Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Verlixir: Verification of Message-Passing Systems

Author:
Matthew Neave

Supervisor:
Dr. Naranker Dulay

Second Marker:
Charles Pert

June 17, 2024

Abstract

Distributed algorithms are difficult to prove and reason about. With the rapid rise in cloud-based clusters, the need for developing robust distributed algorithms is seen as a critical requirement by service providers and has sparked new interest in developing tools for reasoning about distributed algorithms.

This project proposes and evaluates Verlixir, a verification-aware language for specifying and verifying message-passing systems. Verlixir supports three modes of operation. Simulation mode can be used to execute and observe the system in a controlled environment. Verification mode can verify the system against a set of properties. Finally, parameterization mode can be used to guarantee the system behaves across different configurations.

A qualitative evaluation of Verlixir demonstrates the capability to detect violations of properties in distributed algorithms, such as Paxos and Two-Phase Commit. Verlixir enhances Elixir programs with linear temporal logic properties, predicates and function contracts to specify the desired behaviour of systems. Property violations produce Elixir-friendly counterexamples that can be used to debug the system. Counterexamples show process interleaving and message-passing interleaving that results in a property violation.

Verlixir enables developers to define message-passing systems in terms of safety and liveness. Distributed algorithms can be prototyped and verified in a safe environment. Verlixir replaces the need for writing complex models in bespoke specification languages, which have to be maintained alongside implementation in modern programming languages. This is all achieved while maintaining the original program semantics, such that, after verification, the system can be run in production.

Acknowledgements

I am grateful to my supervisor, Dr. Naranker Dulay, for his guidance and support throughout this project.

My sincerest gratitude goes to my family, for their continued support through my studies.

Finally, a word of thanks to my friends, for their encouragement and motivation.

Contents

Chapter 1

Introduction

With the rise of cloud-based clusters, developing robust distributed algorithms is becoming an increasingly difficult problem and the need for vigorous methodologies to verify the correctness of these algorithms has intensified. Distributed systems are interesting, as they provide performance and reliability benefits over centralised systems. They also provide scalability, modularity, and availability improvements [68].

Modern programming languages have been developed to support distributed algorithms that rely on message passing as a means of communication. Common message passing abstractions involve the use of channels (e.g. Go [10]) or actors [30] (e.g. Erlang [13]). At a high level, message-passing systems can be easier to reason about than a common alternative, shared memory. However, message-passing systems are often distributed across multiple nodes, which can introduce challenges in reasoning about the correctness of a system [69].

Verification tools have been developed to support determining the correctness of systems. For example, first-order automated theorem provers such as Z3 [20] and formal specification languages like TLA+ [6]. These tools allow systems to be modelled, and specifications to be defined that can then be used to prove properties over these systems. However, despite the capabilities these tools provide, they often place a burden on developers to write and maintain models of systems alongside their actual implementation. This often leads to a paradigm shift away from system implementations that were designed in, for example, imperative programming languages such as C. Modern programming languages, for example Dafny [18], solve this issue by directly integrating Floyd-Hoare style logic verification alongside the implementation. This report aims to extend this notion to distributed, message-passing systems.

This report discusses the modelling of message passing, actor-based programs and the verification of their adherence to a specification, using Elixir as a target language to support the verification of real-world systems.

1.1 Objectives

Much work has gone into verifying algorithms and programs such as various theorem provers and model checkers. While these tools were initially designed to allow developers to write specifications for how an algorithm should behave in bespoke specification language, more recently verification tools have been designed that can be directly applied to programs written in programming languages such as C [43]. A more recent advancement is support for verifying concurrent programs. However, much of this work has used global shared memory as an implementation for specifying process communication [57]. A common inter-process communication, alternative to shared memory, is message passing. We aim to extend the research in this area, to support the simulation and verification of message-passing systems, in particular, those which follow the actor model [30]. This project sets out to accomplish the following objectives:

- Integrate the verification of formal specifications into modern, message-based programming languages.

- Check that the behaviour of a system is consistent across different configurations.
- Design a framework for simulating the behaviour of large-scale distributed systems.
- Ensure verified systems can be run directly in production.
- Apply the aforementioned techniques and tooling to real-world systems implemented in Elixir.

The current research in the area of verifying modern programming languages presents many challenges for extending this notion to a message-passing system. We define a verification-aware language, as one which combines implementation and verification capabilities into a single language. State-of-the-art verification-aware languages such as Dafny avoid concurrent execution due to the challenges it can introduce to verification [51]. To verify a distributed system in this context, it is instead left to the user to model the system in a manner that it can be sequentially executed. Tools such as Gomela [44], support the verification of concurrent execution, where communication is achieved across channels in Go. However, they do not support formal specifications or the extensive verification of safety and liveness properties. Instead, they are limited to detecting deadlocks. The existing research means we must sacrifice either, the ability to verify concurrent execution or the ability to verify complex properties.

1.2 Contributions

This report introduces Verlixir, a verification-aware programming language for message passing. We believe Verlixir to be the first language to support the verification of safety and liveness properties of message-passing, actor-based systems. Verlixir programs compile to byte-code, so they can be run in production environments, such as on the Erlang Virtual Machine. They also guarantee system correctness under specified system properties.

Verlixir is capable of verifying multiple properties of highly concurrent programs and reporting back counterexamples. Chapter 4 provides an overview of what Verlixir is and how it can be used. We then provide a detailed explanation of the design and implementation of Verlixir in chapter 5.

A subsequent contribution of this project is LTLixir. LTLixir is a specification language, which supports the direct integration of linear temporal logic, propositional logic and contract design, with Elixir programs. Specifications can be simulated and verified using Verlixir. This allows for simple prototyping of complex, real-world distributed systems, in a controlled environment.

Chapter 2

Background

This chapter aims to provide all the required background knowledge to understand the concepts discussed in the report. The contribution of the report involves extending a concurrent, message-passing language to be verification-aware. We define a verification-aware language as a language that strongly couples specifications and implementations. To understand how this can be done with a message-passing language, we must first form strong fundamentals in concurrency and existing verification techniques.

We start by introducing core concurrency concepts in section 2.1. We will explore memory models, safety, liveness and fairness. Section 2.2 will introduce model checking and compare some of the existing state-of-the-art model checkers. Finally, we will discuss what a verification-aware language is and why they are important to modern system design in 2.3

2.1 Concurrency

Concurrency introduces a notion for multiple components of a program to execute out-of-order. We saw in the previous section how multiple processes can be composed and treated as a single execution. For example, given two processes with disjoint alphabets, the parallel composition can result in any interleaving. This section aims to explore further in-depth the principles of concurrency, moving away from a mathematical representation and looking at higher-level concepts such as consistency models and temporal logic.

Concurrency is a core concept in classical distributed algorithms, such as the Paxos algorithm [54], Raft [52] and Dining Philosophers [56]. The capability for concurrency has grown with better hardware. Concurrency introduces the idea of consistency models [32] to help reason about executions of multi-threaded systems. Lamport introduced sequential consistency [33] as a strong safety property for concurrent systems. Sequential consistency can be informally reasoned about by considering a single-core processor: if multiple threads are executed in parallel on a single-core processor, only one instruction can be executed at a time. This means that the result of any execution forms a total order, consistent with the order of operations on each individual process. For example, consider a new composition where a sequence of events has been executed by a teacher, $(teach \rightarrow teach \rightarrow)$, and a student is scheduled to execute next. Under the sequential consistency model, the student must observe the same order of events as the teacher.

Weaker memory models exist, which allow us to model systems that do not guarantee sequential consistency (for example, systems running on multi-core processors). Under these models, the instructions of a thread may be reordered (i.e. execute out-of-order) which introduces weak behaviours that we would not observe under sequential consistency. For example, total store ordering is a weaker memory model, that allows the reordering of write-read operations on different memory locations within a single thread. For the examples discussed in this report, we will assume a sequentially consistent model.

2.1.1 The Actor Model

The actor model [63] is a model of concurrent computation that treats actors as primitive components of the system. A concurrently executed actor is defined in terms of three key behaviours.

- An actor can send a number of messages to other actors
- An actor can create a number of new actors
- An actor can act in response to a message it receives

Tony Hoare introduced Communicating Sequential Processes (CSP), a mathematical notation for defining processes and interactive systems [27]. CSP provides a framework for reasoning about the behaviour of concurrent systems which has influenced distributed algorithms [28], model checking [29] and many other related research fields.

The CSP model uses channels to communicate between actors. A channel is a communication medium that allows processes to send and receive messages. Under this model, an actor can output a value v on a channel c, using the ! operator.

$$c!v$$

Similarly, an actor can input a value x on a channel c using the ? operator.

$$c?x$$

The concept of channels has been used in modern programming languages, such as Go [10], as well as verification modelling languages like Promela [50]. Akin to channels, we introduce mailboxes. Under a mailbox model, an actor has a designated mailbox, where messages are stored. An actor can read messages from its mailbox and act upon them. For example, Elixir [11], handles all communication through process-owned mailboxes. Channels and mailboxes give us an alternative model to that of shared memory.

In a shared memory model, a shared memory region is established in which multiple processes can read and write. Figure 2.1 shows a basic example of two processes that write to a shared in-memory array. Due to how often we see shared memory used in large-scale distributed systems, much work has been done in the verification of these systems using shared memory models. For example, Jon Mediero Iturrioz used Dafny [18] to prove the correctness of concurrent programs that implement shared memory [17].

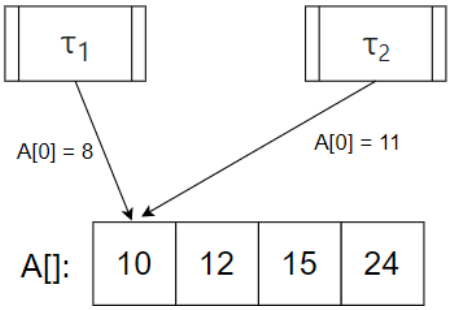

Figure 2.1: An example of two processes writing to a shared in-memory array

By contrast, figure 2.2 shows an example of how actors behave using mailboxes. The mailbox is not necessarily first in, first out (FIFO) but often implementations tend to be.

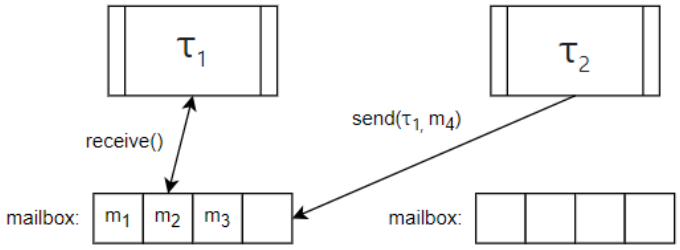

Figure 2.2: An example of actors sending and receiving messages under the actor model

2.1.2 Temporal Logic

First-order logic, or predicate logic uses quantifiers to reason about the truth of statements. For example, the statement $(\forall x \in \mathbb{N} \,.\, x > 0)$ is true for all natural numbers $\mathbb{N}$, and uses the universal quantifier $\forall$, to quantify over all x. We can also use the existential quantifier $\exists$, to reason about the existence of an element in a set. First-order logic is a powerful tool for reasoning about the truth of statements, but it cannot reason about time and change. We introduce modal logic for this purpose.

$$\langle \mathrm{A} \rangle \quad \models \quad p \mid \top \mid \neg\langle \mathrm{A} \rangle \mid \langle \mathrm{A} \rangle \wedge \langle \mathrm{A} \rangle \mid \Box\langle \mathrm{A} \rangle$$

Where A is a modal formula, p is an atomic proposition, $\top$ represents 'truth' and $\Box A$ reads box A. Using rules from first-order logic we can introduce disjunct, implication and if-and-only-if. We can also introduce the second modal operator, $\Diamond A$ which reads diamond A.

$$\Diamond\langle A \rangle \models \neg\Box\neg\langle A \rangle$$

Depending on the circumstances that box and diamond are applied, they have different readings. For example, in temporal logic, $\Box A$ can read as 'always A' and $\Diamond A$ can read as 'sometimes A', informally, it can be useful to think of $\Box$ similarly to $\forall$ and $\Diamond$ to $\exists$.

Saul Kripke introduced Kripke semantics [34] for reasoning about temporal logic. For example, consider modelling a system based on our student and teacher processes. We let M be the model of the system and s represent a singular state the system can be in. Typically, s is the initial state of the system. If we are given a temporal formula A, we can now define the syntax for the truth of A in state s of the model M.

$$(M, s) \models A$$

To reason formally about what it means for A to hold in state s, Kripke provided formal definitions for the base and inductive definitions of A.

We finally extend this understanding of temporal logic to linear temporal logic (LTL) sometimes written as linear-time temporal logic. LTL allows us to reason about the time and change of a model. Before we provide some examples, we introduce a final temporal operator, U, which reads until. The formula $\phi U \psi$ defines the truth of ϕ until ψ holds. We call this 'strong until' as there must exist a state where ψ becomes true. 'Weak until' W, can also be defined, which loosens the restrictions such that ϕ could hold for the entire execution.

We can now explore a few basic examples of LTL formulas as well as provide some intuition behind them. We define a set of atomic propositions, $AP = \{study, sleep, tired, exam\}$.

$\Box$ sleep	Always sleeping
tired U sleep	Tired until sleeping
$\Box$(study $\Rightarrow$ tired)	Studying implies always tired
$\Box$ study $\Rightarrow \Diamond$ exam	Always studying implies eventually an exam

Alongside LTL, other forms of temporal logic exist, such as Computation Tree Logic (CTL) [35] and Alternating-time Temporal Logic (ATL) [38]. CTL introduces path quantifiers to reason about specific traces through a model, and ATL introduces the idea of agents, where agents can work in coalitions to achieve a goal in the system. Temporal logic is an important concept in the model checking of systems [36], see chapter 2.2.

2.1.3 Safety and Liveness

Safety and liveness are properties that can be specified about systems. A safety property can be intuitively thought of as a property such that nothing bad happens, and a liveness property is where something good will happen. For example, something bad could be a deadlock in a system, and something good could be that the system will eventually reach a consensus. We define safety informally as, given a finite execution E and a state s such that s is the final state in the execution, we can say that a safety property P holds if P is true in s and all previous states in E. If s violates P, then E violates P. Unlike with safety, we cannot determine the truth of a liveness property at s, we must instead inspect an infinite execution E'. We express these properties as temporal formulas. For example, we can specify a simple liveness property to ensure our student will always study again.

$$\Box\Diamond \text{study}$$

To help understand why this is a liveness property, consider two states, s_1 and s_2. Take the assignment of *study* to be $\{s_1\}$ i.e., *study* is true only in s_1. Regardless of if we want to reason about the truth of the formula at s_1 or s_2, we cannot, as the box operator requires the formula to hold in all states. Hence, we would have to inspect the current state, as well as an infinite future execution from the state, to determine the truth of the formula. Because no single state exists where we can evaluate the truth of the formula, we can convince ourselves it is a liveness property.

We use the same assignment of study to reason about a new property.

$$\Box \text{ study}$$

We can understand intuitively why this property is a safety property by considering s_2. As s_2 is not in the assignment of study (i.e. *study* is false in s_2), any execution that passes through s_2 will violate the property. As we can determine the truth of the property with a finite execution, we can deem the property a safety property.

By using both safety and liveness properties, we can define a 'correct' system, through the evaluation of these temporal formulae.

2.1.4 Fairness

Fairness introduces more properties that can be defined using temporal formulae. Fairness properties do not target the specification of the system in the same way that other properties we have looked at do. Instead, fairness properties are constraints on the scheduling of the system. They aim to fairly select which process to execute next. Without fairness, a system could favour the scheduling of process A while never progressing with process B. We will discuss two flavours of fairness, weak fairness and strong fairness. Properties that hold under weak fairness also hold under strong fairness, hence strong implies weak. To define the fairness properties, we must first define what it means for an event to be enabled. An event (or action) A of a process algebra is enabled if it can be executed in the current state. We will use the notation A_E to denote an enabled event, for example, $study_E$ denotes the study event is executable in the current state. We now define weak fairness (WF) and strong fairness (SF).

$$WF\,A \equiv \Diamond\Box A_E \Rightarrow \Box\Diamond A$$
$$SF\,A \equiv \Box\Diamond A_E \Rightarrow \Box\Diamond A$$

We can informally define weak fairness as: if an event is continuously enabled, it is executed infinitely often. Similarly, strong fairness reads: if an event is repeatedly enabled, it is executed infinitely often.

2.2 Model Checking

Model checking is the process of determining if a finite-state machine (FSM) is correct under a provided specification. It typically involves enumerating all possible states of an FSM and ensuring the correctness of each state. For example, given a model M and a property φ, if no state of M violates φ, then we can say M satisfies φ. In software development, model checkers are beneficial in providing guarantees for safety-critical systems as well as concurrent systems. Concurrent systems can often cause issues with uncommon instruction execution interleaving that are not easily identifiable until long into a runtime. For example, deadlocks can occur when instructions being run by two processes are dependent on one another making progress. A simple example of a deadlock that can occur, is the following interleaving of instructions executed by two processes, τ_1 and τ_2.

τ_1 : acquire lock A	τ_2 : acquire lock B
τ_1 : acquire lock B	τ_2 : acquire lock A
τ_1 : release locks	τ_2 : release locks

An interleaving such as (τ_1, τ_2, τ_1, τ_2, ...) results in τ_1 blocking until it can acquire lock B, and τ_2 blocking until it can acquire lock A, hence the program is in a deadlock. Due to the nature of concurrent systems, we could run our program and never experience this interleaving of instructions from occurring, hence we could deem our program deadlock-free. Instead, by abstracting our program as a model, and verifying the correctness using a model checker, we could exhaustively check all possible states (interleaving of concurrent processes) and catch this deadlock.

Alongside determining progress can be made within a system, model checkers are also used to guarantee the correctness of a specification. To demonstrate, we model a very simple 24-hour clock, where at each time step, we progress time by an hour.

$$\tau_1 : \text{time} \leftarrow \text{time} + 1$$

Unlike the previous example, this process can always make progress so will not result in a deadlock, however, it is not a correct implementation of a 24-hour clock. We would like our 24-hour clock to only represent times in the range 1 to 24. By introducing a specification alongside our model, we can use a model checker to determine if all the states of our program adhere to the specification. In this instance, we would need to specify a bound over our time variable.

$$\{\text{time} \mid \text{time} \in \mathbb{N}, 1 \leq \text{time} \leq 24\}$$

This is a simple example of a specification, that we can write in a specification language and use in tandem with our model to check the correctness of using a model checker. For a given model M and a property ϕ, we formally define model checking as the process of computing if $M \models \phi$ (satisfiability).

2.2.1 A Comparison Of Model Checkers

Many model checkers have been invented for this reason, each with different focuses and specification languages. This section will comment on some of the more common model checkers and discuss their functionalities. We first provide an overview of the capabilities and limitations of many model checkers, before providing a more in-depth look into model checkers best aligned with the goals of this report. Table 2.1 provides a comparison of some available model checkers.

The table compares support for Linear Temporal Logic (LTL) and Computation Tree Logic (CTL). It also shows which model checkers support probabilistic systems. For example, some systems exhibit random behaviour, such as wireless sensor networks, which rely on random wait times [67]. Finally, it shows which systems support modelling concurrent behaviour.

Model Checker	LTL Support	CTL Support	Probabilistic	Concurrency Support
PAT	Yes	No	Yes	Yes
BLAST	No	No	No	Limited
SPIN	Yes	No	No	Yes
TLC	Yes	No	No	Yes
PRISM	Yes	Yes	Yes	Yes
NuSMV	Yes	Yes	No	Yes
UPPAAL	No	No	Yes	Yes

Table 2.1: Comparison of Model Checkers

PAT

Process Analysis Toolkit (PAT) is a self-contained framework to support composing, simulating and reasoning of concurrent, real-time systems [2]. PAT is based on Tony Hoare's CSP and extends the language using its library called CSP#. CSP# is a superset language of the original CSP, hence all classical CSP models can be verified with PAT. PAT has shown to be capable of verifying classical concurrent algorithms such as the dining philosophers problem. Alongside its verification capabilities, the PAT toolkit can be used to simulate real-world scenarios over specifications.

PAT's ability to determine the correctness of classical process algebra means it is a strong, widely applicable model checker.

BLAST

BLAST is an automatic verification tool for checking the temporal safety properties of C programs. Given a C program and a temporal safety property, BLAST either statically proves the program satisfies the property or provides an execution path that exhibits a violation of the property [3].

Where BLAST differs from PAT, is that it no longer relies on process algebra. The model checker is capable of running directly on a subset of C programs where no intermediate modelling is required. As an end-user tool, this is more generally applicable than PAT; there is no burden on developers to think about how to model their systems with process algebra and instead can directly get safety guarantees from their programs. BLAST handles the translation of C programs to an abstract reachability tree (ART), a labelled tree that represents a portion of the reachable state space of the program. Using a context-free reachability algorithm on this representation of a C program means temporal properties can be checked, without the end programmer being required to think about what the control-flow automata for the program will look like.

BLAST falls short when model-checking large C programs. More importantly, it is unable to provide any guarantees on concurrent programs. We are primarily concerned with concurrent systems in this report, as Elixir is a language that is designed for building concurrent systems.

PRISM

PRISM is a probabilistic model checker, a tool for formal modelling and analysis of systems that exhibit random behaviour or probabilistic behaviour [4]. It has been used to analyse systems implementing randomised distributed algorithms. PRISM relies on Markov decision processes and probabilistic automata to model systems which are both probabilistic and concurrent.

PRISM also supports specifications with temporal logic. This means that PRISM can be used to specify liveness properties under probabilities. For example, we could specify a property, that the probability of a crash is at most 0.01, using the formula: $P_{\leq 0.01}[$ F "crash" $]$.

PRISM has support for many features that are ideal for verifying systems. The main limitation of PRISM is that the modelling language is not true to modern programming languages. It models a high-level system design, and would not support some of the intricacies of a modern language such as Elixir, naturally.

TLC

In 1980, Leslie Lamport formulated the Temporal Logic of Action (TLA) [5]. TLA is a logic system for specifying and reasoning about concurrent systems. Both the systems and their properties are represented in the same logic, so that the assertion that a system meets its specification, can be expressed by a logical implication.

TLA is capable of specifying complex systems but in a typically verbose manner. Leslie Lamport introduced TLA+ [6], combining mathematical ideas with concepts from programming languages to create a specification language that would allow mathematicians to write specifications in 20 lines as opposed to 20 pages.

Furthering on from Leslie Lamport's discovery of these specification languages, Lamport created TLC [7], a model checker for the verification of TLA+ specifications. Similarly to BLAST, TLC builds a finite-state machine from the specification so the model checker can verify and debug invariance properties over it. TLC has been used to verify many large-scale, real-world systems specified in TLA+. Not only does it verify temporal properties of TLA+ specifications, but it can also model check PlusCal [8] algorithms. PlusCal is an algorithm language aimed to resemble that of pseudocode, but PlusCal algorithms can be automatically translated to TLA+ specifications to be reasoned about formally with TLC. We have already come across the concept of model-checking algorithms as opposed to specifications with BLAST, but instead of being strictly bound to the C programming language, PlusCal provides a more general framework agnostic of a choice of programming language allowing developers to separate reasoning about algorithms from their respective programs.

SPIN

Spin is an efficient verification system for models of distributed software systems. It has been used to detect design errors in applications ranging from high-level descriptions of distributed algorithms to detailed code [9]. Spin has a specification language, Process Meta Language (Promela), which the model checker uses to prove the correctness of asynchronous process interactions. Spin supports asynchronous process communication through channels, where processes can send and receive messages. Spin constructs labelled transition systems for respective processes from Promela specifications, which it goes on to use for scheduling and to reason about properties of the model. Because many programming languages, such as Go [10] rely on the creation of channels for asynchronous communication between processes, Promela becomes a natural solution to modelling these systems.

Summary

We have discussed a selection of model-checkers and what their primary focus is. Many existing model-checkers have been originally designed to prove specifications over sequential models. Some have taken this further and applied model checking directly over programming languages, such as BLAST. Other model-checkers have introduced some primitives for reasoning about concurrency. TLC allows for the specification of processes and using structures can begin to specify shared memory. Similarly, Spin allows processes to be specified and supports the creation of channels for communication. Despite this, none of the model checkers discussed, include message-passing as a first-class construct. To reason about message-passing models, such as the actor model, work has to be done to formalise actor-based constructs. This makes specifying actor-based systems, such as systems written in Elixir, a non-trivial task.

2.3 Related Work

Much work has gone into model checking, theorem-proving and verifying the implementations of systems. For Elixir, there are tools such as dialyzer [23], which statically analyse Elixir programs for type errors or dead code. Whilst these tools provide Elixir developers better guarantees, it does not verify the correctness of a system as a whole. Elixir also has libraries for property-based testing, such as PropEr [46], which can be used to generate random test cases for a system. Property-based testing randomly generates inputs to test a system, which can be useful for finding edge cases that

unit tests may not cover. However, property-based testing does not provide guarantees about the correctness of a system, instead, it is used to find bugs in code. Work has also gone into verifying message-passing in Elixir, using binary session types [45]. This approach ensures two processes, communicating over compatible protocols, avoid certain communication errors (i.e. hanging messages), but has not been extended to multiparty session types, so is not appropriate for verifying all actor-based systems.

There is also existing work in the greater verification of real-world software. Much of this is done on sequentially executed programs as concurrency introduces a new level of complexity. For example, both C programs and Go programs have previously been targetted as good options for model checking [44, 43]. While these tools provide system guarantees, they primarily focus on detecting deadlocks or dataraces within a system and do not support other safety or liveness properties.

2.3.1 Theorem Proving

This report discusses model checking as a solution to software verification. An important alternative to model checking is theorem proving, in particular with proof assistants.

Theorem proving takes a formal approach to program verification. In theorem proving, axioms are applied to a set of statements to determine if a particular statement holds. For example, Z3 [20] is a satisfiability modulo theories (SMT) solver developed by Microsoft that can verify propositional logic assertions.

In both model-based and proof-based verification approaches, we begin with a formula ϕ. For a set of formulas Γ, a proof-based approach consists of finding a proof that $\Gamma \vdash \phi$. Provability holds if and only if semantic entailment ($\Gamma \models \phi$) holds. That means, to show provability, we must show that ϕ holds for all models. In a model-based approach, we instead want to show that ϕ holds for a single model. Hence, model-based approaches can be simpler than proof-based approaches [66].

That being said, some existing verification-aware languages take proof-based approaches, for example, Dafny, which will be discussed later in the chapter.

2.3.2 Design by Contract

Design by Contract (DbC) is an approach to software design, where software components are defined using formal specifications [64]. A contract is composed of three elements.

- **Pre-condition**: A condition that must be true before a function is executed.
- **Post-condition**: A condition that must be true after a function is executed.
- **Invariant**: A condition that must be true before and after a function is executed.

These properties are known as assertions. Some programming languages directly support assertions, such as Eiffel [65]. When formalising a contract, an agreement is established between two parties, the client and the contractor. The client is entitled to receive a result that satisfies the post-condition. Similarly, the contractor is protected by the pre-condition, such that, no unintended behaviour occurs.

Design by contract is a contribution to software reliability. In modern languages, the client and contractor are often functions or methods, which call one another. Verification-aware languages such as Dafny [24] support the specification of contracts, on methods, which can be verified.

2.3.3 Verification-aware Languages

Verification-aware languages are a new trend in programming languages, where the language is designed to support proving the correctness of a program. Examples of these include Lean, Dafny and Boogie. We will explore some of these languages in detail to understand how verification-aware languages can be a powerful tool to reason about the correctness of a system. The conventional

alternatives involve either disregarding formal methods entirely or hand translating a program into a specification language, such as TLA+. A good verification-aware language should naturally integrate the system specification with the implementation. The aim is to reduce the burden on programmers to maintain separate specifications alongside evolving codebases.

Lean

The Lean theorem prover is a proof assistant developed by Leonardo de Moura [22]. Lean is first and foremost, a functional programming language, designed to write correct and maintainable code. Lean can be used as an interactive theorem prover, where developers can write proofs alongside code. It supports many features of modern-day functional languages, such as first-class functions, pattern matching and even multithreading. A proof assistant is a language that allows developers to define objects and specifications over them. They can be used to verify the correctness of programs (similar to a model checker) as they check proofs are correct using logical foundations. The theorem proofs typically involve solving constraint problems, by determining if a first-order formula can be satisfied concerning constraints generated during analysis of functions.

While Lean is itself both a functional programming language and theorem prover, this approach differs in implementation from other theorem provers, such as Dafny, which instead prove theorems using existing backend theorem provers.

Dafny

Dafny is a verification-aware programming language that has native support for inlining specifications that can be verified by a theorem prover [24]. Dafny aims to modernise the approach developers take to designing systems, by encouraging developers to write correct specifications. With the rise of modern theorem provers, this untraditional approach is now realistic. Dafny is an imperative language with methods, variables, loops and many other features of typical imperative programming languages. Dafny programs are equipped with supporting tools to translate to other imperative languages, such as Java and Python.

Dafny verifies the correctness of programs using the theorem prover, Z3 [20]. Developers can write specifications alongside code, such as methods, which can then be directly verified. The format of specifications typically follows those of a Hoare Triple, $\{P\}C\{Q\}$, such that given a precondition, $\{P\}$ holds, if C terminates, a post-condition, $\{Q\}$, will hold. In Dafny, the language reserves the keywords `requires` and `ensures` for pre and post-conditions. Listing 2.1 shows a basic example of a Dafny method, which introduces an `Add` method. The implementation unintentionally introduces a bug such that, any execution paths with an input $\{a \in \mathbb{Z} \mid a < 0\}$ do not necessarily return the sum of the two inputs. Because Dafny places the burden on writing good specifications as opposed to correct code, the underlying theorem prover can use our post-condition to flag that this program is not correct for all execution paths.

```
1  method Add(a: int, b: int) returns (c: int)
2      ensures c == a + b;
3  {
4      if  a < 0 {
5          c := -1;
6      } else {
7          c := a + b;
8      }
9  }
```

Listing 2.1: Example of a method in Dafny.

Listing 2.1 only gives a small insight into the power the Dafny specification language defines. Alongside the evaluation of basic expressions, Dafny allows the use of quantifiers such as the universal quantifier. The introduction of quantifiers, allows us to write pre- and post-conditions over collections of objects, such as sets and arrays. Listing 2.2 shows a basic example of how the

universal quantifier can be used with the underlying theorem prover, to assert all the elements of an array, `a[]`, are strictly positive.

```
forall k: int :: 0 <= k < a.Length ==> 0 < a[k]
```

Listing 2.2: `forall` quantifier in Dafny [25].

Dafny also uses other concepts that support the verification of programs. Assertions can be used to provide guarantees in the middle of a method. Loop invariants can annotate while loops to check a condition holds, upon entering a loop and after every execution of the loop body. Similarly, loop variants can be used to determine termination of while loops, by checking that every execution of a loop body makes progress towards the bound of the loop.

Boogie

Boogie is a modelling language intended as an intermediate verification language (IVL), developed at Microsoft [26]. The language is described as an intermediate language because it is designed to bridge the gap between a program and a program verifier. Many tools that rely on Boogie's intermediate representation, are doing so to translate source code in a native language into a format that can be proved. Dafny is a prime example of a programming language which does so. The Dafny compiler generates Boogie programs that can then be verified by Z3. This provides multiple benefits for Dafny. Firstly, Dafny does not have to concern itself with being dependent on a specific SMT solver, such as Z3. Instead, it can be designed agnostic to the choice of theorem prover, as Boogie will take responsibility for handling interaction with theorem provers. Boogie also bears a closer resemblance to an imperative programming language (like Dafny), so translation between the two is easier than translating to Z3. Listing 2.3 shows an example Boogie program, defining a single procedure, `add`, that represents the translated code from the Dafny example in listing 2.1. Note the similarities between both programming languages, both use `ensures` to capture preconditions and have very similar syntax and control flow. However, now that our program is written in the Boogie IVL, we can directly determine an execution path that violates the precondition using a theorem prover such as Z3.

```
1  procedure add(a: int, b: int) returns (c: int)
2      ensures c == a + b;
3  {
4      if (a < 0)
5      {
6          c := -1;
7      } else {
8          c := a + b;
9      }
10 }
```

Listing 2.3: An example Boogie IVL program.

2.3.4 Gomela

Alongside research into verification-aware languages, there has been work on detecting deadlocks in concurrent systems. These approaches typically involve the use of model checkers to determine if a system can reach a deadlock state. For example, Java Pathfinder [53] is a model checker for Java programs. It can be used to detect deadlocks and data races. The initial version of Java Pathfinder was a translator from Java to Promela. At present, Java Pathfinder now uses a Java Virtual Machine (JVM) implementation directly to model check Java programs.

There has also been work on detecting deadlocks in Go programs. Gomela [44] was proven to catch more deadlocks than GCatch [61] and Godel2 [62]. Gomela focuses on channelled communication between goroutines. Similarly, deadlock detection has been researched for C programs [43].

We have identified Gomela as the most closely related work to that of our objectives. Gomela is not truly verification-aware, as it does not support the inlining of specifications, in the manner that Dafny or Boogie do. However, Gomela translates programs to Promela models, which can be verified by Spin. This report takes a similar approach, but instead targets Elixir programs, and also extends Elixir to be truly verification-aware.

To understand some of the challenges of modelling Elixir programs, which Gomela avoided, we can draw a few comparisons between the two:

- Go communication channels can be bounded, whereas Elixir mailboxes are unbounded. Naturally, unbounded queues can lead to more complex verification problems (state-space explosion).
- Go is statically typed, whereas Elixir is dynamically typed.
- Go has support for shared memory, Elixir's actor model strictly enforces all information sharing to be handled by message-passing.
- Elixir process state is immutable, whereas Go has mutable state.

These comparisons lead to challenges that need to be addressed in this report. However, the above points are also reasons why Elixir is an interesting target language, both for designing real-world systems and for verifying.

None of the existing approaches capture the semantics involved in a pure message-based, actor model. They also do not provide guarantees about the liveness properties of a system. This is a limitation in existing research of verification tools concerning modern programming languages.

2.4 Summary

This chapter has provided an overview of core concepts related to concurrent programs and verification of them. We looked at tools such as model checkers and verification-aware programming languages. In particular, we saw the model checker Spin. This chapter also discussed some of the limitations in existing research, such as a need for new techniques to verify the liveness properties of real-world systems. The next chapter will introduce the Elixir programming language and the Promela modelling language.

Chapter 3

Promela and Elixir

In chapter 4, we will introduce the Verlixir tool. Verlixir involves the parsing of Elixir programs, which are translated into a formal model. This model is written in Process Meta Language (Promela). This chapter will introduce both Promela and Elixir. We will go through the core concepts and syntactic elements that Verlixir relies upon to provide a verification-aware Elixir.

3.1 Promela

Promela is the verification modelling language used by the Spin model checker, to specify concurrent processes modelling distributed systems [9]. This section will discuss some of the core features that allow systems to be modelled and verified with Spin. This section aims to give an overview of the syntax and control of Promela, so any specifications in later sections or the code artifact can be read.

3.1.1 Types and Variables

Promela is statically typed. Variables can be declared once within the current scope and then re-assigned throughout. Variables can be declared locally within the context of a process, or in the global scope, where memory is shared. The types available in Promela, and assignment to variables of these types are similar to many imperative programming languages. Promela supports the types bit, bool, byte, pid, short, int and unsigned. Variable declaration and assignment then naturally follows.

int a = 2;

Promela supports **arrays**. Arrays are typed and declared with a fixed size. Array bounds are constant, so the size cannot change. Only single-dimensional arrays are supported. The syntax for declaring an array is as follows.

int array[10];

We can also extend the basic types using `typedef`. This allows us to define records of multiple nested types. We use these records to build a Promela library, to support model checking Elixir programs, using **embedded C** code and Promela **inlines**. Embedded C code cannot be model checked, but Promela inlines can be. These features allow us to extend Promela, and avoid some of the limitations discussed in section 3.1.6

3.1.2 Control Flow

Promela supports some basic control flow concepts. Firstly, the `skip` expression can be used with no effect when executed, other than possibly changing the control of an executing process. The selection construct `if` can be used to evaluate expressions and execute sequences based on the evaluation of these expressions. The syntax of an if statement is unique in comparison to a typical programming language.

```
1 if
2     :: 1 + 1 < 3 ->
```

```
        printf("Condition 1...");
    :: else ->
        printf("No conditions matched");
fi
```

In Promela, *else* is a reserved keyword that can be used in any condition. An *else* condition will negate all the previous conditions. Repetition can be achieved either through the `do` construct, through the use of labels or with `for` loops. We will primarily focus on `do`, as it is the most suitable for our modelling needs.

```
do
    :: a < 10 ->
        a = a + 1;
    :: a < 10 ->
        a = a + 2;
    :: else ->
        break;
od
```

Unlike `if`, which selects sequentially, `do` will non-deterministically select a true branch to execute. This means for the above example, for a given execution, we cannot say how many iterations are performed. The `break` keyword is reserved for explicitly breaking out of the loop.

An important concept in contract specification is the **assert** keyword. An assertion is a logical statement that is expected to be true at a given point in the program. If an assertion is false, a violation is reported.

3.1.3 Processes

An imperative component of understanding the power of the Spin model checker is understanding how processes can run concurrently. Every Promela model requires an initial process that is spawned in the initial system state and determines the control of the program from the initial state. The `init` keyword is reserved for this purpose. Other processes can be defined using the `proctype` keyword and then spawned with `run`. Each process is assigned a process id (pid) which can be accessed within the context of a process using globally defined read-only variable `_pid`. We can now define two processes, a process active in the initial state and a second process that is spawned.

```
    proctype SomeProcess(int a) {
        printf("Do something with %d\n", a);
    }

    init {
        int p1;
        p1 = run SomeProcess(10);

        printf("Init process spawned at %d\n", _pid);
        printf("Process 1 spawned at %d\n", p1);
    }
```

Listing 3.1: Defining and spawning processes in Promela.

Processes run independently of one another, so a parent process terminating will not necessarily result in the termination of a child. Spin sets a limit of 255 concurrently executing processes. Multiple processes can be spawned in a single transition by using the **atomic** construct, which will ensure that no spawning process is scheduled, until all atomic processes have been scheduled. Similarly to atomicity, `d_step` can be used to enforce multiple statements are treated as a single indivisible step. Unlike `atomic`, `d_step` cannot block or jump.

Instead of `init`, we could have used an **active proctype**. Every **active proctype** is spawned in

the initial state, allowing for more than one process to initially run.

3.1.4 Channels

The final concept to briefly discuss, is the asynchronous communication primitive, channels. Promela allows channels to be specified using the predefined data type `chan`. To correctly specify communication, we often need to allow messages of multiple types to be written to channels. For this reason, Promela introduces `mtype` that allows for the introduction of symbolic names for constant values.

$$\text{mtype} = \{ \text{BROADCAST} \};$$

Now, we can define a channel that expects a message to contain multiple fields and is bound to contain a maximum of 10 messages at any time.

$$\text{chan global\_broadcast} = [10] \text{ of } \{ \text{mtype, int} \};$$

We now input messages to the channel using the (!) operator.

$$\text{global\_broadcast ! BROADCAST, 1;}$$

Similarly, we read messages from the channel in a first-in, first-out (FIFO) order.

$$\begin{aligned} &\text{int x;} \\ &\text{global\_broadcast ? BROADCAST, x;} \end{aligned}$$

Where the variable x stores the resulting `int`, assuming the first message in the channel is of type `BROADCAST`. Sending and receiving from channels also supports an alternative flavour. The (!!) and (??) operators are used for sorted insertion and random selection. **Sorted insertion** (!!), will insert a message into the channel in a sorted order, based on the first field of the message. **Random receive** (??), is not random. As opposed to FIFO, it will select the first message in the channel that matches a given pattern. We call this first-in, first-fireable-out (FIFFO).

3.1.5 Promela Example

We will now provide a simple example of a Promela specification. The specification models Dijkstra's Semaphore. It consists of two labelled processes, and an initial process to coordinate the system. A shared channel is used, with a buffer size of 0, which means the channel is blocking (rendezvous). Listing 3.2 shows the Promela specification.

```
1  #define p  0
2  #define v  1
3
4  chan sema = [0] of { bit };
5
6  proctype dijkstra() {
7      byte count = 1;
8
9      do
10     :: (count == 1) ->
11         sema!p; count = 0
12     :: (count == 0) ->
13         sema?v; count = 1
14     od
15 }
16
17 proctype user() {
18     do
19     :: sema?p;
20         /* critical section */
21     sema!v;
22         /* non-critical section */
23     od
```

```
24 }
25
26 init {
27     run dijkstra();
28     run user();
29     run user();
30     run user()
31 }
```

Listing 3.2: Dijkstra's Semaphore in Promela

The semaphore guarantees that only one of the user processes can enter its critical section at a time.

3.1.6 Limitations

Promela is a powerful language for modelling concurrent systems, but it has limitations for capturing the full complexity of a real-world system. In general, hand-translations of a system into Promela can avoid some of these limitations with careful design. However, we are approaching these limits from an Elixir perspective, where features of Elixir may not be easily translated into Promela.

- **Compute**: Promela is not designed to model complex computations. It does not support floating-point arithmetic, so we are limited to working with integers.
- **Memory**: Promela does not support dynamic memory allocation. This means we cannot model systems that require dynamic memory allocation, for example, linked lists.
- **Functions**: Promela does not support functions. It has no notion of a function call or return. By extension, Promela does not support recursion.
- **Randomness**: a Spin execution may not be deterministic, but it cannot model true randomness.
- **Probability**: there is no mechanism for modelling probabilistic behaviour, all correctness claims are checked unconditionally.
- **Time**: there is no notion of a system block, or related time properties in Promela. This means we cannot model sleeping threads.

In chapter 5, we will discuss how we can overcome some of these limitations. In particular, how we overcome the lack of functions and dynamic memory, by introducing a Promela library, which handles these features.

3.1.7 Summary

This basic introduction to the syntax of the Promela modelling language, aims to make the reader familiar with the syntax involved in writing Promela specifications. It is not an exhaustive guide but should form a basis for understanding specifications present in a later section or the code artifact.

3.2 Elixir

Elixir [11] is a functional programming language built on top of Erlang [13] that runs on the BEAM virtual machine [12]. It is commonly used for building distributed, fault-tolerant applications because it supports concurrency, communication and distribution. Elixir actors are uniquely identified with a process identifier (pid) and associated with an unbounded mailbox. Each mailbox supports communication between actors; one actor can send a message to another actor's mailbox, which is then enqueued and can be received in a First-In-First-Firable-Out (FIFFO) ordering. FIFFO is similar to First-In-First-Out (FIFO) where elements are dequeued in the order they are enqueued. However, Elixir supports receiving messages with pattern-matching such that messages

are received in a FIFO order concerning a certain pattern.

BEAM is a virtual machine that executes user programs in the Erlang Runtime System (ERTS). BEAM is a register machine where all instructions operate on named registers containing Erlang terms such as integers or tuples.

We have recently seen companies adopting Elixir in industry, in particular in domains such as telecoms and instant messaging. The Phoenix Framework [14] is a framework for building interactive web applications natively in Elixir, that can take advantage of Elixir's multi-processing and fault tolerance to build scalable web applications. The audio and video communication application Discord [15] uses Elixir to manage its 11 million concurrent users and the Financial Times [16] have begun migrating from Java to Elixir to enjoy the much smaller memory usage by comparison.

Elixir supports multi-processing in two key ways: nodes and processes. Each **Elixir node** is an instance of BEAM (a single operating system process). When an Elixir program is executed, a new instance of BEAM is instantiated for it to run on. In contrast, an **Elixir process** is lightweight in terms of memory and CPU usage (even in comparison to threads that many other programming languages favour). Elixir processes can run concurrently with one another and are completely isolated from one another. Elixir processes communicate via message passing.

```
1  # Spawn a new process
2  spawn(fn -> 1 + 2 end)
3
4  # Create a new BEAM instance
5  Node.spawn(:"node1@localhost", MyModule, :start, [])
```

Listing 3.3: An example of spawn/1 and spawn/4 in Elixir, for spawning a new lightweight process and a new Elixir node

In Elixir, a receive statement is used to read messages in the mailbox. The receive block looks through the mailbox for a message that matches a given pattern. If no messages match the pattern, the process will block until one does.

```
1  # Example send in Elixir
2  send self(), {:hello, "world"}
3
4  # Example receive block in Elixir
5  receive do
6      {:hello, msg} -> IO.puts msg
7  end
```

Listing 3.4: Example of sending and receiving an Elixir message

3.2.1 Verifiable Feature Set

In chapter 4, we will introduce Verlixir. Verlixir is designed to support a reduced set of core Elixir constructs. We will introduce these constructs here to give an overview of what the tool is capable of supporting.

Everything in Elixir is an expression. This means that every piece of code returns a value. For example, an *if* statement will return a value dependent on the branch taken. This means that any expression can be matched on, using Elixir's match (=) operator. We can use pattern matching to match the shape of an expression's evaluation. The set of expressions supported by Verlixir are:

- **Values**: any value of a basic, primitive type such as integers and booleans. Elixir also has a concept of *atoms*. An atom is identified by a preceding colon (:), and is followed by letters, digits, '_', '@' or a string.

- **Variables**: Elixir is dynamically, strongly typed. Variables are bound to using the match operator. Variable names start with lowercase letters. The '_' character can be used to match an expression of any shape.
- **Data structures**: structures such as lists and tuples are treated as values. Lists are dynamic, whereas tuples are fixed in size. Lists are written as [1, 2, 3] and tuples are written as {1, 2, 3}. We can match the shape of these data structures using pattern matching.
- **Pattern matching**: pattern matching can be used to match the shape of an expression. In the context of a conditional guard, values can be used to evaluate the shape of an expression, whereas variables can be used to bind to the value of an expression. For example, the pattern $\{:ok, value\}$, can be used to assign to the variable *value* if the expression is a tuple shape, with an atom *ok* as the first element.
- **Functions**: functions are defined using the *def* keyword. Functions can group multiple sequentially executable expressions. Any function can be called or spawned as a new process or node.
- **Modules**: functions are grouped into modules.
- **Message passing**: Elixir's actor model supports message passing. Messages are sent and received between actors and their mailboxes.
- **Control flow structures**: Elixir supports many control flow expressions. For example, *if*, *case*, *unless* and *for*. All of these introduce a new scope. They are expressions and can be matched.

The feature set supported has been shown expressive enough to support real-world systems in chapter 6. We omit some features of Elixir. For example, data structures like sets and maps are not supported. We determined lists sufficient for our purposes. The supported feature set can be easily extended to support other data structures.

3.2.2 Matching

Elixir is dynamically typed. This means that the type of a variable is determined at runtime. Elixir relies on **patterns** to assign identifiers to values. In Elixir, everything is an expression. By extension, everything can be matched, with the **match operator** (=).

Combining patterns with the match operator, allows different data structures to be packed and unpacked in expressive ways. For example, we can match the shape of a list, to extract the first element and the rest of the list.

```
1    [head | tail] = [1, 2, 3]
2    IO.puts head # 1
3    IO.puts tail # [2, 3]
```

This notion extends to branching constructs, such as *if* and *receive* statements. Consider the following example.

```
1  x = receive do
2      {:ok, value} ->
3          if value > 10 do
4              value
5          else
6              {:bad_value, value}
7          end
8      {:error} ->
9          [1,2,3]
10 end
```

Notice firstly, that the entire *receive* expression is assigned to x. Also notice, that each branch yields differently typed expressions. The expressiveness of Elixir's matching makes Elixir a powerful language for manipulating data flow. However, it presents challenges for modelling.

We cannot determine the type of an expression at compile time. We also have to consider how the value of x depends on the executed branch.

3.2.3 Type Specifications

Type specifications are imperative for the correctness of Verlixir specifications. Verlixir supports some basic types such as `integer()`, `boolean()`, `atom()` and `pid()`. Type specifications are not enforced by the Elixir compiler, but tools such as Dialyzer and Verlixir rely on them.

In type specifications, message types are typed as `atom()`. The atom `:ok` is reserved to identify a **non-returning function**. In Elixir, all functions return a value, so in this context, a 'non-returning function', is a function that's value is never matched. We briefly demonstrate the type specifications for two functions, the first, is a non-returning function with no arguments and the second function, takes two arguments and returns an integer.

```
1  @spec bind_server() ::  :ok
2  def bind_server do
3      ...
4  end
5
6  @spec add(integer(), integer()) ::  integer()
7  def add a, b do
8      ...
9  end
```

Listing 3.5: Valid type specification examples.

Notice (::) marks the return type of the function. If these values are matched in the function body, they should not be matched to a different type.

Within a correct Verlixir specification, any message should also be typed. To ensure this, any instance of a message should begin with an atom which we will refer to as the **message type**. For example, `{:bind}` and `{:calculate, 10, 20}` are valid specification messages. The message, `{false, 15}`, would be ignored by Verlixir, as it does not begin with an atom.

3.2.4 Summary

In this chapter, we learned about Elixir, the programming language built on top of Erlang and we explored some basic approaches to designing concurrent systems with it. We also saw Promela and how concurrently executing processes can be modelled in it. The next section will explore how these core tools can be used in tandem, to provide developers guarantees over large-scale, distributed Elixir-based systems.

Chapter 4

Verlixir

Verlixir is the main project contribution. Verlixir is designed such that, Elixir programmers get deadlock safety guarantees for free. To strengthen the system guarantees, programmers can write inline specifications and function contracts.

Verlixir supports three modes of operation: simulation, verification and parameterized verification. Simulation mode is used to run a single execution of the system. Verification mode is used to verify the system adheres to the provided specification. Parameterized verification is used to verify the system over multiple configurations.

This chapter aims to inform the reader of the constructs defined in Verlixir. Section 4.1 introduces the Verlixir language. Section 4.2 provides an example of specifying a verifiable system and how Verlixir can be used to detect violations of a specification. The subsequent subsections provide further details of more features of Verlixir, such as specifying temporal properties.

4.1 LTLixir

LTLixir is the multi-purpose specification language that compiles to BEAM byte-code and is supported for verification by Verlixir. Primarily, LTLixir is a subset of Elixir, supporting both sequential and concurrent execution. This subset is expressive enough to implement well-known distributed algorithms such as basic Paxos [54] and the alternating-bit protocol [55]. LTLixir extends Elixir with constructs for specifying temporal properties, specifically LTL properties (where LTLixir derives its name) as well as function contracts for specifying pre- and post-conditions. Specifications can be parameterized to identify violations of properties on specific configurations.

4.2 Constructing a Verifiable Elixir Program

This section will walk through the basic construction of a Verlixir program, and show how we can verify the properties of the program using Verlixir. To begin, we define a server and client process. The server is responsible for creating clients and communicating with them.

```
1  defmodule Server do
2      def start_server do
3          client = spawn(Client, :start_client, [])
4      end
5  end
6
7  defmodule Client do
8      def start_client do
```

```
9               IO.puts "Client booted"
10          end
11      end
```

Listing 4.1: Elixir definition for a server and client module.

Given the implementation, we must now declare an entry point to the system, that Verlixir will use to begin verification. For this example, we can define `Server.server_start` as the entry point using **init**.

```
1       @init
2       def start_server do
3           client = spawn(Client, :start_client, [])
4       end
```

Listing 4.2: Declaring an entry point to the system.

With an entry point specified, we can begin using the available tools. By default, Verlixir reports the presence of deadlocks and livelocks in the system. When specifying systems in Verlixir, we do not lose the capability to compile our program to BEAM byte-code, hence the system can still run as a regular Elixir program.

More interestingly, we can now use Verlixir before the Erlang Run-Time System (ERTS) to verify the system adheres to our specification. With no additional properties defined, by running Verlixir, we are ensuring that every possible execution results in a program termination. The presence of a deadlock or livelock will be reported. We first run a simulation of the system.

```
1       $ ./verlixir -s basic_example.ex
2       Client booted
```

Alternatively, we can run the verifier on the specification.

```
1       $ ./verlixir -v basic_example.ex
2       Model checking ran successfully. 0 error(s) found.
3       The verifier terminated with no errors.
```

4.2.1 Detecting a Deadlock

Now we have a basic understanding of what is required to write a specification, we will use Verlixir to detect a deadlock in the system. Deadlocks in Elixir programs can be introduced by circular waits, where two simultaneously executing processes, are both waiting for a message from the other.

```
1   defmodule Server do
2       @init
3       def start_server do
4           client = spawn(Client, :start_client, [])
5           receive do
6               {:im_alive} -> IO.puts "Client is alive"
7           end
8       end
9     end
10
11  defmodule Client do
12      def start_client do
13          receive do
14          {:binding} -> IO.puts "Client bound"
```

```
15          end
16      end
17  end
```

Listing 4.3: A simple Elixir system with a deadlock.

In this example, any execution of the system will result in a deadlock; the system can be considered deterministic in this regard. In many real-world systems with multiple processes, the presence of a deadlock can be difficult to detect due to multiple interleavings.

A simulation of the system using Verlixir will report a timeout (something which the ERTS would not report). Already, running our specification using Verlixir, provides more information than running the Elixir program. Let's now run the verifier.

```
$ ./verlixir -v basic_example.ex
Model checking ran successfully. 1 error(s) found.
The program likely reached a deadlock. Generating trace.
[8] (proc_0) start_server:4 [receive do]
[9] (proc_0) start_server:4 [receive do]
[10] (proc_0) start_server:5 [{:im_alive} -> IO.puts "Client is alive"]
[13] (proc_1) start_client:13 [{:binding} -> IO.puts "Client bound"]
<<< END OF TRAIL, FINAL STATES: >>>
[14] (proc_1) start_client:13 [{:binding} -> IO.puts "Client bound"]
[15] (proc_0) start_server:5 [{:im_alive} -> IO.puts "Client is alive"]
```

If an error is found, Verlixir will profile the type of error; in this case, it has determined the program likely deadlocked. Once determining the error type, an error trace is produced to debug the source of the error. The underlying model derived from the Verlixir specification does not have a one-to-one mapping to the original Elixir code, hence, heuristics are applied, to determine where in the Elixir program the trail is produced from.

Alongside the process name, we can see the line number in the Elixir file. The remaining information on a trail line is less relevant to most users. The first number on a line is the step number (some of these may be omitted for simplicity). The `proc_n` refers either to a process number or function call stack depth.

Alongside the error trace, Verlixir also reports a trace of all messages being sent and received through the system. If we take a look at the messages produced in this case, we see that no messages were ever sent. This could give a further indication as to why the deadlock has arisen.

We can read the trail in sequential order to learn the interleaving that resulted in the error. In this instance, we can see the server reaches line 5 where it waits for an `:im_alive` message from the client and similarly, the client is waiting for a `:binding` message.

4.2.2 Linear Temporal Logic

We now introduce Linear Temporal Logic (LTL) to our systems to allow us to write more interesting Verlixir specifications.

Let's re-design the server and client processes, so we can introduce temporal properties to reason about. The server will now spawn n clients, bind the clients to itself and then await a response from all three clients.

```
def start_server do
    client_n = 3
    alive_clients = 0
    for _ <- 1..client_n do
        client = spawn(Client, :start_client, [])
        send(client, {:bind, self()})
```

```
        end
        alive_clients = check_clients(client_n, alive_clients)
    end
```

The implementation of the client process and the `check_clients/2` function have been omitted. Without understanding their implementation, we can still use our specification to verify the system acts as intended. We introduce our first LTL formula, which verifies that eventually, the number of alive clients is equal to n. To introduce an LTL formula, we can use **@ltl**. An LTL formula is assigned, as a string, to a function. The LTL grammar is defined as the following.

$$\begin{aligned}
\langle \text{ltl} \rangle &\models \langle \text{operand} \rangle \mid (\langle \text{ltl} \rangle) \mid \langle \text{ltl} \rangle \langle \text{binop} \rangle \langle \text{ltl} \rangle \mid \langle \text{unop} \rangle \langle \text{ltl} \rangle \\
\langle \text{operand} \rangle &\models \mathit{true} \mid \mathit{false} \mid \mathit{var} \mid \mathit{int} \mid \mathit{elixir_expr} \\
\langle \text{unop} \rangle &\models \Box \mid \Diamond \mid ! \\
\langle \text{binop} \rangle &\models U \mid W \mid V \mid \&\& \mid || \mid \rightarrow \mid \leftrightarrow
\end{aligned}$$

We want to verify that eventually, the number of alive clients equals the number the server created. We can write this using the formula $\Diamond(\mathit{alive\_clients} \equiv \mathit{client\_n})$. Using the LTL attribute, we can update our server process.

```
    @init true
    @spec start_server() :: :ok
    @ltl "<>(alive_clients == client_n)"
    def start_server do
        ...
    end
```

Listing 4.4: Example LTL property

Let us run Verlixir on the system.

```
$ ./verlixir -v basic_example.ex
Model checking ran successfully. 0 error(s) found.
The verifier terminated with no errors.
```

We can update the LTL formula, by replacing `clients_n` with the number 1 ($\Diamond(\mathit{alive\_clients} \equiv 1)$). We run the verifier again.

```
$ ./verlixir -v basic_example.ex
Model checking ran successfully. 1 error(s) found.
The program is livelocked, or an LTL property was violated. Generating trace.
... trace omitted ...
```

To help with readability, we can define inline predicates to use in LTL formulae. The predicates that can be defined are formed from a subset of the LTL grammar, without the temporal modalities. Inline predicates can refer to variables in the scope of the function. For example, we can define a predicate $\mathit{all\_alive}$ as $\mathit{alive\_clients} \equiv \mathit{client\_n}$. Using this predicate, we can strengthen our LTL formula to $(!\mathit{all\_alive})\ \mathcal{U}\ (\Box \mathit{all\_alive})$. Informally, this formula states that there is a moment in time where $\mathit{alive\_clients} \equiv \mathit{client\_n}$, and from that moment onwards, this property holds until termination. We can update our `start_server` function to reflect this.

```
    @ltl "(!all_alive)U([]all_alive)"
    def start_server do
      client_n = 3
      alive_clients = 0
      predicate all_alive, alive_clients == client_n
      ...
```

```
7       end
```

4.2.3 Contracts

Verlixir also supports Design by Contracts, using pre- and post-conditions in function definitions. These are particularly useful for ensuring proper bounds on the system, that help define correct execution. Consider a process that should send and receive a single message each round. We can define a pre-condition on a bounded parameter to ensure the process does not run indefinitely. Let us refactor the client process, to complete a number of rounds (determined by the server) before terminating.

```
1       defmodule Client do
2           @spec start_client() :: :ok
3           def start_client do
4               receive do
5                   {:bind, sender, round_limit} ->
                        next_round(server, round_limit)
6               end
7           end
8
9           @spec next_round(pid(), integer()) :: :ok
10          def next_round(server, rounds) do
11              send(server, {:im_alive})
12              remaining_rounds = rounds - 1
13              next_round(server, remaining_rounds)
14          end
15      end
```

We now introduce the **defv** macro from the LTLixir specification language. The `defv` macro is used to create **contracts**, using pre- and post-conditions. Pre-conditions check conditions regarding the values of function arguments on entry to the function and similarly, post-conditions can assert conditions on values within the scope of the function on exit.

With this definition, we gain assurances that the implementation of the function behaves as we expect and that no other function interacts with it in a manner that violates our expected behaviour. We can create a contract for our client. We assert that, on each function entry, the number of rounds is positive, and on exit, the remaining number of rounds has decreased.

```
1       defv next_round(server, rounds), pre: rounds >= 0,
            post: remaining_rounds < rounds do
2           ...
3       end
```

We can run Verlixir on the system to verify the pre-condition holds for every possible execution.

```
1       $ ./verlixir -v basic_example.ex
2       Model checking ran successfully. 1 error(s) found.
3       An LTL, pre- or post-condition was violated. Generating trace.
4       Violated: assertion violated (rounds>=0) (at depth 45).
5       ... trace omitted ...
```

Verlixir reports an error, in particular, it notes the violation of an assertion. An assertion violation can be a violation of an LTL formula, and pre-condition or a post-condition. In this case, it outputs the assertion that was violated `rounds >= 0`, which we are aware is our pre-condition.

4.2.4 Parameterized Systems

Up to this point, we have declared various system properties such as `client_n`, `alive_client` and `rounds`. In reality, the value assigned to these properties could be determined by many factors and it may not be known to the developer at the time of writing the specification. To support this, Verlixir allows us to declare these properties as parameters, in particular, we want to declare concurrency parameters. We define concurrency parameters as variables that impact the behaviour of a distributed system (we will simply refer to them as parameters going forward). For example, in a consensus algorithm such as Paxos, we may have variables to determine the number of acceptors, proposers and size of a quorum. We can declare these variables as parameters in the specification in order to verify the system for multiple possible configurations. Typically, the values used in these auto-generated configurations will be small values, as large values may lead to a state space that becomes difficult to explore.

We use `@model` to mark variables as parameters. It takes a tuple of atoms referencing variables in the function scope. We can apply this definition to our existing server process.

```
1   @model {:client_n, :number_of_rounds}
2   def start_server do
3     client_n = 3
4     number_of_rounds = 2
5     predicate all_alive, alive_clients == client_n *
          number_of_rounds
6     ...
7   end
```

Listing 4.5: Example of declaring concurrency parameters in specification.

We can declare as many parameters as required. The values matched in the declaration of the variables will be ignored if we run Verlixir in parameterized mode. To run the verifier, we can use the $-p$ flag. To run the parameterized verification, we use the same command but with the $-p$ flag.

```
1   $ ./verlixir -p 3 basic_example.ex
2   Generating models.
3   Generated 9 models.
```

We can introduce a bug into our program that will cause a violation of the specification, to show the output of the verifier under these circumstances. To introduce a bug, we are going to conditionally call `check_clients/2` if `client_n > 1` holds.

```
1   @model {:number_of_rounds}
2   def start_server do
3       ...
4       alive_clients = if number_of_rounds > 1 do
5           check_clients(client_n * number_of_rounds, alive_clients)
6       else
7           0
8       end
9   end
```

This will now result in cases where the temporal property is violated, as `alive_clients` $\equiv$ `client_n` $*$ `number_of_rounds` will not hold for all configurations. Note, that we have reduced the parameters down to just `{number_of_rounds}`. The system is theoretically capable of handling any number of parameters in its search. However, the computational cost of exploring the state space grows exponentially with the number of parameters.

```
1   $ ./verlixir -p 3 basic_example.ex
```

```
2 Generating models.
3 Generated 3 models.
4 Violations found in models:
5 Model with params: {"number_of_rounds": '1'}
6 Assign these parameters to the system and re-run the verifier in verification
      mode to gather a trace.
```

The system found a violation for the assignment of 1 to `number_of_rounds`. To investigate, we could run the verifier on this configuration. It's also useful to note that Verlixir was executed with $-p3$, this sets the range of values, from 0 to (p-1), for each parameter.

4.3 Summary

We have now given a high-level overview of Verlixir, the verification toolchain capable of verifying Elixir programs written using the LTLixir specification language. We saw how to use Verlixir to simulate executions of the system; verify our system's adherence to a specification and parameterize concurrency parameters for exploration. This chapter also explained how to use LTLixir constructs, to reason about temporal properties, as well as how pre- and post-conditions can drive functional correctness. The next chapter will begin to explore the implementation of Verlixir.

Chapter 5

Verification of Message-Based Systems

This chapter provides an in-depth discussion into the design decisions that were made during the development of Verlixir. The chapter will begin by providing a high-level overview of where the relevant components fit into the toolchain, as well as providing an architectural overview of the tool. Section 5.2 will describe the main techniques Verlixir applies in the analysis and modelling of a specification and section 5.3 will discuss the design of the LTLixir specifiation language. Finally, section 5.4 will describe how the outputs generated by Verlixir are derived.

5.1 Verlixir Toolchain

We will first introduce where the new tools fit into the greater toolchain. Figure 5.1 shows a high-level overview of the toolchain. Given an LTLixir specification, either, the Elixir program can be compiled and run on the ERTS, or the program can be modelled by Verlixir and verified with Spin.

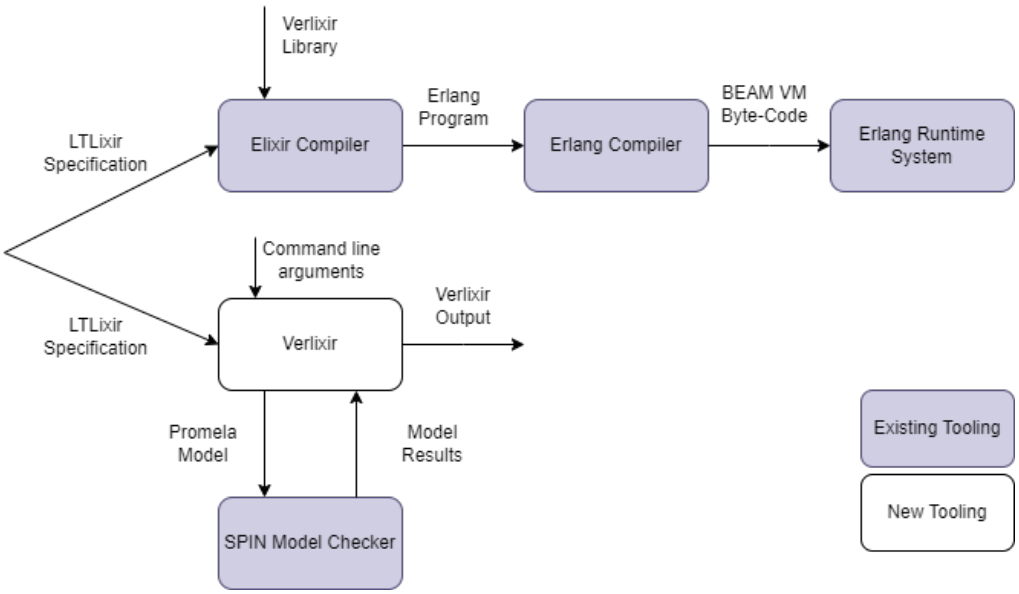

Figure 5.1: High-level overview of the verifiable Elixir toolchain.

Figure 5.2 provides an in-depth insight into the architecture underlying Verlixir.

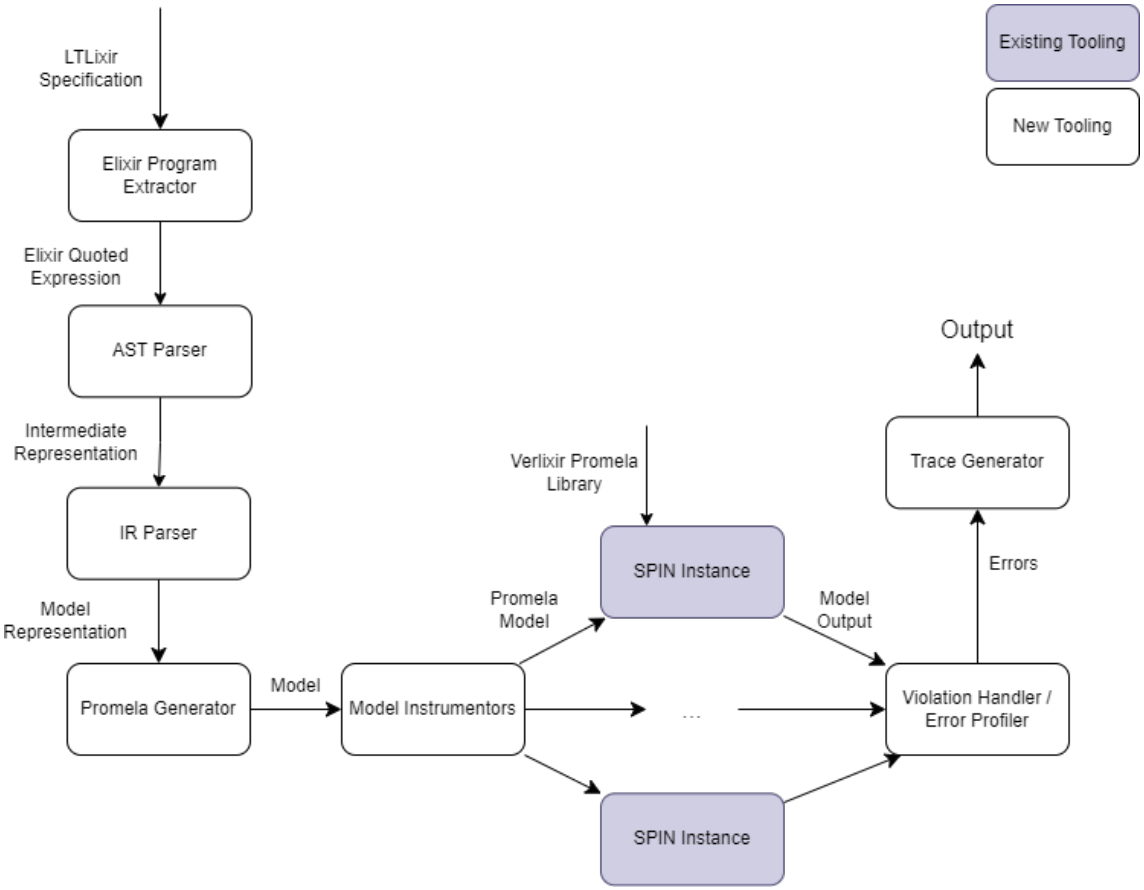

Figure 5.2: Verlixir design.

Before we discuss the design of Verlixir, we will summarise the components of the tool.

- **LTLixir Specification**: Specification of an Elixir program to be parsed by Verlixir.
- **Elixir Extractor**: Converts the Elixir program into a quoted expression.
- **AST Parser**: Parses the quoted expression into an intermediate representation.
- **IR Parser**: Parses the intermediate representation, collecting relevant information for the model generator.
- **Promela Generator**: Writes a Promela model from the intermediate representation.
- **Model Instrumentors**: Instruments the model for appropriate verification (depending on system and user requirements).
- **Violation Handler**: Handles the output of the model checker by mapping violations back to the original Elixir program.
- **Trace Generator**: Generates an error trace from the mapped Elixir violations.

We briefly mention the technologies used in the design of the Verlixir artifact. Alongside the core Verlixir system, we introduce an Elixir library and a Promela library.

The Elixir library is imported into programs that wish to access the LTLixir specification constructs we have introduced. In particular, contracts and predicates. The library is written in Elixir, using metaprogramming. With the library, comes the Elixir program extractor. The extractor, is an Elixir module, that converts Elixir syntax into a parsable quoted expression.

The Promela library is a hybrid between Promela and C. It is required to extend Promela beyond its basic capabilities; aligning it with the Elixir program semantics. The model generator

uses the Promela library to model some Elixir constructs, for example, functions, data structures, message passing and more. Promela compiles models to C, so a C library incurs no overhead. However, C code can not be model-checked.

The core Verlixir offering (intermediate representation and Promela generator) is written in Rust. Rust is highly performant and provides software reliability guarantees that are relevant in asserting the correctness of the tool. Rust is highly optimised for parallelism, which we rely upon for optimising the model-checking process.

5.2 Modelling Elixir Programs

The primary work done by Verlixir is determining how to internally represent an Elixir program. Given an Elixir program, with an inlined specification following the LTLixir semantics, Verlixir must both model the system and the properties of the specification. This section will outline the techniques used to achieve this.

5.2.1 High-level Overview

The internals of how the system is used to produce models of Elixir programs can be categorised into three umbrellas:

- **Parsing**: takes an Elixir program and generates a quoted expression. Then, lexical analysis is performed on the quoted expression.
- **Intermediate Representation**: takes the parsed expression and generates an intermediate representation by extracting features relevant to model the program and specification.
- **Writing**: takes the intermediate representation and generates a model in a target language.

The writer currently only supports the generation of Promela models, which can be verified using the model checker Spin, see section 5.4. Although this component of Verlixir can be split into these three stages, as they all work to achieve the same goal we will treat them as one and consider this component the `model generator`.

To help understand the model generator, we begin by providing a high-level mapping of the supported set of Elixir constructs to Promela in table 5.1.

Elixir Expression	**Promela Expression**	**Definition**
Functions	Processes	Every function call (recursive or else) spawns a new process. Function calls always block the parent process. If a function returns a value, the caller will await a rendezvous with the callee.
Process spawning	Processes	Process spawning translates naturally to Promela, using the *run* keyword.
Boolean / arithmetic expressions	Expressions	The translation, for the supported set of operators, is direct to Promela for basic data types. Data structures, like lists, have custom inline code fragments for supported operators.
Matching (=)	Declaration, assignment	The first match in scope is translated as a declaration. Subsequent matches are re-assignments. In the case where the left-hand expression of a match is non-basic (i.e. tuple), a stack may be used to determine declaration ordering.

Types (integer, boolean, atom)	int, bool, mtype	Basic types include integers and booleans. Atoms are treated as message types (mtypes) and are globally unique.
Lists	Dynamically-sized arrays	Promela arrays have been extended to support dynamically sized lists.
For comprehensions	For loops	A for comprehension over a range of values, translate to a Promela for loop. More complex comprehensions typically involve inlines.
If statements	If statements	If statements translate directly, with additional instrumentation as Promela blocks until a condition is matched.
Send	Channel append (!!)	A message will be packed into a new message structure, and then appended to the relevant mailbox channel.
Receive	Channel receive (??)	Receive involves matching the correct mailbox and message type. We can then remove the message from the channel and unpack the values.
LTL	LTL	Promela supports LTL. However, any variables or Elixir expressions present have to be specially translated before we translate the LTL formula.
Contracts	Assertions	Assertions are placed in process entry and exit points to ensure correctness.
Predicates	Global inlines	Predicates are recursively searched and moved to the global scope as inlines.

Table 5.1: Overview of the translation of Elixir Expressions to Promela.

The remainder of this section will discuss the techniques applied in the model generator, and specifically how the generator targets Promela as an output language.

5.2.2 Sequential Execution

First, we explore how to model sequential execution. Elixir relies on a few parent identifiers that generally describe the structure of a program. We discuss a few flavours of these:

- **Blocks**: blocks are a simple but core concept. An Elixir block contains multiple Elixir expressions separated by newlines or semi-colons.
- **Do**: Elixir control structures such as *if* and *receive* all use the *do* keyword for a new expression. The child expression could be a single expression or a nested block.
- **Functions**: functions are essentially named or anonymous blocks, that can be spawned as processes.
- **Modules**: multiple functions can be grouped in a module. All Elixir code runs inside processes, so typically grouping functions in a module is a way to group functions that are related to the task a process performs.

These three constructs are examples of the primary building blocks of an Elixir program. With each, a new level of scope is introduced. Declared variables from parent scopes are accessible in child scopes, but any match to re-assign a variable from the parent scope will not persist. Instead, the structure of an Elixir program expects you to match the variable to the child scope, and return the intended assignment to the parent scope. Assuming there is no assignment to these constructs, then constructing a model is straightforward. We can derive the parent-child hierarchy directly.

We hold multiple types of symbol tables, to represent the different constructs such as modules, functions and blocks. Within these symbol table types, we further can assign child symbol tables to account for the nesting of these scoped constructs.

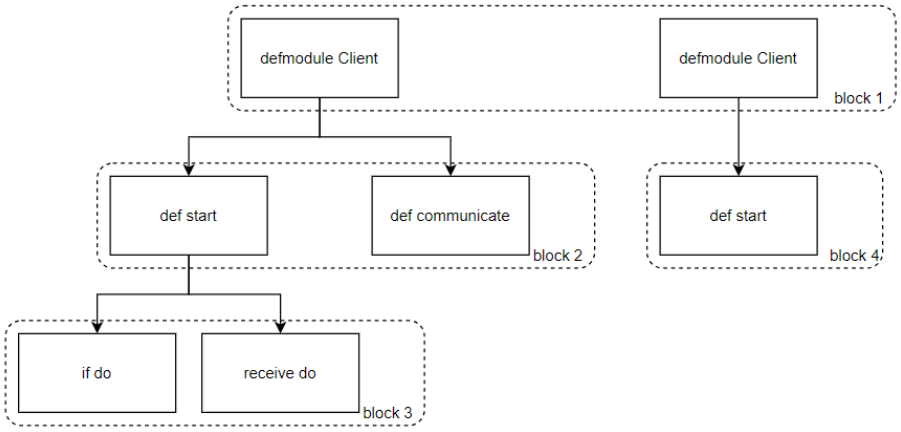

Figure 5.3: An example symbol table scope hierarchy.

Traversal, Scoping and Variable Declarations

So far, we assumed no expressions were matched to variables. To support assignment, we are required to track the execution of a scope in more detail. In the intermediate representation, a match is represented as either a **declaration** or an **assignment**. Given a declaration, we can infer the variable type and assign this in the symbol table. Now the variable is declared, any subsequent match in the same scope level is considered an assignment.

If we now match an expression that introduces a new child scope (such as a receive or if), we must explore every possible branch, determine the **returning expression** of the branch and use the relevant identifiers to assign the return value to the parent scope. To achieve this, the expression is traversed using a depth-first search with a stack of lists. The stack manages the scope nesting and the lists manage the variable identifiers. Let's explore an example.

```
1    {player, action} = receive do
2        {:move, player, direction} ->
3            {player, "moved #{direction}"}
4        {:attack, player, target} ->
5            send health, {target, -2}
6            {player, "attacked #{target}"}
7    end
```

Listing 5.1: Representing variable declarations using the match operator.

In the example, we are matching with a tuple. We push *player* and *action* to the **identifier list** and then descend into the first guard (conditioned by the `:move` atom). This is a singular expression, so it must be the return value of this guard. We can peek the scope stack to access the list of variables, and iterate through them assigning the relevant values. Assuming this is a declaration, we also add the identifiers to the **current scopes symbol table**. We can mark direction as a *string* as this is easily inferred, but we leave *player* as *unknown* until we can gather more information about the type.

We now traverse the second branch. This follows the `block` construct, so we require pushing

an empty list to the stack. We recursively apply this process until we reach the last expression in the block. Reaching the last expression, we can pop the stack (removing the child scope level) and then peek the stack to access the list of variables from the parent scope. We assign these using the same method, this time asserting the types align with the symbol table or inferring more information about *unknown* types if possible. Figure 5.4 shows the stack and lists for the second receive guard.

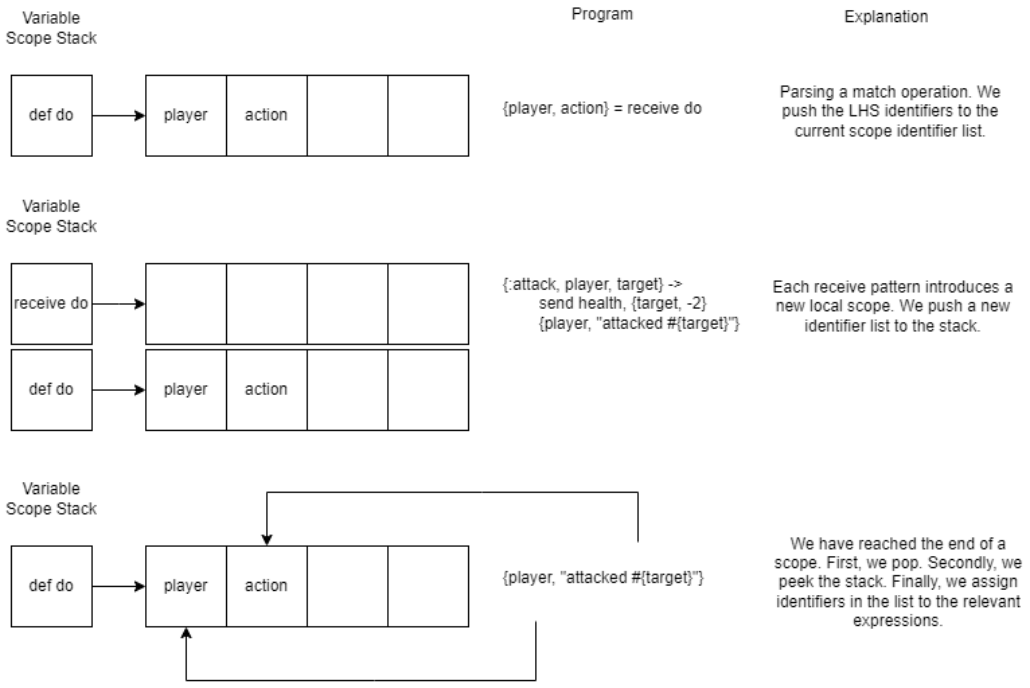

Figure 5.4: Example variable stack and identifier lists. The stack relates to the scope level, we push and pop as we traverse the receive guard. Only values waiting for assignments are added to the identifier list.

Functions, Returning and Recursion

Like the other constructs, functions introduce a new scope level. Promela does not support functions. We model functions using Promela processes and rendezvous communication channels. To implement functions, we apply various techniques that implement the behaviour of Elixir functions.

Let's consider a function call. The caller must declare a new communication channel, used to output the return value on. We also declare a new variable to read the return value of the function call, typed using the type specification of the callee. We can now spawn a new process and pass the function arguments, alongside some additional information.

We first pass the return channel, which is also used for determining whether the callee has terminated. By creating the return channel with a buffer size of 0, the caller will block until the callee returns. We also pass a process identifier. All processes are identified by this process ID; by passing this to the callee, the callee can take actions as if it were communicating as the parent process. We then pass the remaining arguments as if we were making an Elixir function call.

The caller will now block until the callee sends a message over the return channel. The callee can proceed as normal, and by using the discussed traversal techniques, all exit points will send the final expression over the return channel. This approach easily extends to **recursion**, as the callee can spawn a new instance of itself and block until a signal is received. In figure 5.5, we show an example of a recursive function call.

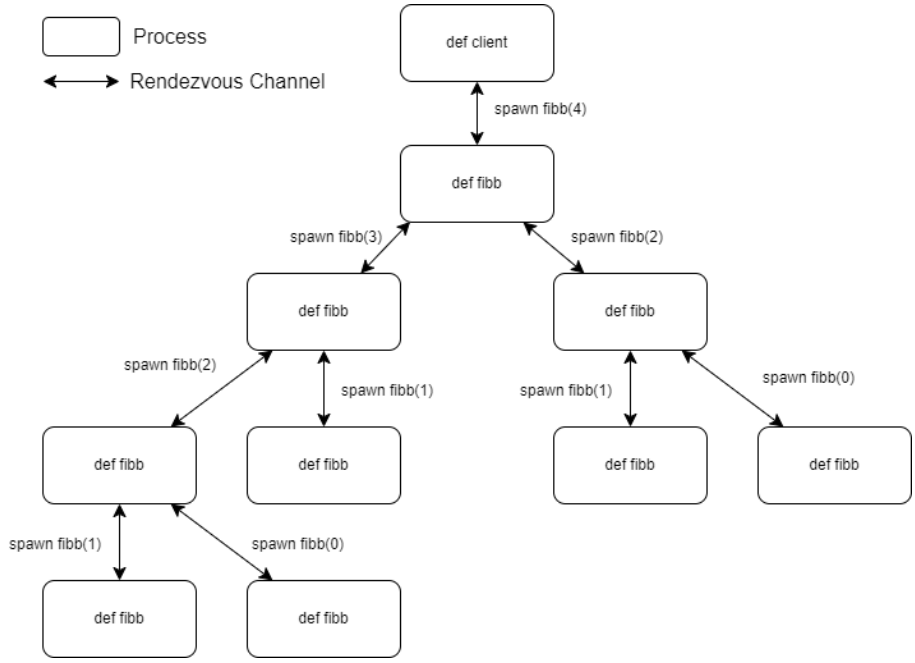

Figure 5.5: An example of a recursive function call. In total, 9 processes are spawned to calculate the factorial of 4. Callers block until they rendezvous with callees. If we consider the call stack as a graph, it is traversed in a depth-first manner.

5.2.3 Concurrent Memory Model

Now we have explored the basics of modelling Elixir programs, we extend the ideas to programs with multiple processes running concurrently. To correctly design a model of a concurrent Elixir system, there are a few core principles we must capture.

- Spawning processes.
- Sending messages.
- Receiving messages.

We will first explain how we model the spawning of a new process, before taking a deeper look into more complex concurrency primitives as well as a memory model to extend the existing capabilities of Promela.

Each Elixir `function` is already translated to a `proctype`. We need to tell apart function calls from process spawns. During a spawn, we pass a process identifier to the new child process. In this case, the process identifier is a reserved identifier, which prompts the child to ask the scheduler for a new id. This ensures all parents are uniquely identifiable, which is crucial for communication. The process identifier can be captured in the caller's symbol table and used to communicate with the process.

Actors, Mailboxes and Message Passing

Once we have another process's identifier in our symbol table, we can begin communication between processes. To model Elixir actors, we must model the three core components of an actor system: sending, receiving and spawning. We now look at sending and receiving.

Constructing Messages

A message is internally comprised of two components: the **message type** and the **message body**. When a message type is used in the context of a send or receive it is added to a global set of message types. Tracking this set globally is important to model the entirety of the system.

The message body consists of multiple message arguments. We can store any primitive type within a message argument by inferring the type from the send or receive expression. If a type cannot be inferred in the context, we reserve a byte array to store the message argument, but to avoid this causing memory issues, we limit the size of the byte array to a small fixed size.

```
typedef __message_component {
    byte data1[2];
    int data2;
    byte data3[2];
    bool data4;
    bit data5;
};
typedef __message_body {
    __message_component m1;
    __message_component m2;
    __message_component m3;
    ...
};
```

Listing 5.2: Example of a message body, from the Verlixir Promela library.

Sending Messages

Now that we can construct a message, we can begin to model how messages can be sent and received. Using the global message type set, we construct a mailbox for each message type. The mailboxes are indexed by using a process identifier. We use a **sorted insert** (!!), so that messages are grouped by their intended target. When a process sends a message, we triage which mailbox the message should be sent to, using the message type and use the process identifier from the symbol table. We also attach the message body to the mailbox. In figure 5.6, we show an example of how a mailbox is used.

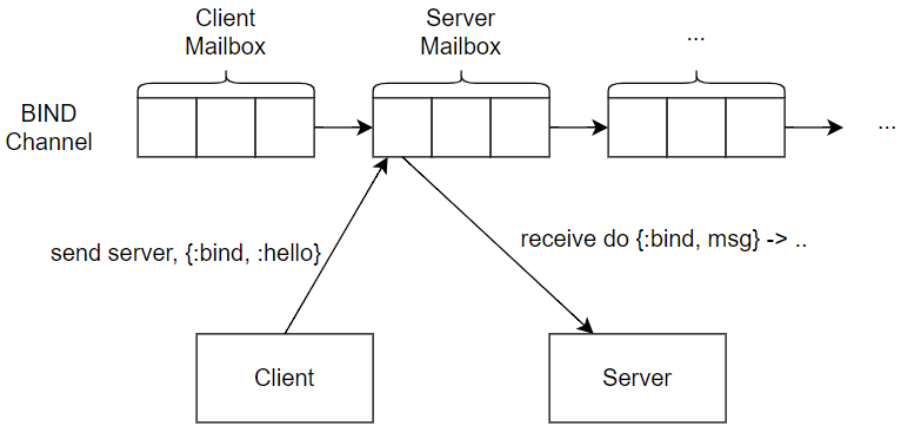

Figure 5.6: An example of modelling the Elixir mailbox.

Receiving Messages

Receiving a message is a little more complex. We must now consider pattern matching. To begin pattern matching, we can again use the message type to determine which mailbox to check. If

more than just the message type is required in the pattern matching of a guard, we must pre-empt elements of the message body to determine which elements are important for pattern matching and which elements are identifiers that need assignments.

In order to generate a model for the patterns, we introduce a blocking statement consisting of multiple conditions. When one of the conditions is satisfied (i.e. a message has been pattern-matched), we stop blocking, execute the block relevant for the matched guard and then break out of the control structure.

Before we can execute the block, we must assign all the remaining identifiers from the receive pattern, to the relevant values in the received message body. To do this, we first introduce a new dummy variable which is assigned the value of the message body. We can then access the dummy variable to assign the remaining identifiers from the guard. During the assignment, we have no indication of the type of the identifier. Hence, we can temporarily assign a message argument to the identifier, and extract the correct attribute from the message argument when we have more information about the type.

Unlike Elixir actors, Verlixir bounds multiple communication primitives. This is to ensure state explosion in the model checker is less likely to occur. As mentioned, we are strict in the bounding of byte arrays that can be passed as message arguments. We also only supply a small number of mailboxes per message type, furthermore, we bound the number of messages that can buffer in a per-process mailbox.

Listing 5.3 shows an example of a receive. We are receiving a *vote* message. We always match on `__pid`, to ensure the message is intended for the current process. We read the body into `__rec_v_0` and then extract the individual components. We

```
__message_body __rec_v_0;
do
:: __VOTE ?? eval(__pid), __rec_v_0 ->
    __message_component x = __rec_v_0.m1.data2;
    ... body ...
    break;
od
```

Listing 5.3: Example of a receive pattern. Translation of `{:vote, x}`.

Dynamic Memory Allocation

Promela does not support dynamic memory allocation. Elixir uses dynamic memory for its structures like lists, maps and sets. The Verlixir Promela library introduces two structures to handle all list operations across the system. Promela supports statically sized arrays. These arrays cannot be passed to other processes. Hence, an array's lifetime is limited to its scope. To get around this limitation, the two structures we introduce are called `memory` and `linked_list`.

First, we describe the design of `linked_list`. It is in fact not a traditional linked list, but it behaves similarly, as we can perform many operations on this structure that we could not perform on a statically sized array. For example, we support prepending and appending, which causes dynamic resizes. In actuality, none of these operations resize the list. A list has an upper bound in its size, which is statically set for all model generation. Let's name this limit L.

```
typedef __linked_list {
    __node vals[L];
}
```

Listing 5.4: The structure of a list.

For example for a limit, 10, means the maximum number of elements that can be appended to the list is 10. The list starts as empty and is handled by the `node` nested structure.

```
typedef __node {
    int val;
    bool allocated;
}
```

Listing 5.5: Example of a list node typed 'int'.

A `node` stores a single value in a list, as well as a flag to indicate if the value has been allocated. Now, given a sequence of nodes, the order of the nodes represents the order of the Elixir list for all allocated nodes. For example, for a list, ls, $ls[0]$ and $ls[9]$ can be contiguous list elements, if they are both allocated and there are no allocations in between.

Given this representation, we now see why all operations are in linear time. For example, for insertion (prepend or append), we must assign a pointer to a node and iterate through all nodes to find an unallocated node to insert into. We also support many other list operations in the Elixir standard library, all of which use similar logic.

We now extend this implementation to introduce dynamic memory. The implementation is similar to how lists are implemented. We introduce a new field to the `linked_list` structure, called `allocated`. This represents the allocation of a list in memory. We can then similarly introduce memory.

```
typedef __memory {
    __linked_list lists[M];
}
```

Listing 5.6: Memory intermediate representation. Limit of M lists.

We introduce a single, globally defined instance of this structure, named `__memory`. All processes share this memory for their list allocations. All list operations are treated as a single, indivisible step using **atomic**. If an Elixir process declares a new list, the IR will allocate a list from memory. The model will iterate through all the lists in memory, to find an unallocated list and return this, as a pointer, to the process. This *pointer* is generated as an `int`, which indexes into memory.

Finally, with all these definitions in place we can support the passing of Elixir lists as function arguments. When we detect a function call, or `send` expression, passing a list, we first allocate a pointer to a new location in memory. We then copy all the values from the list into the new allocation and then pass the pointer as an argument. With this memory in place, we can now support Elixir lists and operations on them.

Iteration and Higher Order Functions

Promela supports for loops, but for our Promela library, these are not sufficient. A **for comprehension** can be represented as a linear scan through all `linked_list` elements. We find the allocated elements and apply the comprehension body to these. We can introduce temporary dummy variables to represent Elixir's `<-` operator.

If we want to match on a for comprehension, we must also introduce a pointer into a second `linked_list` which tracks new allocations into the matched block independently of the scan through an existing list. A for comprehension over an Elixir range construct, `n..m`, can be represented with a for loop.

A map operation (such as from the `Enum` Elixir library) is represented similarly to a for comprehension in the IR. Instead of inlining the body, in order to hold a more fair representation of the Elixir program, dummy anonymous processes are stored to represent higher-order functions.

As a closing note on memory, we describe the representation of randomness. Again, Promela

does not inherently support randomness which has influenced the IR design. To represent a function, such as **random**, from the `Enum` library, we represent this using multiple truth conditions, which can be selected non-deterministically. One of these conditions will return an allocated list element and one will increment the current list pointer. To ensure termination, if we point to the end of the list before returning a value, we simply return the last allocated value we saw. This is not true randomness, and is not fair to all elements in the list, hence, random operations are strongly discouraged in LTLixir, but are theoretically supported.

Note that as Promela does not support functions, when we generate a model, the Verlixir Promela library is inlined into the model.

5.3 Specification Language

LTLixir is a specification language that can be used to reason about the time and change of an Elixir system. This section will primarily discuss the design decisions behind the additional constructs we introduce to Elixir. LTLixir is an Elixir library, built with Elixir's metaprogramming capabilities.

We call any construct prefixed with `@` a system annotation. All system annotations are treated as ghost variables, meaning they do not affect the runtime of the Elixir program.

5.3.1 System Initialisation

@init marks the entry point for the three operational modes. `@init` is captured in the intermediate representation; every function marked with `@init` is instrumented as **active** in the model. Active processes are spawned in the initial system state.

5.3.2 Type Specifications

For each argument parsed when creating a new function in the IR, we insert a new symbol table entry using the type provided in the type specification. The return type is a special case, as the function could be labelled **non-returning**. We instrument non-returning functions separately from returning ones.

In returning functions, we must ensure every path the process can take returns a value. Similarly, to ensure function calls calls are handled correctly, a non-returning function must return a dummy value at each exit point. The dummy value is used to rendezvous with the caller, signalling the callee has finished execution.

5.3.3 Concurrency Parameters

A concurrency parameter is a parameter which may cause a change in the behaviour of the system. Every parameter passed to `@model` is marked as a concurrency parameter in the IR. Instead of parsing the Elixir declaration for the parameter, we assign a reserved `__PARAM` value to the identifier. The model instrumentor will replace these reserved values with various configurations of the parameter. We discuss how these configurations are generated when we discuss the modes of operation, in section 5.4.

5.3.4 Linear Temporal Logic Formulae

The specification language supports the introduction of temporal properties through system annotations. The Verlixir library parses these as a string. The model generator will parse the string,

and translate it into a Promela LTL formula.

We store a separate symbol table for **temporal variables**. Temporal variables refer to Elixir variables that are used in LTL formulae. When we parse the program and find the declaration of a temporal variable, we instrument it as a global variable in the model. We now consider this a system property, instead of a property bound to a single process. The Elixir program does not, of course, allow any other processes to modify these variables, so the semantics remain unchanged.

We create a single model for each temporal formula, which allows them to be checked in parallel. This is an important optimisation, as model checking can be computationally expensive. Any non-Elixir constructs used in temporal properties, translate directly into Promela LTL.

The extraction of these properties is in fact a little more complex than explained. We have to consider the nesting of identifiers within Elixir constructs. There is also the possibility that some temporal variables are used as concurrency parameters. There are multiple reasons why local identifiers could conflict when moved to a global context, so conflicts also have to be carefully resolved. To preserve the semantics of the LTL formula, any assignment to a temporal variable must be captured in a single, **atomic** step. Due to the nature of Elixir's match, capturing this in the model is not trivial.

5.3.5 Predicates

Predicates rely on Elixir's metaprogramming capabilities. A predicate has an identifier and a condition. The condition can be evaluated either at runtime or by the model checker. We implement a `predicate` as an Elixir **macro**. Macros are compile-time constructs that are invoked with Elixir's AST as input, and produce a superset of the AST as output. The `predicate` macro matches the identifier to the condition and inserts it back into the AST. The identifier can be used in any Elixir construct where a condition is expected.

A defined predicate can be used in an LTL formula. We globally define the predicate in the model, such that we can use the predicate identifier in the LTL formula. Given the predicate condition, we must recursively search the condition for any nested identifiers. We move these identifiers to the global scope, and replace the nested identifier with the global identifier. All predicates and nested identifiers are now temporal variables. This step introduces the same complexities as the temporal variables previously discussed.

5.3.6 Function Contracts

A function contract is defined with `defv`. As with predicates, `defv` is an Elixir macro, which directly modifies the Elixir AST. Semantically, a contract acts similarly to `def`. The macro inserts an expression into the AST which contains the function body and **pre- and post-conditions**. Syntactically, these can be defined using the terms `pre:` and `post:` in the function declaration, before the body of the function.

```
1 defv add_positives(x, y), pre: x > 0 && y > 0, post: ret == x + y do
2     ...
3 end
```

Listing 5.7: An example contract.

The `defv` macro is intended for verification. It also instruments the execution of Elixir programs, such that at runtime, any violation of a condition will be flagged. This is achieved by capturing the pre- and post-conditions. In the absence of a condition, we insert the `:ok` atom into the AST. Otherwise, a quoted expression is constructed by first unquoting the condition, evaluating its truth and then possibly flagging a violation. When a call is made to the function, we build a final quoted expression to instrument the call. This final expression consists of the following steps:

- We capture a function call and delay the evaluation of the function body.
- We evaluate the precondition check.
- We evaluate the function body, buffering the result.
- We evaluate the postcondition check.
- We return the buffered result.

Relying on evaluating these conditions at runtime does not prove the correctness of a specification. However, by using contracts with Verlixir, we can ensure function calls and returns are consistent with their specification.

Verlixir models pre- and post-conditions as **assertions**. Assertions are placed at the entry and exit points of a process. Naturally, the model checker reports assertion violations. We map these assertion violations back to the original Elixir program. The algorithm to determine exit points is the same as we used when returning values. A slight complexity arises if the return value is a non-trivial expression, for example, a receive or if statement. In this case, we must ensure the evaluation of the expression occurs before the post-condition check. This can be complex, as expressions can be nested.

5.4 Simulation and Verification

We have now explored the intermediate representation constructed by Verlixir, alongside some implementation details specific to using Promela as a target language. This section will describe the remainder of the Verlixir design, which touches on simulation, verification, generating multiple models and using Spin as a target model checker. The default execution of the Verlixir executable, will produce a single Promela model, for a given input LTLixir specification. For Verlixir to produce outputs, it should be executed in an operational mode: $\{-s, -v, -p\}$ for simulation, verification or parameter exploration.

If $-s$ is passed, Spin is run in simulation mode. If $-v$ is passed, Verlixir will first determine the existence of LTL properties within the model. For each LTL property, we run an instance of Spin specifying the LTL property we want to evaluate. We do this using Spin's $-search$ parameter, which will generate and run the verifier from the model. In the event we do not detect an LTL property, we use $-search$ to check for the existence of deadlocks or non-progress cycles (livelocks). If $-p$ is passed, multiple models are generated instead of one, and all of these are checked as if $-v$ was passed for each.

5.4.1 Simulation

Simulation mode will use the Spin scheduler to execute a single execution through the generated state space. A simulation can timeout in the case of a deadlock. If a timeout is produced, we inform the user of the timeout but do not provide further information as that is left for verification mode. Elixir calls to the `IO` library can be reproduced in simulation mode, as we display them to the user if they are executed. Simulation mode can be more beneficial than examining the output of running an Elixir program. The scheduler takes enabled transitions, based on the current state, whereas, the Elixir scheduler, runs in real-time, which could result in a process interleaving never being observed.

5.4.2 Verification

Verification mode will run the Spin verifier. If an error is detected, the output is captured by the Verlixir error profiler, which triages the issue to generate relevant outputs for the user to digest. For example, if a non-acceptance cycle is produced by Spin, Verlixir captures this and reports to the user that either an LTL property was violated (liveness property) or the system is potentially livelocked. Verlixir will report the entire trace that led to this cycle, showing which process was

responsible for a transition between states as well as reading from the Elixir module the line of Elixir code that led to that transition.

Reporting Violations

The mapping between Elixir expressions and how they are modelled is not a one-to-one mapping of line-numbers. Instead a single Elixir expression can result in multiple Promela expressions and multiple states in the state-space Spin traverses. Still, all the relevant Elixir expressions are captured and reported in the trace produced. For this error, Verlixir will also report the cycle that led to non-terminating processes. It will report where the cycle begins, then give a trace of executed Elixir lines and which function executed them. The error profiler also captures and reports the processes which have terminated, or a blocking expression (such as a receive with no accepting guards). For a given model M, the profiler can determine the following truth conditions for the specification ϕ. If the specification holds for an initial state, S_i, we have $(M, S_i) \models \phi$. If the specification holds for all initial states, the specification is valid: $M \models \phi$. The error profiler may report a violation, in which case for a violating specification ϕ_v we have $(M, S_i) \models \phi_v$. For example, deadlocks, non-accept cycles, out-of-memory, assertion violations are possible violating specifications all represented by $\phi_v \in \{\phi_{dl}, \phi_{cycle}, \phi_{memory}, \phi_{assert}\}$. In the event that $M \models \phi_{memory}$, we could re-run the verification process using directives to reduce memory usage. If this is required, Spin will no longer perform an exhaustive search. If the specification is true under these conditions, we can only say the model partially models the specification, $(M, S_i) \models \phi_p$. A non-exhaustive search should only be applied as a last resort, if it is infeasible to perform verification by reducing the system complexity and can be performed using the $-r$ flag.

Fairness

Similarly, in the event $M \models \phi_{cycle}$, it may be useful to introduce weak fairness to the system. Weak fairness can be applied by passing the $-f$ flag when in verification mode. Other flavours of fairness should be introduced using LTL formulas. If the weak fairness flag is active, scheduling decisions will consider how process-level non-determinism is resolved. For example, if a non-progress cycle is detected by a single process infinitely executing, even when other active processes are not being scheduled, by re-running the verifier with weak fairness applied, infinitely enabled transitions will eventually be scheduled to be taken. This may instead report a new error, consisting of a fair non-progress cycle, or even avoid the non-progress cycle entirely.

5.4.3 Parameterization

The Verlixir IR passes all detected parameters to the Verlixir model runner. The model runner is responsible for generating multiple models depending on the number of parameters provided, N and the range of search values, M, which can be parsed as a command line argument using $-p\ M$. The model runner generates a total of M^N Promela models. All of these models are run in $-search$ mode and violations are reported. Verlixir reports an acceptance score, α, determining how many of the models were accepted by the verifier: $\alpha := 1 - \frac{|V|}{M^N}$, where V are violating models. After outputting α, we note all the elements of V so the user can generate a trace for the violation using verification mode.

5.5 Modelling Paxos

We will now demonstrate the model generation process for a larger example, the basic Paxos algorithm. A thorough explanation of the problem will be discussed in section 6.1.1. For now, we will just concern ourselves with the syntax of the Elixir program and how it is parsed and translated into Promela.

The entire program is large, so we will just look at a single Elixir $defmodule$. In particular,

we will look at the *learner* module. When the system is distributed, the *learner* could be an Elixir node, or an Elixir process. For our model, it does not matter. We will now take a look at the Elixir implementation of one of the functions of the learner, *wait_learned*, focusing primarily on the syntactic elements.

```
@spec wait_learned(list(), integer(), integer()) :: :ok
@ltl """
[]((p->!<>q) && (q->!<>p))
<>(r)
[](s)
"""
def wait_learned(acceptors, p_n, learned_n) do
    predicate p, final_value == 31
    predicate q, final_value == 42
    predicate r, final_value != 0
    predicate s, final_value == 0 || final_value == 31 || final_value == 42

    if p_n == learned_n do
        for acceptor <- acceptors do
            send acceptor, {:terminate}
        end
    else
        receive do
            {:learned, final_value} ->
            IO.puts("Learned final_value:")
            IO.puts(final_value)
        end
        wait_learned(acceptors, p_n, learned_n + 1)
    end
end
```

Listing 5.8: A function from the *learner* module.

Listing 5.8 shows an Elixir function, *wait_learned*. The function specifies some LTL properties, using the multi-line LTL syntax. The LTL properties rely on the predicates defined in the function body. The function also receives some arguments. In the body, we see a conditional branch, where one branch involves a *for* comprehension, through a list performing some message sending. The other branch receives a value. At the end of the *else* condition, we recurse. Let's now break down the Promela model.

```
#define p ((final_value == 31))
#define q ((final_value == 42))
#define r ((final_value != 0))
#define s ((((final_value == 0) || (final_value == 31)) || (final_value == 42)))

proctype wait_learned (int acceptors;int p_n;int learned_n;chan ret;int
    __pid;int __ret_f) {
    chan ret1 = [1] of { int };
    atomic{
        if
        :: __pid == - 1 -> __pid = _pid;
        :: else -> skip;
        fi;
    }
```

Listing 5.9: Promela function definition.

Listing 5.9 shows the first translation result of Promela from the Elixir function. Firstly, we see the predicates are pulled from the body and placed into the global context.

The function is translated to a process type, named *wait_learned*. The process receives the same arguments as the Elixir function, using the type specification to type the arguments. Additionally, we receive some meta-arguments *___pid* and *___ret_f*. The first line of the function body creates a channel, named *ret*1. Channels of this nature will be used in function calls.

Finally, we see an *atomic* block, which performs a check on the received *___pid*. If the process ID passed is -1, we assign a new process ID from the process scheduler. Otherwise, we continue to act as the caller, by retaining the passed process ID. Next, we look at the first branch of the Elixir *if* statement. In particular, we will look at the *for* comprehension.

```
1  if
2  :: (p_n == learned_n) ->
3      atomic {
4          __list_ptr_old = __list_ptr;
5          __list_ptr = 0;
6          __list_ptr_new = 0;
7          do
8          :: __list_ptr >= LIST_LIMIT || __list_ptr_new >= LIST_LIMIT ->
9              __list_ptr = __list_ptr_old;
10             break;
11         :: else ->
12             if
13             :: LIST_ALLOCATED(acceptors,__list_ptr) ->
14                 int acceptor;
15                 acceptor = LIST_VAL(acceptors,__list_ptr);
16                 atomic {
17                     MessageList msg_0;
18                     __TERMINATE!!acceptor,TERMINATE,msg_0;
19                 }
20                 ;
21                 __list_ptr_new++;
22                 __list_ptr++;
23             :: else -> __list_ptr++;
24             fi
25         od
26     }
```

Listing 5.10: Promela if statement and for comprehension.

Listing 5.10 shows the translation for the *if* statement. Each condition in the *if* is determined with a (::) operator. In this listing, we just have the first condition, the *else* will be seen in a later listing.

Next, we have the translation of the for comprehension. This is wrapped in an atomic block, as it relies on accessing shared memory. It looks daunting, but all that is taking place is a linear scan through the list. We check each element, determine if it has been assigned and if it has, then the comprehension body (lines 14 through 19) is executed with the list element.

In this case, the comprehension body is a *send*. To send, we pack the message elements into structure, and then send this data to the channel matching our *terminate* atom, and the mailbox matching *acceptor*.

Now let's look at the *else* condition.

```
1  :: else ->
2      MessageList rec_v_5;
3      do
4      :: __LEARNED??eval(__pid),LEARNED,rec_v_5 ->
5          final_value = rec_v_5.m1.data2;
```

```
6           printf("Learned final_value:\n");
7           break;
8       od;
9       int __temp_cp_arr_2;
10      __copy_memory_to_next(__temp_cp_arr_2,acceptors);
11      int __ret_placeholder_1;
12      run wait_learned(acceptors,p_n,learned_n + 1,ret1,__pid,1);
13      ret1?__ret_placeholder_1;
14  fi;
```

Listing 5.11: Promela receive and recursion.

Listing 5.11 shows the translation of the *else* condition. Again, this is guarded by a (::) operator. We do two things here: first we receive a message and then we recurse.

The receive only has one guard. We translate this guard by reading from the *learned* channel, looking into our mailbox (marked by `__pid`). The data we read from the channel is read into `rec_v_5`, which we unpack into the relevant variables.

Secondly, we make a recursive function call. As one of the arguments passed is a list, we must ensure we pass the list by value. To do this, we get a new pointer from memory, and copy the values from our local list into the new memory location.

We next create a value to store the returned value from the recursive call. Actually, in this case the function's type definition marked the function as 'non-returning'. We recursively call the function by spawning a new instance of the process type. We pass the parameters, as well as the relevant meta-arguments. In this case, the meta-arguments are:

- The channel to send the return value to, *ret*1. This was defined at the beginning of the function body.
- We pass our process id, so the spawned function can continue to take actions on our behalf.
- We pass 1. 1 marks a non-returning function.

We then wait to read on *ret*1 to rendezvous with the callee.

Let's finally look at the remaining translated code.

```
1       atomic {
2           if
3           :: __ret_f -> ret!0;
4           :: else -> skip;
5           fi;
6       }
7   }
8
9
10  ltl ltl_1 { [](p -> ! <> q) && (q -> ! <> p)) };
11  ltl ltl_2 { <> (r) };
12  ltl ltl_3 { [](s) };
```

Listing 5.12: Translation of LTL properties.

Listing 5.12 shows the remainder of the translated code. To end the function, we have a final *atomic* block. Within this block, we check the value of the passed *___ret_f* value. If this is 1, the function is non-returning. To handle the resolution of the call stack, we propagate a ghost 0 through the return channels. If it was a returning function, the return would already have been handled, and hence we can *skip*.

Finally, after the function and back in the global context, we define our LTL properties. We split the multi-line property into multiple properties so they can be checked in parallel. They rely

on the predicates defined, which were defined in the global context as they are properties of the system, not a single process.

5.6 Summary

This chapter has discussed some core design concepts behind Verlixir. We first looked at a high-level overview of where the tools fit into a wider toolchain and gave a basic introduction to their architecture. We saw how Elixir was extended using metaprogramming to support inline specifications. We then introduced the core techniques essential to constructing an intermediate representation of a specification, as well as how Promela can be used as a target modelling language for Elixir systems. Finally, we looked at how Verlixir interacts with the model checker Spin to simulate and verify programs while capturing traces of Elixir programs. We will now evaluate the effectiveness of this tooling.

Chapter 6

Evaluation

In this chapter, we aim to evaluate the expressiveness of Verlixir's design. First, we perform a qualitative analysis of modelling classical distributed algorithms such as basic Paxos in section 6.1. Then section 6.2 will delve into a comparison against current state-of-the-art model-checking techniques for modern-day programming languages and verification-aware languages.

6.1 Analysing Distributed Systems

Verifying the correctness of real-world distributed systems is a major motivation for this project. Critical real-time systems (such as in air-traffic control or healthcare [39, 40]) should not fail and should rely on rigorous verification techniques to guarantee production code is correct. All code examples in this section are available in the appendix.

6.1.1 Basic Paxos

Paxos is an example of a distributed algorithm [41]. It is a consensus algorithm, where many processes are tasked to agree on a value. Processes may propose what this value should be, but only one value should be agreed upon. The safety requirements (SR) for consensus are:

- **SR1**: Only a value that has been proposed may be chosen.
- **SR2**: Only a single value is chosen.
- **SR3**: A process never learns that a value has been chosen unless it actually has.

The system's liveness requirement is that a proposed value is eventually chosen and if a value is chosen then a process can learn the chosen value.

Informal Specification

There are many flavours to the Paxos algorithm. We will informally present a basic, one-shot Paxos. We introduce three roles in the system: proposer, acceptor and learner. The Paxos algorithm performs two steps: prepare and accept. A proposer will broadcast a prepare message to all the acceptors, who will respond with a promise. When the proposer has received a promise from a quorum q of acceptors, it will broadcast an accept message. If more than q acceptors accept, then the value is chosen, and the learners are informed.

To evaluate the expressiveness of Verlixir, we first must write the Paxos specification in LTLixir. The specifications of proposer, acceptor and learner are similar to those presented in pseudocode by Marzullo, Mei and Meling [42]. We now present the key differences in our Elixir specification to a traditional Paxos design.

All processes contain two functions, a start function to introduce relevant initial configuration and a main loop to process messages. Every acceptor initializes an accepted proposal, value and minimum proposal to -1 and then processes *prepare* and *accept* messages until receiving a *terminate* message, signifying consensus has been reached. A termination clause is important to ensure the

completion of a round of Paxos. A proposer receives its configuration in the form of a *bind* message, before executing its protocol. If during phase two, when asking acceptors to accept a value, a quorum of acceptors rejects the proposal, the proposer will inform the system it has reached consensus on value 0. Traditionally, the proposer would retry with a higher proposal number, but we aim to avoid infinite paths so instead introduce this terminating condition. The learner awaits a *learn* message from all proposers. We only ever consider a single learner and the learner is also responsible for spawning the proposers and acceptors, choosing their values and assigning proposal numbers for the single round of Paxos. We finally set up the learner such that it spawns three acceptors and two proposers. The learner decides the values the proposers will propose, which for this example will be 31 and 42. Of course, in a different context, these values may come from other sources within a larger system, however, notional values are sufficient for our purposes.

With our implementation complete, we introduce the three safety requirements established. To achieve this, we introduce a value $final\_value$ which the learner receives from proposers. This value is initialized to 0 and set to the agreed value of consensus. Let's specify the temporal formula required to express our safety requirements. We first introduce four predicates into our specification (note the use of 0 both represents a state where consensus is unreached, or a value received from a rejected proposer).

predicate chooseA: final_value == 31
predicate chooseB: final_value == 42
predicate chosen: final_value $\neq$ 0
predicate learned: final_value == 0 $\vee$ final_value == 31 $\vee$ final_value == 42

We can now use the predicates to simplify the formulation of the safety requirements.

SR1 : $\Diamond$ chosen	Only a proposed value is chosen
SR2 : $\Box$ (chooseA $\rightarrow \neg\Diamond$chooseB) $\wedge$ (chooseB $\rightarrow \neg\Diamond$chooseA)	Only a single value is chosen
SR3 : $\Box$ learned	Only proposed values are learned

We now have a complete specification of the basic Paxos algorithm in LTLixir. Note that SR1 could be considered a liveness requirement, as a result of slight modifications on the original SRs to align with our specific implementation decisions. We can run Verlixir on the model to verify the safety requirements. When we run the verification mode, we see that no SRs are violated. This justifies that both the informal Paxos specification we defined is correct regarding our SRs and that the implementation of the specification is also correct.

```
1    Model ran successfully. 0 error(s) found.
2    The verifier terminated with no errors.
```

This gives a good indication that the expressiveness of Verlixir is sufficient to model and verify distributed systems. However, we also should investigate how Verlixir can express errors for a more complex system such as Paxos.

Counterexample 1

We introduce a bug into the proposer's protocol. The proposer will now wait for a majority of acceptors to accept the proposal and only be rejected if a majority of acceptors reject the proposal. This is a violation of the protocol, as we only need a single rejection (within the accepting quorum) for a proposer to retry with a higher proposal number. We can now run the verifier on the model again to see if the bug is detected. Verlixir reports a violation of SR2, which is expected. In particular, we are told there is a violation SR2 due to ($final\_value == 31$). We can infer that the learner was informed the chosen value is 42, but a later proposer informed the learner the chosen value is 31. Verlixir detects this bug, informing us that SR2 was violated and then produces its counterexample. Digesting this counterexample can take some time, as the interleaving of process communication that triggers this bug involves approximately 50 messages and 800 steps. The full

message log is available in the appendix. We will provide a simpler interpretation to help reason that Verlixir has correctly identified the bug (derived from the message log) in figure 6.1.

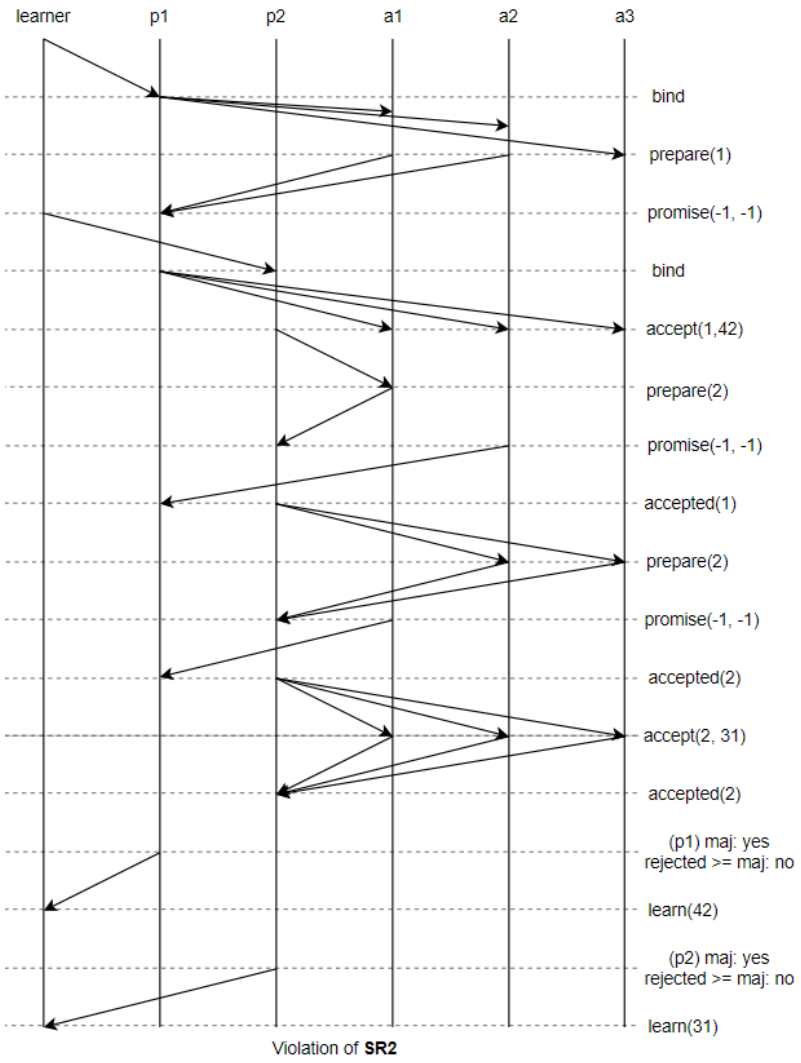

Figure 6.1: Violation of Paxos specification due to proposer bug. Note that the figure only shows the ordering of *receive* events. We see that $p1$ forms a quorum of *accepted* messages from $\{a1, a2\}$. Although one of these acceptors rejects the proposal (by sending a higher proposal number than $p1$ expected), the bug would require a majority of acceptors to have rejected the proposal, so $p1$ asks the learner to learn its value regardless.

Counterexample 2

We now explore a second counterexample, again, the Paxos specification and message log can be found in the appendix. This time, we introduce a bug into the proposer, such that if the proposer receives a $\{prepared, proposalNumber, value\}$ message from an acceptor with a higher proposal number, it propagates this proposal number forward. A correct Paxos implementation should keep the same proposal number, but propagate the value forward. We again get a violation of SR2, where the mutual exclusion of values is violated. The violation is the same as counterexample one but caused by a different interleaving.

6.1.2 Consistent Hash Ring

Consistent hashing is a distributed hashing technique designed to support dynamic loads of nodes in a system [48]. It has been used in large real-world systems to help scalability and load balancing [47]. Consistent hashing requires choosing a hash space and distributing both system nodes and system requests over the hash space. The hash-space is logically considered a ring due to the wrap-around semantics of the distribution applied over the hashing function.

We will look at a simple version of a consistent hash ring involving a handler and a ring manager. The handler receives requests from the outside world and sends these to the ring manager to be distributed. The ring manager is responsible for taking these requests and determining which node should be responsible for handling them. The ring can dynamically grow and shrink in size depending on the load from handlers.

To model the system, we are primarily concerned with one liveness property. Every incoming request should eventually be forwarded to the correct node in the ring. A more detailed implementation may involve the nodes communicating to determine the correct node for requests, identify faults and handoff information when nodes join or leave the ring. We will abstract this behaviour within our ring manager for now, and introduce some temporal properties to specify the system's correctness. Firstly, we will introduce some predicates to help simplify the LTL formulae.

$$\forall i \in \{1..4\} \text{ predicate } \text{pos}_\text{i}\text{: assigned\_node} == \text{node}_\text{i}$$
$$\forall j \in \{1..3\} \text{ predicate } \text{req}_\text{j}\text{: next\_request} == \text{V[j]}$$
$$\text{where } V = \{1 \to 42, 2 \to 31, 3 \to 25\}$$

These predicates p_i specify assignments of a value to a node in the ring and r_j specify the next request to be processed by the ring manager. We can now introduce our liveness property, which we do so by breaking into components to capture specific details of the system.

$\phi_1 : \Box(\text{req1} \to \Diamond\text{pos1})$	Hashing assigns correctly
$\phi_2 : \Box(\text{req2} \to \Diamond\text{pos3})$	Hashing assigns correctly
$\phi_3 : \Box(\text{req3} \wedge n\_nodes == 3 \to \Diamond\text{pos1})$	Hashing wrap-around semantics
$\phi_4 : \Box(\text{req3} \wedge n\_nodes == 4 \to \Diamond\text{pos4})$	Hashing for ring resizing

We use these properties to ensure the correctness of the system, by using an understanding of how the system hashes requests to enable verification of evolving behaviour. For example, we use the variable n_nodes to distinguish between different behaviour patterns depending on the loads of the system. In particular, ϕ_1 and ϕ_2 ensure that the ring manager assigns requests to the next sequential node in the ring. ϕ_3 is responsible for ensuring the wrap-around semantics. When the hash value of a request is larger than the last node's range, it should be assigned to the first node. ϕ_4 is responsible for ensuring that as the ring grows, the ring manager adjusts its assignment of requests so that the new node now receives its relevant load.

We can attach these LTL properties to the handler model, S, to determine that our incoming requests are being handled as expected. We can run them with Verlixir, which determines there are no violations of the properties, and our hashing is performing as intended.

Evolving the System Requirements

Up to now, we have been strict in our liveness properties. In other flavours of the system, we may not concern ourselves with the exact node a request is assigned to, but rather that the request is assigned to a node. Our current implementation enforces a synchronisation between the handler and the ring manager. Let's introduce a bug into the system that breaks this synchronization. Currently, our handler will wait for ring resizing to complete before sending more requests. We will modify the ring manager to dynamically resize asynchronously to the handler requests. This introduces a violation of our liveness properties, as we can no longer guarantee that every request is assigned to a specific node.

When we run Verlixir on the updated model, S', we see that $S' \not\models \phi_4$. The erroneous message log, alongside both specifications, can be found in the appendix. We will provide a simplified interpretation of the message log to help reason that Verlixir has correctly identified the bug in figure 6.2.

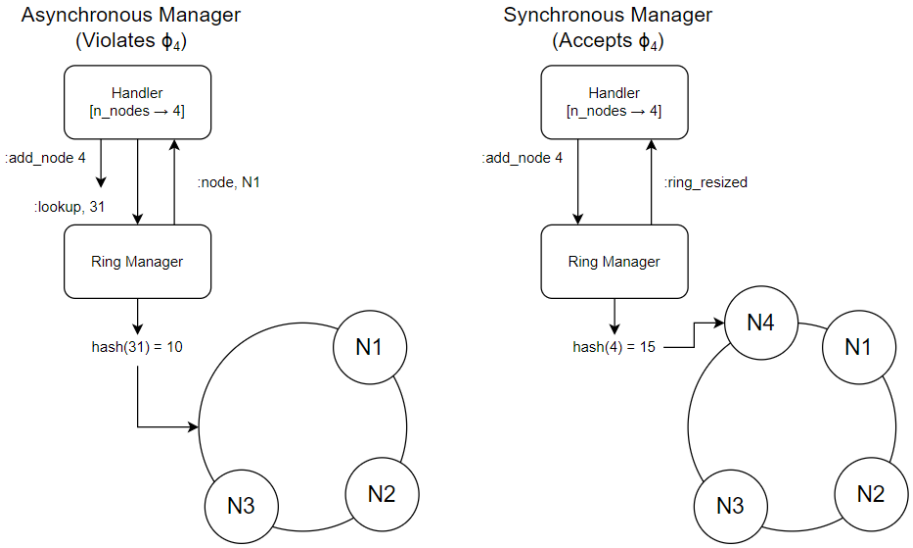

Figure 6.2: Violating and accepting consistent hash ring implementations. The violating model shows the handler sending *lookup* requests without awaiting confirmation of ring resize. This violates the liveness property ϕ_4, which specifically requires the manager to assign 31 to node 4. The accepting implementation waits for confirmation of a resize before continuing with requests. Note that *n_nodes* is the number of nodes the handler believes to be in the ring, not the actual number.

In this instance, instead of considering this an error, we may instead want to refine the system requirements. To do this, we can introduce a new liveness property to specify a weaker system, where we only care about requests being distributed to nodes.

$$\phi_5 : \Box(\text{sent\_request} \rightarrow \Diamond\text{assigned\_node})$$

Verlixir reports that $S \models \phi_5$ and $S' \models \phi_5$.

6.1.3 Two-Phase Commit

The two-phase commit protocol is a distributed algorithm used to update resources on multiple nodes in a single operation. It can be used to ensure replication of data is consistent across multiple nodes. For example, Spanner [49] uses a two-phase commit between leaders of replica groups to preserve the atomicity of transactions.

The protocol involves a coordinator and multiple participants. The coordinator is responsible for communicating with the participants to complete the two phases of the protocol. The first phase is the prepare phase, where the coordinator asks participants to prepare for a transaction. The second phase is the commit phase, where the coordinator asks participants to commit the transaction. In our design, the coordinator will also be responsible for terminating the participants after the transaction has been completed. When a participant is asked to prepare for a transaction, it will check the conditions it requires to commit a transaction and then reply with a *prepared* or *abort* message. The condition is typically application-specific, for example, it could be ensuring the participant has access to a lock required for a write operation. If a single participant aborts the transaction, the coordinator will ask all participants to abort. If all participants are

prepared, the coordinator will ask the participants to commit.

We informally specify the system with a safety and liveness property. The safety property is that all participants must agree on the same outcome of the transaction. The liveness property is that eventually, an outcome is agreed on. We can now formalise these by constructing predicates and LTL formulae.

predicate commit: commit_count == participant_count
predicate abort: abort_count == participant_count

The *commit_count* is the number of participants that have committed the transaction and similarly, the *abort_count* is the number of participants that have aborted the transaction. The *participant_count* is a property specified by the system. We will reason about the protocol using a system with three participants. We can now introduce the LTL formulae to specify the system.

SR1 : $\Box$ (commit $\rightarrow \neg\Diamond$abort)	Mutual exclusion
SR2 : $\Box$ (abort $\rightarrow \neg\Diamond$commit)	Mutual exclusion
LR1 : $\Diamond\Box$ commit $\vee$ $\Diamond\Box$ abort	Eventual agreement on commit or abort

Verlixir verifies the model is correct under these properties. We can now introduce a bug into the system. As we have mainly looked at safety properties so far, we will introduce a bug that may trigger a violation of LR1. To do this, we randomly make a participator 'faulty'. Faulty participators will always abort the transaction, regardless of the coordinator's request and even if they agree to commit. We can now run the Verlixir on the updated model to see if the system violates the specification. Verlixir reports that the system violates LR1, and provides a counterexample of an execution that violates the specification. We can again use a diagram to represent the violating execution, as shown in figure 6.3.

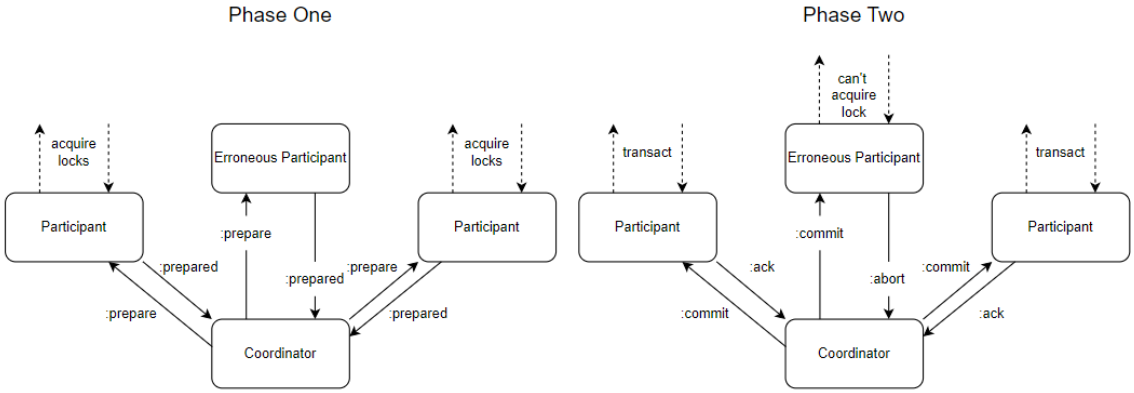

Figure 6.3: Implementation of two-phase commit that violates specification (LR1).

We can see from figure 6.3, that in the counterexample reported by Verlixir, the erroneous participant may reply to the coordinator with a *prepared* message, before acquiring the relevant locks. This means the coordinator has the relevant information to ask participants to commit. The erroneous participant will attempt to commit, determine it does not have the required locks to do so and then abort. Verlixir reports this violates the liveness property LR1, as the system never reaches a state where all participants either commit or abort.

For this counterexample, the message log produced by Verlixir makes the violation relatively obvious.

```
send [1, PREPARE, ...]
send [3, PREPARE, ...]
send [5, PREPARE, ...]
...
send [7, PREPARED, ...]
...
send [7, PREPARED, ...]
...
send [7, PREPARED, ...]
...
send [1, COMMIT, ...]
send [3, COMMIT, ...]
send [5, COMMIT, ...]
...
send [7, TRANSACTION_COMMIT, ...]
...
send [7, TRANSACTION_COMMIT, ...]
...
send [7, TRANSACTION_ABORT, ...]
...
```

Listing 6.1: Message log produced by counterexample violating LR1.

Listing 6.1 shows a reduced version of the message log produced. We can see the first broadcast of *prepare* messages is sent to all participants[1] and subsequently the participants reply with *prepared* messages. The coordinator then broadcasts a *commit* message, which is received by all participants. However, in the response to *commit* we notice that one participant sends an unexpected *transaction_abort* message. This explains the violation of LR1, as well as gives an indication as to where in the execution it was violated.

6.1.4 Dining Philosophers

The dining philosophers problem is a classic concurrency problem used to illustrate the challenges of concurrent programming. The problem involves a group of philosophers sitting at a table in a circle. Between each philosopher is a fork. A philosopher can either be thinking or eating. To eat, a philosopher must first pick up both the fork on their left and right side.

There are many flavours of this algorithm with increasing complexity to handle the concurrency primitives. To evaluate the expressiveness of Verlixir, we will specify the system with a naive approach:

- A philosopher will ask to be sat at the table.
- A philosopher will attempt to pick up the fork to their left.
- A philosopher will attempt to pick up the fork to their right.
- A philosopher will eat for some time.
- A philosopher will put down the left fork, right fork and then leave the table.

In particular, the process algebra for a philosopher, *Phil* is as follows:

Phil = (sit → pickupLeft → pickupRight → eat → putdownLeft → putdownRight → leave → SKIP)

The issue with this approach is that if all the philosophers pick up the fork to their left, then they will all be waiting for the fork to their right. This is a known deadlock, and we would expect Verlixir to be expressive enough to detect this. When we run Verlixir on the dining philosopher specification, it reports a violation of the safety property that a deadlock should not occur. We can read the trail produced by Verlixir, to determine our understanding of how a deadlock may

[1] The numbers 1, 3 and 5 are the process identifiers assigned to the participants by Verlixir. Although this reveals some of the internals Verlixir uses to model programs, being aware of what these are can help in understanding errors. They do not have any significance in the actual program.

arise in this naive implementation, is correct.

To summarise the trail produced by Verlixir, we will demonstrate a process composition for two philosophers, $Phil_0$ and $Phil_1$ that results in a deadlock. We denote the fork on the philosopher's left as $Fork_i$ and on the right as $Fork_{(i+1)\%n}$ for philosopher $Phil_i$, where n is the number of philosophers; similarly, for a fork, the philosophers are located at i and $(i-1)\%n$. We will use L and R interchangeably to express left and right.

$$\text{Phil}_i = (\text{sit}_i \rightarrow \text{pickup}_{iL} \rightarrow \text{pickup}_{iR} \rightarrow \text{eat}_i \rightarrow \text{putdown}_{iL} \rightarrow \text{putdown}_{iR} \rightarrow \text{leave}_i \rightarrow \text{SKIP})$$
$$\text{Fork}_i = (\text{pickup}_{Li} \rightarrow \text{putdown}_{Li} \rightarrow \text{Fork}_i) \mid (\text{pickup}_{Ri} \rightarrow \text{putdown}_{Ri} \rightarrow \text{Fork}_i)$$
$$\text{DiningPhilosophers}(2) = \text{Phil}_0 \parallel \text{Phil}_1 \parallel \text{Fork}_0 \parallel \text{Fork}_1$$

Which when run using Verlixir, produces the violating trace:

$$\text{sit}_0 \rightarrow \text{sit}_1 \rightarrow \text{pickup}_{00} \rightarrow \text{pickup}_{11} \rightarrow \text{STOP}$$

We can observe that even without explicitly specifying system properties, Verlixir is capable of detecting a deadlock in systems. We can now look at a shortened version of the message log produced by Verlixir to reason the counterexample produced, aligns with our intuition.

```
1  The program likely reached a deadlock. Generating trace.
2  <<<Message Events>>>
3  send [2,PICKUP,4]
4  send [4, OK]
5  send [3,PICKUP,5]
6  send [5, OK]
7  send [2,PICKUP,5]
8  send [3,PICKUP,4]
9  <<<Deadlock>>>
```

Listing 6.2: Message log produced by counterexample of a Dining Philosophers deadlock.

The trace in listing 6.2 shows four *pickup* messages being sent. The values in these messages do not necessarily directly correspond to anything in the Elixir program, however, we can use some intuition to work out what is going on. For any message log, the first element will be the intended receiver. Hence, the first message shows a philosopher with process identifier 4 sending a *pickup* message to a fork with process identifier 2. Continuing this logic, we can see the interleaving results in the philosophers with identifiers 4 and 5, both attempting to pick up the fork to their left. We see the forks confirming they have been acquired, before the philosophers attempt to pick up the fork to their right. This is where the trace ends, as Verlixir reports a non-terminating state has been reached and the system has deadlocked.

We can see that if we map the process identifiers for philosophers and forks to the numbering system we used in the process algebra, the trace aligns with our operational semantics for the system. This gives us confidence that Verlixir is correctly identifying deadlocks in the system.

6.1.5 Raft Leader Election

The final algorithm we will use in the evaluation of Verlixir's expressiveness is Raft [52]. The Raft consensus algorithm was introduced to be a simpler consensus algorithm than Paxos. It is a consensus algorithm for managing a replicated log. The result produced is similar to multi-Paxos and is as efficient as Paxos. Unlike Paxos, Raft explicitly coordinates through a leader.

We split the design of Raft into two components: leader election and log replication. We will primarily be focusing on leader election in this evaluation. The Raft design involves four process types: clients, followers, candidates and leaders. Clients communicate directly with leaders to propose log entries for replication. Leaders will commit these log messages and forward them onto followers. There can be multiple leaders at any moment, however, the Raft algorithm is divided into 'terms'. If multiple leaders forward log entries to a follower, it will only respect the leader elected for the highest term. If a follower does not receive a message from a leader within a certain

time frame, it will mark itself as a candidate and initiate a new leader election.

Leader election involves selecting a new term number and asking all followers to vote for the candidate. If a candidate receives a majority of votes, it marks itself as the leader for the term. The consensus algorithm is designed such that only one leader can be elected per term, in the case of a split vote then no leader is elected.

Instrumenting Raft

We first use Verlixir to determine we can reach consensus on a round of leader election. To do this, we configure our implementation by introducing a coordinator to initiate the first round of leader election. The coordinator does the following:

- Set $n\_nodes = 3$
- Set $n\_rounds = 1$
- Spawn 3 followers
- Bind a unique new term number to each follower
- Broadcast a *start_election* message to all followers
- Await an *elected* message from a leader

We instrument the program in such a way that we enforce an election to take place, and with our intuition of the Raft algorithm, as all the followers have unique term numbers, we expect one of them to be elected as the leader (the follower with the highest unique term number). With the system instrumented in such a way, the only liveness requirement we are interested in is the eventual termination of the program (considering the program will only terminate when the coordinator receives an elected message). When we run Verlixir on the instrumented program, we observe every execution is a terminating one. This gives us confidence the Raft specification is able to reach consensus on a leader.

Introducing Timeouts

Next, we slowly expand our system requirements and incrementally test correctness as we go. Instead of instrumenting the system with a coordinator to begin an election, a true Raft system will automatically begin rounds of leader election. Raft does this using timeouts. Every participant in the system is able to initiate an election if they fail to receive a message from the leader within a specified time frame. We can now introduce this behaviour into our system. The coordinator no longer broadcasts a *start_election* message; instead, the followers will rely on the timeout to initiate elections. We again run Verlixir on the updated model, and observe that the system is still able to reach consensus on a leader (as determined by termination).

Safety Requirements

To complete our consensus specification, we are interested in the following safety requirement:

- **SR**: Only a single leader is elected per term.

Although up to this point, we have shown the liveness property, that the system eventually terminates with a leader holds, this is not actually a guaranteed requirement of the system. Due to split votes, we cannot always guarantee election. What we can guarantee is that if a leader is elected, it is the only leader for that term.

We introduce the SR into our system by increasing *n_rounds* so that we can observe multiple participants being elected as leaders. We can then introduce the SR formally with LTL:

$$\text{SR} = \Box(\text{elected\_term} \neq \text{previously\_elected\_term})$$

LTLixir does not explicitly support the ($\neq$) operator, so we can rewrite the SR in LTLixir as:

```
@ltl """
!<>[](elected_term == previously_elected_term)
"""
```

This is a more implementation-specific approach. We can run the system with *n_rounds* as 2, then the variables will be set to the values of the two rounds. The LTL will hence ensure that we never always have an execution where the elected terms are equivalent.

We can finally verify the system with this property and determine whether the system is correct with respect to our specification.

6.2 Verlixir vs. Existing Work

We will now compare Verlixir to existing state-of-the-art work in verification-aware languages and modern programming language verification tools. We believe Verlixir is the first tool to support a pure message-passing model of computation, and hence much of the design has introduced novel techniques to support this. We will first discuss some of these techniques that differentiate Verlixir from existing work. We will then provide a direct comparison between Verlixir and existing tools, before discussing the future of Verlixir.

6.2.1 Difference in Approach

To our knowledge, Verlixir is the first verification-aware language to support a pure message-based model. This is a significantly different approach to existing tools, which have either ignored concurrency, used shared-memory models or communication over channels.

To support a shared-nothing model, we ensure all heap memory is kept local to processes and all data sharing is achieved explicitly through communication. We applied heuristics to bound the communication possible between processes, by modelling infinite mailboxes as finite Promela channels. A challenge with a shared-nothing model is supporting passing data structures. To support this, we introduced a technique to pass data structures between processes by storing data structures in global memory and providing processes with pointers to access and send structures through messages.

Any Elixir function can be used to spawn a new process. Elixir functions are also often highly recursive. To handle both the spawning and recursion of a function, we introduced a method to determine the nature of a function's usage at runtime and instrument the behaviour depending on the function's usage. For example, a function being spawned as a new process needs to determine a unique process identifier, whereas a function being called naturally needs access to the parent process identifier and also needs to communicate with the parent through a rendezvous channel to ensure the parent waits for the child to complete.

Elixir also uses a receive-anything pattern, where due to the language's dynamic typing and matching, determining how a message should be processed can involve peeking into the message to examine its contents. To enable verification of this, we introduced a method to process messages in a non-blocking manner, where messages can be received and parsed in a first-in-first-fireable-out (FIFFO) order.

To give a higher-level overview of where Verlixir differentiates itself from existing tools, we provide a table of comparison in table 6.1.

	Verlixir	Java PathFinder	Gomela	Dafny	Lean
Concurrency	Actor-based	Shared-memory	Channel-based	✗	✗
Propositional logic	✓	Limited	✗	✓	✓
Predicate logic	✗	✗	✗	✓	✓
Temporal logic	✓	✗	✗	✗	✗
Model checking	Spin	Built in	Spin	✗	✗
Theorem proving	✗	✗	✗	Z3	Built in
LTL	✓	✗	✗	✗	✗
Safety	✓	Deadlock	Deadlock	✗	✗
Liveness	✓	✗	✗	✗	✗
Weak Fairness	✓	✗	✗	✗	✗

Table 6.1: Comparison of Verlixir to existing tools.

6.2.2 Verlixir vs. Related Work

A large component of Verlixir is the translation of Elixir to Promela. As Verlixir is the first of its kind, we have no benchmark to evaluate the translation progress. However, we can provide insights into the translation process, results, and future work.

Model Performance

We have avoided an in-depth analysis of the performance of the models produced by the translator as this was not an initial focus of the research. However, to provide some insight into the performance of the models produced, under the Spin model checker, we can examine the results produced by the Dining Philosophers deadlock example.

- **Depth**: the depth of the state space explored is 198. Spin explores using a depth-first search.
- **Process Count**: the number of processes spawned by the model is 18. This is due to the implementation of function calls relying on process spawning. Spin only handles 255 concurrent processes, but no models we have produced so far have come close to this limit.
- **Execution Time**: the execution time of the model for Dining Philosophers sits at 0.21 seconds. The state space is being explored at a rate of 652 states per second.
- **State Vector**: the state vector is 18168 bytes. This exceeds the default state vector Spin uses (1024 bytes). Because of the available state vector compression techniques, increasing the limit by a large factor is no concern.

System simulation has been pushed to the limits. The limiting factor for simulation is the concurrent process limit of 255. For example, we can run the algorithms we evaluated, with over 100 nodes, but any higher number could begin to reach the concurrent limit.

It would not be feasible to verify systems of this size. We do not propose a process limit for verification, as it depends on each system and for the hardware Verlixir is run on. We did not perform analysis on alternative backends, so no direct comparison could be made. However, the results produced by Spin are sufficient for our purposes.

Optimisations

Verlixir was primarily designed to suit any backend. In particular, concurrent model checkers such as Spin, PAT or PRISM. Given Spin is the targetted model checker, we introduced some Spin-specific optimisations to the design in order to reduce the state space and time complexity of model checking. For example, consider how we optimised the modelling over the Elixir mailbox over multiple iterations:

- **Singular Mailbox**: the original mailbox was designed such that every process has its mailbox (like Elixir). Receiving messages involves the same approach Elixir uses. Messages were stored in a FIFO queue. To receive a message, we scan the queue pushing to a stack, until we find a message that matches the pattern. We can then take the message from the queue, and re-apply the stack.
- **Multiple Mailboxes**: we optimised this approach to reduce scanning by splitting the mailbox up such that each possible message type in the system has its own, per-process mailbox. This reduces the time required to search the state space as channel orderings are more deterministic.
- **Random read, sorted insert**: we further improve by moving from sequential select ? Promela's random select ?? operator to match messages in the mailbox. We do this to reduce the number of process interleavings that need to be explored in the state space. Finally, now we are using random selection, we can also use sorted insert !!. Because we are reading randomly, the order of messages in the mailbox is not important, so by sorting the mailboxes we reduce the state space.

Future Optimisations

There is lots of scope to further optimise the translation process. The existing framework has been designed such that this is easy to achieve. For example, we currently translate all Elixir functions to Promela functions. This incurs an overhead that may not be strictly necessary. By statically analysing the behaviour of a recursive function, we could apply heuristics to unroll the recursion and inline the function using an imperative style loop.

As alluded to earlier, there is room to replace the Spin backend with another model checker. Although we identified Spin as sufficient for our purposes, as explored in section 2.2, other model checkers support features that Spin does not. For example, probabilistic model checkers can more accurately model systems that involve randomness, such as gossip protocols. We aim for the design of the IR to be flexible enough to support swapping out backends.

6.3 Summary

This chapter has highlighted the expressiveness of Verlixir by modelling and verifying distributed algorithms that are frequently used in both industry and academia. We have shown the toolchain is capable of supporting these specifications, verifying properties over them and providing clear feedback when properties are violated. We have also compared Verlixir to existing tools, highlighting the differences in capabilities provided for various verification-aware languages. Finally, we have evaluated the translation process from Elixir to Promela, providing insights into the optimisations that have been made and the potential for future work.

Chapter 7

Conclusion

In this paper, we aimed to provide modelling and verification techniques for message-passing systems. To evaluate these techniques, we target Elixir, the actor-based, concurrent programming language. As part of our research, we introduced Verlixir, a verification-aware language for message-passing systems.

This project aimed to achieve some key objectives required for specifying and implementing distributed algorithms in a 'correct' manner. The specification language supported by Verlixir, alongside Verlixir's operational modes, moves towards achieving these objectives.

Verlixir provides a simulator capable of simulating large-scale distributed systems. These systems can be implemented in a modern programming language, and simulated in a controlled environment. This allows developers to prototype complex systems and observe their behaviour before deploying them in a production environment. Because Verlixir directly integrates with Elixir, the system can be easily transferred from simulation to production.

We also have empowered developers to verify the behaviour of systems, ensuring formal safety and liveness properties are adhered to. These properties can be specified alongside the implementation of the system, and verified without the need for a separate model. Counterexamples are produced in an Elixir-friendly format, allowing developers to debug the system and understand why a property was violated; particularly focusing on the message-passing interleaving that caused the violation.

To extend confidence, Verlixir supports modelling systems across different configurations. This ensures systems behave consistently across different environments. This is particularly useful for distributed systems that are subject to network partitions, or systems that are required to scale up or down.

7.1 Future Work

Throughout the course of research, we have identified several areas for future work. We have indicated some of these areas previously, but we will discuss them in more detail here. An obvious extension would involve supporting a larger Elixir feature set, but we will go into depth on more unique extensions.

A large inspiration for this project was the verification-aware language, Dafny. The Dafny language sets out to achieve similar claims to Verlixir, but takes a different approach. Dafny fundamentally relies on theorem proving for verification, whereas Verlixir uses model checking. In particular, Dafny transpiles to Boogie, a verification-aware intermediate language. Boogie currently then uses Z3 for theorem proving. Theorem proving provides formal proofs of correctness, which give stronger claims than the pre- and post-condition checks we assert in model checking. We believe an ideal solution to verify message-passing systems could involve a combination of both theorem proving and model checking. We could extend Verlixir to support theorem proving within Elixir processes, and then continue to use model checking for verifying inter-process communication. This would

provide a stronger guarantee of correctness for the system as a whole.

We explored the differences between Verlixir and other verification-aware languages. A key insight we drew from this is the lack of capability for injecting faults into models. Many distributed algorithms have been designed with an approach such that given there are no more than n faults, the system will still follow the specification. In order to truly scrutinise these algorithms, we need the capability to inject faults into the system. A simple extension to Verlixir could involve injecting random terminating faults into processes, and then observing the system's fault tolerance under these conditions.

A second insight gathered from comparing tools is the lack of support for verifying computation tree logic (CTL) in modern programming languages. CTL is a temporal logic that allows us to reason about paths in a system. A simple example of where CTL could be useful is in verifying the Paxos algorithm. To ensure fairness between all proposers, we could specify and verify a CTL property that states: there exists a path where each proposal is accepted. We believe that extending Verlixir to support CTL would provide a more comprehensive verification framework for message-passing systems.

7.2 Ethical Considerations

Throughout the report, refer to critical distributed systems used in the contexts of air traffic control and healthcare. While the tools and research discussed are designed to improve the reliability of these systems, relying on them in isolation is not sufficient. Testing distributed systems is hard, and it is important that rigorous testing is performed by multiple parties using multiple techniques. That said, we believe Verlixir provides an advancement in the verification of distributed systems.

7.3 Final Remarks

The goal of this project was to design a tool that could verify real-world distributed algorithms which are used in research and industry. Throughout researching Verlixir, many notional examples were used to demonstrate the capabilities of the tool. Through the evaluation, we have shown these capabilities extend to real, well-known algorithms such as Paxos and Raft. In achieving this, we believe Verlixir has the potential to be a valuable tool for verifying distributed systems.

Historically, model checking has required hand-translation of code into a model. Verlixir lowers the barrier to entry for verifying systems. If a programmer can write their implementation using the provided LTLixir set, the system verification comes for free. We believe moving towards a world where verification-aware languages become the standard, will greatly improve the reliability of code. Instead of programmers thinking about how to implement a system, they can focus on what the system should do. A shift in mindset away from implementation specifics, towards reasoning about the safety and liveness of a system means that we can describe our systems more holistically.

With the recent rise in artificially intelligent copilots [58, 59, 60], the future of programming could move towards a more declarative style. We programmers could simply be burdened to write the system specification, in terms of safety and liveness, and a large-language model could generate the implementation.

Bibliography

[1] Communicating Sequential Processes Available from: http://www.usingcsp.com/cspbook.pdf.

[2] Process Analysis Toolkit (PAT) 3.5 User Manual Available from: https://pat.comp.nus.edu.sg/wp-source/resources/OnlineHelp/pdf/Help.pdf.

[3] The software model checker BLAST Available from: https://www.sosy-lab.org/research/pub/2007-STTT.The_Software_Model_Checker_BLAST.pdf.

[4] PRISM Model Checker Available from: https://www.prismmodelchecker.org/.

[5] Lamport L. The temporal logic of actions. ACM Transactions on Programming Languages and Systems (TOPLAS). 1994 May 1;16(3):872-923. Available from: https://lamport.azurewebsites.net/pubs/lamport-actions.pdf.

[6] Lamport L. Specifying concurrent systems with TLA+. Calculational System Design. 1999 Apr 23:183-247.Available from: https://lamport.azurewebsites.net/tla/xmxx01-06-27.pdf.

[7] Yu Y, Manolios P, Lamport L. Model checking TLA+ specifications. InAdvanced Research Working Conference on Correct Hardware Design and Verification Methods 1999 Sep 27 (pp. 54-66). Berlin, Heidelberg: Springer Berlin Heidelberg.Available from: https://lamport.azurewebsites.net/pubs/yuanyu-model-checking.pdf.

[8] Lamport L. The PlusCal algorithm language. InInternational Colloquium on Theoretical Aspects of Computing 2009 Aug 16 (pp. 36-60). Berlin, Heidelberg: Springer Berlin Heidelberg.Available from: https://lamport.azurewebsites.net/pubs/pluscal.pdf.

[9] The Model Checker SPIN Available from: https://spinroot.com/spin/Doc/ieee97.pdf.

[10] Build simple, secure, scalable systems with Go Available from: https://go.dev/.

[11] Elixir is a dynamic, functional language for building scalable and maintainable applications. Available from: https://elixir-lang.org/.

[12] A brief introduction to BEAM Available from: https://www.erlang.org/blog/a-brief-beam-primer/.

[13] Practical functional programming for a parallel world Available from: https://www.erlang.org/.

[14] Phoenix Peace of mind from prototype to production Available from: https://phoenixframework.org/.

[15] Discord Available from: https://discord.com/.

[16] Financial Times Available from: https://www.ft.com/.

[17] Mediero Iturrioz J. Verification of Concurrent Programs in Dafny. Available from: https://addi.ehu.es/bitstream/handle/10810/23803/Report.pdf?isAllowed=y&sequence=2.

[18] The Dafny Programming and Verification Language Available from: dafny.org

[19] Elixir, Macros, Our First Macro Available from: https://hexdocs.pm/elixir/macros.html#our-first-macro.

[20] De Moura L, Bjørner N. Z3: An efficient SMT solver. InInternational conference on Tools and Algorithms for the Construction and Analysis of Systems 2008 Mar 29 (pp. 337-340). Berlin, Heidelberg: Springer Berlin Heidelberg. Available from: https://link.springer.com/content/pdf/10.1007/978-3-540-78800-3_24.pdf.

[21] Hoare CA. An axiomatic basis for computer programming. Communications of the ACM. 1969 Oct 1;12(10):576-80. Available from: https://dl.acm.org/doi/10.1145/363235.363259

[22] Lean and its Mathematical library Available from: https://leanprover-community.github.io/.

[23] dialyzer Available from: https://www.erlang.org/doc/man/dialyzer.html.

[24] Leino KR. Dafny: An automatic program verifier for functional correctness. InInternational conference on logic for programming artificial intelligence and reasoning 2010 Apr 25 (pp. 348-370). Berlin, Heidelberg: Springer Berlin Heidelberg. Available from: https://link.springer.com/chapter/10.1007/978-3-642-17511-4_20.

[25] Nipkow T. Getting started with Dafny: A guide. Software Safety and Security: Tools for Analysis and Verification. 2012;33:152. Available from: https://dafny.org/dafny/OnlineTutorial/guide.

[26] Barnett M, Chang BY, DeLine R, Jacobs B, Leino KR. Boogie: A modular reusable verifier for object-oriented programs. InFormal Methods for Components and Objects: 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures 4 2006 (pp. 364-387). Springer Berlin Heidelberg. Available from: https://link.springer.com/chapter/10.1007/11804192_17.

[27] Hoare CA. Communicating sequential processes. Englewood Cliffs: Prentice-hall; 1985 Jan.

[28] Lynch NA. Distributed algorithms. Elsevier; 1996 Apr 16. Available from: https://lib.fbtuit.uz/assets/files/5.-NancyA.Lynch.DistributedAlgorithms.pdf.

[29] Clarke EM. Model checking. InFoundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18-20, 1997 Proceedings 17 1997 (pp. 54-56). Springer Berlin Heidelberg.

[30] Agha G. Actors: a model of concurrent computation in distributed systems. MIT press; 1986 Dec 17.

[31] Available from: https://hexdocs.pm/elixir/processes.html#links

[32] Herlihy, M. & Shavit, N. (2008), The art of multiprocessor programming. , Morgan Kaufmann. Available from: https://cs.ipm.ac.ir/asoc2016/Resources/Theartofmulticore.pdf

[33] Lamport, 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE transactions on computers, 100(9), pp.690-691.

[34] Kripke, S.A., 1980. Naming and necessity. Harvard University Press.

[35] Emerson, E.A., 1990. Temporal and modal logic. In Formal Models and Semantics (pp. 995-1072). Elsevier.

[36] Baier, C. and Katoen, J.P., 2008. Principles of model checking. MIT press.

[37] Huth, M. and Ryan, M., 2004. Logic in Computer Science: Modelling and reasoning about systems. Cambridge university press.

[38] Alur, R., Henzinger, T.A. and Kupferman, O., 2002. Alternating-time temporal logic. Journal of the ACM (JACM), 49(5), pp.672-713.

[39] Smith, P.J., Spencer, A.L. and Billings, C.E., 2007. Strategies for designing distributed systems: case studies in the design of an air traffic management system. Cognition, Technology & Work, 9, pp.39-49.

[40] Sarkar, B.K. and Sana, S.S., 2020. A conceptual distributed framework for improved and secured healthcare system. International Journal of Healthcare Management, 13(sup1), pp.74-87.

[41] Lamport, L., 2001. Paxos made simple. ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001), pp.51-58.

[42] Marzullo, K., Mei, A. and Meling, H., 2013. A simpler proof for paxos and fast paxos. Course notes.

[43] Jiang, K., 2009. Model checking c programs by translating c to promela.

[44] Dilley, N. and Lange, J., 2020. Bounded verification of message-passing concurrency in Go using Promela and Spin. arXiv preprint arXiv:2004.01323.

[45] Tabone, G. and Francalanza, A., 2022. Session Fidelity for ElixirST: A Session-Based Type System for Elixir Modules. arXiv preprint arXiv:2208.04631.

[46] Hebert, F., 2019. Property-Based Testing with PropEr, Erlang, and Elixir: Find Bugs Before Your Users Do.

[47] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W., 2007. Dynamo: Amazon's highly available key-value store. ACM SIGOPS operating systems review, 41(6), pp.205-220.

[48] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M. and Lewin, D., 1997, May. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In Proceedings of the twenty-ninth annual ACM symposium on Theory of computing (pp. 654-663).

[49] Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J.J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P. and Hsieh, W., 2013. Spanner: Google's globally distributed database. ACM Transactions on Computer Systems (TOCS), 31(3), pp.1-22.

[50] The Concise Promela Reference. Available from: https://spinroot.com/spin/Man/Quick.html.

[51] Leino, K.R.M., 2018. Modeling concurrency in Dafny. In Engineering Trustworthy Software Systems: Third International School, SETSS 2017, Chongqing, China, April 17–22, 2017, Tutorial Lectures 3 (pp. 115-142). Springer International Publishing.

[52] Ongaro, D. and Ousterhout, J., 2014. In search of an understandable consensus algorithm. In 2014 USENIX annual technical conference (USENIX ATC 14) (pp. 305-319).

[53] Havelund, K. and Pressburger, T., 2000. Model checking java programs using java pathfinder. International Journal on Software Tools for Technology Transfer, 2, pp.366-381.

[54] Chandra, T.D., Griesemer, R. and Redstone, J., 2007, August. Paxos made live: an engineering perspective. In Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing (pp. 398-407).

[55] Schwartz, R.L. and Melliar-Smith, P.M., 1981, April. Temporal logic specification of distributed systems. In ICDCS (pp. 446-454).

[56] Lehmann, D. and Rabin, M.O., 1981, January. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (pp. 133-138).

[57] Havelund, K. and Pressburger, T., 2000. Model checking java programs using java pathfinder. International Journal on Software Tools for Technology Transfer, 2, pp.366-381.

[58] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł. and Polosukhin, I., 2017. Attention is all you need. Advances in neural information processing systems, 30.

[59] Dakhel, A.M., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M.C. and Jiang, Z.M.J., 2023. Github copilot ai pair programmer: Asset or liability?. Journal of Systems and Software, 203, p.111734.

[60] Orenes-Vera, M., Martonosi, M. and Wentzlaff, D., 2023. From RTL to SVA: LLM-assisted generation of Formal Verification Testbenches. arXiv preprint arXiv:2309.09437.

[61] Liu, Z., Zhu, S., Qin, B., Chen, H. and Song, L., 2021, April. Automatically detecting and fixing concurrency bugs in go software systems. In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (pp. 616-629).

[62] Gabet, J. and Yoshida, N., 2020. Static race detection and mutex safety and liveness for go programs. In 34th European Conference on Object-Oriented Programming (ECOOP 2020). Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[63] Hewitt, C., Bishop, P. and Steiger, R., 1973. Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence. In Advance papers of the conference (Vol. 3, p. 235). Menlo Park, CA: Stanford Research Institute.

[64] Jazequel, J.M. and Meyer, B., 1997. Design by contract: The lessons of Ariane. Computer, 30(1), pp.129-130.

[65] Eiffel. Available from: https://www.eiffel.org/doc/eiffel/Learning_Eiffel.

[66] Huth, M. and Ryan, M., 2004. Logic in Computer Science: Modelling and reasoning about systems. Cambridge university press.

[67] Duflot, M., Kwiatkowska, M., Norman, G. and Parker, D., 2006. A formal analysis of Bluetooth device discovery. International journal on software tools for technology transfer, 8, pp.621-632.

[68] Cachin, C., Guerraoui, R. and Rodrigues, L., 2011. Introduction to reliable and secure distributed programming. Springer Science & Business Media.

[69] Lamport, L., A Science of Concurrent Programs.

Appendix A

Full Code Listings

A.1 Verlixir Example

```
1  import VaeLib
2
3  defmodule Server do
4    @init true
5    @ltl "(q)U([]p)"
6    @spec start_server() :: :ok
7    @model {:number_of_rounds}
8    def start_server do
9      predicate p, alive_clients == client_n * number_of_rounds
10     predicate q, !p
11     client_n = 1
12     number_of_rounds = 2
13     alive_clients = 0
14     for _ <- 1..client_n do
15       client = spawn(Client, :start_client, [])
16       send(client, {:bind, self(), number_of_rounds})
17     end
18     alive_clients = if number_of_rounds > 1 do
19       check_clients(client_n * number_of_rounds, alive_clients)
20     else
21       0
22     end
23   end
24
25   @spec check_clients(integer(), integer()) :: integer()
26   def check_clients(expected_clients, current_clients) do
27     if expected_clients == current_clients do
28       current_clients
29     else
30       receive do
31         {:im_alive} -> check_clients(expected_clients, current_clients
               + 1)
32       end
33     end
34   end
35 end
36
37 defmodule Client do
38   @spec start_client() :: :ok
39   def start_client do
```

```
    {server, rounds} = receive do
      {:bind, sender, round_limit} -> {sender, round_limit}
    end
    next_round(server, rounds)
  end

  @spec next_round(pid(), integer()) :: :ok
  defv next_round(server, rounds), pre: rounds >= 0 do
    if rounds == 0 do
      :ok
    else
      send(server, {:im_alive})
      next_round(server, rounds - 1)
    end
  end
end
```

A.2 Paxos

A.2.1 First paxos implementation with a bug

```
import VaeLib

defmodule Acceptor do

  @spec start_acceptor() :: :ok
  def start_acceptor do
    acceptedProposal = -1
    acceptedValue = -1
    minProposal = -1
    accept_handler(acceptedProposal, acceptedValue, minProposal)
  end

  @spec accept_handler(integer(), integer(), integer()) :: :ok
  def accept_handler(acceptedProposal, acceptedValue, minProposal) do
    receive do
      {:prepare, n, proposer} ->
        if n > minProposal do
          send proposer, {:promise, acceptedProposal, acceptedValue}
          accept_handler(acceptedProposal, acceptedValue, n)
        else
          send proposer, {:promise, acceptedProposal, acceptedValue}
          accept_handler(acceptedProposal, acceptedValue, minProposal)
        end
      {:accept, n, value, proposer} ->
        if n >= minProposal do
          send proposer, {:accepted, n}
          accept_handler(n, value, n)
        else
          send proposer, {:accepted, minProposal}
          accept_handler(acceptedProposal, acceptedValue, minProposal)
        end
      {:terminate} ->
        IO.puts("Terminating acceptor")
    end
  end
```

```
36  end
37
38  defmodule Proposer do
39    @spec start_proposer() :: :ok
40    def start_proposer do
41      receive do
42        {:bind, acceptors, proposal_n, value, maj, learner} ->
              proposer_handler(acceptors, proposal_n, value, maj, learner)
43      end
44    end
45
46    @spec proposer_handler(list(), integer(), integer(), integer(),
          integer()) :: :ok
47    def proposer_handler(acceptors, proposal_n, value, maj, learner) do
48      for acceptor <- acceptors do
49        send acceptor, {:prepare, proposal_n, self()}
50      end
51
52      receive_prepared(proposal_n, value, maj, 0, 0)
53      {prepared_n, prepared_value} = receive do
54        {:majority_prepared, n, v} -> {n, v}
55      end
56
57      for acceptor <- acceptors do
58        send acceptor, {:accept, prepared_n, prepared_value, self()}
59      end
60
61      accepted_n = receive_accepted(maj, prepared_n, 0, 0)
62
63      if accepted_n != -1 do
64        # Value chosen
65        send learner, {:learned, prepared_value}
66      else
67        # Value was rejected
68        send learner, {:learned, 0}
69      end
70    end
71
72    @spec receive_prepared(integer(), integer(), integer(), integer(),
          integer()) :: :ok
73    def receive_prepared(proposal_n, value, maj, highest_seen_proposal,
        count) do
74      if count >= maj do
75        send self(), {:majority_prepared, proposal_n, value}
76      else
77        receive do
78          {:promise, acceptedProposal, acceptedValue} ->
79            if acceptedValue != -1 && acceptedProposal >
                  highest_seen_proposal do
80              receive_prepared(proposal_n, acceptedValue, maj,
                    acceptedProposal, count + 1)
81            else
82              receive_prepared(proposal_n, value, maj,
                    highest_seen_proposal, count + 1)
83            end
84        end
85      end
86    end
```

```
87
88    @spec receive_accepted(integer(), integer(), integer(), integer()) ::
          integer()
89    def receive_accepted(maj, prepared_n, rejections, count) do
90      if count >= maj do
91        if rejections >= maj do # BUG IS HERE
92          -1
93        else
94          prepared_n
95        end
96      else
97        receive do
98          {:accepted, n} ->
99            if n > prepared_n do
100             receive_accepted(maj, prepared_n, rejections + 1, count + 1)
101           else
102             receive_accepted(maj, prepared_n, rejections, count + 1)
103           end
104       end
105     end
106   end
107 end
108
109 defmodule Learner do
110
111   @spec start() :: :ok
112   @init true
113   def start do
114     n_acceptors = 3
115     quorum = 2
116     n_proposers = 2
117     vals = [42, 31]
118     acceptors = for _ <- 1..n_acceptors do
119       spawn(Acceptor, :start_acceptor, [])
120     end
121
122     for i <- 1..n_proposers do
123       proposer = spawn(Proposer, :start_proposer, [])
124       val_i = i - 1
125       val = Enum.at(vals, val_i)
126       send proposer, {:bind, acceptors, i, val, quorum, self()}
127     end
128     wait_learned(acceptors, n_proposers, 0)
129   end
130
131   @spec wait_learned(list(), integer(), integer()) :: :ok
132   @ltl """
133   [](p->!<>q) && (q->!<>p))
134   <>(r)
135   [](s)
136   """
137   def wait_learned(acceptors, p_n, learned_n) do
138     predicate p, final_value == 31
139     predicate q, final_value == 42
140     predicate r, final_value != 0
141     predicate s, final_value == 0 || final_value == 31 || final_value ==
          42
142
```

```
143       if p_n == learned_n do
144         for acceptor <- acceptors do
145           send acceptor, {:terminate}
146         end
147       else
148         receive do
149           {:learned, final_value} ->
150             IO.puts("Learned final_value:")
151             IO.puts(final_value)
152         end
153         wait_learned(acceptors, p_n, learned_n + 1)
154       end
155     end
156   end
```

A.2.2 First paxos bug message log

```
1     Never claim moves to line 6  [(1)]
2   138:   proc 7 (start_proposer:1) test_out.pml:314 Recv
      7,BIND,0,0,2,0,0,0,0,0,1,0,0,0,0,0,42,0,0,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 <-
      queue 20 (__BIND)
3   154:   proc 8 (proposer_handler:1) test_out.pml:350 Send
      1,PREPARE,0,0,1,0,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
      -> queue 23 (__PREPARE)
4   158:   proc 8 (proposer_handler:1) test_out.pml:350 Send
      3,PREPARE,0,0,1,0,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
      -> queue 23 (__PREPARE)
5   162:   proc 8 (proposer_handler:1) test_out.pml:350 Send
      5,PREPARE,0,0,1,0,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
      -> queue 23 (__PREPARE)
6   197:   proc 6 (accept_handler:1) test_out.pml:262 Recv
      5,PREPARE,0,0,1,0,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
      <- queue 23 (__PREPARE)
7   203:   proc 6 (accept_handler:1) test_out.pml:272 Send
      7,PROMISE,0,0,-1,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
      -> queue 26 (__PROMISE)
8   205:   proc 9 (receive_prepared:1) test_out.pml:425 Recv
      7,PROMISE,0,0,-1,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
      <- queue 26 (__PROMISE)
9   219:   proc 2 (accept_handler:1) test_out.pml:262 Recv
      1,PREPARE,0,0,1,0,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
      <- queue 23 (__PREPARE)
10  262:   proc 4 (accept_handler:1) test_out.pml:262 Recv
      3,PREPARE,0,0,1,0,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
      <- queue 23 (__PREPARE)
11  278:   proc 0 (:init::1) test_out.pml:521 Send
      10,BIND,0,0,3,0,0,0,0,0,2,0,0,0,0,0,31,0,0,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ->
      queue 20 (__BIND)
12  292:   proc 4 (accept_handler:1) test_out.pml:272 Send
      7,PROMISE,0,0,-1,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
      -> queue 26 (__PROMISE)
13  300:   proc 11 (receive_prepared:1) test_out.pml:425 Recv
      7,PROMISE,0,0,-1,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
      <- queue 26 (__PROMISE)
14  344:   proc 2 (accept_handler:1) test_out.pml:272 Send
      7,PROMISE,0,0,-1,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
      -> queue 26 (__PROMISE)
15  346:   proc 10 (start_proposer:1) test_out.pml:314 Recv
      10,BIND,0,0,3,0,0,0,0,0,2,0,0,0,0,0,31,0,0,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 <-
      queue 20 (__BIND)
```

```
370:   proc 14 (receive_prepared:1) test_out.pml:421 Send
    7,MAJORITY_PREPARED,0,0,1,0,0,0,0,0,42,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 46 (__MAJORITY_PREPARED)
372:   proc 8 (proposer_handler:1) test_out.pml:363 Recv
    7,MAJORITY_PREPARED,0,0,1,0,0,0,0,0,42,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 46 (__MAJORITY_PREPARED)
382:   proc 8 (proposer_handler:1) test_out.pml:386 Send
    1,ACCEPT,0,0,1,0,0,0,0,0,42,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 47 (__ACCEPT)
386:   proc 8 (proposer_handler:1) test_out.pml:386 Send
    3,ACCEPT,0,0,1,0,0,0,0,0,42,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 47 (__ACCEPT)
390:   proc 8 (proposer_handler:1) test_out.pml:386 Send
    5,ACCEPT,0,0,1,0,0,0,0,0,42,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 47 (__ACCEPT)
421:   proc 15 (proposer_handler:1) test_out.pml:350 Send
    1,PREPARE,0,0,2,0,0,0,0,0,10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 23 (__PREPARE)
425:   proc 15 (proposer_handler:1) test_out.pml:350 Send
    3,PREPARE,0,0,2,0,0,0,0,0,10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 23 (__PREPARE)
429:   proc 15 (proposer_handler:1) test_out.pml:350 Send
    5,PREPARE,0,0,2,0,0,0,0,0,10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 23 (__PREPARE)
452:   proc 16 (accept_handler:1) test_out.pml:262 Recv
    1,PREPARE,0,0,2,0,0,0,0,0,10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 23 (__PREPARE)
458:   proc 16 (accept_handler:1) test_out.pml:272 Send
    10,PROMISE,0,0,-1,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 26 (__PROMISE)
460:   proc 12 (accept_handler:1) test_out.pml:282 Recv
    5,ACCEPT,0,0,1,0,0,0,0,0,42,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 47 (__ACCEPT)
466:   proc 12 (accept_handler:1) test_out.pml:291 Send
    7,ACCEPTED,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 48 (__ACCEPTED)
474:   proc 17 (accept_handler:1) test_out.pml:282 Recv
    1,ACCEPT,0,0,1,0,0,0,0,0,42,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 47 (__ACCEPT)
492:   proc 17 (accept_handler:1) test_out.pml:296 Send
    7,ACCEPTED,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 48 (__ACCEPTED)
494:   proc 13 (accept_handler:1) test_out.pml:262 Recv
    3,PREPARE,0,0,2,0,0,0,0,0,10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 23 (__PREPARE)
500:   proc 13 (accept_handler:1) test_out.pml:272 Send
    10,PROMISE,0,0,-1,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 26 (__PROMISE)
512:   proc 18 (receive_accepted:1) test_out.pml:459 Recv
    7,ACCEPTED,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 48 (__ACCEPTED)
542:   proc 21 (accept_handler:1) test_out.pml:262 Recv
    5,PREPARE,0,0,2,0,0,0,0,0,10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 23 (__PREPARE)
554:   proc 20 (receive_prepared:1) test_out.pml:425 Recv
    10,PROMISE,0,0,-1,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 26 (__PROMISE)
560:   proc 23 (accept_handler:1) test_out.pml:282 Recv
    3,ACCEPT,0,0,1,0,0,0,0,0,42,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 47 (__ACCEPT)
578:   proc 23 (accept_handler:1) test_out.pml:296 Send
    7,ACCEPTED,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 48 (__ACCEPTED)
```

```
37 580:   proc 21 (accept_handler:1) test_out.pml:272 Send
       10,PROMISE,0,0,1,0,0,0,0,0,42,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
       -> queue 26 (__PROMISE)
38 582:   proc 24 (receive_prepared:1) test_out.pml:425 Recv
       10,PROMISE,0,0,-1,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
       <- queue 26 (__PROMISE)
39 602:   proc 25 (receive_prepared:1) test_out.pml:421 Send
       10,MAJORITY_PREPARED,0,0,2,0,0,0,0,0,31,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
       -> queue 46 (__MAJORITY_PREPARED)
40 616:   proc 15 (proposer_handler:1) test_out.pml:363 Recv
       10,MAJORITY_PREPARED,0,0,2,0,0,0,0,0,31,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
       <- queue 46 (__MAJORITY_PREPARED)
41 626:   proc 15 (proposer_handler:1) test_out.pml:386 Send
       1,ACCEPT,0,0,2,0,0,0,0,0,31,0,0,0,0,0,10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
       -> queue 47 (__ACCEPT)
42 630:   proc 15 (proposer_handler:1) test_out.pml:386 Send
       3,ACCEPT,0,0,2,0,0,0,0,0,31,0,0,0,0,0,10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
       -> queue 47 (__ACCEPT)
43 634:   proc 15 (proposer_handler:1) test_out.pml:386 Send
       5,ACCEPT,0,0,2,0,0,0,0,0,31,0,0,0,0,0,10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
       -> queue 47 (__ACCEPT)
44 659:   proc 27 (receive_accepted:1) test_out.pml:459 Recv
       7,ACCEPTED,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
       <- queue 48 (__ACCEPTED)
45 681:   proc 29 (accept_handler:1) test_out.pml:282 Recv
       5,ACCEPT,0,0,2,0,0,0,0,0,31,0,0,0,0,0,10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
       <- queue 47 (__ACCEPT)
46 699:   proc 29 (accept_handler:1) test_out.pml:291 Send
       10,ACCEPTED,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
       -> queue 48 (__ACCEPTED)
47 701:   proc 28 (receive_accepted:1) test_out.pml:459 Recv
       10,ACCEPTED,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
       <- queue 48 (__ACCEPTED)
48 707:   proc 30 (receive_accepted:1) test_out.pml:454 Send 1 -> queue 78 (ret)
49 711:   proc 22 (accept_handler:1) test_out.pml:282 Recv
       1,ACCEPT,0,0,2,0,0,0,0,0,31,0,0,0,0,0,10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
       <- queue 47 (__ACCEPT)
50 717:   proc 27 (receive_accepted:1) test_out.pml:466 Recv 1 <- queue 78 (ret1)
51 719:   proc 27 (receive_accepted:1) test_out.pml:467 Send 1 -> queue 54 (ret)
52 721:   proc 18 (receive_accepted:1) test_out.pml:471 Recv 1 <- queue 54 (ret2)
53 723:   proc 18 (receive_accepted:1) test_out.pml:472 Sent 1 -> queue 22 (ret)
54 724:   proc  8 (proposer_handler:1) test_out.pml:396 Recv 1 <- queue 22 (ret2)
55 740:   proc 22 (accept_handler:1) test_out.pml:291 Send
       10,ACCEPTED,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
       -> queue 48 (__ACCEPTED)
56 742:   proc 30 (receive_accepted:1) test_out.pml:459 Recv
       10,ACCEPTED,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
       <- queue 48 (__ACCEPTED)
57 748:   proc  8 (proposer_handler:1) test_out.pml:401 Send
       0,LEARNED,0,0,42,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
       -> queue 90 (__LEARNED)
58 762:   proc 31 (receive_accepted:1) test_out.pml:454 Send 2 -> queue 89 (ret)
59 766:   proc 30 (receive_accepted:1) test_out.pml:471 Recv 2 <- queue 89 (ret2)
60 768:   proc 30 (receive_accepted:1) test_out.pml:472 Send 2 -> queue 81 (ret)
61 772:   proc 28 (receive_accepted:1) test_out.pml:471 Recv 2 <- queue 81 (ret2)
62 774:   proc 28 (receive_accepted:1) test_out.pml:472 Sent 2 -> queue 41 (ret)
63 775:   proc 15 (proposer_handler:1) test_out.pml:396 Recv 2 <- queue 41 (ret2)
64 791:   proc 19 (wait_learned:1) test_out.pml:558 Recv
       0,LEARNED,0,0,42,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
       <- queue 90 (__LEARNED)
65 Never claim moves to line 5  [(!(!((final_value==42))))]
66 Never claim moves to line 16 [(1)]
67 807:   proc 26 (accept_handler:1) test_out.pml:282 Recv
```

```
    3,ACCEPT,0,0,2,0,0,0,0,0,31,0,0,0,0,0,10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 47 (__ACCEPT)
813:   proc 26 (accept_handler:1) test_out.pml:291 Send
    10,ACCEPTED,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 48 (__ACCEPTED)
815:   proc 15 (proposer_handler:1) test_out.pml:401 Send
    0,LEARNED,0,0,31,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 90 (__LEARNED)
823:   proc 32 (wait_learned:1) test_out.pml:558 Recv
    0,LEARNED,0,0,31,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 90 (__LEARNED)
spin: _spin_nvr.tmp:15, Error: assertion violated
spin: text of failed assertion: assert(!((final_value==31)))
Never claim moves to line 15 [assert(!((final_value==31)))]
spin: trail ends after 826 steps
```

Listing A.1: Message passing caused by the proposer's protocol bug.

A.2.3 Second paxos implementation with a bug

```elixir
import VaeLib

defmodule Acceptor do

  @spec start_acceptor() :: :ok
  def start_acceptor do
    acceptedProposal = -1
    acceptedValue = -1
    minProposal = -1
    accept_handler(acceptedProposal, acceptedValue, minProposal)
  end

  @spec accept_handler(integer(), integer(), integer()) :: :ok
  def accept_handler(acceptedProposal, acceptedValue, minProposal) do
    receive do
      {:prepare, n, proposer} ->
        if n > minProposal do
          send proposer, {:promise, acceptedProposal, acceptedValue}
          accept_handler(acceptedProposal, acceptedValue, n)
        else
          send proposer, {:promise, acceptedProposal, acceptedValue}
          accept_handler(acceptedProposal, acceptedValue, minProposal)
        end
      {:accept, n, value, proposer} ->
        if n >= minProposal do
          send proposer, {:accepted, n}
          accept_handler(n, value, n)
        else
          send proposer, {:accepted, minProposal}
          accept_handler(acceptedProposal, acceptedValue, minProposal)
        end
      {:terminate} ->
        IO.puts("Terminating acceptor")
    end
  end
end

defmodule Proposer do
```

```
39      @spec start_proposer() :: :ok
40      def start_proposer do
41        receive do
42          {:bind, acceptors, proposal_n, value, maj, learner} ->
                proposer_handler(acceptors, proposal_n, value, maj, learner)
43        end
44      end
45
46      @spec proposer_handler(list(), integer(), integer(), integer(),
            integer()) :: :ok
47      def proposer_handler(acceptors, proposal_n, value, maj, learner) do
48        for acceptor <- acceptors do
49          send acceptor, {:prepare, proposal_n, self()}
50        end
51
52        receive_prepared(proposal_n, value, maj, 0, 0)
53        {prepared_n, prepared_value} = receive do
54          {:majority_prepared, n, v} -> {n, v}
55        end
56
57        for acceptor <- acceptors do
58          send acceptor, {:accept, prepared_n, prepared_value, self()}
59        end
60
61        accepted_n = receive_accepted(maj, prepared_n, 0, 0)
62
63        if accepted_n != -1 do
64          # Value chosen
65          send learner, {:learned, prepared_value}
66        else
67          # Value was rejected
68          send learner, {:learned, 0}
69        end
70      end
71
72      @spec receive_prepared(integer(), integer(), integer(), integer(),
            integer()) :: :ok
73      def receive_prepared(proposal_n, value, maj,
    highest_seen_proposal, count) do
74        if count >= maj do
75          send self(), {:majority_prepared, proposal_n, value}
76        else
77          receive do
78            {:promise, acceptedProposal, acceptedValue} ->
79              if acceptedProposal > highest_seen_proposal do
80                receive_prepared(acceptedProposal, acceptedValue, maj,
                      acceptedProposal, count + 1) # BUG IS HERE
81              else
82                receive_prepared(proposal_n, value, maj,
                      highest_seen_proposal, count + 1)
83              end
84          end
85        end
86      end
87
88      @spec receive_accepted(integer(), integer(), integer(), integer())
            :: integer()
89      def receive_accepted(maj, prepared_n, rejections, count) do
```

```
90        if count >= maj do
91          if rejections >= 1 do
92            -1
93          else
94            prepared_n
95          end
96        else
97          receive do
98            {:accepted, n} ->
99              if n > prepared_n do
100               receive_accepted(maj, prepared_n, rejections + 1, count +
                      1)
101             else
102               receive_accepted(maj, prepared_n, rejections, count + 1)
103             end
104         end
105       end
106     end
107   end
108
109   defmodule Learner do
110
111     @spec start() :: :ok
112     @init true
113     def start do
114       n_acceptors = 3
115       quorum = 2
116       n_proposers = 2
117       vals = [42, 31]
118       acceptors = for _ <- 1..n_acceptors do
119         spawn(Acceptor, :start_acceptor, [])
120       end
121
122       for i <- 1..n_proposers do
123         proposer = spawn(Proposer, :start_proposer, [])
124         val_i = i - 1
125         val = Enum.at(vals, val_i)
126         send proposer, {:bind, acceptors, i, val, quorum, self()}
127       end
128       wait_learned(acceptors, n_proposers, 0)
129     end
130
131     @spec wait_learned(list(), integer(), integer()) :: :ok
132     @ltl "[]((p->!<>q) && (q->!<>p))"
133     @ltl "<>(r)"
134     @ltl "[](s)"
135     def wait_learned(acceptors, p_n, learned_n) do
136       if p_n == learned_n do
137         for acceptor <- acceptors do
138           send acceptor, {:terminate}
139         end
140       else
141         receive do
142           {:learned, final_value} ->
143             predicate p, final_value == 31
144             predicate q, final_value == 42
145             predicate r, final_value != 0
146             predicate s, final_value == 0 || final_value == 31 ||
```

```
                    final_value == 42
147                 IO.puts("Learned final_value:")
148                 IO.puts(final_value)
149               end
150               wait_learned(acceptors, p_n, learned_n + 1)
151             end
152           end
153         end
```

A.2.4 Second paxos bug message log

```
1   ltl ltl_1: [] ((((! ((final_value==31))) || (! (<> ((final_value==42))))) && ((!
        ((final_value==42))) || (! (<> ((final_value==31))))))
2   ltl ltl_2: <> ((final_value!=0))
3   ltl ltl_3: [] ((((final_value==0)) || ((final_value==31))) || ((final_value==42)))
4   starting claim 8
5   Never claim moves to line 6  [(1)]
6   132:   proc  0 (:init::1) test_out.pml:521 Send
        7,BIND,0,0,2,0,0,0,0,0,1,0,0,0,0,0,42,0,0,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        -> queue 20 (__BIND)
7   138:   proc  7 (start_proposer:1) test_out.pml:314 Recv
        7,BIND,0,0,2,0,0,0,0,0,1,0,0,0,0,0,42,0,0,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        <- queue 20 (__BIND)
8   154:   proc  8 (proposer_handler:1) test_out.pml:350 Send
        1,PREPARE,0,0,1,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        -> queue 23 (__PREPARE)
9   158:   proc  8 (proposer_handler:1) test_out.pml:350 Send
        3,PREPARE,0,0,1,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        -> queue 23 (__PREPARE)
10  162:   proc  8 (proposer_handler:1) test_out.pml:350 Send
        5,PREPARE,0,0,1,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        -> queue 23 (__PREPARE)
11  197:   proc  6 (accept_handler:1) test_out.pml:262 Recv
        5,PREPARE,0,0,1,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        <- queue 23 (__PREPARE)
12  203:   proc  6 (accept_handler:1) test_out.pml:272 Send
        7,PROMISE,0,0,-1,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        -> queue 26 (__PROMISE)
13  205:   proc  9 (receive_prepared:1) test_out.pml:425 Recv
        7,PROMISE,0,0,-1,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        <- queue 26 (__PROMISE)
14  227:   proc  4 (accept_handler:1) test_out.pml:262 Recv
        3,PREPARE,0,0,1,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        <- queue 23 (__PREPARE)
15  233:   proc  4 (accept_handler:1) test_out.pml:272 Send
        7,PROMISE,0,0,-1,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        -> queue 26 (__PROMISE)
16  235:   proc 10 (receive_prepared:1) test_out.pml:425 Recv
        7,PROMISE,0,0,-1,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        <- queue 26 (__PROMISE)
17  300:   proc  0 (:init::1) test_out.pml:521 Send
        13,BIND,0,0,3,0,0,0,0,0,2,0,0,0,0,0,0,31,0,0,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        -> queue 20 (__BIND)
18  308:   proc 13 (start_proposer:1) test_out.pml:314 Recv
        13,BIND,0,0,3,0,0,0,0,0,2,0,0,0,0,0,0,31,0,0,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        <- queue 20 (__BIND)
19  324:   proc 15 (proposer_handler:1) test_out.pml:350 Send
        1,PREPARE,0,0,2,0,0,0,0,0,13,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        -> queue 23 (__PREPARE)
20  328:   proc 15 (proposer_handler:1) test_out.pml:350 Send
```

```
    3,PREPARE,0,0,2,0,0,0,0,0,13,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 23 (__PREPARE)
21  332:   proc 15 (proposer_handler:1) test_out.pml:350 Send
    5,PREPARE,0,0,2,0,0,0,0,0,13,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 23 (__PREPARE)
22  355:   proc 14 (accept_handler:1) test_out.pml:262 Recv
    3,PREPARE,0,0,2,0,0,0,0,0,13,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 23 (__PREPARE)
23  361:   proc 14 (accept_handler:1) test_out.pml:272 Send
    13,PROMISE,0,0,-1,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 26 (__PROMISE)
24  363:   proc 12 (receive_prepared:1) test_out.pml:421 Send
    7,MAJORITY_PREPARED,0,0,1,0,0,0,0,0,42,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 42 (__MAJORITY_PREPARED)
25  365:   proc  8 (proposer_handler:1) test_out.pml:363 Recv
    7,MAJORITY_PREPARED,0,0,1,0,0,0,0,0,42,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 42 (__MAJORITY_PREPARED)
26  413:   proc  8 (proposer_handler:1) test_out.pml:386 Send
    1,ACCEPT,0,0,1,0,0,0,0,0,42,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 43 (__ACCEPT)
27  417:   proc  8 (proposer_handler:1) test_out.pml:386 Send
    3,ACCEPT,0,0,1,0,0,0,0,0,42,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 43 (__ACCEPT)
28  421:   proc  8 (proposer_handler:1) test_out.pml:386 Send
    5,ACCEPT,0,0,1,0,0,0,0,0,42,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 43 (__ACCEPT)
29  458:   proc 16 (receive_prepared:1) test_out.pml:425 Recv
    13,PROMISE,0,0,-1,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 26 (__PROMISE)
30  474:   proc  2 (accept_handler:1) test_out.pml:282 Recv
    1,ACCEPT,0,0,1,0,0,0,0,0,42,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 43 (__ACCEPT)
31  486:   proc 11 (accept_handler:1) test_out.pml:262 Recv
    5,PREPARE,0,0,2,0,0,0,0,0,13,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 23 (__PREPARE)
32  492:   proc 11 (accept_handler:1) test_out.pml:272 Send
    13,PROMISE,0,0,-1,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 26 (__PROMISE)
33  510:   proc 18 (accept_handler:1) test_out.pml:282 Recv
    3,ACCEPT,0,0,1,0,0,0,0,0,42,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 43 (__ACCEPT)
34  518:   proc 18 (accept_handler:1) test_out.pml:296 Send
    7,ACCEPTED,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 58 (__ACCEPTED)
35  520:   proc 17 (receive_prepared:1) test_out.pml:425 Recv
    13,PROMISE,0,0,-1,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 26 (__PROMISE)
36  542:   proc 19 (receive_accepted:1) test_out.pml:459 Recv
    7,ACCEPTED,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 58 (__ACCEPTED)
37  548:   proc 20 (accept_handler:1) test_out.pml:282 Recv
    5,ACCEPT,0,0,1,0,0,0,0,0,42,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 43 (__ACCEPT)
38  556:   proc 20 (accept_handler:1) test_out.pml:296 Send
    7,ACCEPTED,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 58 (__ACCEPTED)
39  570:   proc  2 (accept_handler:1) test_out.pml:291 Send
    7,ACCEPTED,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 58 (__ACCEPTED)
40  586:   proc 24 (receive_prepared:1) test_out.pml:421 Send
    13,MAJORITY_PREPARED,0,0,2,0,0,0,0,0,31,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 42 (__MAJORITY_PREPARED)
41  588:   proc 23 (receive_accepted:1) test_out.pml:459 Recv
```

```
    7,ACCEPTED,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 58 (__ACCEPTED)
42  600:   proc 15 (proposer_handler:1) test_out.pml:363 Recv
    13,MAJORITY_PREPARED,0,0,2,0,0,0,0,0,31,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 42 (__MAJORITY_PREPARED)
43  610:   proc 15 (proposer_handler:1) test_out.pml:386 Send
    1,ACCEPT,0,0,2,0,0,0,0,0,31,0,0,0,0,0,13,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 43 (__ACCEPT)
44  614:   proc 15 (proposer_handler:1) test_out.pml:386 Send
    3,ACCEPT,0,0,2,0,0,0,0,0,31,0,0,0,0,0,13,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 43 (__ACCEPT)
45  618:   proc 15 (proposer_handler:1) test_out.pml:386 Send
    5,ACCEPT,0,0,2,0,0,0,0,0,31,0,0,0,0,0,13,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 43 (__ACCEPT)
46  655:   proc 26 (accept_handler:1) test_out.pml:282 Recv
    5,ACCEPT,0,0,2,0,0,0,0,0,31,0,0,0,0,0,13,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 43 (__ACCEPT)
47  661:   proc 27 (receive_accepted:1) test_out.pml:454 Send 1 -> queue 65 (ret)
48  665:   proc 23 (receive_accepted:1) test_out.pml:471 Recv 1 <- queue 65 (ret2)
49  667:   proc 21 (accept_handler:1) test_out.pml:282 Recv
    3,ACCEPT,0,0,2,0,0,0,0,0,31,0,0,0,0,0,13,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 43 (__ACCEPT)
50  673:   proc 26 (accept_handler:1) test_out.pml:291 Send
    13,ACCEPTED,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 58 (__ACCEPTED)
51  675:   proc 23 (receive_accepted:1) test_out.pml:472 Send 1 -> queue 52 (ret)
52  677:   proc 21 (accept_handler:1) test_out.pml:291 Send
    13,ACCEPTED,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 58 (__ACCEPTED)
53  679:   proc 25 (accept_handler:1) test_out.pml:262 Recv
    1,PREPARE,0,0,1,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 23 (__PREPARE)
54  687:   proc 25 (accept_handler:1) test_out.pml:278 Send
    7,PROMISE,0,0,1,0,0,0,0,0,42,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 26 (__PROMISE)
55  699:   proc 27 (receive_accepted:1) test_out.pml:459 Recv
    13,ACCEPTED,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 58 (__ACCEPTED)
56  723:   proc 19 (receive_accepted:1) test_out.pml:466 Recv 1 <- queue 52 (ret1)
57  737:   proc 31 (receive_accepted:1) test_out.pml:459 Recv
    13,ACCEPTED,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 58 (__ACCEPTED)
58  755:   proc 30 (accept_handler:1) test_out.pml:262 Recv
    1,PREPARE,0,0,2,0,0,0,0,0,13,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 23 (__PREPARE)
59  761:   proc 32 (receive_accepted:1) test_out.pml:454 Send 2 -> queue 93 (ret)
60  765:   proc 30 (accept_handler:1) test_out.pml:272 Send
    13,PROMISE,0,0,1,0,0,0,0,0,42,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 26 (__PROMISE)
61  767:   proc 19 (receive_accepted:1) test_out.pml:467 Sent 1 -> queue 22 (ret)
62  768:   proc  8 (proposer_handler:1) test_out.pml:396 Recv 1 <- queue 22 (ret2)
63  772:   proc 31 (receive_accepted:1) test_out.pml:471 Recv 2 <- queue 93 (ret2)
64  780:   proc  8 (proposer_handler:1) test_out.pml:401 Send
    0,LEARNED,0,0,42,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 100 (__LEARNED)
65  782:   proc 22 (wait_learned:1) test_out.pml:558 Recv
    0,LEARNED,0,0,42,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 100 (__LEARNED)
66  788:   proc 32 (accept_handler:1) test_out.pml:282 Recv
    1,ACCEPT,0,0,2,0,0,0,0,0,31,0,0,0,0,0,13,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 43 (__ACCEPT)
67  794:   proc 32 (accept_handler:1) test_out.pml:291 Send
    13,ACCEPTED,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

```
    -> queue 58 (__ACCEPTED)
796:   proc 31 (receive_accepted:1) test_out.pml:472 Send 2 -> queue 79 (ret)
798:   proc 27 (receive_accepted:1) test_out.pml:471 Recv 2 <- queue 79 (ret2)
800:   proc 27 (receive_accepted:1) test_out.pml:472 Sent 2 -> queue 41 (ret)
801:   proc 15 (proposer_handler:1) test_out.pml:396 Recv 2 <- queue 41 (ret2)
Never claim moves to line 5  [(!(!((final_value==42))))]
Never claim moves to line 16 [(1)]
805:   proc 15 (proposer_handler:1) test_out.pml:401 Send
    0,LEARNED,0,0,31,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    -> queue 100 (__LEARNED)
817:   proc 33 (wait_learned:1) test_out.pml:558 Recv
    0,LEARNED,0,0,31,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    <- queue 100 (__LEARNED)
spin: _spin_nvr.tmp:15, Error: assertion violated
spin: text of failed assertion: assert(!((final_value==31)))
Never claim moves to line 15 [assert(!((final_value==31)))]
spin: trail ends after 820 steps
```

Listing A.2: Message passing caused by the proposer's protocol bug.

A.3 Consistent Hash Table

A.3.1 Working Consistent Hash Table

```
import VaeLib

defmodule ConsistentHashRing do

  @spec start_ring(list(), integer()) :: :ok
  def start_ring(nodes, n) do
    node_positions = Enum.map(nodes, fn n -> hash(n) end)
    ring_handler(nodes, node_positions, n)
  end

  @spec ring_handler(list(), list(), integer()) :: :ok
  defp ring_handler(nodes, node_positions, n) do
    receive do
      {:lookup, key, sender} ->
        position = hash(key)
        node = find_closest_node(nodes, node_positions, position, 0, n)
        send sender, {:ring_pos, node}
        ring_handler(nodes, node_positions, n)

      {:add_node, node, sender} ->
        new_nodes = nodes ++ [node]
        new_node_position = hash(node)
        new_node_positions = node_positions ++ [new_node_position]
        send sender, {:node_accepted}
        ring_handler(new_nodes, new_node_positions, n + 1)

      {:terminate} ->
        IO.puts("Terminating ring handler")
    end
  end

  @spec find_closest_node(list(), list(), integer(), integer(),
      integer()) :: integer()
  defp find_closest_node(nodes, node_positions, position, i, n) do
```

```
      if i >= n do
        Enum.at(nodes, 0)
      else
        check_node = Enum.at(nodes, i)
        check_pos = Enum.at(node_positions, i)

        if check_pos >= position do
          check_node
        else
          find_closest_node(nodes, node_positions, position, i + 1, n)
        end
      end
    end

    @spec hash(integer()) :: integer()
    defp hash(key) do
      # Example hardcoded hash values for keys and nodes
      case key do
        # Keys
        42 -> 1
        25 -> 8
        31 -> 10

        # Nodes
        1 -> 2
        2 -> 5
        3 -> 9
        4 -> 15
      end
    end
  end

  defmodule Client do

    @init true
    @spec start() :: :ok
    @ltl "[](r1 -> <>(p1))"
    @ltl "[](r2 -> <>(p3))"
    @ltl "[](r3 && n_nodes==3 -> <>(p1))"
    @ltl "[](r3 && n_nodes==4 -> <>(p4))"
    def start do
      n_nodes = 3
      nodes = for i <- 1..n_nodes do
        i
      end
      ring = spawn(ConsistentHashRing, :start_ring, [nodes, n_nodes])

      next_key = 42
      send ring, {:lookup, next_key, self()}
      ring_position = receive do
        {:ring_pos, node} ->
          IO.puts("Key 42 is assigned to")
          IO.puts node
          node
      end

      next_key = 25
      send ring, {:lookup, next_key, self()}
```

```
92        ring_position = receive do
93          {:ring_pos, node} ->
94            IO.puts("Key 25 is assigned to")
95            IO.puts node
96            node
97        end
98
99        next_key = 31
100       send ring, {:lookup, next_key, self()}
101       ring_position = receive do
102         {:ring_pos, node} ->
103           IO.puts("Key 31 is assigned to")
104           IO.puts node
105           node
106       end
107
108       # Dynamically grow the ring
109       send ring, {:add_node, 4, self()}
110       n_nodes = n_nodes + 1
111
112       receive do
113         {:node_accepted} ->
114           IO.puts("Node 4 added to the ring")
115       end
116
117       next_key = 31
118       send ring, {:lookup, next_key, self()}
119       ring_position = receive do
120         {:ring_pos, node} ->
121           IO.puts("Key 31 is assigned to")
122           IO.puts node
123           node
124       end
125
126       predicate p1, ring_position == 1
127       predicate p2, ring_position == 2
128       predicate p3, ring_position == 3
129       predicate p4, ring_position == 4
130       predicate r1, next_key == 42
131       predicate r2, next_key == 25
132       predicate r3, next_key == 31
133
134       send ring, {:terminate}
135     end
136   end
```

A.3.2 Promela for Consistent Hash Table

```
1  int n_nodes;
2  int ring_position;
3  #define p1 ((ring_position == 1))
4  #define p2 ((ring_position == 2))
5  #define p3 ((ring_position == 3))
6  #define p4 ((ring_position == 4))
7  int next_key = 42;
8  #define r1 ((next_key == 42))
9  #define r2 ((next_key == 25))
```

```
#define r3 ((next_key == 31))

proctype __anonymous_0 (int n; chan ret; int __pid) {
chan ret1 = [1] of { int }; /*7*/
if
::__pid==-1 -> __pid = _pid;
::else->skip;
fi;
run hash(n, ret1, __pid); /*7*/
int __ret_placeholder_1; /*7*/
ret1 ? __ret_placeholder_1; /*7*/
ret ! __ret_placeholder_1; /*7*/
}

proctype start_ring (int nodes;int n; chan ret; int __pid) {
chan __anonymous_ret_0 = [0] of { int };
chan ret2 = [1] of { int }; /*8*/
if
::__pid==-1 -> __pid = _pid;
::else->skip;
fi;
int node_positions;
__get_next_memory_allocation(node_positions);
atomic {
int __iter;
__iter = 0;
do
:: __iter >= LIST_LIMIT -> break;
:: else ->
if
:: LIST_ALLOCATED(nodes, __iter) ->
run __anonymous_0(LIST_VAL(nodes, __iter),__anonymous_ret_0,__pid);
LIST_ALLOCATED(node_positions, __iter) = true;
__anonymous_ret_0 ? LIST_VAL(node_positions, __iter);
__iter++;
:: else -> __iter++;
fi
od
}
int __temp_cp_arr_0;
__copy_memory_to_next(__temp_cp_arr_0, nodes);
int __temp_cp_arr_1;
__copy_memory_to_next(__temp_cp_arr_1, node_positions);
run ring_handler(__temp_cp_arr_0,__temp_cp_arr_1,n, ret2, __pid); /*8*/
}

proctype ring_handler (int nodes;int node_positions;int n; chan ret; int __pid) {
chan ret1 = [0] of { int }; /*15*/
chan ret2 = [0] of { int }; /*16*/
chan ret3 = [1] of { int }; /*18*/
chan ret4 = [0] of { int }; /*22*/
chan ret5 = [1] of { int }; /*25*/
if
::__pid==-1 -> __pid = _pid;
::else->skip;
fi;
MessageList rec_v_0; /*13*/
do /*13*/
:: __LOOKUP ?? eval(__pid),LOOKUP, rec_v_0 -> /*14*/
int key; /*14*/
key = rec_v_0.m1.data2; /*14*/
int sender; /*14*/
sender = rec_v_0.m2.data2; /*14*/
```

```
73  int position;
74  position = run hash(key, ret1, __pid); /*15*/
75  ret1 ? position; /*15*/
76  int node;
77  int __temp_cp_arr_2;
78  __copy_memory_to_next(__temp_cp_arr_2, nodes);
79  int __temp_cp_arr_3;
80  __copy_memory_to_next(__temp_cp_arr_3, node_positions);
81  node = run find_closest_node(__temp_cp_arr_2,__temp_cp_arr_3,position,0,n, ret2,
        __pid); /*16*/
82  ret2 ? node; /*16*/
83  MessageList msg_0; /*17*/
84  msg_0.m1.data2 = node; /*17*/
85  __RING_POS !! sender,RING_POS, msg_0; /*17*/
86  int __temp_cp_arr_4;
87  __copy_memory_to_next(__temp_cp_arr_4, nodes);
88  int __temp_cp_arr_5;
89  __copy_memory_to_next(__temp_cp_arr_5, node_positions);
90  run ring_handler(__temp_cp_arr_4,__temp_cp_arr_5,n, ret3, __pid); /*18*/
91  break;
92  :: __ADD_NODE ?? eval(__pid),ADD_NODE, rec_v_0 -> /*20*/
93  node = rec_v_0.m1.data2; /*20*/
94  sender = rec_v_0.m2.data2; /*20*/
95  int new_nodes;
96  __get_next_memory_allocation(new_nodes);
97  __list_append_list(new_nodes, nodes);
98  __list_append(new_nodes, node);
99  int new_node_position;
100 new_node_position = run hash(node, ret4, __pid); /*22*/
101 ret4 ? new_node_position; /*22*/
102 int new_node_positions;
103 __get_next_memory_allocation(new_node_positions);
104 __list_append_list(new_node_positions, node_positions);
105 __list_append(new_node_positions, new_node_position);
106 MessageList msg_1; /*24*/
107 __NODE_ACCEPTED !! sender,NODE_ACCEPTED, msg_1; /*24*/
108 int __temp_cp_arr_6;
109 __copy_memory_to_next(__temp_cp_arr_6, new_nodes);
110 int __temp_cp_arr_7;
111 __copy_memory_to_next(__temp_cp_arr_7, new_node_positions);
112 run ring_handler(__temp_cp_arr_6,__temp_cp_arr_7,n + 1, ret5, __pid); /*25*/
113 break;
114 :: __TERMINATE ?? eval(__pid),TERMINATE, rec_v_0 -> /*27*/
115 printf("Terminating ring handler\n");
116 break;
117 od;
118 }
119
120 proctype find_closest_node (int nodes;int node_positions;int position;int i;int n; chan
        ret; int __pid) {
121 chan ret1 = [1] of { int }; /*43*/
122 if
123 ::__pid==-1 -> __pid = _pid;
124 ::else->skip;
125 fi;
126 if
127 :: (i >= n) -> /*0*/
128 ret ! __list_at(nodes, 0)
129 :: else ->
130 int check_node;
131 check_node = __list_at(nodes, i)
132 int check_pos;
133 check_pos = __list_at(node_positions, i)
```

```
134 if
135 :: (check_pos >= position) -> /*0*/
136 ret ! check_node; /*41*/
137 :: else ->
138 int __temp_cp_arr_8;
139 __copy_memory_to_next(__temp_cp_arr_8, nodes);
140 int __temp_cp_arr_9;
141 __copy_memory_to_next(__temp_cp_arr_9, node_positions);
142 run find_closest_node(__temp_cp_arr_8,__temp_cp_arr_9,position,i + 1,n, ret1, __pid);
    /*43*/
143 int __ret_placeholder_1; /*43*/
144 ret1 ? __ret_placeholder_1; /*43*/
145 ret ! __ret_placeholder_1; /*43*/
146 fi;
147 fi;
148 }
149
150 proctype hash (int key; chan ret; int __pid) {
151 if
152 ::__pid==-1 -> __pid = _pid;
153 ::else->skip;
154 fi;
155 do
156 :: key == 42 ->
157 ret ! 1; /*0*/
158 break;
159 :: key == 25 ->
160 ret ! 8; /*0*/
161 break;
162 :: key == 31 ->
163 ret ! 10; /*0*/
164 break;
165 :: key == 1 ->
166 ret ! 2; /*0*/
167 break;
168 :: key == 2 ->
169 ret ! 5; /*0*/
170 break;
171 :: key == 3 ->
172 ret ! 9; /*0*/
173 break;
174 :: key == 4 ->
175 ret ! 15; /*0*/
176 break;
177 od
178 }
179
180 active proctype start () {
181 chan ret1 = [1] of { int };
182 int __pid = 0;
183 if
184 ::__pid==-1 -> __pid = _pid;
185 ::else->skip;
186 fi;
187 n_nodes = 3;
188 int nodes;
189 __get_next_memory_allocation(nodes);
190 int i;
191 for(i : 1 .. n_nodes) { /*76*/
192 int __tmp;
193 __tmp = i; /*77*/
194 __list_append(nodes, __tmp);
195 }
```

```
196 int ring;
197 atomic {
198 ring = run start_ring(nodes,n_nodes,ret1,-1); /*79*/
199 }
200 MessageList msg_0; /*82*/
201 msg_0.m1.data2 = next_key; /*82*/
202 msg_0.m2.data2 = __pid; /*82*/
203 __LOOKUP !! ring,LOOKUP, msg_0; /*82*/
204 MessageList rec_v_1; /*83*/
205 do /*83*/
206 :: __RING_POS ?? eval(__pid),RING_POS, rec_v_1 -> /*0*/
207 int node; /*0*/
208 node = rec_v_1.m1.data2; /*0*/
209 printf("Key 42 is assigned to\n");
210 printf("node\n");
211 ring_position = node; /*87*/
212 break;
213 od;
214 next_key = 25;
215 MessageList msg_1; /*91*/
216 msg_1.m1.data2 = next_key; /*91*/
217 msg_1.m2.data2 = __pid; /*91*/
218 __LOOKUP !! ring,LOOKUP, msg_1; /*91*/
219 MessageList rec_v_2; /*92*/
220 do /*92*/
221 :: __RING_POS ?? eval(__pid),RING_POS, rec_v_2 -> /*0*/
222 node = rec_v_2.m1.data2; /*0*/
223 printf("Key 25 is assigned to\n");
224 printf("node\n");
225 ring_position = node; /*96*/
226 break;
227 od;
228 next_key = 31;
229 MessageList msg_2; /*100*/
230 msg_2.m1.data2 = next_key; /*100*/
231 msg_2.m2.data2 = __pid; /*100*/
232 __LOOKUP !! ring,LOOKUP, msg_2; /*100*/
233 MessageList rec_v_3; /*101*/
234 do /*101*/
235 :: __RING_POS ?? eval(__pid),RING_POS, rec_v_3 -> /*0*/
236 node = rec_v_3.m1.data2; /*0*/
237 printf("Key 31 is assigned to\n");
238 printf("node\n");
239 ring_position = node; /*105*/
240 break;
241 od;
242 MessageList msg_3; /*109*/
243 msg_3.m1.data2 = 4; /*109*/
244 msg_3.m2.data2 = __pid; /*109*/
245 __ADD_NODE !! ring,ADD_NODE, msg_3; /*109*/
246 n_nodes = n_nodes + 1;
247 MessageList rec_v_4; /*112*/
248 do /*112*/
249 :: __NODE_ACCEPTED ?? eval(__pid),NODE_ACCEPTED, rec_v_4 -> /*113*/
250 printf("Node 4 added to the ring\n");
251 break;
252 od;
253 next_key = 31;
254 MessageList msg_4; /*118*/
255 msg_4.m1.data2 = next_key; /*118*/
256 msg_4.m2.data2 = __pid; /*118*/
257 __LOOKUP !! ring,LOOKUP, msg_4; /*118*/
258 MessageList rec_v_5; /*119*/
```

```
do /*119*/
:: __RING_POS ?? eval(__pid),RING_POS, rec_v_5 -> /*0*/
node = rec_v_5.m1.data2; /*0*/
printf("Key 31 is assigned to\n");
printf("node\n");
ring_position = node; /*123*/
break;
od;
MessageList msg_5; /*134*/
__TERMINATE !! ring,TERMINATE, msg_5; /*134*/
}


ltl ltl_1 { [](r1 -> <>(p1)) };
ltl ltl_2 { [](r2 -> <>(p3)) };
ltl ltl_3 { [](r3 && n_nodes==3 -> <>(p1)) };
ltl ltl_4 { [](r3 && n_nodes==4 -> <>(p4)) };
```

Listing A.3: Promela of consistent hash table

A.3.3 Buggy Consistent Hash Table

```
import VaeLib

defmodule ConsistentHashRingB do

  @spec start_ring(list(), integer()) :: :ok
  def start_ring(nodes, n) do
    node_positions = Enum.map(nodes, fn n -> hash(n) end)
    ring_handler(nodes, node_positions, n)
  end

  @spec ring_handler(list(), list(), integer()) :: :ok
  defp ring_handler(nodes, node_positions, n) do
    receive do
      {:lookup, key, sender} ->
        position = hash(key)
        node = find_closest_node(nodes, node_positions, position, 0, n)
        send sender, {:ring_pos, node}
        ring_handler(nodes, node_positions, n)

      {:add_node, node} ->
        new_nodes = nodes ++ [node]
        new_node_position = hash(node)
        new_node_positions = node_positions ++ [new_node_position]
        ring_handler(new_nodes, new_node_positions, n + 1)

      {:terminate} ->
        IO.puts("Terminating ring handler")
    end
  end

  @spec find_closest_node(list(), list(), integer(), integer(),
      integer()) :: integer()
  defp find_closest_node(nodes, node_positions, position, i, n) do
    if i >= n do
      Enum.at(nodes, 0)
    else
```

```
36          check_node = Enum.at(nodes, i)
37          check_pos = Enum.at(node_positions, i)
38
39          if check_pos >= position do
40            check_node
41          else
42            find_closest_node(nodes, node_positions, position, i + 1, n)
43          end
44        end
45      end
46
47      @spec hash(integer()) :: integer()
48      defp hash(key) do
49        # Example hardcoded hash values for keys and nodes
50        case key do
51          # Keys
52          42 -> 1
53          25 -> 8
54          31 -> 10
55
56          # Nodes
57          1 -> 2
58          2 -> 5
59          3 -> 9
60          4 -> 15
61        end
62      end
63    end
64
65    defmodule ClientB do
66
67      @init true
68      @spec start() :: :ok
69      @ltl "[](r1 -> <>(p1))"
70      @ltl "[](r2 -> <>(p3))"
71      @ltl "[](r3 && n_nodes==3 -> <>(p1))"
72      @ltl "[](r3 && n_nodes==4 -> <>(p4))"
73      def start do
74        n_nodes = 3
75        nodes = for i <- 1..n_nodes do
76          i
77        end
78        ring = spawn(ConsistentHashRingB, :start_ring, [nodes, n_nodes])
79
80        next_key = 42
81        send ring, {:lookup, next_key, self()}
82        ring_position = receive do
83          {:ring_pos, node} ->
84            IO.puts("Key 42 is assigned to")
85            IO.puts node
86            node
87        end
88
89        next_key = 25
90        send ring, {:lookup, next_key, self()}
91        ring_position = receive do
92          {:ring_pos, node} ->
93            IO.puts("Key 25 is assigned to")
```

```
94            IO.puts node
95            node
96        end
97
98        next_key = 31
99        send ring, {:lookup, next_key, self()}
100       ring_position = receive do
101         {:ring_pos, node} ->
102           IO.puts("Key 31 is assigned to")
103           IO.puts node
104           node
105       end
106
107       # Dynamically grow the ring
108       send ring, {:add_node, 4}
109       n_nodes = n_nodes + 1
110
111       next_key = 31
112       send ring, {:lookup, next_key, self()}
113       ring_position = receive do
114         {:ring_pos, node} ->
115           IO.puts("Key 31 is assigned to")
116           IO.puts node
117           node
118       end
119
120       predicate p1, ring_position == 1
121       predicate p2, ring_position == 2
122       predicate p3, ring_position == 3
123       predicate p4, ring_position == 4
124       predicate r1, next_key == 42
125       predicate r2, next_key == 25
126       predicate r3, next_key == 31
127
128       send ring, {:terminate}
129     end
130   end
```

A.3.4 Buggy Hash Table Logs

```
1  The program is livelocked, or an ltl property was violated. Generating trace.
2  <<<Message Events>>>
3  [1] (hash:1) send [2]
4  [2] (__anonymous_0:1) recv [2]
5  [3] (start_ring:1) recv [2]
6  [4] (hash:1) send [5]
7  [5] (__anonymous_0:1) recv [5]
8  [6] (start_ring:1) recv [5]
9  [7] (hash:1) send [9]
10 [8] (__anonymous_0:1) recv [9]
11 [9] (start_ring:1) recv [9]
12 [10] (start:1) send
       [1,LOOKUP,0,0,42,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
13 [11] (ring_handler:1) recv
       [1,LOOKUP,0,0,42,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
14 [12] (ring_handler:1) recv [1]
15 [13] (ring_handler:1) recv [1]
16 [14] (ring_handler:1) send
```

```
    [0,RING_POS,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[15] (start:1) recv
    [0,RING_POS,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[16] (start:1) send
    [1,LOOKUP,0,0,25,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[17] (ring_handler:1) recv
    [1,LOOKUP,0,0,25,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[18] (ring_handler:1) recv [8]
[19] (find_closest_node:1) send [3]
[20] (find_closest_node:1) recv [3]
[21] (find_closest_node:1) send [3]
[22] (find_closest_node:1) recv [3]
[23] (ring_handler:1) recv [3]
[24] (ring_handler:1) send
    [0,RING_POS,0,0,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[25] (start:1) recv
    [0,RING_POS,0,0,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[26] (start:1) send
    [1,LOOKUP,0,0,31,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[27] (ring_handler:1) recv
    [1,LOOKUP,0,0,31,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[28] (ring_handler:1) recv [10]
[29] (find_closest_node:1) send [1]
[30] (find_closest_node:1) recv [1]
[31] (find_closest_node:1) send [1]
[32] (find_closest_node:1) recv [1]
[33] (find_closest_node:1) send [1]
[34] (find_closest_node:1) recv [1]
[35] (ring_handler:1) recv [1]
[36] (ring_handler:1) send
    [0,RING_POS,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[37] (start:1) recv
    [0,RING_POS,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[38] (start:1) send
    [1,ADD_NODE,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[39] (start:1) send
    [1,LOOKUP,0,0,31,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[40] (ring_handler:1) recv
    [1,LOOKUP,0,0,31,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[41] (ring_handler:1) recv [10]
[42] (find_closest_node:1) send [1]
[43] (find_closest_node:1) recv [1]
[44] (find_closest_node:1) send [1]
[45] (find_closest_node:1) recv [1]
[46] (find_closest_node:1) send [1]
[47] (find_closest_node:1) recv [1]
[48] (ring_handler:1) recv [1]
[49] (ring_handler:1) send
    [0,RING_POS,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[50] (ring_handler:1) recv
    [1,ADD_NODE,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[51] (ring_handler:1) recv [15]
[52] (start:1) recv
    [0,RING_POS,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[53] (start:1) send
    [1,TERMINATE,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[54] (ring_handler:1) recv
    [1,TERMINATE,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
```

Listing A.4: Message passing caused by the proposer's protocol bug.

A.4 Two-Phase Commit

A.4.1 LTLixir 2PC

```
1  import VaeLib
2
3  defmodule Coordinator do
4    @spec start_coordinator(list(), integer(), integer(), integer()) ::
         :ok
5    def start_coordinator(participants, transaction_id, value,
     n_participants) do
6      coordinator_handler(participants, transaction_id, value, 0,
           n_participants)
7    end
8
9    @spec coordinator_handler(list(), integer(), integer(), integer(),
         integer()) :: :ok
10   defp coordinator_handler(participants, transaction_id, value, phase,
     n_participants) do
11     case phase do
12       0 -> # Phase 1: Prepare
13         for participant <- participants do
14           send participant, {:prepare, transaction_id, value, self()}
15         end
16         receive_prepare_responses(participants, transaction_id, value,
               0, 0, n_participants)
17       1 -> # Phase 2: Commit
18         for participant <- participants do
19           send participant, {:commit, transaction_id, self()}
20         end
21         wait_for_acks(participants, 0, n_participants, 1)
22     end
23   end
24
25   @spec receive_prepare_responses(list(), integer(), integer(),
         integer(), integer(), integer()) :: :ok
26   defp receive_prepare_responses(participants, transaction_id, value,
     count, acks, n_participants) do
27     if count >= n_participants do
28       if acks == n_participants do
29         coordinator_handler(participants, transaction_id, value, 1,
               n_participants)
30       else
31         IO.puts("Transaction aborted")
32         for participant <- participants do
33           send participant, {:abort, transaction_id, self()}
34         end
35         wait_for_acks(participants, 0, n_participants, 0)
36       end
37     else
38       receive do
39         {:prepared, response_transaction_id, participant} ->
40           if response_transaction_id == transaction_id do
41             receive_prepare_responses(participants, transaction_id,
                   value, count + 1, acks + 1, n_participants)
42           else
43             receive_prepare_responses(participants, transaction_id,
                   value, count, acks, n_participants)
```

```
44            end
45          {:abort, response_transaction_id, participant} ->
46            if response_transaction_id == transaction_id do
47              receive_prepare_responses(participants, transaction_id,
                    value, count + 1, acks, n_participants)
48            else
49              receive_prepare_responses(participants, transaction_id,
                    value, count, acks, n_participants)
50            end
51        end
52      end
53    end
54
55    @spec wait_for_acks(list(), integer(), integer(), integer()) :: :ok
56    defp wait_for_acks(participants, count, n_participants, committed) do
57      if count >= n_participants do
58        if committed == 1 do
59          IO.puts("Transaction committed")
60        end
61        for participant <- participants do
62          send participant, {:terminate}
63        end
64      else
65        receive do
66          {:ack, _participant} ->
67            wait_for_acks(participants, count + 1, n_participants,
                committed)
68        end
69      end
70    end
71  end
72
73  defmodule Participant do
74    @spec start_participant(integer()) :: :ok
75    def start_participant(client) do
76      participant_handler(client)
77    end
78
79    @spec participant_handler(integer()) :: :ok
80    defp participant_handler(client) do
81      receive do
82        {:prepare, transaction_id, value, coordinator} ->
83          prepare = decide_to_prepare(value)
84          if prepare do
85            send coordinator, {:prepared, transaction_id, self()}
86          else
87            send coordinator, {:abort, transaction_id, self()}
88          end
89          participant_handler(client)
90        {:commit, transaction_id, coordinator} ->
91          commit(transaction_id, client)
92          send coordinator, {:ack, self()}
93          participant_handler(client)
94        {:abort, transaction_id, coordinator} ->
95          abort(transaction_id, client)
96          send coordinator, {:ack, self()}
97          participant_handler(client)
98        {:terminate} ->
```

```
99            IO.puts("Terminating participant")
100         end
101       end
102
103       @spec decide_to_prepare(integer()) :: boolean()
104       defp decide_to_prepare(value) do
105         # Example decision logic i.e. ensure all locks are required to
              make the commit
106         # We use some arbitrary random logic
107         cmps = [10, 90]
108         cmp = Enum.random(cmps)
109         if value < cmp do
110           true
111         else
112           false
113         end
114       end
115
116       @spec commit(integer(), integer()) :: :ok
117       defp commit(transaction_id, client) do
118         IO.puts("Committing transaction")
119         send client, {:transaction_commit}
120       end
121
122       @spec abort(integer(), integer()) :: :ok
123       defp abort(transaction_id, client) do
124         IO.puts("Aborting transaction")
125         send client, {:transaction_abort}
126
127       end
128     end
129
130     defmodule Client do
131       @init true
132       @spec start() :: :ok
133       def start do
134         n_participants = 3
135         participants = for _ <- 1..n_participants do
136           spawn(Participant, :start_participant, [self()])
137         end
138
139         transaction_id = 1
140         value = 42
141         coordinator = spawn(Coordinator, :start_coordinator,
              [participants, transaction_id, value, n_participants])
142
143         await_transaction_result(0, 0, n_participants)
144       end
145
146       @spec await_transaction_result(integer(), integer(), integer()) ::
              :ok
147       @ltl "<>[]p || <>[]q"
148       @ltl "[](p -> !<>[]q)"
149       @ltl "[](q -> !<>[]p)"
150       def await_transaction_result(n_c, n_a, n_p) do
151         commit_count = n_c
152         abort_count = n_a
153         participant_count = n_p
```

```
154         predicate p, commit_count == participant_count
155         predicate q, abort_count == participant_count
156         if n_c + n_a >= n_p do
157           IO.puts("All participants have responded")
158         else
159           receive do
160             {:transaction_commit} ->
161               await_transaction_result(n_c + 1, n_a, n_p)
162             {:transaction_abort} ->
163               await_transaction_result(n_c, n_a + 1, n_p)
164           end
165         end
166       end
167     end
```

A.4.2 Buggy 2PC Logs

```
1  The program is livelocked, or an ltl property was violated. Generating trace.
2  <<<Message Events>>>
3  [1] (coordinator_handler:1) send
       [1,PREPARE,0,0,1,0,0,0,0,0,42,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
4  [2] (coordinator_handler:1) send
       [3,PREPARE,0,0,1,0,0,0,0,0,42,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
5  [3] (coordinator_handler:1) send
       [5,PREPARE,0,0,1,0,0,0,0,0,42,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
6  [4] (participant_handler:1) recv
       [5,PREPARE,0,0,1,0,0,0,0,0,42,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
7  [5] (participant_handler:1) recv [1]
8  [6] (participant_handler:1) send
       [7,PREPARED,0,0,1,0,0,0,0,0,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
9  [7] (receive_prepare_responses:1) recv
       [7,PREPARED,0,0,1,0,0,0,0,0,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
10 [8] (participant_handler:1) recv
       [3,PREPARE,0,0,1,0,0,0,0,0,42,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
11 [9] (participant_handler:1) recv [1]
12 [10] (participant_handler:1) send
       [7,PREPARED,0,0,1,0,0,0,0,0,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
13 [11] (receive_prepare_responses:1) recv
       [7,PREPARED,0,0,1,0,0,0,0,0,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
14 [12] (participant_handler:1) recv
       [1,PREPARE,0,0,1,0,0,0,0,0,42,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
15 [13] (participant_handler:1) recv [1]
16 [14] (participant_handler:1) send
       [7,PREPARED,0,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
17 [15] (receive_prepare_responses:1) recv
       [7,PREPARED,0,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
18 [16] (coordinator_handler:1) send
       [1,COMMIT,0,0,1,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
19 [17] (coordinator_handler:1) send
       [3,COMMIT,0,0,1,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
20 [18] (coordinator_handler:1) send
       [5,COMMIT,0,0,1,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
21 [19] (participant_handler:1) recv
       [3,COMMIT,0,0,1,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
22 [20] (commit:1) send
       [0,TRANSACTION_COMMIT,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
23 [21] (participant_handler:1) send
       [7,ACK,0,0,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
24 [22] (wait_for_acks:1) recv
       [7,ACK,0,0,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
```

```
25 [23] (participant_handler:1) recv
       [5,COMMIT,0,0,1,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
26 [24] (commit:1) send
       [0,TRANSACTION_COMMIT,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
27 [25] (participant_handler:1) send
       [7,ACK,0,0,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
28 [26] (wait_for_acks:1) recv
       [7,ACK,0,0,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
29 [27] (participant_handler:1) recv
       [1,COMMIT,0,0,1,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
30 [28] (commit:1) send
       [0,TRANSACTION_ABORT,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
31 [29] (participant_handler:1) send
       [7,ACK,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
32 [30] (wait_for_acks:1) recv
       [7,ACK,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
33 [31] (wait_for_acks:1) send
       [1,TERMINATE,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
34 [32] (wait_for_acks:1) send
       [3,TERMINATE,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
35 [33] (wait_for_acks:1) send
       [5,TERMINATE,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
36 [34] (participant_handler:1) recv
       [1,TERMINATE,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
37 [35] (participant_handler:1) recv
       [5,TERMINATE,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
38 [36] (participant_handler:1) recv
       [3,TERMINATE,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
39 [37] (await_transaction_result:1) recv
       [0,TRANSACTION_COMMIT,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
40 [38] (await_transaction_result:1) recv
       [0,TRANSACTION_COMMIT,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
41 [39] (await_transaction_result:1) recv
       [0,TRANSACTION_ABORT,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
```

Listing A.5: Message passing caused by the proposer's protocol bug.

A.5 Dining Philosophers

A.5.1 Dining Philosophers Deadlock Logs

```
1  The program likely reached a deadlock. Generating trace.
2  <<<Message Events>>>
3  [1] (start:1) send
       [4,BIND,0,0,1,0,0,0,0,0,2,0,0,0,0,0,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
4  [2] (start_phil:1) recv
       [4,BIND,0,0,1,0,0,0,0,0,2,0,0,0,0,0,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
5  [3] (phil_loop:1) send
       [1,SIT,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
6  [4] (table_loop:1) recv
       [1,SIT,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
7  [5] (table_loop:1) send
       [4,OK,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
8  [6] (wait:1) recv
       [4,OK,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
9  [7] (wait:1) send [0]
10 [8] (phil_loop:1) recv [0]
11 [9] (phil_loop:1) send
       [2,PICKUP,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
12 [10] (start:1) send
       [2,LPHIL,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
```

```
[11] (start_fork:1) recv
    [2,LPHIL,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[12] (start:1) send
    [3,RPHIL,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[13] (start:1) send
    [5,BIND,0,0,1,0,0,0,0,0,3,0,0,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[14] (start_phil:1) recv
    [5,BIND,0,0,1,0,0,0,0,0,3,0,0,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[15] (phil_loop:1) send
    [1,SIT,0,0,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[16] (table_loop:1) recv
    [1,SIT,0,0,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[17] (table_loop:1) send
    [5,OK,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[18] (wait:1) recv
    [5,OK,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[19] (wait:1) send [0]
[20] (phil_loop:1) recv [0]
[21] (phil_loop:1) send
    [3,PICKUP,0,0,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[22] (start:1) send
    [3,LPHIL,0,0,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[23] (start_fork:1) recv
    [3,LPHIL,0,0,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[24] (start_fork:1) recv
    [3,RPHIL,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[25] (fork_loop:1) recv
    [3,PICKUP,0,0,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[26] (fork_loop:1) send
    [5,OK,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[27] (wait:1) recv
    [5,OK,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[28] (wait:1) send [0]
[29] (phil_loop:1) recv [0]
[30] (phil_loop:1) send
    [2,PICKUP,0,0,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[31] (start:1) send
    [2,RPHIL,0,0,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[32] (start_fork:1) recv
    [2,RPHIL,0,0,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[33] (fork_loop:1) recv
    [2,PICKUP,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[34] (fork_loop:1) send
    [4,OK,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[35] (wait:1) recv
    [4,OK,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
[36] (wait:1) send [0]
[37] (phil_loop:1) recv [0]
[38] (phil_loop:1) send
    [3,PICKUP,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]

<<<Error Trace>>>
[1] (proc_0) start:120 [send lfork, {:lphil, phil}]
[2] (proc_4) start_phil:20 [end]
[3] (proc_6) phil_loop:27 [wait()]
[4] (proc_1) table_loop:0 []
[5] (proc_1) table_loop:6 [table_loop()]
[6] (proc_7) wait:53 [end]
[8] (proc_6) phil_loop:28 []
[9] (proc_6) phil_loop:31 [wait()]
[10] (proc_0) start:121 [send rfork, {:rphil, phil}]
[11] (proc_2) start_fork:0 []
[12] (proc_0) start:122 [end]
```

```
[13] (proc_0) start:120 [send lfork, {:lphil, phil}]
[14] (proc_5) start_phil:20 [end]
[15] (proc_9) phil_loop:27 [wait()]
[16] (proc_8) table_loop:0 []
[17] (proc_8) table_loop:6 [table_loop()]
[18] (proc_10) wait:53 [end]
[20] (proc_9) phil_loop:28 []
[21] (proc_9) phil_loop:31 [wait()]
[22] (proc_0) start:121 [send rfork, {:rphil, phil}]
[23] (proc_3) start_fork:0 []
[24] (proc_3) start_fork:0 []
[25] (proc_12) fork_loop:0 []
[26] (proc_12) fork_loop:76 [fork_loop(1, lphil, rphil)]
[27] (proc_10) wait:53 [end]
[29] (proc_9) phil_loop:32 [IO.puts "lfork"]
[30] (proc_9) phil_loop:35 [wait()]
[31] (proc_0) start:122 [end]
[32] (proc_2) start_fork:0 []
[33] (proc_15) fork_loop:0 []
[34] (proc_15) fork_loop:76 [fork_loop(1, lphil, rphil)]
[35] (proc_7) wait:53 [end]
[37] (proc_6) phil_loop:32 [IO.puts "lfork"]
[38] (proc_6) phil_loop:35 [wait()]
[39] (proc_17) wait:52 [{:ok} -> :ok]
[40] (proc_16) fork_loop:86 [{:putdown, phil} ->]
[41] (proc_15) fork_loop:77 [else]
[42] (proc_14) wait:52 [{:ok} -> :ok]
[43] (proc_13) fork_loop:86 [{:putdown, phil} ->]
[44] (proc_12) fork_loop:77 [else]
[45] (proc_11) table_loop:4 [{:sit, phil} ->]
[47] (proc_9) phil_loop:36 [IO.puts "rfork"]
[48] (proc_8) table_loop:7 [{:leave, phil} ->]
[50] (proc_6) phil_loop:36 [IO.puts "rfork"]
[51] (proc_5) start_phil:20 [end]
[52] (proc_4) start_phil:20 [end]
[53] (proc_3) start_fork:64 [end]
[54] (proc_2) start_fork:64 [end]
[55] (proc_1) table_loop:7 [{:leave, phil} ->]
[56] (proc_0) start:126 [{:done} -> :ok]
```

Listing A.6: Dining Philosophers Verlixir Report.

A.5.2 Dining Philosophers in Elixir

```elixir
defmodule Table do
  @spec table_loop() :: :ok
  def table_loop do
    receive do
      {:sit, phil} ->
        send phil, {:ok}
        table_loop()
      {:leave, phil} ->
        send phil, {:ok}
        table_loop()
      {:terminate} -> :ok
    end
  end
end

defmodule Philosopher do
```

```
17    @spec start_phil(integer()) :: :ok
18    def start_phil(coordinator) do
19      receive do
20        {:bind, table, lfork, rfork} -> phil_loop(coordinator, table,
              lfork, rfork)
21      end
22    end
23
24    @spec phil_loop(integer(), integer(), integer(), integer()) :: :ok
25    def phil_loop(coordinator, table, lfork, rfork) do
26      # ... think ... #
27      send table, {:sit, self()}
28      wait()
29
30      # ... sitting ... #
31      send lfork, {:pickup, self()}
32      wait()
33      IO.puts "lfork"
34
35      send rfork, {:pickup, self()}
36      wait()
37      IO.puts "rfork"
38
39      # ... eating ... #
40      send table, {:leave, self()}
41      wait()
42      send lfork, {:putdown, self()}
43      wait()
44      send rfork, {:putdown, self()}
45      wait()
46
47      send coordinator, {:done}
48    end
49
50    @spec wait() :: :ok
51    def wait() do
52      receive do
53        {:ok} -> :ok
54      end
55    end
56  end
57
58  defmodule Fork do
59    @spec start_fork() :: :ok
60    def start_fork do
61      receive do
62        {:lphil, lphil} ->
63          receive do
64            {:rphil, rphil} -> fork_loop(0, lphil, rphil)
65          end
66      end
67    end
68
69    @spec fork_loop(integer(), integer(), integer()) :: :ok
70    def fork_loop(allocated, lphil, rphil) do
71      # allocated: 0 => none, 1 => left, 2 => right
72      if allocated == 0 do
73        receive do
```

```
        {:pickup, phil} ->
          if phil == lphil do
            send phil, {:ok}
            fork_loop(1, lphil, rphil)
          else
            send phil, {:ok}
            fork_loop(2, lphil, rphil)
          end
        {:terminate} ->
          :ok
      end
    else
      receive do
        {:putdown, phil} ->
          send phil, {:ok}
          fork_loop(0, lphil, rphil)
        {:terminate} ->
          :ok
      end
    end
  end
end


defmodule Coordinator do
  @init true
  @spec start() :: :ok
  def start do
    n = 4

    table = spawn(Table, :table_loop, [])

    forks = for _ <- 1..n do
      spawn(Fork, :start_fork, [])
    end

    phils = for i <- 1..n do
      spawn(Philosopher, :start_phil, [self()])
    end

    j = n-1
    for i <- 0..j do
      phil = Enum.at(phils,i)
      lfork = Enum.at(forks,i)
      r_i = rem(i+1, n)
      rfork = Enum.at(forks, r_i)
      send phil, {:bind, table, lfork, rfork}
      send lfork, {:lphil, phil}
      send rfork, {:rphil, phil}
    end

    for i <- 1..n do
      receive do
        {:done} -> :ok
      end
    end
    IO.puts "All philosophers have finished eating!"

```

```
      for fork <- forks do
        send fork, {:terminate}
      end
      send table, {:terminate}
    end
  end
```

A.5.3 Promela Translation of Dining Philosophers

```
proctype table_loop (chan ret;int __pid;int __ret_f) {
  chan ret1 = [1] of { int };/* 7*/ /* table_loop()*/
  chan ret2 = [1] of { int };/* 10*/ /* table_loop()*/
      atomic{
          if
          :: __pid == - 1 -> __pid = _pid;
          :: else -> skip;
          fi;
      }
      MessageList rec_v_0;/* 4*/ /* receive do*/
      do/* 4*/ /* receive do*/
      :: __SIT??eval(__pid),SIT,rec_v_0 -> /* 0*/
          int phil;/* 0*/
          phil = rec_v_0.m1.data2;/* 0*/
          atomic {
          MessageList msg_0;/* 6*/ /* send phil,{:ok}*/
          __OK!!phil,OK,msg_0;/* 6*/ /* send phil,{:ok}*/
          }
          int __ret_placeholder_1;/* 7*/ /* table_loop()*/
          run table_loop(ret1,__pid,1);/* 7*/ /* table_loop()*/
          ret1?__ret_placeholder_1;/* 7*/ /* table_loop()*/
          break;
      :: __LEAVE??eval(__pid),LEAVE,rec_v_0 -> /* 0*/
          phil = rec_v_0.m1.data2;/* 0*/
          atomic {
          MessageList msg_1;/* 9*/ /* send phil,{:ok}*/
          __OK!!phil,OK,msg_1;/* 9*/ /* send phil,{:ok}*/
          }
          int __ret_placeholder_2;/* 10*/ /* table_loop()*/
          run table_loop(ret2,__pid,1);/* 10*/ /* table_loop()*/
          ret2?__ret_placeholder_2;/* 10*/ /* table_loop()*/
          break;
      :: __TERMINATE??eval(__pid),TERMINATE,rec_v_0 -> /* 11*/ /* {:terminate} -> :ok*/
          break;
      od;
      atomic{
          if
          :: __ret_f -> ret!0;
          :: else -> skip;
          fi;
      }
  }

  proctype start_phil (int coordinator;chan ret;int __pid;int __ret_f) {
  chan ret1 = [1] of { int };/* 20*/ /* {:bind,table,lfork,rfork} ->
       phil_loop(coordinator,table,lfork,rfork)*/
      atomic{
          if
          :: __pid == - 1 -> __pid = _pid;
          :: else -> skip;
          fi;
```

```
51          }
52          MessageList rec_v_1;/* 19*/ /* receive do*/
53          do/* 19*/ /* receive do*/
54            :: __BIND??eval(__pid),BIND,rec_v_1 -> /* 20*/ /* {:bind,table,lfork,rfork} ->
                  phil_loop(coordinator,table,lfork,rfork)*/
55              int table;/* 20*/ /* {:bind,table,lfork,rfork} ->
                    phil_loop(coordinator,table,lfork,rfork)*/
56              table = rec_v_1.m1.data2;/* 20*/ /* {:bind,table,lfork,rfork} ->
                    phil_loop(coordinator,table,lfork,rfork)*/
57              int lfork;/* 20*/ /* {:bind,table,lfork,rfork} ->
                    phil_loop(coordinator,table,lfork,rfork)*/
58              lfork = rec_v_1.m2.data2;/* 20*/ /* {:bind,table,lfork,rfork} ->
                    phil_loop(coordinator,table,lfork,rfork)*/
59              int rfork;/* 20*/ /* {:bind,table,lfork,rfork} ->
                    phil_loop(coordinator,table,lfork,rfork)*/
60              rfork = rec_v_1.m3.data2;/* 20*/ /* {:bind,table,lfork,rfork} ->
                    phil_loop(coordinator,table,lfork,rfork)*/
61              int __ret_placeholder_1;/* 20*/ /* {:bind,table,lfork,rfork} ->
                    phil_loop(coordinator,table,lfork,rfork)*/
62              run phil_loop(coordinator,table,lfork,rfork,ret1,__pid,1);/* 20*/ /*
                    {:bind,table,lfork,rfork} -> phil_loop(coordinator,table,lfork,rfork)*/
63              ret1?__ret_placeholder_1;/* 20*/ /* {:bind,table,lfork,rfork} ->
                    phil_loop(coordinator,table,lfork,rfork)*/
64              break;
65          od;
66          atomic{
67              if
68              :: __ret_f -> ret!0;
69              :: else -> skip;
70              fi;
71          }
72      }
73
74      proctype phil_loop (int coordinator;int table;int lfork;int rfork;chan ret;int
            __pid;int __ret_f) {
75      chan ret1 = [1] of { int };/* 28*/ /* wait()*/
76      chan ret2 = [1] of { int };/* 32*/ /* wait()*/
77      chan ret3 = [1] of { int };/* 36*/ /* wait()*/
78      chan ret4 = [1] of { int };/* 41*/ /* wait()*/
79      chan ret5 = [1] of { int };/* 43*/ /* wait()*/
80      chan ret6 = [1] of { int };/* 45*/ /* wait()*/
81          atomic{
82              if
83              :: __pid == - 1 -> __pid = _pid;
84              :: else -> skip;
85              fi;
86          }
87          atomic {
88          MessageList msg_0;/* 27*/ /* send table,{:sit,self()}*/
89          msg_0.m1.data2 = __pid;/* 27*/ /* send table,{:sit,self()}*/
90          __SIT!!table,SIT,msg_0;/* 27*/ /* send table,{:sit,self()}*/
91          }
92          int __ret_placeholder_1;/* 28*/ /* wait()*/
93          run wait(ret1,__pid,1);/* 28*/ /* wait()*/
94          ret1?__ret_placeholder_1;/* 28*/ /* wait()*/
95          atomic {
96          MessageList msg_1;/* 31*/ /* send lfork,{:pickup,self()}*/
97          msg_1.m1.data2 = __pid;/* 31*/ /* send lfork,{:pickup,self()}*/
98          __PICKUP!!lfork,PICKUP,msg_1;/* 31*/ /* send lfork,{:pickup,self()}*/
99          }
100         int __ret_placeholder_2;/* 32*/ /* wait()*/
101         run wait(ret2,__pid,1);/* 32*/ /* wait()*/
102         ret2?__ret_placeholder_2;/* 32*/ /* wait()*/
```

```
103         printf("lfork\n");
104         atomic {
105         MessageList msg_2;/* 35*/ /* send rfork,{:pickup,self()}*/
106         msg_2.m1.data2 = __pid;/* 35*/ /* send rfork,{:pickup,self()}*/
107         __PICKUP!!rfork,PICKUP,msg_2;/* 35*/ /* send rfork,{:pickup,self()}*/
108         }
109         int __ret_placeholder_3;/* 36*/ /* wait()*/
110         run wait(ret3,__pid,1);/* 36*/ /* wait()*/
111         ret3?__ret_placeholder_3;/* 36*/ /* wait()*/
112         printf("rfork\n");
113         atomic {
114         MessageList msg_3;/* 40*/ /* send table,{:leave,self()}*/
115         msg_3.m1.data2 = __pid;/* 40*/ /* send table,{:leave,self()}*/
116         __LEAVE!!table,LEAVE,msg_3;/* 40*/ /* send table,{:leave,self()}*/
117         }
118         int __ret_placeholder_4;/* 41*/ /* wait()*/
119         run wait(ret4,__pid,1);/* 41*/ /* wait()*/
120         ret4?__ret_placeholder_4;/* 41*/ /* wait()*/
121         atomic {
122         MessageList msg_4;/* 42*/ /* send lfork,{:putdown,self()}*/
123         msg_4.m1.data2 = __pid;/* 42*/ /* send lfork,{:putdown,self()}*/
124         __PUTDOWN!!lfork,PUTDOWN,msg_4;/* 42*/ /* send lfork,{:putdown,self()}*/
125         }
126         int __ret_placeholder_5;/* 43*/ /* wait()*/
127         run wait(ret5,__pid,1);/* 43*/ /* wait()*/
128         ret5?__ret_placeholder_5;/* 43*/ /* wait()*/
129         atomic {
130         MessageList msg_5;/* 44*/ /* send rfork,{:putdown,self()}*/
131         msg_5.m1.data2 = __pid;/* 44*/ /* send rfork,{:putdown,self()}*/
132         __PUTDOWN!!rfork,PUTDOWN,msg_5;/* 44*/ /* send rfork,{:putdown,self()}*/
133         }
134         int __ret_placeholder_6;/* 45*/ /* wait()*/
135         run wait(ret6,__pid,1);/* 45*/ /* wait()*/
136         ret6?__ret_placeholder_6;/* 45*/ /* wait()*/
137         atomic {
138         MessageList msg_6;/* 47*/ /* send coordinator,{:done}*/
139         __DONE!!coordinator,DONE,msg_6;/* 47*/ /* send coordinator,{:done}*/
140         }
141         atomic{
142             if
143             :: __ret_f -> ret!0;
144             :: else -> skip;
145             fi;
146         }
147     }
148
149     proctype wait (chan ret;int __pid;int __ret_f) {
150         atomic{
151             if
152             :: __pid == - 1 -> __pid = _pid;
153             :: else -> skip;
154             fi;
155         }
156         MessageList rec_v_2;/* 52*/ /* receive do*/
157         do/* 52*/ /* receive do*/
158           :: __OK??eval(__pid),OK,rec_v_2 -> /* 53*/ /* {:ok} -> :ok*/
159             break;
160         od;
161         atomic{
162             if
163             :: __ret_f -> ret!0;
164             :: else -> skip;
165             fi;
```

```
166         }
167     }
168
169     proctype start_fork (chan ret;int __pid;int __ret_f) {
170     chan ret1 = [1] of { int };/* 64*/ /* {:rphil,rphil} -> fork_loop(0,lphil,rphil)*/
171         atomic{
172             if
173             :: __pid == - 1 -> __pid = _pid;
174             :: else -> skip;
175             fi;
176         }
177         MessageList rec_v_3;/* 61*/ /* receive do*/
178         do/* 61*/ /* receive do*/
179           :: __LPHIL??eval(__pid),LPHIL,rec_v_3 -> /* 0*/
180             int lphil;/* 0*/
181             lphil = rec_v_3.m1.data2;/* 0*/
182             MessageList rec_v_4;/* 63*/ /* receive do*/
183             do/* 63*/ /* receive do*/
184               :: __RPHIL??eval(__pid),RPHIL,rec_v_4 -> /* 0*/
185                 int rphil;/* 0*/
186                 rphil = rec_v_4.m1.data2;/* 0*/
187                 int __ret_placeholder_1;/* 64*/ /* {:rphil,rphil} ->
                        fork_loop(0,lphil,rphil)*/
188                 run fork_loop(0,lphil,rphil,ret1,__pid,1);/* 64*/ /* {:rphil,rphil} ->
                        fork_loop(0,lphil,rphil)*/
189                 ret1?__ret_placeholder_1;/* 64*/ /* {:rphil,rphil} ->
                        fork_loop(0,lphil,rphil)*/
190                 break;
191             od;
192             break;
193           od;
194         atomic{
195             if
196             :: __ret_f -> ret!0;
197             :: else -> skip;
198             fi;
199         }
200     }
201
202     proctype fork_loop (int allocated;int lphil;int rphil;chan ret;int __pid;int __ret_f)
            {
203     chan ret1 = [1] of { int };/* 77*/ /* fork_loop(1,lphil,rphil)*/
204     chan ret2 = [1] of { int };/* 80*/ /* fork_loop(2,lphil,rphil)*/
205     chan ret3 = [1] of { int };/* 89*/ /* fork_loop(0,lphil,rphil)*/
206         atomic{
207             if
208             :: __pid == - 1 -> __pid = _pid;
209             :: else -> skip;
210             fi;
211         }
212         if
213         :: (allocated == 0) -> /* 0*/
214             MessageList rec_v_5;/* 73*/ /* receive do*/
215             do/* 73*/ /* receive do*/
216             :: __PICKUP??eval(__pid),PICKUP,rec_v_5 -> /* 0*/
217                 int phil;/* 0*/
218                 phil = rec_v_5.m1.data2;/* 0*/
219                 if
220                 :: (phil == lphil) -> /* 0*/
221                     atomic {
222                     MessageList msg_0;/* 76*/ /* send phil,{:ok}*/
223                     __OK!!phil,OK,msg_0;/* 76*/ /* send phil,{:ok}*/
224                     }
```

```
225                 int __ret_placeholder_1;/* 77*/ /* fork_loop(1,lphil,rphil)*/
226                 run fork_loop(1,lphil,rphil,ret1,__pid,1);/* 77*/ /*
                        fork_loop(1,lphil,rphil)*/
227                 ret1?__ret_placeholder_1;/* 77*/ /* fork_loop(1,lphil,rphil)*/
228             :: else ->
229                 atomic {
230                 MessageList msg_1;/* 79*/ /* send phil,{:ok}*/
231                 __OK!!phil,OK,msg_1;/* 79*/ /* send phil,{:ok}*/
232                 }
233                 int __ret_placeholder_2;/* 80*/ /* fork_loop(2,lphil,rphil)*/
234                 run fork_loop(2,lphil,rphil,ret2,__pid,1);/* 80*/ /*
                        fork_loop(2,lphil,rphil)*/
235                 ret2?__ret_placeholder_2;/* 80*/ /* fork_loop(2,lphil,rphil)*/
236             fi;
237             break;
238           :: __TERMINATE??eval(__pid),TERMINATE,rec_v_5 -> /* 82*/ /* {:terminate} ->
                    */
239             break;
240         od;
241     :: else ->
242         MessageList rec_v_6;/* 86*/ /* receive do*/
243         do/* 86*/ /* receive do*/
244         :: __PUTDOWN??eval(__pid),PUTDOWN,rec_v_6 -> /* 0*/
245             phil = rec_v_6.m1.data2;/* 0*/
246             atomic {
247             MessageList msg_2;/* 88*/ /* send phil,{:ok}*/
248             __OK!!phil,OK,msg_2;/* 88*/ /* send phil,{:ok}*/
249             }
250             int __ret_placeholder_3;/* 89*/ /* fork_loop(0,lphil,rphil)*/
251             run fork_loop(0,lphil,rphil,ret3,__pid,1);/* 89*/ /*
                    fork_loop(0,lphil,rphil)*/
252             ret3?__ret_placeholder_3;/* 89*/ /* fork_loop(0,lphil,rphil)*/
253             break;
254         :: __TERMINATE??eval(__pid),TERMINATE,rec_v_6 -> /* 90*/ /* {:terminate} -> */
255             break;
256         od;
257     fi;
258     atomic{
259         if
260         :: __ret_f -> ret!0;
261         :: else -> skip;
262         fi;
263     }
264 }
265
266 active proctype start () {
267 chan ret1 = [1] of { int };
268 chan ret2 = [1] of { int };
269 chan ret3 = [1] of { int };
270     int __pid = 0;
271     atomic{
272         if
273         :: __pid == - 1 -> __pid = _pid;
274         :: else -> skip;
275         fi;
276     }
277     int n = 4;
278     int table;
279     atomic {
280         table = run table_loop(ret1,- 1,0);/* 104*/ /* table =
                spawn(Table,:table_loop,[])*/
281     }
282     int forks;
```

```
283         __get_next_memory_allocation(forks);
284         for(__dummy_iterator : 1 .. n) {/* 106*/ /* forks = for _ < - 1..n do*/
285             int __tmp;
286             atomic {
287                 __tmp = run start_fork(ret2,- 1,0);/* 107*/ /* spawn(Fork,:start_fork,[])*/
288             }
289             __list_append(forks,__tmp);
290         }
291         int phils;
292         __get_next_memory_allocation(phils);
293         int i;
294         for(i : 1 .. n) {/* 110*/ /* phils = for i < - 1..n do*/
295             int __tmp;
296             atomic {
297                 __tmp = run start_phil(__pid,ret3,- 1,0);/* 111*/ /*
                        spawn(Philosopher,:start_phil,[self()])*/
298             }
299             __list_append(phils,__tmp);
300         }
301         int j;
302         j = n - 1;
303         for(i : 0 .. j) {/* 115*/ /* for i < - 0..j do*/
304             int phil;
305             phil = __list_at(phils,i)
306             int lfork;
307             lfork = __list_at(forks,i)
308             int r_i;
309             r_i = (i + 1) % (n);/* 118*/ /* r_i = rem(i + 1,n)*/
310             int rfork;
311             rfork = __list_at(forks,r_i)
312             atomic {
313             MessageList msg_0;/* 120*/ /* send phil,{:bind,table,lfork,rfork}*/
314             msg_0.m1.data2 = table;/* 120*/ /* send phil,{:bind,table,lfork,rfork}*/
315             msg_0.m2.data2 = lfork;/* 120*/ /* send phil,{:bind,table,lfork,rfork}*/
316             msg_0.m3.data2 = rfork;/* 120*/ /* send phil,{:bind,table,lfork,rfork}*/
317             __BIND!!phil,BIND,msg_0;/* 120*/ /* send phil,{:bind,table,lfork,rfork}*/
318             }
319             atomic {
320             MessageList msg_1;/* 121*/ /* send lfork,{:lphil,phil}*/
321             msg_1.m1.data2 = phil;/* 121*/ /* send lfork,{:lphil,phil}*/
322             __LPHIL!!lfork,LPHIL,msg_1;/* 121*/ /* send lfork,{:lphil,phil}*/
323             }
324             atomic {
325             MessageList msg_2;/* 122*/ /* send rfork,{:rphil,phil}*/
326             msg_2.m1.data2 = phil;/* 122*/ /* send rfork,{:rphil,phil}*/
327             __RPHIL!!rfork,RPHIL,msg_2;/* 122*/ /* send rfork,{:rphil,phil}*/
328             }
329         }
330         for(i : 1 .. n) {/* 125*/ /* for i < - 1..n do*/
331             MessageList rec_v_7;/* 126*/ /* receive do*/
332             do/* 126*/ /* receive do*/
333             :: __DONE??eval(__pid),DONE,rec_v_7 -> /* 127*/ /* {:done} -> :ok*/
334                 break;
335             od;
336         }
337         printf("All philosophers have finished eating!\n");
338         atomic {
339             __list_ptr_old = __list_ptr;
340             __list_ptr = 0;
341             __list_ptr_new = 0;
342             do
343             :: __list_ptr >= LIST_LIMIT || __list_ptr_new >= LIST_LIMIT ->
344                 __list_ptr = __list_ptr_old;
```

```
345                 break;
346             :: else ->
347                 if
348                 :: LIST_ALLOCATED(forks,__list_ptr) ->
349                     int fork;
350                     fork = LIST_VAL(forks,__list_ptr);
351                     atomic {
352                     MessageList msg_3;/* 133*/ /* send fork,{:terminate}*/
353                     __TERMINATE!!fork,TERMINATE,msg_3;/* 133*/ /* send fork,{:terminate}*/
354                     }
355                     ;
356                     __list_ptr_new++;
357                     __list_ptr++;
358                 :: else -> __list_ptr++;
359                 fi
360             od
361         }
362         atomic {
363         MessageList msg_4;/* 135*/ /* send table,{:terminate}*/
364         __TERMINATE!!table,TERMINATE,msg_4;/* 135*/ /* send table,{:terminate}*/
365         }
366     }
```

Listing A.7: Dining Philosophers Promela translation.

A.6 Raft

A.6.1 Raft Consensus in Elixir

```
1   defmodule RaftNode do
2   @spec start_node(integer(), integer(), integer()) :: :ok
3   def start_node(id, n_peers, client) do
4     peers = receive do
5       {:bind, peers} -> peers
6     end
7     term = 10 * id
8     node_handler(id, peers, n_peers, 0, term, 0, client)
9   end
10
11  @spec node_handler(integer(), list(), integer(), atom(), integer(),
        integer(), integer()) :: :ok
12  defp node_handler(id, peers, n_peers, state, term, vote_count, client)
      do
13    receive do
14      {:request_vote, candidate_term, candidate_id, reply_to} ->
15        if candidate_term > term do
16          send(reply_to, {:vote_granted, id})
17          node_handler(id, peers, n_peers, 0, candidate_term, vote_count,
                client)
18        else
19          node_handler(id, peers, n_peers, state, term, vote_count,
                client)
20        end
21
22      {:vote_granted, _voter_id} ->
23        new_vote_count = vote_count + 1
24        if state == 1 and new_vote_count >= n_peers / 2 + 1 do
25          send(client, {:elected, term})
```

```
26            for peer <- peers do
27              send(peer, {:append_entries, term, id}) # Send log here
28            end
29            node_handler(id, peers, n_peers, 2, term, new_vote_count,
                  client)
30          else
31            node_handler(id, peers, n_peers, state, term, new_vote_count,
                  client)
32          end
33
34        {:start_election} ->
35          for peer <- peers do
36            send(peer, {:request_vote, term + 1, id, self()})
37          end
38          node_handler(id, peers, n_peers, 1, term + 1, 0, client)
39
40        {:terminate} ->
41          IO.puts("Terminating node")
42
43        after 1000 ->
44          send self(), {:start_election}
45          node_handler(id, peers, n_peers, state, term, 0, client)
46      end
47    end
48  end
49
50  defmodule Client3 do
51    @init true
52    @spec start() :: :ok
53    @ltl """
54    !<>[](elected_term == previously_elected_term)
55    """
56    def start do
57      # follower -> 0, candidate -> 1, leader -> 2
58      n_nodes = 3
59      n_peers = n_nodes - 1
60      rounds = 2
61      previously_elected_term = -1
62      elected_term = 0
63      nodes = for id <- 1..n_nodes do
64        spawn(RaftNode, :start_node, [id, n_peers, self()])
65      end
66
67      for p_id <- nodes do
68        send(p_id, {:bind, nodes})
69      end
70
71      for _ <- 1..rounds do
72        {elected_term, previously_elected_term} = receive do
73          {:elected, term} ->
74            IO.puts("Node elected")
75            {term, elected_term}
76        end
77      end
78
79      for p_id <- nodes do
80        send(p_id, {:terminate})
81      end
```

```
82    end
83  end
```