Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Automatic Verification of Actor-Based Systems DRAFT

Author:
Matthew Neave

Supervisor:
Dr. Naranker Dulay

Second Marker:
TBD

May 22, 2024

Contents

List of Figures

Listings

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. Modern programming languages have been developed to support distributed algorithms that rely on message-passing as a means of communication between sequential nodes executing in parallel. Common message-passing abstractions involve the use of channels (e.g. Go [10]) or actors [30] (e.g. Erlang [13]). Message-passing abstractions can be simple and more natural to reason about than a common alternative in shared-memory concurrency, however, it can also become more difficult to verify a program implements a given specification.

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 power 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 such as Dafny [18] solve this issue by directly integrating Floyd-Hoare style logic verification alongside the implementation.

Elixir [11] is a functional programming language built on top of Erlang 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.

This report discusses the automatic modelling of 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. An even 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. This project sets out to accomplish the following objectives:

- Design novel modelling techniques for actor-based systems.
- Determine how specifications can be succinctly specified for actor-based systems.

- Design a toolkit for automation of the model checking and verification processes.
- Apply the aforementioned techniques and tooling to real-world systems using Elixir as an implementation of the actor model.

Chapter 2

Background

2.1 Communicating Sequential Processes

Communicating Sequential Processes (CSP) was discovered by Tony Hoare, it provides us with 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. This section will give a brief introduction to some process algebra introduced to model some simple parallel processes.

CSP defines processes and events. The alphabet of a process, αP is the set of all events. For example, the alphabet of a student process S could consist of two events.

$$\alpha S = \{study, sleep\}$$

The process with the alphabet A which never engages in the events of A is called $STOP_A$. $STOP$ can be considered a constant, and it is used to define a process that never engages in any available action. $STOP$ is used to describe behaviour of a broken object, such that the object terminates unsuccessfully. Similarly, $SKIP_A$ is defined as a process which does nothing but terminates successfully. We can now construct a sequence of events for a process that takes three actions and then breaks.

$$(study \rightarrow sleep \rightarrow study \rightarrow STOP_{\alpha S})$$

Similarly, a sequence of events for a process that takes an action and then terminates successfully.

$$(study \rightarrow SKIP_{\alpha S})$$

Using the same alphabet, we now define two simple processes modeling a student L and a strict teacher T, who never accepts students sleeping.

$$L = (study \rightarrow sleep \rightarrow L)$$
$$T = (study \rightarrow study \rightarrow T)$$

Note that both processes are recursively defined, hence a valid `trace` for L could be $\langle study, sleep, study, sleep \rangle$.

We can now introduce the process algebra for concurrency, using the parallel composition operator ('||'). To help reason about concurrent processes, we also introduce a fixed-point constructor, $\mu X \bullet F(X)$, to define anonymous recursive processes. This now lets us denote a process that behaves like a system of composed processes, where both processes have the same algebra αS.

$$(T \,||\, L)$$

Using the definitions for T and L, and the recursive process definition, we have.

$$(T \,||\, L) = (study \rightarrow STOP)$$

This is the composition of both processes. As both begin with a `study` event, so does the composition. However, after `study` each component process is prepared to take different events, as these are different, the processes can not agree on what action to take next. The resulting `STOP` is known as a deadlock. Alternatively, had the student and teacher processes been defined with

behaviour composed without deadlock, we could describe that behaviour with process algebra. In this example, the student and teacher have multiple actions that can be taken, denoted by the choice, |, notation.

$$\alpha = \{learn, revise, exam\}$$
$$L = (learn \rightarrow L \,|\, exam \rightarrow L \,|\, revise \rightarrow exam \rightarrow L)$$
$$T = revise \rightarrow (exam \rightarrow T \,|\, learn \rightarrow T)$$

$$(T \,||\, L) = \mu X \bullet (revise \rightarrow exam \rightarrow X)$$

The composition can be described as a single process. Also, note the use of recursion with the fixed-point operator $\mu X \bullet F(X)$ [27, p.74], where X marks recursion under the composed process. Under this model, to compose two processes, we require simultaneous participation of the same event from both processes. Therefore, each event in the composed process can be attributed to events in both individual participants.

We extend this notion to concurrency by considering separate alphabets for each process. Again, take our student L and teacher T. If the alphabets of both processes differ, an event in the alphabet of L that is not in the alphabet of T is of no concern to T, thus becomes a valid event in the composition of the processes. Similarly to before, events that are in both alphabets can only be composed if they can be simultaneously taken by both processes individually.

$$\alpha L = \{study, exam\}$$
$$\alpha T = \{teach, exam\}$$
$$L = study \rightarrow exam$$
$$T = teach \rightarrow exam$$

$$(T \,||\, L) = \mu X \bullet ((study \rightarrow teach | teach \rightarrow study) \rightarrow exam)$$

Finally, we will briefly look at how Hoare modeled communication between processes. Hoare designed communication over channels. A pair $c.v$ represents communication taking place over a channel c and v is the value of a message being passed. Hoare describes the set of all messages that a process P can communicate on channel c as $\{v \,|\, c.v \in \alpha P\}$.

$$\alpha c(P) = \{v \,|\, c.v \in \alpha P\}$$

Functions to extract the channel and message components from the pair $c.v$ are also defined.

$$channel(c.v) = c$$
$$message(c.v) = v$$

With this understanding, we can finally model sending and receiving messages to channels. Given a process P and a value $v \in \alpha c(P)$, a process can output v on the channel c using the ! operator (similar to sending a message to a channel in GO [10]).

$$(c!v \rightarrow P)$$

Similarly, we can read messages from channels (receive a message) using the ? operator. A process can input any value x on the channel c, and then behave under $P(x)$.

$$(c?x \rightarrow P(x))$$

That concludes a brief overview of the process algebra proposed by Tony Hoare. We saw how event sequencing constructs processes, how processes can be composed and the special communication event $c.v$ which allows message-passing over channels. Tools have been built from similar syntax and concepts, for example, Promela 2.4.2, which models channels and messages with a similar approach.

2.2 Concurrency

Concurrency introduces the ability 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 distinct 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 [**?**], Raft [**?**] and Dining Philosophers [**?**]. 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 behaviors 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.

2.2.1 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 modeling 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), or 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.

TODO DIAGRAM OF GLOBALLY, EVENTUALLY, UNTIL

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

We can also explore what it means for the formula to hold in a given state of a model.

TODO DIAGRAM OF MODEL, WITH SOME FORMULA FROM ABOVE, COMPARING THE TRUTH IN A STATE AND VALIDITY IN A MODEL

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 2.3.

2.2.2 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 construct what it means for a system to be correct, through the evaluation of these temporal formulae.

2.2.3 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 favor the scheduling of process A while never progressing with process B. We will discuss two flavors 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 intuitively understand weak fairness as the property that 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.

2.3 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 interleavings 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. By instead abstracting our program as a model, and verifying the correctness using a model checker, we could exhaustively check all possible states (interleavings 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 just 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.

2.3.1 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.

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 is more interesting than PAT is that it no longer relies on process algebra. The model checker is capable of being run directly on a subset of C programs, 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 labeled 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. A strong driving factor in why developers choose to design systems in Elixir is its concurrent capabilities.

PRISM

PRISM is a probabilistic model checker, a tool for formal modelling and analysis of systems that exhibit random behavior or probabilistic behavior [4]. It has been used to analyse systems implementing random distributed algorithms.

TLC

In 1980, Leslie Lamport discovered 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.

```
---------------------- MODULE HourClock ----------------------
EXTENDS Naturals
VARIABLE hr
HCini == hr \in (1 .. 12)
HCnxt == hr' = IF hr # 12 THEN hr + 1 ELSE 1
HC == HCini /\ [][HCnxt]_hr
--------------------------------------------------------------
THEOREM HC => []HCini
==============================================================
```

Figure 2.1: An example TLA+ Specification for an HourClock [6]

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 labeled 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.

```
1 mtype = { HELLO };
2 chan channel = [10] of { mtype };
3
4 init {
5     channel ! HELLO;
6 }
```

Listing 2.1: Example of a Promela specification that enqueues a message in a channel

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.

2.3.2 Theorem Proving

Theorem proving is another process to verify programs. 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.

2.3.3 Hoare Logic

Hoare Logic was discovered in 1969 by Tony Hoare [21]. Hoare Logic defines the Hoare Triple, an essential idea in describing how code execution changes the state of a computation. A Hoare Triple

is composed of a pre-condition assertion P, a post-condition assertion Q, and a command C.

$$\{P\}C\{Q\}$$

Note how the postcondition is the same as the precondition for this command.

Hoare Logic provides axioms and inference rules required to construct a simple imperative programming language. If P holds in the given state and C terminates, then Q will hold after. Below is an example of a simple Hoare Triple for the `skip` command, which leaves the program state unchanged.

$$\{P\}\text{skip}\{P\}$$

Hoare describes many more rules that allow for assignment, composition, consequence and so forth. These rules have led to the development of modern-day theorem provers, such as Z3, which will detailed more later.

To help understand, we show a concrete example of a Hoare Triple. In this example, the pre-condition P and post-condition Q represent the known program state for a variable, x. We informally describe a command C as an assignment to x that modifies the known state of the program.

$$\{x \to 1\}\ \text{x} := \text{x} + 1\ \{x \to 2\}$$

To formalise this notation, we should define rules for commands, but for brevity, these have been omitted.

2.4 Existing 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 tools like such provide Elixir developers better guarantees their code is correct, it does not verify the correctness of a system as a whole.

2.4.1 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.

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 instead of necessarily correct code. 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 postcondition, $\{Q\}$, will hold. In Dafny, the language reserves the keywords `requires` and `ensures` for pre and postconditions. Listing 2.2 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 postcondition 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 2.2: Example of a method in Dafny.

Listing 2.2 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 postconditions over collections of objects, such as sets and arrays. Listing 2.3 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 2.3: `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 modeling 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 2.4 shows an example Boogie program, defining a single procedure, `add`, that represents the translated code from the Dafny example in listing 2.2. 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 2.4: An example Boogie IVL program.

2.4.2 Promela

Promela is the verification modeling language used by the Spin model checker, to specify concurrent processes modeling distributed systems [9]. This section will discuss some of the core features that allow systems to be modeled 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.

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 if 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.

$$\text{int a} = 2;$$

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.

$$\begin{aligned}&\text{if}\\&:: \text{exp\_1} \text{ -> } \ldots\\&:: \text{exp\_2} \text{ -> } \ldots\\&\text{fi}\end{aligned}$$

Repetition can be achieved either through the `do` construct, through the use of labels or with `for` loops.

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 2.5: Defining and spawning processes in Promela.

Multiple processes can be declared to run in the initial state by marking them as `active` processes. 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.

Channels

The final concept to briefly discuss is the asynchronous communication primitive, channels. Recall Hoare's definition of channels 2.1, defining a channel c that can input and output values. Promela echos this definition, allowing 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 purpose 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.

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

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

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

Where the variable x stores the resulting `int` assuming the first message in the channel is of type `BROADCAST`.

Summary

This basic introduction to the syntax of the Promela modelling language aims to make the reader familiar with the syntax and control 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.

2.5 Summary

This chapter has provided an overview of core concepts related to concurrent programs and verification of them. We saw process algebra that can be used to model and reason about concurrent processes, as well as Hoare Logic and its definition of the Hoare Triple as a fundamental property in

verification. We also looked at applications based on this theory, such as model checkers, theorem provers and programming languages. Much work related to the topic of verifying programming languages was explored, but importantly, we learned about SPIN, and how concurrent programs can be modeled in Promela to be model-checked for deadlocks, race conditions and incompleteness. We also learned about Boogie, the intermediate verification language that can verify programmatic assumptions using Z3. The next chapter will discuss the Elixir programming language.

Chapter 3

Elixir TODO refactor

Elixir is a dynamic, functional language for building scalable and maintainable applications [11]. Elixir programs run on the BEAM virtual machine[12], which is also used to run the Erlang programming language [13]. Elixir was designed by José Valim and first released in 2012. Elixir is built on top of Erlang and hence inherits many of the abstractions designed for building distributed systems. This chapter aims to give a brief overview of Elixir, it is not a complete guide but rather aims to give non-Elixir programmers a basic understanding of the language.

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.

Elixir has begun to see use 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 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 not an operating system process. 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 3.1: An example of spawn/1 and spawn/4 in Elixir for spawning a new lightweight process and a new Elixir node

3.1 Shared Memory and Message Passing

Two key concepts in inter-process communication (IPC) are shared memory models and message-passing models. They are two techniques used to allow processes to send signals or share data between each other. In a shared memory model, a shared memory region is established in which multiple processes can read and write. Figure 3.1 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].

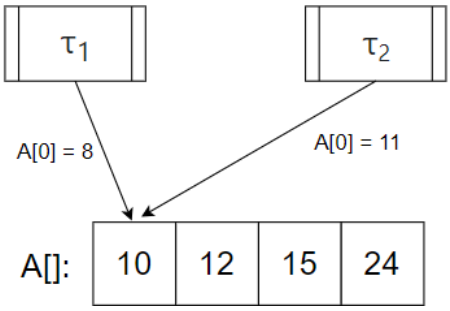

Figure 3.1: An example of two processes writing to a shared in-memory array

Elixir instead uses a message-passing model for IPC. More specifically, Elixir uses an actor-based model, where each process (actor) has its state and a message box to receive messages from other actors. Actors are responsible for sending a finite number of messages to other actors, spawning new actors and changing their behaviour based on the handling of messages received in the mailbox. Figure 3.2 shows an example of how actors behave. The mailbox is not necessarily first in, first out (FIFO) but often implementations tend to be.

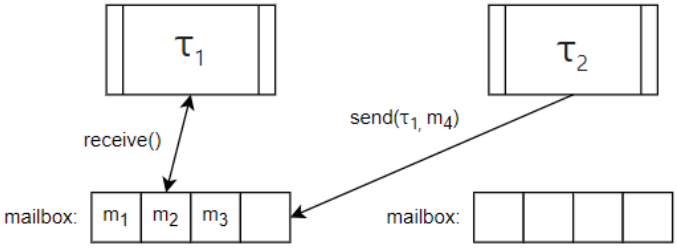

Figure 3.2: An example of actors sending and receiving messages under the actor model

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 a given 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 3.2: An example of spawn/1 and spawn/4 in Elixir for spawning a new lightweight process and a new Elixir node

3.2 Quote and Unquote

The quote and unquote constructs in Elixir give us a deeper insight into how the programming language is implemented. Elixir is fundamentally made of tuples with three elements consisting of an atom[1] that identifies the tuple, an array of metadata and finally the data. For example, the function call `sum(1, 2)` would be represented by the tuple `(:sum, [], [1, 2])` and similarly, the variable `total` would be represented by the tuple `(:total, [], Elixir)`. Using these building blocks, Elixir can begin to build what is known as a quoted expression, which is a nesting of tuples

[1] In Elixir, atoms are named constants, whose values are their own name. They can be identified by a preceding colon, for example, `:hello`.

in a tree-like structure. In many other programming languages, this tree-like structure is referred to as an abstract syntax tree (AST).

The quote and unquote constructs allow us to transition between Elixir syntax and quoted expressions. Using the `quote/2`[2] macro on an Elixir block, such as `quote do: sum(1, 2)` will return the quoted expression representing the block, in this case, `(:sum, [], [1, 2])`. Similarly, the `unquote/1` macro can be used within a quoted expression to inject code directly into the underlying expression. Figure 3.3 shows a small example of how unquote can be applied within a quoted expression to inject a variable.

```
x = 2
quote do: sum(1, unquote(x))
```

Listing 3.3: Elixir example of `quote/2` and `unquote/1`.

For ease of reading, we will use the terms quoted-expression and AST interchangeably for the remainder of the report.

3.3 Metaprogramming

Metaprogramming is a technique that allows developers to write a program that outputs another program. It means a program can be designed to read or transform other programs. In Elixir, metaprogramming is often used to extend the language by directly modifying the generated quoted expressions by a program. This is achieved through the quote and unquote constructs alongside macros. Macros allow for transforming code and expanding a module.

In Elixir, `defmacro/2` is used to define new macros, which itself is a macro. Macros receive quoted expressions as arguments and typically inject these expressions into code before returning another quoted expression. Listing 3.4 introduces how `defmacro/2` can be used to define the `unless/2` macro used in the standard library. Unless is the opposite of an `if/2` statement, it will execute an expression if a conditional check evaluates to false.

```
defmacro unless(clause, do: expression) do
  quote do
    if(!unquote(clause), do: unquote(expression))
  end
end
```

Listing 3.4: Elixir example of the `unless/2` macro as defined in the standard library [19].

Macros are both lexical and explicit. That means it is impossible to inject macros globally and it is impossible to run macros without explicit invocation. By leveraging the use of functions, quoted expressions and macros, we can begin to develop a domain-specific language (DSL). For example, constructing a DSL that overrides the standard implementations for many Elixir constructs in a style that makes verifying the correctness of Elixir programs more trivial. By default, Elixir is very difficult to verify. Elixir provides an ExUnit module, with an `assert/1` macro which could be used for loop invariants, preconditions and postconditions but doesn't support an approach that favours writing verification-aware code. As many Elixir programs are concurrent, and as Elixir uses the actor model, verifying an arbitrary Elixir program that has not been restricted or extended using macros is a challenge.

Another useful feature often associated with the development of DSLs in Elixir is attributes. Attributes can be used to store additional information, as a temporary storage. Attributes also work as constants, or simply to annotate code which can be useful for other developers or the virtual machine. Listing 3.5 shows a basic example of annotating a function with an attribute.

```
@doc "Calculate the sum of two numbers, x and y"
def sum(x, y) do
    x + y
```

[2] In Elixir, it is common to name functions or macros alongside their number of arguments. The function `spawn/1` refers to the function spawn, with 1 argument.

```
4 end
```

Listing 3.5: Example use of attributes in Elixir.

3.4 Additional Contructs

Various other constructs in Elixir are useful to be aware of to understand the format of quoted expressions.

3.4.1 Return Values and Types

Elixir is a dynamically typed language, hence any introduced type specifications are never used by the compiler in optimisations or type-checks, however, using annotations[3] Elixir does support type specification which can be relevant for documentation and additional tooling. Unlike many imperative programming languages where function return values must be explicitly defined using a keyword such as `return`, Elixir functions simply return the evaluation of a statement. This could also be the final statement if many statements are sequentially executed in a block.

3.4.2 Charlists

Elixir introduces linked lists as a data structure to store elements. Elixir lists are similar to other programming languages, where they are displayed as comma-separated values enclosed in square braces. If a list is made exclusively of non-negative integers, where every integer has a Unicode code point, then the list may be interpreted as a charlist. If the list contains only printable ASCII characters, then it is often stored and displayed in ASCII format. For example, the list `[97, 98, 99]` would be stored in the AST as `'abc'`.

3.5 Summary

In this chapter, we learned about Elixir, the programming language built on top of Erlang and we explored so basic approaches to designing concurrent systems with 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.

[3] Annotations are used to add additional metadata to code. They are prefixed with an asperand, for example, `@type` or `@doc`.

Chapter 4

Veriflixir

Veriflixir is the main project contribution. The Veriflixir toolchain supports the simulation and verification of a set of Elixir programs. This set is named LTLixir and is detailed in section 4.1. This chapter aims to inform the reader of the constructs defined in LTLixir and how Veriflixir can be used to reason about them. 4.1 introduces the LTLixir language and its constructs. 4.2 provides an example of specifying a verifiable system and how Veriflixir can be used to detect violations of a specification. The subsequent subsections provide further details of more interesting features of LTLixir, 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 Veriflixir. Primarily, LTLixir is a subset of Elixir supporting both sequential and concurrent execution. This subset is expressive enough to well-known distributed algorithms such as basic paxos [?] and the alternating-bit protocol [?]. LTLixir extends Elixir with constructs for specifying temporal properties, specifically LTL properties (where LTLixir derives its name) as well as Floyd-Hoare style logic 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 an LTLixir program, and show how we can verify the properties of the program using Veriflixir. 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.

To begin, the server spawns a single client process, which writes to stdio. To ensure correctness when verifying properties of the system, we remove ambiguity by being particular in our naming of the functions `start_server` and `start_client`. Notice we could name both functions `start`, but this ambiguity can make it more difficult to digest a trail produced by Veriflixir. With the system implemented, we must now declare an entry point to the system that Veriflixir will use to

begin verification. For this example, we can define `Server.server_start` as the entry point using an attribute `vae_init`.

```
@vae_init true
def start_server do
    client = spawn(Client, :start_client, [])
end
```

Listing 4.2: Declaring an entry point to the system.

Although we set the attribute `vae_init` to `true`, note it is not required that other functions are set to `false`, this is already implied. With an entry point specified, we can begin using the available tools. By default, Veriflixir reports the presence deadlocks and livelocks in the system. When specifying systems in LTLixir, 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. For example, using mix [?].

```
$ mix run -e Server.start_server
Generated app
Client booted
```

More interestingly, we can now use Veriflixir before the Erlang Run-Time System (ERTS) to verify the system adheres to our specification. With no additional properties defined, by running Veriflixir we are ensuring that every possible execution results in a program termination. The presence of a deadlock or livelock will be reported. We can run the Veriflixir executable by passing optional arguments as the path to the specification file. For example, we use the simulator flag $-s$ to run a single simulation of the system.

```
$ ./veriflixir -s basic_example.ex
Client booted
```

Alternatively, we can use the $-v$ flag to run the verifier on the specification.

```
$ ./veriflixir -v basic_example.ex
Model checking ran successfully. 0 error(s) found.
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 Veriflixir to detect a deadlock in the system. By default, the verifier will detect the presence of deadlocks and livelocks 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. To demonstrate this, we modify the existing client and server by introducing a circular wait. The server will now spawn the process, and expect to receive a message from the client, meanwhile, the client will expect to receive a message from the server. The resulting server and client processes are modeled below.

```
defmodule Server do
@vae_init true
def start_server do
  client = spawn(Client, :start_client, [])
  receive do
    {:im_alive} -> IO.puts "Client is alive"
  end
end
end

```

```
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 simple 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. Let's take another look at what happens when we execute the program, run a simulation and run the verifier.

```
1  $ mix run -e Server.start_server
2  Generated app
3  Client booted
```

Notice when running the Elixir program, nothing is output to stdio, even though a naive Elixir programmer could think one of the two `IO.puts` statements is executed. Of course, we know this not to be the case but let's compare the outputs from running Veriflixir.

```
1    $ ./veriflixir -s basic_example.ex
2    timeout
```

The simulator terminates, reporting a timeout. Already, running our specification using Veriflixir provides more information than running the Elixir program. Let's now run the verifier.

```
1    $ ./veriflixir -v basic_example.ex
2    Model checking ran successfully. 1 error(s) found.
3    The program likely reached a deadlock. Generating trace.
4    [8] (proc_0) init:4 [receive do]
5    [9] (proc_0) init:4 [receive do]
6    [10] (proc_0) init:5 [{:im_alive} -> IO.puts "Client is alive"]
7    [13] (proc_1) start_client:13 [{:binding} -> IO.puts "Client bound"]
8    <<< END OF TRAIL, FINAL STATES: >>>
9    [14] (proc_1) start_client:13 [{:binding} -> IO.puts "Client bound"]
10   [15] (proc_0) init:5 [{:im_alive} -> IO.puts "Client is alive"]
```

Running the verifier produces much more output. Let's break down step by step the output produced by Veriflixir. The first line of the output informs us that the verifier successfully terminated on the input, along with how many errors were found. If an error is found, Veriflixir will use heuristics to profile the type of error, in this case, it has determined the program likely deadlocked. Once determining the error type, an error trail is produced to debug the source of the error. The underlying model derived from the LTLixir specification does not have a one-to-one mapping to the original Elixir code, hence again heuristics are applied to determine where in the Elixir program the trail is produced from. In this case, we can see reference to `init`, the entry point to the system (annotated previously by `@vae_init`). Alongside the process, we can see a line number referring to a line number in the Elixir file, as well as the line of code the line refers to. With the exception of the system entry point, all other function names are labeled as in the original program, for example, *start_client*. 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 both to process numbers and function call stack depth.

Now we understand how to read a single line of the trail, 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-time Temporal Logic to our systems to allow us to write more interesting LTLixir specifications. Before doing so, we must detour to type specifications. Type specifications had been previously omitted from examples, but they are imperative for the correctness of LTLixir specifications. LTLixir supports some basic types such as `integer()`, `boolean()`, `atom()` and `pid()`. Internally, Veriflixir treats process identifiers as integers, so `integer()` and `pid()` can be used interchangeably in specifications, for the following examples, we refer to process identifiers as integers. Message passing in specifications should be typed using atoms. 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, `false` would be ignored by Veriflixir. In type specifications, message types are typed as `atom()`. The atom `:ok` is reserved to identify non-returning functions. 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 start_server() :: :ok
2    def start_server do
3        ...
4    end
5
6    @spec add(integer(), integer()) :: integer()
7    def add a, b do
8        ...
9    end
```

Listing 4.4: 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.

Let's now 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. To achieve this, we introduce two variables `client_n` and `alive_clients` that will later be used in our temporal specification.

```
1    def start_server do
2        client_n = 3
3        alive_clients = 0
4        for _ <- 1..client_n do
5            client = spawn(Client, :start_client, [])
6            send(client, {:bind, self()})
7        end
8        alive_clients = check_clients(client_n, alive_clients)
9    end
```

The implementation of the client process and the `check_clients/2` function have been redacted, 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 the `@ltl` attribute. The attribute assigns an LTL formula, as a string, to function. The LTL grammar is defined as the following.

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

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

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

Listing 4.5: Example LTL property.

The entire program can be found in appendix **??**. Let us run Veriflixir on the system.

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

Let's update the LTL formula, by replacing `clients_n` with the number 1 ($\Diamond(alive\_clients \equiv 1)$). We run the verifier again.

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

The examples show how LTL properties can be specified and used to verify the system. Given the first LTL property was accepted, we know the formula holds. This is likely because the implementations of the client and `check_clients/2` are correct, but we should also specify properties to check the validity of these functions instead of making this assumption.

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 p as $alive\_clients \equiv client\_n$. The predicates can take the name of any variable but should not reference variables that are already in scope. To help construct our example, let's also define a predicate q and set it to $\neg p$. Using these predicates, we can strengthen our LTL formula to $(q)\mathcal{U}(\Box p)$. Informally, this formula states that there is a moment in time where $alive\_clients \equiv client\_n$, and from that moment onwards this property holds until termination. We can update our `start_server` function to reflect this.

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

4.2.3 Pre- and Post-Conditions

Veriflixir has been designed to target system designs that are bounded in execution. As many real-world systems run long-lived processes, it is important that when writing LTLixir specifications there is a focus on termination conditions. For many distributed algorithms this could be completing a round of communication, reaching a consensus, awaiting a specific message to be received or reaching a stable system state. To aid this, LTLixir supports Floyd-Hoare style 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. The client will receive a `:bind` message from the server, deciding how many rounds to complete and then will recurse until it has completed all the rounds.

```
defmodule Client do
    @spec start_client() :: :ok
    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
    def next_round(server, rounds) do
        send(server, {:im_alive})
        next_round(server, rounds - 1)
    end
end
```

We can again run this with mix, and we can observe that the server terminates. Although our clients did not terminate, there was no indication of this. Processes in Elixir are isolated, so the termination of the server does not imply the termination of the clients (links can resolve this [31]). It has become challenging to determine if our client is truely behaving as intended.

```
$ mix run -e Server.start_server
Generated app
$
```

To help reason about this, we introduce the `defv` macro from the LTLixir specification language. The `defv` macro is used to define pre- and post-conditions on functions. 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. For example, we could define a function `add_positives/2` that takes two strictly positive numbers and sums them.

```
defv add_positives(a, b), pre: a > 0 and b > 0, post: result == a + b do
    ...
end
```

Listing 4.6: Example usage of pre- and post-conditions in LTLixir

With this definition, we gain assurances that the implementation of the function behaves as we expect and that no other function interacts with the function in a manner that violates our expected behavior. Let's take a similar approach with our client to help understand why the program does not terminate.

```
@spec next_round(pid(), integer()) :: :ok
defv next_round(server, rounds), pre: rounds >= 0 do
    ...
end
```

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

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

Veriflixir 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. The depth refers to how many transitions were taken in the execution path before violation, to most users this information is not useful.

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, LTLixir allows us to declare these properties as parameters, in particular, we want to declare concurrency parameters. We define concurrency parameters are 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.

To mark a variable as a parameter, we again make use of attributes. The `@params` attribute takes a tuple of atoms referencing variables in the function scope. For example, `@params :x, :y` declares that x and y are configurable parameters that the verifier will explore. We can apply this definition to our existing server process.

```
@params {:client_n, :number_of_rounds}
def start_server do
  client_n = 3
  number_of_rounds = 2
  predicate p, alive_clients == client_n *
      number_of_rounds
  ...
end
```

Listing 4.7: 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 Veriflixir in parameterized mode. To run the verifier, we can use the $-p$ flag. The verifier will calculate an acceptance confidence, $\alpha \in [0, 1]$, where an α of 1 indicates all configurations were accepted by the verifier and an α of 0 indicates no configurations were accepted. In the case $\alpha < 1$, the verifier will output configurations that lead to violations in the system. Then, the system can run in verification mode to reproduce a trail that results in the violation. To run the parameterized verification, we use the same command but with the $-p$ flag.

```
$ ./veriflixir -p basic_example.ex
Generating models.
Generated 9 models.
Acceptance confidence: 1.
```

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.

```
@params {:number_of_rounds}
def start_server do
```

```
    ...
    alive_clients = if number_of_rounds > 1 do
        check_clients(client_n * number_of_rounds, alive_clients)
    else
        0
    end
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, hence in reality we should use our understanding of the system to determine which parameters to test at a time.

```
$ ./veriflixir -p 3 basic_example.ex‘
Generating models.
Generated 3 models.
Acceptance confidence: 0.6666666666666667.
Violations found in models:
Model with params: {"number_of_rounds": '1'}
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 using the $-v$ flag on this configuration. We also now see our acceptance confidence has dropped below 1, due to an error produced by one of the configurations. It's also useful to note that Veriflixir was executed with $-p3$, this sets the range of values for the parameter, large numbers are not recommended and most users will find 3 sufficient.

4.3 Summary

We have now given a high-level overview of Veriflixir, the verification toolchain capable of verifying Elixir programs written using the LTLixir specification. We saw how to use Veriflixir to simulate executions of the system, verify our system's adherence to a specification and parameterize concurrency parameters for exploration. This chapter also exposed 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 behind Veriflixir.

Chapter 5

Design of Veriflixir

This chapter provides an in-depth insight into the design decisions that were made during the development of Veriflixir and LTLixir. 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 discuss the design of LTLixir, section 5.3 will describe the main techniques Veriflixir applies in the analysis and modeling of a specification. Finally, section 5.4 will describe the derivations of the outputs generated by Veriflixir.

5.1 Veriflixir Toolchain - Mostly diagram todo

Toolchain: LTLixir -> Veriflixir -> Spin -> Veriflixir -> Output |-> BEAM byte-code -> beam

Architecture: Elixir program Elixir parser IR representation Promela writer Instrumentors (modelrunner/modelgenerator) File writer Spin model checker Violationhandlers Trace generator Output

5.2 Specification Language

LTLixir is a specification language that can be used to reason about the time and change of an Elixir system using LTL formulae. An LTLixir specification is the input to Veriflixir and is used to guide the model generation process. Primarily, LTLixir supports a restricted subset of Elixir, with additional constructs to support specification properties. This section will primarily discuss the design decisions behind the additional constructs. For information on how LTLixir is transformed in model generation, see section 5.3.4.

There is only one construct in LTLixir that is imperative to successfully interact with the simulator and verifier. `@vae_init` is an attribute that is assigned a value, $v \in \{true, false\}$. The attribute is treated by the Elixir compiler as a regular attribute, so will not cause issues when running within the ERTS. Importantly, for Veriflixir, it is used to determine an entry point to the system.

All the remaining constructs of the specification are optional to the verification of the system. Similarly to the `@vae_init` attribute, `@spec`, `@ltl` and `@params` are all attributes that are assigned values. The `@spec` is already well-defined in the Elixir language, although it is noted that type specifications do not instrument Elixir programs in any way. Instead, `@spec` can be used by analysis tools that run on Elixir programs. In our case, we reduce the possible type specifications which can be defined. There are two key differences between an Elixir type specification and an LTLixir type specification. Firstly, LTLixir does not support the use of custom types introduced through the `@type` attribute. Only a set of primitive types are accepted within a specification: $\{integer(), boolean(), atom()\}$. The second restriction applied by LTLixir is the return type `:ok`. This atom is reserved in type specifications to mark functions whose return values are never matched on. The actual return type of the function is not relevant in this instance as an agreement is made with the developer that the function will never be matched. The `@ltl` attribute assigns a string value. This string value must follow the imposed rules on LTL formula by LTLixir. Namely, it must follow a formula of the form introduced in section 4.2 with the additional allowance of the Elixir constructs identifiers, boolean operations and binary operations. All the identifiers must be

well-defined within the scope of the function and should not be declared in a nested child scope within the function. The final attribute LTLixir supports is `@params`. `@params` is assigned a tuple of atoms. Similarly to the `@ltl` attribute, the tuple should consist of atom identifiers from the function scope.

The next construct we will discuss is `defv`. The `defv` construct no longer 'annotates' information about the specification of a function, instead it is directly inlined into the function definition. `defv` defines a 'verifiable function'. A verifiable function is a third function type definition alongside `def` and `defp`. Semantically, it acts similar to `def`, as it is a public function that can be accessed from external module calls. The `defv` construct is what Elixir refers to as a macro. A macro is used to extend the Elixir syntax through metaprogramming. The macro introduces two new tuples to the arguments of the quoted expression representing the function. Syntactically, these tuples 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 5.1: Example of the `defv` syntax.

The introduced pre- and post-conditions introduce quoted expressions of the following form.

$$\text{pre-condition} \models \{\text{pre: } \rightsquigarrow \langle \text{condition} \rangle \rightsquigarrow\}$$
$$\text{post-condition} \models \{\text{post: } \rightsquigarrow \langle \text{condition} \rangle \rightsquigarrow\}$$

The `defv` is intended for verification. It also instruments the execution of Elixir programs such that at runtime, any violation of a condition provided either as a pre- or post-condition will be detected and flagged. This is achieved by capturing the conditions attributed by either `:pre` or `:post`. Using these captures, conditions are generated by first determining the relevant existence of an evaluative condition. Either a basic passable quoted expression is constructed using the `:ok` atom in the absence of a condition or a quoted expression is constructed by first unquoting the condition, evaluating its truth and then flagging an error if the condition is violated. We do this for both pre- and post-conditions, so we are left with a quoted expression representing the evaluation of each. When a call is made to the function, we build a final quoted expression to instrument the call. This final expression consists of first unquoting the pre-condition, evaluating the condition, matching the result of evaluating the function's body and capturing this value, then unquoting the post-condition, evaluating the condition and finally returning the buffered result. To summarise:

- 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.

It is important to reiterate that the runtime instrumentation of these conditions should only serve a niche purpose. Relying on evaluating these conditions at runtime does not prove the correctness of a specification. The `defv` construct should primarily be used to augment the verifier, not replace it.

The final construct LTLixir introduces is predicates. Predicates are a way to define a set of conditions that can be used in LTL formulas to help readability and reasoning. They provide no formal improvement over constructing verbose LTL properties. Predicates are defined in-line, using the `predicate` macro. The macro serves two purposes. Firstly, it instruments the Elixir program by introducing a new identifier to the function scope, which is assigned to a condition. Secondly, it flags the condition as a predicate to the verifier, so it can be recognized as a valid identifier in an LTL formula. The macro takes a new identifier and a condition as arguments and inserts a new quoted expression matching the identifier to the condition into the parent quoted expression. This predicate could be used as a condition within guards of control statements such as `if` and `receive`, but primarily is used within LTL formulae.

5.3 Modelling Elixir Programs

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

5.3.1 High-level Overview

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

- **Parsing**: given an Elixir program, generate a quoted expression that represents the program and performs lexical analysis and parsing on the quoted expression.
- **Intermediate Representation**: given a parsed tree of the quoted expression, generate an intermediate representation by extracting features relevant to model the program and specification.
- **Writing**: take the intermediate representation and generate 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 Veriflixir 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`. 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.

5.3.2 Sequential Execution

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

- **Blocks**: blocks are a simple but core concept. An Elixir block contains multiple Elixir expressions separated by newlines or semi-colons.
- **Modules**: multiple functions can be grouped together 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.
- **Do**: Elixir control structures such as `if` and `receive` all use the `do` keyword as syntactic sugar for a new expression. The child expression could be a single expression or a nested block.

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 attended 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.

TODO DIAGRAM OF THE TYPES OF SYMBOL TABLES AND THEIR HIERARCHY

Traversal, Scoping and Variable Declarations

Of course, that assumed a variable was not assigned to the child scope. In this case, we are required to track the execution of a scope in more detail. Let's first re-iterate Elixir's `match` operator and describe how the model generator handles it. Every expression in Elixir returns a value, and hence every expression can be matched on. We can match the entire return value of an expression to a single identifier, or use pattern matching to either have more granular control on how the

return value is matched. We could also form guards for more complex checks used in conditional statements (such as `if` and `receive`). Elixir is a dynamically, strongly typed language. Dynamic typing enforces restrictions on the set of Elixir expressions we support. Strong typing helps type inference in model generation. In the intermediate representation (IR), a match is represented as either a declaration or an assignment. Given a declaration to an integer, 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 this assignment's inferred type does not align with the declared type, we raise an error.

If we now match on 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 in using a depth-first search with a stack of lists. The stack manages the scope levels and the list manages the variable identifiers. Let's explore an example.

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

Listing 5.2: Representing variable declarations using the match operator.

In the example, we are matching on 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. TODO DIAGRAM ON THE STACKS?

Functions, Type Specifications and Return Types

Like the other constructs, functions introduce a new scope level. Multiple functions within a module and can be used to determine the control flow for a process. Before we discuss about the how functions are represented, we must quickly discuss types again. A correct LTLixir specification should provide type information for all function arguments and the return type. These properties are used to aid the type inference. The return type also determines how we model our function. If the return type is `:ok`, the function is non-returning, and we can treat the function as a standard sequential execution (but with its own local internal representation). If a return type is provided, we use a similar technique by recursively traversing expressions to determine all exit points of the function.

Promela does not support functions. We model functions using Promela processes and rendezvous communication channels. To implement functions using our writer, we apply various techniques that mimick how Elixir functions behave. Firstly, the caller must declare a new rendezvous communication channel (using a buffer size of 0). A rendezvous channel has no buffer and hence communication over the channel is entirely synchronous. We also declare a new variable to store the return value of the function, typed using the type specification of the callee. We can now spawn a new process (a process mocking the function) and pass two imperative arguments alongside the arguments specified in the Elixir function. We first pass the channel, which acts as both a return value and signaling method for the function termination. 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 the parent process in communication. We then pass the remaining arguments as if we were making a true function call. The caller will now block until the callee sends

a message over the signaling channel. The callee can proceed as normal, and by using the discussed traversal techniques, all exit points will send the final expression over the signaling channel. This approach easily extends to recursion, as the callee can spawn a new instance of itself and block until a signal is received.

TODO DIAGRAM OF RECURSIVE FUNCTION CALLS.

5.3.3 Concurrent Memory Model

Now we have explored the basics of modeling 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.
- Bounding communication.

We will first explain how we model the spawning of a new process, before taking a deeper look into more interesting concurrency primitives as well as a memory model to extend the existing capabilities of Promela. The Veriflixir IR for spawning processes resembles the Elixir spawn very closely. We capture a function that acts as the entry to the process which we can then model as a Promela process. As with a function call, we pass a process identifier to the new child process. In this case, the process identifier is a reserved identifier that signals to the new process to take a unique process identifier assigned automatically by the system. This is an important mechanism to ensure all parents are uniquely identifiable, which is crucial for communication. This process identifier can be captured in the caller's symbol table and used to communicate with the process.

TODO DIAGRAM OF FUNCTION CALL VS SPAWN AND HOW IT IMPACTS PID

Actors, Mailboxes and Message Passing

Once we have another process's identifier in our symbol table, we can begin communication between processes. Elixir implements this communication using the actor model, where each running process in the system is an actor that can send messages and receive messages using a mailbox. The mailbox can be considered a first-in-first-fireable-out queue. The IR must capture this ordering, otherwise it could misrepresent the execution of an Elixir program.

We begin by describing the design of a message. A message is comprised of two components, the `message type` and the `message body`. In Elixir, the message type is denoted with an atom, which the IR simply treats as a distinct type within the system. When the 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, even if in reality, the message type has different meanings in local contexts. The message body consists of multiple message arguments. At this point, the IR does not refer to the message argument by its type but purely treats it as an argument that is expected to be passed in communication. 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.

Now that we can construct a message, we can begin to model how messages can be sent and received. The IR keeps close to the mailbox system the actor-based implementation uses. Using the global message type set, we construct a per-process mailbox for each message type. The per-process mailboxes are packed into an arrays of mailboxes, where the index of the array aligns with the process identifiers we had previously been allocating. 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 to index into the correct communication channel to attach the message.

TODO EXAMPLE OF SENDING A MESSAGE TO ONE OF THE PROCESS' MAILBOXES

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

matching of a guard, we must preempt 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 guards, 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 guard 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 entire 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. A note on typing: the IR considers message types interchangeable with integers, hence comparisons can be made between the two which can be useful for matching. Sending messages is a non-blocking operation, unlike when we used rendezvous channels to model functions, if we send a message and do not care about receiving a returning message, we can continue executing the process body.

Unlike Elixir actors, Veriflixir 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.

Dynamic Memory Allocation

Although not a direct fallout of the Elixir concurrency prinicples, Promela does not support dynamic memory allocation such as memory allocations (malloc) or memory de-allocations (free). This lack of support has lead to design choices in the generation of Elixir lists that if mishandled could lead to a mis-representation of Elixir concurrency in the target specification language. Immutability is a core feature of the Elixir runtime, hence the Elixir list operations introduce behaviours that must be carefully handled. For example, the ++ operator can be used to append or prepend elements to a list. If this is used to prepend, the operation is constant time, however the IR will represent this (as well as most list operations) as an operation in linear time. Let's explore the memory model to understand why.

The IR introduces two structures to handle all list operations across the system. These structures are stored in the global scope context. The reason for this comes from Promela limitations. Promela supports statically sized arrays, and for good reasons regarding reducing state-space explosion, 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`. Firstly, it is in fact not a traditional linked list, but it behaves similar as we can perform many operations on this structure that we could not perform on a statically size array. For example, we support prepending and appending in order to dynamically resize the list. In actualallity, none of these operations resize the list. A list has an upper bound in it's size, which is statically set for all model generation. Let's name this limit L.

```
1 typedef linked_list {
2     node vals[L];
3 }
```

Listing 5.3: 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, this is handed by the `node` nested structure.

```
1 typedef node {
2     int val;
3     bool allocated;
4 }
```

Listing 5.4: Example of an int list node.

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 *true* 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 inbetween.

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

```
1 typedef memory {
2     linked_list lists[10];
3 }
```

Listing 5.5: Memory intermediate representation.

With this structure, we introduce a single globally defined instance of this structure named `__memory`. All processes share this singular memory for their list allocations. All list operations are treated as a single, indivisible step so this does not introduce concurrent execution concerns. If an Elixir process declares a new list, the IR will represent this as a memory allocation. The model runtime will iterate through all the lists in memory to find an unallocated list, as return this as a pointer to the process. Of course, as dynamic memory allocations is not supported in Promela, this *pointer* is generated as an `int`, which indexes into memory.

Finally, with all these definitions in place we can finally 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 allocated list and then pass the pointer as an argument. With this mocked memory in place, we can now support Elixir lists and operations on lists, we will discuss a few of these operations next.

Promela supports for loops, but for our custom dynamic memory these do not suffice to represent Elixir for comprehensions. Using a similar approach we used to append list elements, a for comprehension can be represnted as a linear scan through all `linked_list` elements to find the allocated elements, and then only applying the comprehension body to these. We can introduce temporary dummy variables to represent Elixir's `<-` operation. 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) are represented similarly to for comprehensions 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 that can be selected from 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, all of these Elixir functions are in-lined into the process body. Any reference to a function *return* is actually simply generated as an assignment.

5.3.4 Supporting LTLixir

In section 5.2 we discussed how Elixir was extended to support inline specifications. We now describe how these are modeled in the IR and generated in Promela.

System Entry

The `@vae_init` LTLixir attribute represents an entry point to the system. We represent this using a flag in the IR for each process, controlling if the process should be running in the first state of the model or not.

Type Specifications

A `@spec` attribute is parsed when parsing a function definition. For each argument being 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 of this as it could be the non-returning `:ok` type. If this is received, we insert a reserved non-returning type into the symbol table, and interface with this by supporting a `table.returning_function()` $\rightarrow$ `boolean` call to determine if a function if returning or not. If a function is non-returning, it would be an error to attempt to look up the type of the function in any context. If a function is returning, we must ensure all expressions which are returning (i.e. last in a block) write to the rendezvous channel, as discussed in section 5.3.2.

Concurrency Parameters

A `@params` attribute also instruments the IR of a function. For all of the parameters in the tuple of identifiers, we mark these before continuing with storing the remainder of the function. When a declaration of one of these identifers is found in an expression, we avoid the usual handling of this and instead assign a dummy value `__PARAM` to the identifer. This dummy value is used by the model generator to create multiple configurations of a single model, which will be discussed more in section 5.4.

Linear Temporal Time Formulae

The final attribute is `@ltl`. When an LTL formula is parsed, we create a set of all identifers (or predicates) present in the formula. We first instrument the function body by marking all these identifers as LTL identifers. When we detect the declaration of an LTL identifer to an expression, we extract this declaration out of the function and move it to the global context. Internally, we now consider this a system property as opposed to a property of a single process. When we have extracted all LTL identifer declarations, we can handle the conditions of the LTL formula, such as the temporal properties. We construct formulae from the system properties using the same operators the user defined in the attribute. We finally mark these formulae as LTL formulae in the IR, which means the Promela model generator can create LTL properties from them to be constructed with the rest of the state-space during verification.

Predicates

Similarly to LTL properties, any predicates are tracked in a similar way. Any identifers that construct a predicate must be extracted to the global context and flagged as system properties. We also globally define the predicates using the system properties in order to ensure they are valid within an LTL formuae.

Verifiable Functions

The `defv` construct is handled as the the other function definitions are as described in section 5.3.2. Verifiable functions are still internally represented as processes, however we now also extract the pre- and post-conditions from the definition. A pre-condition instruments a function-body by inserting an additional condition that's truth is evaluated before the rest of the function body. A post-condition flag is set in the functions definition in the event one is parsed. When a post-condition flag is set, we ensure to instrument all returning expressions of a function to assert the truth of the post-condition. As in the IR, functions actually write to a rendezvous channel instead actually returning, we perform this violation check after the return point of a function. When the verifier is ran on a model, every state in the state-space is exhausitvely checked, hence any possible evaluation of a pre- or post-condition will be exhaustively checked in the verification process. This provides better assurance than relying on the condition checks during the Elixir runtime.

5.4 Simulation and Verification

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

If $-s$ is passed, spin is ran in vanilla simulation mode: `./spin model.pml`. If $-v$ is passed, Veriflixir will first determine the existence of LTL properies 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 case 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 re-produced in simulation mode as we display them to the user if they are executed. These are not as proficient as using `IO` in the Elixir runtime. Simulation mode can be more beneficial than examining the output of running and Elixir program, as the scheduler takes enabled transitions based on the current state whereas the Elixir scheduler run in real-time which can result in some concurrent interleavings 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 Veriflixir 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, Veriflixir captures this and reports to the user that either an LTL property was violated (liveness property) or the system is potentially livelocked. Veriflixir will report the entire trace that lead 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 lead to that transition. 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, Veriflixir will also report the cycle that lead to non-terminating processes. It will report where the cycle begins, then give a trace of executed Elixir lines and which function exectued 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 specifiation 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, $(M, S_i) \models \phi_p$. 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. 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 there are other

active processes that 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 Veriflixir IR passes all detected parameters to the Veriflixir model runner. The model runner is reponsible for generating multiple models depending of the number of paramters 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 ran in $-search$ mode and violations are reported. Veriflixir reports an acceptance score, α, determing how many of the model were accepted by the verifier: $\alpha \coloneqq 1 - \frac{|V|}{M^N}$, 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 Summary, Limitations and Future Work

Chapter 6

Evaluation

6.1 Manual Verification

6.1.1 Manual Deadlock Model

6.1.2 Verification-aware Elixir Deadlock

6.1.3 Comparison

6.2 Alternating-bit Protocol

6.2.1 Informal Specification

6.2.2 Experimental Analysis

6.2.3 Limitations

6.3 Basic Paxos

6.3.1 Informal Specification

6.3.2 Comparison with 'Model Checking Paxos in Spin' Paper

Chapter 7

Conclusion

7.1 Future Work

7.2 Ethical Considerations?

7.3 Final Remarks

Appendix A

First Appendix

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.