equation for a symbolic evaluator of an expression becomes

$$\mathcal{V}(e) = \{(d, B_d) \mid d \text{ is a possible value of } e, B_d \text{ is the BDD under which } e = d\}. \quad (3.9)$$

For a binary operator $\odot$ and operands e_1 and e_2 , if

$$\mathcal{V}(e_1) = \{(a, B_a)\}, \quad \mathcal{V}(e_2) = \{(b, C_b)\}, \quad (3.10)$$

then the result map is obtained by pairwise combination of the expressions:

$$\mathcal{V}(e_1 \odot e_2) = \{(a \odot b, B_a \wedge C_b)\}, \quad (3.11)$$

3.5.4 Reachability Analysis

To perform reachability analysis and get the boolean formula for all reachable states, let $Reach_k(\mathbf{x})$ denote the set of states reachable after k transitions which is a Boolean formula over the current-state variables $\mathbf{x}$. Initially, the reachable set is equal to the initial-state formula:

$$Reach_0(\mathbf{x}) = \mathcal{I}(\mathbf{x}). \quad (3.12)$$

At each iteration, the symbolic checker computes the post-image of the current reachable set. The transition relation $\mathcal{R}(\mathbf{x}, \mathbf{x}')$ relates current-state variables $\mathbf{x}$ to next-state variables $\mathbf{x}'$. The post-image is defined as:

$$Post'(\mathbf{x}') = \exists \mathbf{x} (Reach_k(\mathbf{x}) \wedge \mathcal{R}(\mathbf{x}, \mathbf{x}')). \quad (3.13)$$

In the formula mentioned above, $\exists \mathbf{x}$ is existentially quantified over the finite vector of Boolean current-state variables. In BDD-based symbolic model checking, this operation is implemented by Boolean variable abstraction: the current-state variables $\mathbf{x}$ are removed from the formula, leaving a Boolean formula only over the next-state variables $\mathbf{x}'$. Therefore, the result remains a propositional Boolean formula.

After computing the post-image over next-state variables, the next-state variables are renamed back to current-state variables:

$$Post(\mathbf{x}) = Post'(\mathbf{x}')[\mathbf{x}'/\mathbf{x}]. \quad (3.14)$$

The reachable set is then updated by adding the newly reached states:

$$Reach_{k+1}(\mathbf{x}) = Reach_k(\mathbf{x}) \vee Post(\mathbf{x}). \quad (3.15)$$

The algorithm terminates when no new states are discovered:

$$Reach_{k+1}(\mathbf{x}) \equiv Reach_k(\mathbf{x}). \quad (3.16)$$

The fixpoint condition $Reach_{k+1}(\mathbf{x}) \equiv Reach_k(\mathbf{x})$ is computed by comparing the BDDs showing both Boolean formulas, because BDDs are canonical under a fixed variable ordering, identical BDDs represent the same reachable-state set.

3.5.5 Assert Formula Evaluation

Each boolean formula we want to assert is represented as a logical formula φ that is evaluated over the reachable state space of the model. Let

$$M = (S, S_0, R)$$

be the reachable transition structure constructed by the state-space generator, where S is the finite set of reachable states, $S_0 \subseteq S$ is the set of initial states, and $R \subseteq S \times S$ is the transition relation. A transition $(s, s') \in R$ means that the model can move from state s to state s' in one execution step.

For every boolean formula φ , the model checker computes its *satisfaction set*

$$Sat(\varphi) = \{ s \in S \mid M, s \models \varphi \}. \quad (3.17)$$

That is, $Sat(\varphi)$ contains exactly those reachable states in which the formula φ evaluates to true.

The final truth check for an assert statement is performed by comparing the satisfaction set with the set of initial states:

$$S_0 \subseteq Sat(\varphi). \quad (3.18)$$

It means that a boolean formula is considered true only if it holds in every initial state of the reachable model. Equivalently, the assertion fails iff there exists at least one

initial state in which φ is false:

$$S_0 \cap (S \setminus Sat(\varphi)) \neq \emptyset. \quad (3.19)$$

If this set is non-empty, the checker selects one state from it and reports it as a witness state showing that the assertion does not hold.

The set-theoretic semantics of Boolean operations is shown in Table 3.6.

Table 3.6: Set-theoretic semantics of Boolean state formulas

Formula	Semantics
$Sat(\neg\varphi)$	$S \setminus Sat(\varphi)$
$Sat(\varphi \wedge \psi)$	$Sat(\varphi) \cap Sat(\psi)$
$Sat(\varphi \vee \psi)$	$Sat(\varphi) \cup Sat(\psi)$
$Sat(\varphi \rightarrow \psi)$	$(S \setminus Sat(\varphi)) \cup Sat(\psi)$
$Sat(\varphi \leftrightarrow \psi)$	$(Sat(\varphi) \cap Sat(\psi)) \cup ((S \setminus Sat(\varphi)) \cap (S \setminus Sat(\psi)))$

3.5.6 Assessing CTL Formulas

CTL formulas are evaluated over states; therefore, for every CTL formula φ , the model checker computes its *satisfaction set*:

$$Sat(\varphi) = \{ s \in S \mid s \models \varphi \}. \quad (3.20)$$

The computation is performed symbolically: sets of states are represented by BDDs, and CTL operators are evaluated as symbolic set operations over the reachable state space boolean formula.

The minimal implemented CTL basis is:

$$true, \quad \vee, \quad \neg, \quad EX, \quad EU, \quad EG. \quad (3.21)$$

The Boolean cases are evaluated directly as set operations:

$$Sat(true) = S, \quad (3.22)$$

$$Sat(\neg\varphi) = S \setminus Sat(\varphi), \quad (3.23)$$

$$Sat(\varphi_1 \vee \varphi_2) = Sat(\varphi_1) \cup Sat(\varphi_2). \quad (3.24)$$

Thus, negation is implemented as set complement, and disjunction is implemented as set union.

Temporal operators are evaluated using predecessor computation. Let $X \subseteq S$ be a set of states. The existential predecessor operator $Pre_{\exists}(X)$ returns all states that have at least one successor in X :

$$Pre_{\exists}(X) = \{ s \in S \mid \exists s' \in S. (s, s') \in R \wedge s' \in X \}. \quad (3.25)$$

In other words, a state belongs to $Pre_{\exists}(X)$ if it is possible to reach a state in X after one transition.

The primitive next-state operator $EX\varphi$ is true in a state if there exists at least one successor state satisfying φ :

$$Sat(EX\varphi) = Pre_{\exists}(Sat(\varphi)). \quad (3.26)$$

The primitive existential until operator $E[\varphi U \psi]$ is true in a state if there exists a path starting from that state such that ψ eventually becomes true and φ holds in every preceding state. Its satisfaction set is computed as the least fixpoint:

$$Sat(E[\varphi U \psi]) = \mu Z. Sat(\psi) \cup (Sat(\varphi) \cap Pre_{\exists}(Z)). \quad (3.27)$$

Here, μZ denotes the least fixpoint, i.e. the smallest set of states satisfying the equation. To derive it, the checker computes it as follows:

1. initialize

$$Z_0 = Sat(\psi);$$

2. iteratively compute

$$Z_{i+1} = Z_i \cup (Sat(\varphi) \cap Pre_{\exists}(Z_i));$$

3. stop when $Z_{i+1} = Z_i$.

Since the reachable state space S is finite and the sequence of sets is monotonically increasing, this iteration always terminates.

The primitive existential globally operator $EG\varphi$ is true in a state if there exists an infinite path starting from that state along which φ remains true forever. It could be expressed as the greatest fixpoint:

$$Sat(EG\varphi) = \nu Z. Sat(\varphi) \cap Pre_{\exists}(Z). \quad (3.28)$$

Here, νZ denotes the greatest fixpoint, i.e. the largest set of states satisfying the equation. In other words, the checker starts from the largest possible set and removes states that cannot remain inside $Sat(\varphi)$ indefinitely:

1. initialize

$$Z_0 = S;$$

2. iteratively compute

$$Z_{i+1} = Sat(\varphi) \cap Pre_{\exists}(Z_i);$$

3. stop when $Z_{i+1} = Z_i$.

Since the sequence is monotonically decreasing over the finite state space, this computation also terminates.

All other CTL operators are reduced to the primitive basis $\{true, \vee, \neg, EX, EU, EG\}$, as described in Table 2.5.

After computing $Sat(\varphi)$ for the full CTL assertion, the final check is:

$$S_0 \subseteq Sat(\varphi). \quad (3.29)$$

If this condition holds, then every initial state satisfies the CTL assertion. If it does not hold, then at least one initial state violates the assertion. In that case, the checker selects one state from

$$S_0 \cap (S \setminus Sat(\varphi)) \quad (3.30)$$

and reports it as a witness state showing that the CTL assertion is false.

3.5.7 Assessing LTL Formulas

Unlike CTL, LTL is interpreted over linear execution paths. In the checker, an LTL assertion is evaluated over the explicit reachable graph using `BitSet` state sets. The implemented primitive basis is:

$$true, \quad \neg, \quad \vee, \quad X, \quad U. \quad (3.31)$$

For an atomic predicate p , the satisfaction set is:

$$Sat(p) = \{ s \in S \mid p \text{ evaluates to true in } s \}. \quad (3.32)$$

Boolean operators are evaluated as set operations:

$$Sat(true) = S, \quad (3.33)$$

$$Sat(\neg\varphi) = S \setminus Sat(\varphi), \quad (3.34)$$

$$Sat(\varphi_1 \vee \varphi_2) = Sat(\varphi_1) \cup Sat(\varphi_2). \quad (3.35)$$

The implemented next operator uses universal successor semantics:

$$Sat(X\varphi) = \{ s \in S \mid \forall s' \in S. ((s, s') \in R \Rightarrow s' \in Sat(\varphi)) \}. \quad (3.36)$$

Therefore, a state satisfies $X\varphi$ if all of its immediate successors satisfy φ .

The until operator $\varphi U \psi$ is evaluated as a least fixpoint:

$$Sat(\varphi U \psi) = \mu Z. Sat(\psi) \cup (Sat(\varphi) \cap Pre_{\forall}(Z)), \quad (3.37)$$

where

$$Pre_{\forall}(Z) = \{ s \in S \mid \forall s' \in S. ((s, s') \in R \Rightarrow s' \in Z) \}. \quad (3.38)$$

Operationally, the checker initializes $Z_0 = Sat(\psi)$ and repeatedly adds every state satisfying φ whose successors are already in Z_i , until a fixpoint is reached.

After computing $Sat(\varphi)$ for the full LTL assertion, the checker validates it against the initial states:

$$S_0 \subseteq Sat(\varphi). \quad (3.39)$$

If this condition is false, then some initial state violates the assertion. The checker selects one state from

$$S_0 \cap (S \setminus Sat(\varphi)) \quad (3.40)$$

and reports it as a witness for the failed LTL assertion.

Chapter 4

Results

4.1 Explicit vs. Symbolic Model Checking

During initial development and testing, the proposed tool used an explicit-state model checker. However, this approach led to very long execution times and high memory usage, primarily due to the extremely large state space. As a result, model execution and verification took a significant amount of time, making it impractical for use with large models.

Due to these issues, a symbolic model checker was implemented for the language and used in our case study. The symbolic model checker could operate on a very large total state space, yet it was still limited by the size of the reachable state space and the complexity of representing model transitions using Boolean formulas. The implementation and model source code are publicly available at

<https://github.com/lemontree01/final-compiler>

and

https://drive.google.com/file/d/1W_wuoRigSLpxLLAtwCd7bx2hamjZByBo/view?usp=sharing.

The software developed for this work includes a parser, abstract syntax tree components, model flattener, state graph generator, symbolic BDD-based model checker, CTL formula evaluator, and LTL formula evaluator. The core implementation that

was written by hand contains 67 Java source files consisting of approximately 8,370 lines of code, and 83 type declarations such as classes, interfaces, and enums.

4.2 Case Study: NuSMV Models

4.2.1 Model 0: A Simple Model for LTL and CTL Operators

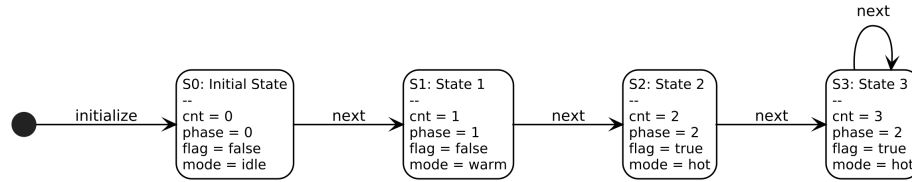


Figure 4-1: The diagram of the model.

A simple model was created to ensure that basic formulas containing LTL and CTL operators were properly verified. This model represents a small deterministic finite-state controller with four state variables:

- **cnt** — a bounded counter with values from 0 to 3
- **phase** — a stage indicator ranging from 0 to 2
- **flag** — a boolean variable
- **mode** — an enumeration with possible values of *idle*, *warm*, and *hot*

The initial variable values are as follows:

$$\text{cnt} = 0, \quad \text{phase} = 0, \quad \text{flag} = \textit{FALSE}, \quad \text{mode} = \textit{idle}$$

At each state transition:

- **cnt** increases by 1 until it reaches 3, after which it remains unchanged.
- **phase** depends on the current value of **cnt**:
 - if **cnt** = 0, then the next value of **phase** is 1

- if `cnt = 1`, then the next value of `phase` is 2
- if `cnt >= 2`, then `phase` remains 2
- `flag` becomes `TRUE` once `cnt >= 1`; otherwise, it remains `FALSE`.
- `mode` changes according to the value of `cnt`:
 - `warm` when `cnt = 0`
 - `hot` when `cnt >= 1`

As shown in Table 4.1, the model checker gave the same verification results as the equivalent model when checked using NuSMV. Using NuSMV the model was verified in **0.06 seconds** with memory use of **34,048 KB**. The model checker used **0.191 seconds** and **1,021 MB** of memory.

Property	Proposed Language & Model Checker	NuSMV
<code>AX(cnt <= 3)</code>	true	true
<code>EX(cnt <= 3)</code>	true	true
<code>X(cnt <= 3)</code>	true	true
<code>AG(cnt <= 3)</code>	true	true
<code>EG(cnt <= 3)</code>	true	true
<code>EF(cnt == 3)</code>	true	true
<code>AF(cnt == 3)</code>	true	true
<code>EU(cnt <= 3, cnt == 3)</code>	true	true
<code>AU(cnt <= 3, cnt == 3)</code>	true	true
<code>U(cnt <= 3, cnt == 3)</code>	true	true
<code>AG((((cnt + 1) >= 1) & ((cnt * 2) >= cnt)) & (((cnt + 2) / 2) >= 1)) & ((cnt % 2) <= 1)) & ((cnt ** 2) >= cnt)) & (-(cnt) <= 0))</code>	true	true
<code>AG(((mode == "idle") (mode == "warm")) (mode == "hot")) & (((flag == true) (flag == false)) & (((cnt == 0) -> (mode != "hot")) & ((cnt != 0) (phase == 0))))</code>	true	true
<code>EF((cnt == 3) & (phase == 2) & (flag == true) & (mode == "hot"))</code>	true	true
<code>AX(cnt == 3)</code>	false	false
<code>EX(cnt == 4)</code>	false	false
<code>AG(cnt == 0)</code>	false	false
<code>EG(mode == "idle")</code>	false	false
<code>EF((cnt == 3) & (mode == "idle"))</code>	false	false
<code>AF(cnt == 0)</code>	true	true
<code>EU(cnt == 0, cnt == 3)</code>	false	false
<code>AU(cnt == 0, cnt == 3)</code>	false	false
<code>U(cnt == 0, cnt == 3)</code>	false	false
<code>AG((cnt - 5) >= 0)</code>	false	false
<code>AG((flag == true) & (cnt == 0))</code>	false	false

Table 4.1: Comparison of verification results between the Model Checker and NuSMV.

4.2.2 Model 1: Priority-Scheduled Multi-Pipeline Real-Time Processing Model

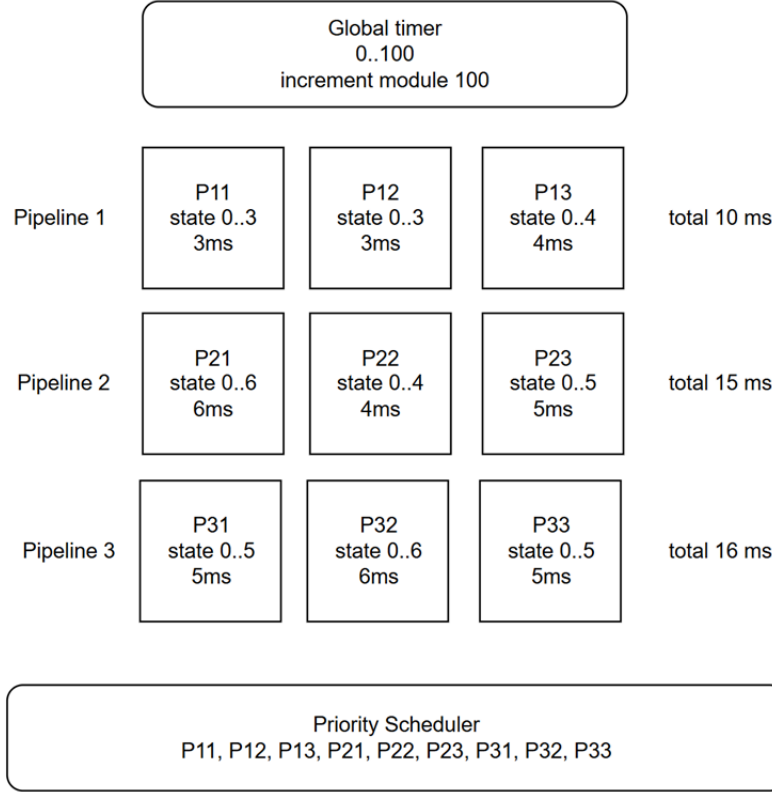


Figure 4-2: The diagram of model 1

The next model used for comparison was an implementation of a data-driven pipeline scheduling system which was originally modeled in NuSMV [53]. The NuSMV source code can be found at [54].

The model consists of 3 independent pipelines, each of which consists of 3 phases which use the same shared processor. Additionally, each phase is represented as a finite-state process with a fixed time. In the given model, Pipeline 1 operates with a period of 20 ms, Pipeline 2 with a period of 50 ms, and Pipeline 3 with a period of 100 ms, and the model includes both a data-driven pipeline (phases executed strictly sequentially) and a synchronized pipeline (according to their activation time). An

error condition is defined if a new period starts when the processor is not idle, leading to a violation. The author of the model expressed the specification as $AG \neg error$. In addition, the model computes the minimum and maximum completion times for each pipeline.

Overall, the model represents 9 periodic tasks that are grouped into 3 pipelines. All of these tasks compete for a single processor. Additionally, each task contains a local variable state, that defines the current execution stage inside the task. The task progresses, only if it has assigned priority using `processor_granted`. The error occurs if the release of a task occurs while it is not in the idle state.

Table 4.2: Overview of the periodic pipeline scheduling model

Element	Description
Global timer	Defines current instance of time and tells which task to start execution at: <i>timeout20</i> , <i>timeout50</i> , and <i>timeout100</i> .
Pipeline 1	Period 20 ms; stages: $P11 \rightarrow P12 \rightarrow P13$; execution times: 3, 3, 4 ms.
Pipeline 2	Period 50 ms; stages: $P21 \rightarrow P22 \rightarrow P23$; execution times: 6, 4, 5 ms.
Pipeline 3	Period 100 ms; stages: $P31 \rightarrow P32 \rightarrow P33$; execution times: 5, 6, 5 ms.
Pipeline stage	Each stage has an internal state variable representing idle, execution progress, and completion.
Scheduler	A fixed-priority scheduler defines which tasks gets the CPU.
Scheduler Priority order	$P11 > P12 > P13 > P21 > P22 > P23 > P31 > P32 > P33$.
Error condition	An error occurs if a stage is not idle when its next period begins.
Verified property	$AG \neg error$, meaning no stage misses its deadline.

For this example, we compared the original NuSMV model using the NuSMV model checker, against the model written in the proposed language along with the symbolic BDD-based symbolic model checker that was developed. In addition, we generated an equivalent NuSMV model from the proposed language model for comparison to validate the original translation.

Table 4.3: Comparison of the original NuSMV model and the revised implementation

Metric	Original NuSMV	Proposed Language & Model Checker	Generated NuSMV from Proposed Language
Total run time	0.06 s	0.586 s	0.06 s
Memory usage	35.1 MB	61.005 MB	35.1 MB
Lines of code	281	118	426
Reachable state space size	1.0×10^3	1.0×10^3	1.0×10^3
Total state space size	4.276×10^9	4.276×10^9	4.276×10^9
Number of modules	10	10	10

Table 4.4: Results of the original NuSMV model and the revised implementation

Compute Value/Result	Original NuSMV	Proposed Language & Model Checker	Generated NuSMV from Proposed Language
MIN [P11.start, P13.finish]	10	10	10
MAX [P11.start, P13.finish]	10	10	10
MIN [P21.start, P23.finish]	25	25	25
MAX [P21.start, P23.finish]	35	35	35
MIN [P31.start, P33.finish]	95	95	95
MAX [P31.start, P33.finish]	95	95	95
MIN [timeout20, P13.finish]	10	10	10
MAX [timeout20, P13.finish]	10	10	10
MIN [timeout50, P23.finish]	25	25	25
MAX [timeout50, P23.finish]	35	35	35
MIN [timeout100, P33.finish]	95	95	95
MAX [timeout100, P33.finish]	95	95	95
Verification result ($AG \neg error$)	true	true	true

In these trials, the execution time and memory used by the NuSMV checker were still better for both the original NuSMV model and the translated NuSMV model, in comparison to the model written in the proposed language and verified using the new model checker. These results can be seen in Table 4.3. Note, however, the focus is on adding type guards and limiting the state space while improving type safety in models, and overall performance is secondary.

Table 4.4 shows the verification results of the assertion properties, as well as the results of the compute operation for the properties of the three versions of the scheduler model. The matching rows indicate that the proposed-language model and its generated NuSMV translation preserve the behavior of the original NuSMV model.

4.2.3 Model 2: A Shared-Bus Cache Coherence Protocol for Three Processors and One Memory Module

Another benchmark model was taken from [55], which describes a simplified cache protocol of 3 processors and 1 memory communicating over a shared single bus. A processor has 3 possible states: *invalid*, *shared*, or *owned*, and it also issues read and write commands using a shared bus. The overall purpose of the model is to ensure that processors can eventually become readable and writable while ensuring coherence. Short descriptions of the model’s components are shown in Table 4.5, and a diagram of the model workflow is shown in Figure 4-3.

Table 4.5: Brief summary of the cache-coherence model

Component	Role
Cache state	Each processor cache is in invalid , shared , or owned .
Processor	Issues bus commands depending on cache state and snoop result.
Bus-device	Tracks bus control variables such as master , waiting , and reply-stall .
Memory	Responds to read requests, may become busy, and may stall replies.
Shared command CMD	Single global bus command observed by all processors and memory.
Checked properties	Eventual readability, eventual writability, and no simultaneous writability of two processors.

The resulting model has type guards and annotations and more explicit decla-

rations of variables and modules. Table 4.6 compares the original NuSMV model, the proposed-language implementation verified on the new model checker, and the NuSMV model generated from the translation of the proposed-language version verified on NuSMV. As in the previous example, the matching properties and results show that the proposed implementation preserves the behavior of the original cache-coherence model for the checked properties, with the same reachable state space size. The results for the generated NuSMV model are included here to support that the translation back into NuSMV preserves the underlying model.

Table 4.6: Comparison of the original NuSMV model, the proposed-language implementation, and the generated NuSMV translation

Metric/Result	Original NuSMV	Proposed Language & Model Checker	Generated NuSMV from Proposed Language
Total run time	0.04 s	6.79 s	0.04 s
Memory usage	37.2 MB	450.0 MB	37.2 MB
Lines of code	155	220	191
Reachable state space size	8,872	8,872	8,872
Total state space size	1.763×10^{11}	1.763×10^{11}	1.763×10^{11}
Number of modules	5	5	5
Verification result ($AG(EF p0.readable)$)	true	true	true
Verification result ($AG(EF p0.writable)$)	true	true	true
Verification result ($AG \neg (p0.writable \wedge p1.writable)$)	true	true	true

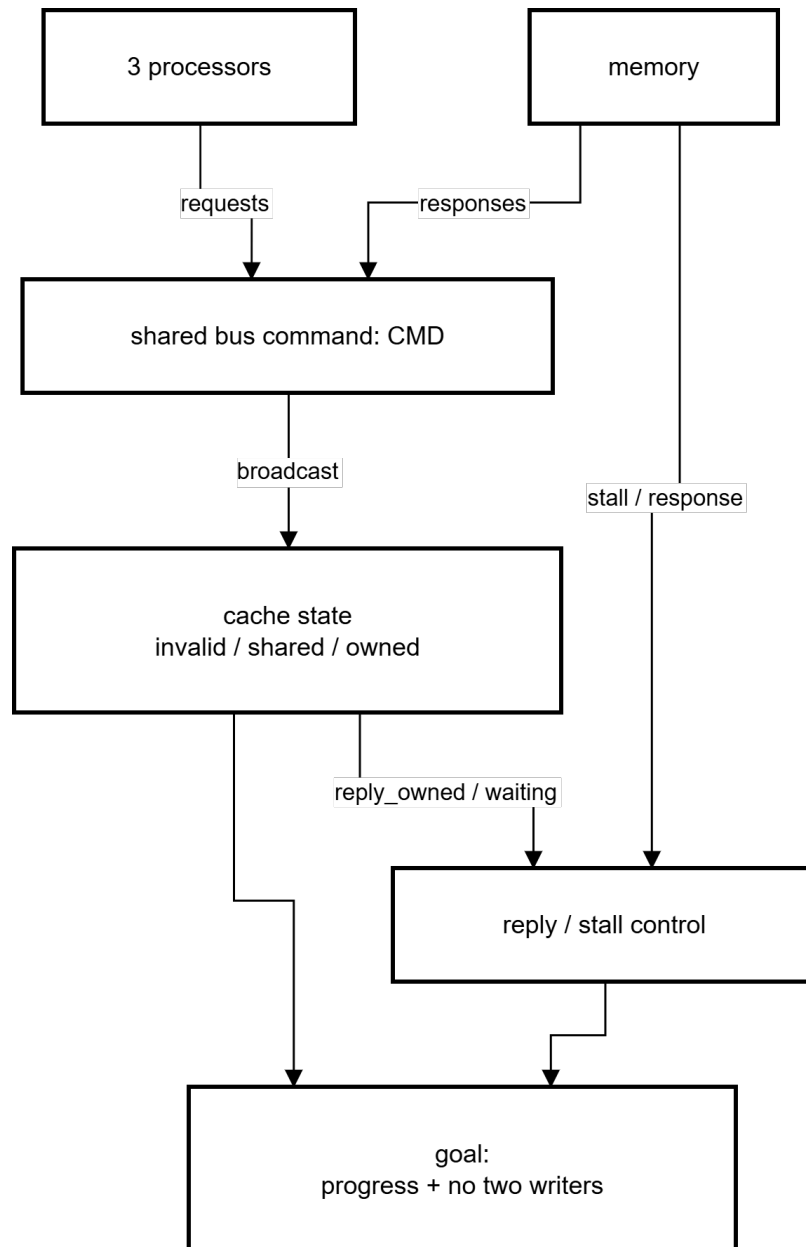


Figure 4-3: The diagram of model 2

4.3 Case Study: SPIN Models

4.3.1 Model 3: Bounded Petri Net Model

The following model is based on [56], and represents a small bounded Petri net consisting of six places and six transitions, as illustrated in Figure 4-4. Each place stores a finite number of tokens, and transitions define the movement of tokens between places according to predefined input and output relations. The original output of running SPIN on the model stated that no errors or assert violations were found while 24 states and 37 transitions were explored.

In the SPIN model, each `Place` variable represents a place, and its value is the token count of that place. The variables are declared and initialized as follows in SPIN:

```
Place P1, P2, P4, P5, RC, CC, RD, CD;
```

```
Place p1, p2, p4, p5, rc, cc, rd, cd;
```

```
init {  
    P1 = 1;  
    p1 = 1;  
}
```

The translation into the proposed language initializes the variables using the syntax:

```
P1 := 1;
```

```
p1 := 1;
```

To translate the transition relations from SPIN to the proposed language is less straightforward. For example, the transition DC is written as follows in SPIN:

```
:: atomic { inp1(P1) -> out2(rc, P2) }    /* DC */
```

This consumes one token from P1 and produces one token in `rc` and one token in P2. To express this in the proposed language, the same transition uses case branching for the individual variables:

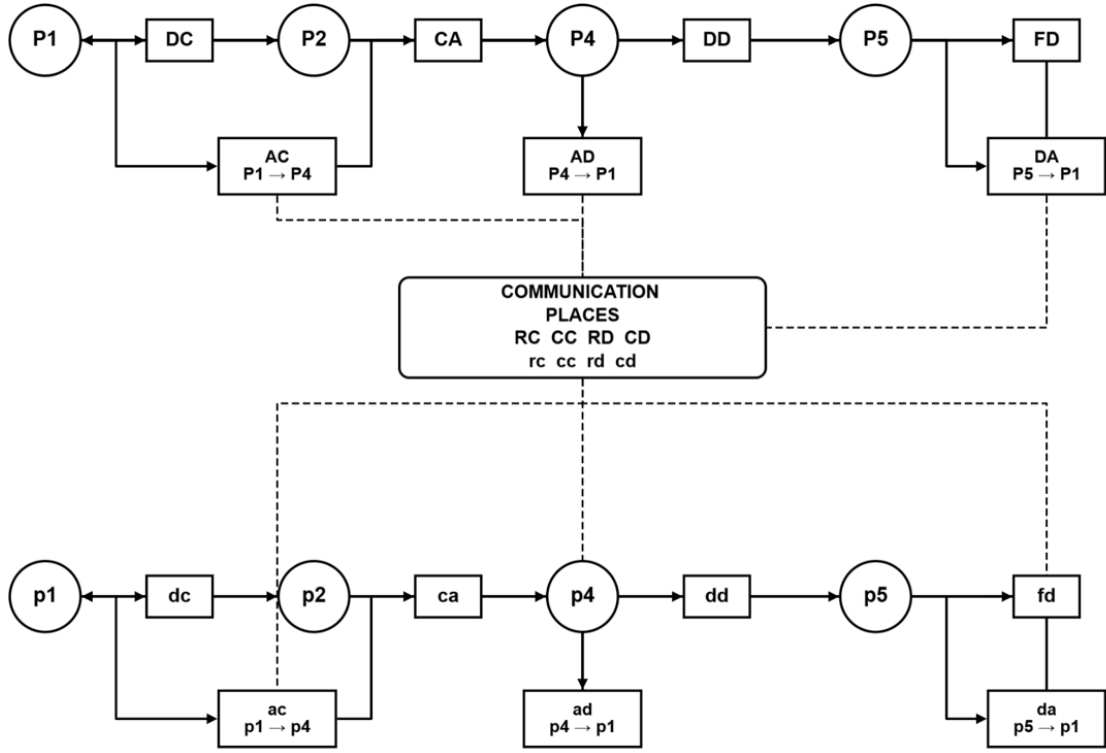


Figure 4-4: The diagram of model 3

```

P1' := case { next_pick_dc => P1 - 1; true => P1; };
P2' := case { next_pick_dc => P2 + 1; true => P2; };
rc' := case { next_pick_dc => rc + 1; true => rc; };

```

Here, the Petri net logic is preserved, while the structure and stepwise execution differs. SPIN/Promela uses native guarded-command choice and atomic execution, while the proposed language does not and relies on a small auxiliary selector variable to encode which enabled transition fires next. Note that we are not attempting to provide a semantically equivalent model in the new language, but rather, a faithful Petri net-level translation.

The Promela model contains embedded assertion checks through the use of the macro `check_model3_invariants` which are used to check that the intended safety conditions hold throughout the reachable state space. In addition to these inline assertions, the model contains three named LTL properties:

- `upper_control_token_conserved: [] upper_control_token_ok`

- `lower_control_token_conserved: [] lower_control_token_ok`
- `all_places_are_1_safe: [] places_are_1_safe`

The first LTL property states that the total number of tokens in the upper control places P1, P2, P4, and P5 always remains equal to one. The second states the same conservation property for the lower control places p1, p2, p4, and p5. The third states that every place in both layers always remains 1-safe, that is, no place ever contains more than one token.

The verification outcomes for these three LTL properties are summarized in Table 4.7. The per-property timing comparison between SPIN, the proposed language, and the translated NuSMV model is reported in Table 4.8. Together, these tables show that all three implementations satisfy the same Model 3 properties, while the measured verification times differ across the tools.

Table 4.7: Model 3 property results across SPIN, the proposed language, and the translated NuSMV model

Property	SPIN	Proposed Language Model Checker	Generated NuSMV
<code>upper_control_token_conserved</code>	True	True	True
<code>lower_control_token_conserved</code>	True	True	True
<code>all_places_are_1_safe</code>	True	True	True

Table 4.8: Model 3 property verification times

Property	SPIN	Proposed Language Model Checker	Generated NuSMV
<code>upper_control_token_conserved</code>	150 ms	240 ms	
<code>lower_control_token_conserved</code>	130 ms	210 ms	130 ms total
<code>all_places_are_1_safe</code>	140 ms	253 ms	

4.3.2 Model 4: ALICE Data Point Processing Framework

The last model of the case study represents the ADAPRO (ALICE Data Point Processing Framework) threading processing framework [57] with one Session environ-

ment, one Supervisor, and two Workers:

$$\text{Session} \rightarrow \text{Supervisor} \rightarrow \{\text{Worker}_1, \text{Worker}_2\}.$$

Session controls the Supervisor and represents the user commands that are input to interact with the system. It may later pause, resume, or stop the Supervisor component. The Supervisor controls the Workers by sending execution commands to them. Each thread in the model uses the same set of control states:

$$\{\text{READY}, \text{STARTING}, \text{RUNNING}, \text{PAUSED}, \text{STOPPING}, \text{STOPPED}, \text{ABORTING}, \text{ABORTED}\}$$

as well as the same command set:

$$\{\text{START}, \text{CONTINUE}, \text{PAUSE}, \text{STOP}, \text{ABORT}\}.$$

For example, the lifecycle for a thread is shown below:

$$\begin{aligned} \text{READY} &\xrightarrow{\text{START}} \text{STARTING} \xrightarrow{\text{CONTINUE}} \text{RUNNING} \xrightarrow{\text{PAUSE}} \text{PAUSED} \\ &\xrightarrow{\text{CONTINUE}} \text{RUNNING} \xrightarrow{\text{STOP}} \text{STOPPING} \rightarrow \text{STOPPED}, \end{aligned}$$

and the abort behavior version is shown here:

$$\text{STARTING/RUNNING/STOPPING} \xrightarrow{\text{ABORT}} \text{ABORTING} \rightarrow \text{ABORTED}.$$

A conceptual diagram of the system can be seen in Figure 4-5.

The following system components were directly translated from SPIN/Promela to the proposed language:

- Session, one Supervisor, two Workers,
- Thread-state domain and command domain,
- Start, resume, pause, stop, and abort effects,

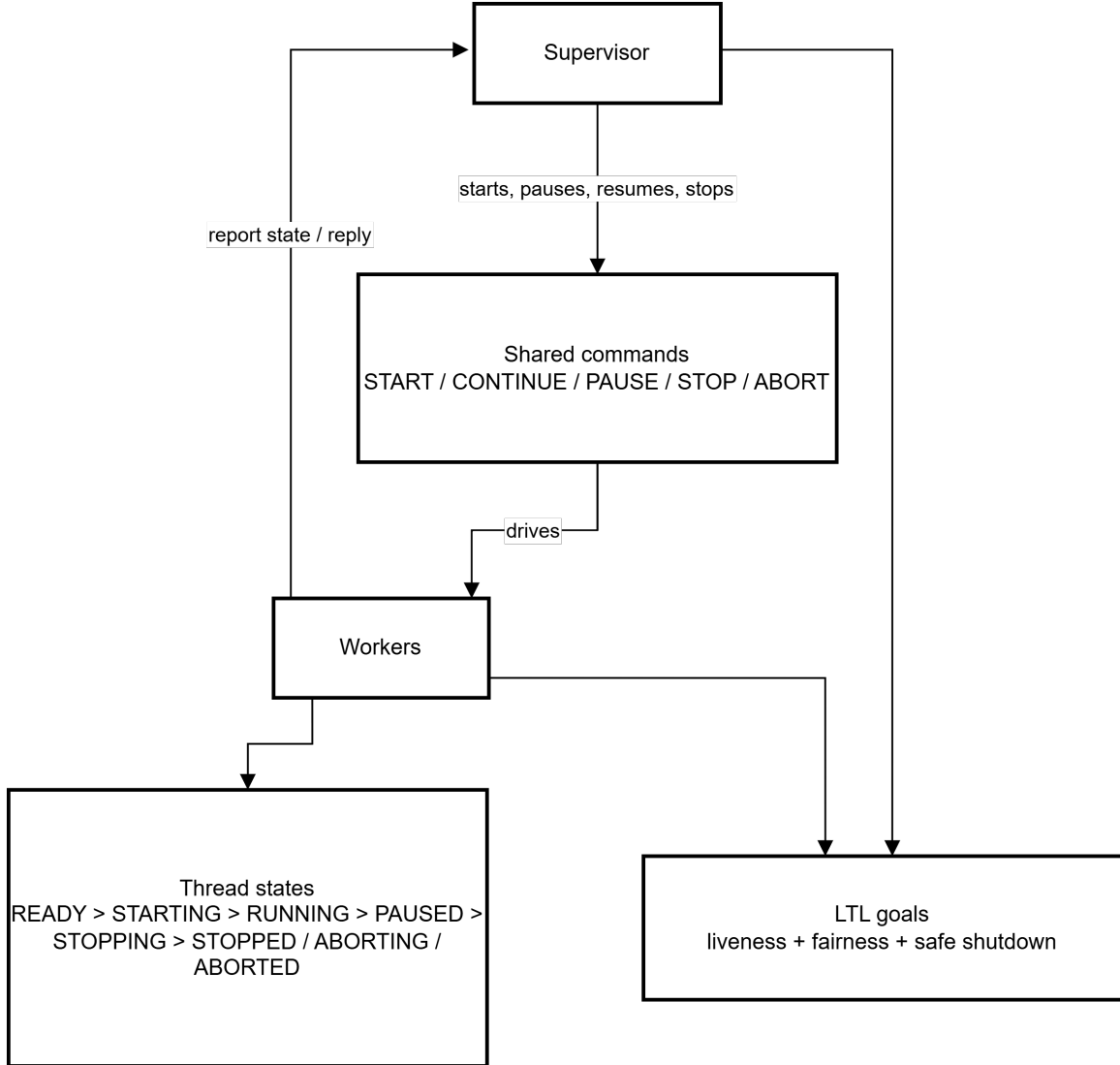


Figure 4-5: The diagram of model 4

- `all_workers_stopped` and `exists_aborted_worker` behavior,
- Special handling of the first Supervisor `CONTINUE`,
- Supervisor and Worker lifecycles

The verification result of the model is shown in Table 4.9. Moreover, the properties that do not fully match and were changed to adapt to the proposed syntax are shown in Table 4.10. In addition, the comparison in Table 4.11 shows that several Model 4 properties are checked and verified faster than the direct CTL-style safety or reachability properties, such as invariants of the form $\text{AG}(p \rightarrow \text{AX}(q))$ or reachability

checks of the form $EF(p)$.

At the same time, the comparison must be treated as not fully identical because not all rows are semantically equivalent. Some proposed-language properties are simplified versions of the corresponding SPIN properties due to expressibility constraints.

Table 4.9: Model 4 property verification results

Property	SPIN	Proposed Language Model Checker
fsm_ready	True	True
fsm_starting*	True	True
fsm_running	True	True
fsm_paused	True	True
fsm_stopping*	True	True
fsm_stopped	True	True
fsm_aborting	True	True
fsm_aborted	True	True
abort_is_undoable*	True	True
supervisor_is_the_first_thread	True	True
supervisor_constructs_workers*	True	True
supervisor_stops	True	True
_after_all_workers_stop*	True	True
supervisor_stops	True	True
_after_some_workers_abort*	True	True
supervisor_never_aborts	True	True
supervisor_halts_last*	True	True
liveness*	True	True
fairness*	True	True

Note. Entries marked with * indicate properties that do not have a directly identical verification expression in the proposed language.

Table 4.10: Model 4 properties that do fully map from SPIN to the proposed language

Property	What does not fully map
fsm_starting	SPIN uses LTL until over executions while the proposed model checks only within allowed and reachable state space.
fsm_stopping	SPIN requires temporal progress from STOPPING while the proposed model modified this to successor safety.
abort_is _undoable	SPIN checks command finality until destruction and reset while the proposed model has no destructor and reset phase.
supervisor _constructs _workers	SPIN refers to construction state 0 while the proposed model starts after construction in READY.
supervisor_stops _after_all _workers_stop	SPIN checks LTL eventual stop unless infinitely often paused while the proposed model uses a CTL-style approximation.
supervisor_stops _after_some _workers _abort	SPIN requires eventual stop unless paused; the proposed model accepts pause, stopping, or stopped as reaction.
supervisor _halts _last	SPIN needs eventual worker halt after supervisor halting while the proposed passing model uses an abstraction.
liveness	SPIN uses weak-until over infinite executions while the proposed model only approximates this behavior.
fairness	SPIN verifies strong fairness but the proposed checker does not implement SPIN fairness search semantics.

Table 4.11: Comparison of SPIN and proposed-language execution time for Model 4

Property	SPIN	Proposed Language Model Checker	Generated NuSMV
fsm_ready	0.01 ms	42.482 ms	
fsm_starting*	2.05 ms	21.293 ms	
fsm_running	12.40 ms	21.644 ms	
fsm_paused	2.80 ms	18.045 ms	
fsm_stopping*	5.49 ms	72.596 ms	
fsm_stopped	3.91 ms	6.455 ms	
fsm_aborting	2.65 ms	14.937 ms	
fsm_aborted	3.04 ms	0.498 ms	
abort_is_undoable*	11.40 ms	0.046 ms	250 ms total
supervisor_is_the_first_thread	1.83 ms	0.120 ms	
supervisor_constructs_workers*	1.68 ms	0.153 ms	
supervisor_stops			
_after_all_workers_stop*	1.72 ms	60.792 ms	
supervisor_stops			
_after_some_workers_abort*	3.22 ms	13.768 ms	
supervisor_never_aborts	1.60 ms	0.510 ms	
supervisor_halts_last*	1.90 ms	0.018 ms	
liveness*	250.00 ms	13.478 ms	
fairness*	31.00 ms	0.033 ms	

Note. The NuSMV value is a combined result for all logical assertion properties. Entries marked with * indicate properties without a directly identical verification expression. Execution times are not strictly comparable.

Chapter 5

Conclusion

5.1 Summary

This thesis developed a proof-of-concept modeling language and verification toolchain for finite-state systems. The main goals were to design and develop a more readable and modern typed modeling language for use with existing models. The results suggest that the proposed modeling language can improve usability by introducing simplified syntax and making declarations and type constraints more explicit.

Overall, the main results of the thesis include:

- Working Proof of Concept: a language and verification toolchain were developed.
- Validation of the approach through a case study involving several different models originally written in other modeling languages.
- Modeling Usability: the language demonstrates clear syntax and type safety.
- Model checking capabilities were implemented (both explicit-state and symbolic approaches were developed).

The proposed syntax was designed to reduce the learning curve of model-checkable modeling languages and to add type annotations with more flexible syntax. Additionally, an attempt was made to implement software tools that parse the proposed

syntax and can verify specified properties written in CTL or LTL using model checking. Although the efficiency of the underlying symbolic model checker was not as good as that of existing solutions like NuSMV or SPIN/Promela, it could be applied to moderately-sized models.

5.2 Future Work

The main focus of future work will be to improve the efficiency of the model checker by implementing similar techniques used in NuSMV [10]; these should reduce the time and memory used during verification. This work will focus on optimizing the BDD representations and applying variable-ordering heuristics [58]. In addition, different verification techniques, such as SAT solvers, could be investigated [59]. Further investigation of explicit-state, symbolic, and model-slicing techniques could also help to better address the state-space explosion problem.

Future work will also include the refinement of the prototype tools. A better interface for the toolchain could be developed, possibly through IDE plugins or a dedicated model-development environment. The modeling language could also be improved by adding additional features for more concise modeling, such as more flexible types, message-passing via channels, macros, reflection, and event-based modeling.

Additionally, attempts to address infinite-state-space models [60] could be made after improving the efficiency of the underlying model checker. This would also include applying various reduction techniques, such as partial-order reduction and improved algorithms for state representation.

Finally, a wider range of systems could be investigated for use in evaluating the language, including heterogeneous systems from multiple domains. This would help to better assess the language’s generality and practical applicability. The expressiveness and syntax of the proposed language could also be improved to model systems from different existing languages with less information loss. Moreover, future development could focus on creating more user-friendly abstractions and constructs, as well as improving the readability of the language. Lastly, visualization tools could be created

to improve the user experience and make system behavior clearer.

Bibliography

- [1] M. H. ter Beek. Review on formal methods for software engineering: Languages, methods, application domains. *Formal Aspects of Computing*, 2025.
- [2] T. McBride and M. Lepmets. Quality assurance in agile safety-critical systems development. In *2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 44–51. IEEE, 2016.
- [3] Jeannette M. Wing. A specifier’s introduction to formal methods. *Computer*, 23(9):8–24, 1990.
- [4] Ekkart Kindler. Safety and liveness properties: A survey. *Bulletin of the European Association for Theoretical Computer Science*, 53:268–272, 1994.
- [5] Aarti Gupta. Formal hardware verification methods: A survey. *Formal Methods in System Design*, 1(2):151–238, 1992.
- [6] Thushan Amarasinghege. Formal computer programming, semantic foundations for software engineering, and lightweight formal methods for programming, April 2025. SSRN preprint.
- [7] A. N. Prior. *Formal Logic*. Oxford University Press, 1963.
- [8] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2001.
- [9] Antti Valmari. The state explosion problem. In *Advanced Course on Petri Nets*, pages 429–528. Springer, Berlin, Heidelberg, 1996.

- [10] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A new symbolic model verifier. In *Computer Aided Verification (CAV 1999)*, volume 1633 of *Lecture Notes in Computer Science*, pages 495–499. Springer, 1999.
- [11] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
- [12] Christine Paulin-Mohring. Introduction to the Coq proof-assistant for practical software verification. In *LASER Summer School on Software Engineering*, pages 45–95. Springer, 2011.
- [13] Lawrence C. Paulson. The Isabelle reference manual. Technical Report UCAM-CL-TR-283, University of Cambridge, Computer Laboratory, 1993.
- [14] Panos Louridas. Static code analysis. *IEEE Software*, 23(4):58–61, 2006.
- [15] P. Chaudhary, L. Agrawal, and A. Ali. Modern programming languages—characteristics and recommendations for instruction. *Issues in Information Systems*, 26(2), 2025.
- [16] A. Bakalov. Higher-level formal languages: A theoretical foundation for software system specification, 2025.
- [17] Egidio Astesiano, Giuseppe Castagna, and Martin Wirsing. CASL: The common algebraic specification language. *Theoretical Computer Science*, 286(2):153–196, 2002.
- [18] Luis Quesada, Fernando Berzal, and Juan C. Cubero. A tool for model-based language specification. *arXiv preprint arXiv:1111.3970*, 2011.
- [19] Ian J. Hayes, Cliff B. Jones, and J. E. Nicholls. Understanding the differences between VDM and Z. *ACM SIGSOFT Software Engineering Notes*, 19(3):75–81, 1994.

- [20] B. A. Catron and S. R. Ray. ALPS: A language for process specification. *International Journal of Computer Integrated Manufacturing*, 4(2):105–113, 1991.
- [21] C. N. Dean and Michael G. Hinchey, editors. *Teaching and Learning Formal Methods*. Morgan Kaufmann, 1996.
- [22] M. D. Fraser, K. Kumar, and V. K. Vaishnavi. Informal and formal requirements specification languages: Bridging the gap. *IEEE Transactions on Software Engineering*, 17(5):454–466, 1991.
- [23] Héctor A. López and Thomas T. Hildebrandt. Three decades of formal methods in business process compliance: A systematic literature review. *arXiv preprint arXiv:2410.10906*, 2024.
- [24] Manfred Broy, Achim D. Brucker, Alessandro Fantechi, Mario Gleirscher, Klaus Havelund, Matthias A. Kuppe, et al. Does every computer scientist need to know formal methods? *Formal Aspects of Computing*, 37(1):1–17, 2024.
- [25] V. K. Pandey, J. Gowrishankar, and S. Kashyap. Accelerating computer architecture development using formal methods. In *2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pages 1–6. IEEE, 2024.
- [26] Wolfgang Grieskamp. Is formal verification practical? In *6th International Workshop on Formal Methods for Blockchains*, 2025.
- [27] Leanna Rierson. *Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance*. CRC Press, 2017.
- [28] Richard Zurawski and MengChu Zhou. Petri nets and industrial applications: A tutorial. *IEEE Transactions on Industrial Electronics*, 41(6):567–583, 1994.
- [29] Rance Cleaveland and Gerald Lüttgen. A logical process calculus. *Electronic Notes in Theoretical Computer Science*, 68(2):33–50, 2002.

- [30] Sheldon B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, 1978.
- [31] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, et al. The nuXmv symbolic model checker. In *International Conference on Computer Aided Verification*, pages 334–342. Springer, 2014.
- [32] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer.
- [33] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.
- [34] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UP-PAAL. In *Formal Methods for the Design of Real-Time Systems*, pages 200–236. Springer, 2004.
- [35] Christine Paulin-Mohring. Introduction to the calculus of inductive constructions. *All About Proofs, Proofs for All*, 55, 2015.
- [36] Gérard Huet. The Gallina specification language: A case study. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 229–240. Springer, 1992.
- [37] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752. Springer, 1992.
- [38] Matt Kaufmann and J Strother Moore. ACL2: An industrial strength version of Nqthm. In *Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS '96)*, pages 23–34. IEEE, 1996.

- [39] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems*, 21(3):502–526, 1999.
- [40] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- [41] Marco Bozzano, Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. *nuXmv 2.1.0 User Manual*. Fondazione Bruno Kessler, 2014.
- [42] Gerard J. Holzmann. *Basic SPIN Manual: Introduction to PROMELA and SPIN*. AT&T Bell Laboratories, Murray Hill, NJ, 1990.
- [43] Nicholas Rescher and Alasdair Urquhart. *Temporal Logic*. Springer Science & Business Media, 2012.
- [44] Thomas Hafer and Wolfgang Thomas. Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree. In *International Colloquium on Automata, Languages, and Programming*, pages 269–279, Berlin, Heidelberg, July 1987. Springer Berlin Heidelberg.
- [45] Alan Martin. Adequate sets of temporal connectives in ctl. *Electronic Notes in Theoretical Computer Science*, 52(1):21–31, 2002.
- [46] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.
- [47] Kristin Yvonne Rozier. Linear temporal logic symbolic model checking. *Computer Science Review*, 5(2):163–203, 2011.
- [48] A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

- [49] Randal E. Bryant. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 236–243. IEEE, 1995.
- [50] Scott E. Hudson, Frank Flannery, and C. Scott Ananian. *CUP User’s Manual*, 1999. Accessed: 5 April 2026.
- [51] G. Klein, S. Rowe, and R. Décamps. *JFlex User’s Manual*, 2023. Accessed: 5 April 2026.
- [52] M. C. Browne, Edmund M. Clarke, and Orna Grümberg. Characterizing Kripke structures in temporal logic. In *Colloquium on Trees in Algebra and Programming*, pages 256–270. Springer, 1987.
- [53] Sergio Campos, Edmund M. Clarke, and Mihalik Minea. Symbolic techniques for formally verifying industrial systems. *Science of Computer Programming*, 29(1–2):79–98, 1997.
- [54] Fondazione Bruno Kessler. `periodic.smv`, n.d. Accessed: 17 March 2026.
- [55] Fondazione Bruno Kessler. A model of the GIGAMAX cache coherence protocol, n.d. Accessed: 17 March 2026.
- [56] SPIN Verification Examples and Exercises. SPIN verification examples and exercises. Michigan State University, 1997. Accessed: 26 March 2026.
- [57] J. Lång, Keijo Heljanko, and M. Nieminen. Model checking a C++ software framework: A case study. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*, pages 1026–1036. ACM, 2019.
- [58] Nina Narodytska and Toby Walsh. Constraint and variable ordering heuristics for compiling configuration problems. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 149–154, January 2007.

- [59] Wei Gong and Xiaolong Zhou. A survey of SAT solver. In *AIP Conference Proceedings*, volume 1836, page 020059. AIP Publishing LLC, June 2017.
- [60] Bernard Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. Université de Liège, 1998.