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 19, 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 2.5.4, 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, when two events unique to the alphabet of two independent processes are enabled at the same time, the 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 and 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 need 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, even if the student is behind in the number of instructions it has executed, it 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.

$$\langle A \rangle \quad \models \quad p \;\mid\; \top \;\mid\; \neg \langle A \rangle \;\mid\; \langle A \rangle \wedge \langle A \rangle \;\mid\; \Box \langle A \rangle$$

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 trust of A in state s of the model M.

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

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, $AP = \{study, sleep, tired, exam\}$.

$\square$sleep	Always sleeping
$tired\,U\,sleep$	Tired until sleeping
$\square(study \Rightarrow tired)$	Studying implies always tired
$\square study \Rightarrow \Diamond exam$	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]. With LTL, these are all important concepts in the model checking of systems [36], see 2.3. 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.

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. Hence, the state s could be a violating state for E. 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.

$$\square\Diamond study$$

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.

$$\square study$$

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 $(\tau_1, \tau_2, \tau_1, \tau_2, \dots)$ 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.

$$\tau_1 : time \leftarrow time + 1$$

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.

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

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.

2.4 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.

2.4.1 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.

2.5 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.

2.5.1 Lean

The Lean theorem prover is a proof assistant developed by Leonardo de Moura [22]. It is the first of a few theorem provers we will discuss to understand how Hoare Logic has developed into software tools. 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.

Lean is both a functional programming language and a theorem prover. This means we can define first-class functions and interactively operate the theorem prover to ensure correctness over them. This approach differs in implementation from other theorem provers, such as Dafny, which instead prove theorems using existing tools.

2.5.2 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.

2.5.3   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.

2.5.4 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

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 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.

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

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.

```
1      proctype SomeProcess(int a) {
2          printf("Do something with %d\n", a);
3      }
4
5      init {
6          int p1;
7          p1 = run SomeProcess(10);
8
9          printf("Init process spawned at %d\n", _pid);
10         printf("Process 1 spawned at %d\n", p1);
11     }
```

Listing 2.5: Defining and spawning processes in Promela.

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.

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

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

$$\text{int x;}$$
$$\text{global\_broadcast ? BROADCAST, x;}$$

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.

2.6   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

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 implement 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.

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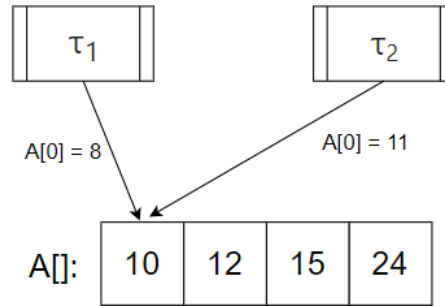
Figure 3.1: An example of two processes writing to a shared in-memory array

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.
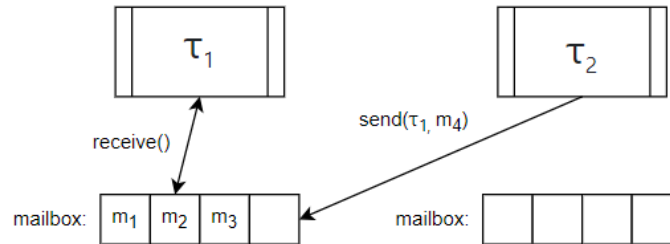

Figure 3.2: An example of actors sending and receiving messages under the actor model

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.

```
1     x = 2
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.

```
1   defmacro unless(clause, do: expression) do
2     quote do
3       if(!unquote(clause), do: unquote(expression))
4     end
5   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.

```
1     @doc "Calculate the sum of two numbers, x and y"
2     def sum(x, y) do
3         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`.

```
1    @vae_init true
2    def start_server do
3        client = spawn(Client, :start_client, [])
4    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 [?].

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

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

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

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

4.2.1 Detecting a Deadlock

Now we have a basic understanding of what is required to write a specification, we will use 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.

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

```
11   defmodule Client do
12     def start_client do
13       receive do
14         {:binding} -> IO.puts "Client bound"
15       end
16     end
17   end
```

Listing 4.3: A simple Elixir system with a deadlock.

In this 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.

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

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

```
1    @vae_init true
2    @spec start_server() :: :ok
3    @ltl "<>(alive_clients==client_n)"
4    def start_server do
5        ...
6    end
```

Listing 4.5: Example LTL property.

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

```
1    $ ./veriflixir -v basic_example.ex
2    Model checking ran successfully. 0 error(s) found.
3    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.

```
1    $ ./veriflixir -v basic_example.ex
2    Model checking ran successfully. 1 error(s) found.
3    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.

```
1    @ltl "(q)U([]p)"
2    def start_server do
3      client_n = 3
4      alive_clients = 0
5      predicate p, alive_clients == client_n
6      predicate q, !p
7      ...
8    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.

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

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

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.

```
1  defv add_positives(a, b), pre: a > 0 and b > 0, post: result == a + b do
2      ...
3  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.

```
1  @spec next_round(pid(), integer()) :: :ok
2  defv next_round(server, rounds), pre: rounds >= 0 do
3      ...
4  end
```

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

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

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.

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

```
1   $ ./veriflixir -p basic_example.ex
2   Generating models.
3   Generated 9 models.
4   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.

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

```elixir
3        ...
4        alive_clients = if number_of_rounds > 1 do
5            check_clients(client_n * number_of_rounds, alive_clients)
6        else
7            0
8        end
9    end
```

This will now result in cases where the temporal property is violated, as `alive_clients` $\equiv$ `client_n * number_of_rounds` will not hold for all configurations. Note that we have reduced the parameters down to just {`number_of_rounds`}. The system is theoretically capable of handling any number of parameters in its search, however the computational cost of exploring the state-space grows exponentially with the number of parameters, hence in reality we should use our understanding of the system to determine which parameters to test at a time.

```
1    $ ./veriflixir -p 3 basic_example.ex'
2    Generating models.
3    Generated 3 models.
4    Acceptance confidence: 0.6666666666666667.
5    Violations found in models:
6    Model with params: {"number_of_rounds": '1'}
7    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.

Chapter 5

Implementation

5.1 Verification-Aware Elixir Toolchain

5.2 Extending Elixir

5.2.1 Linear Temporal Logic

5.2.2 Hoare-style Logic

5.3 Modelling Elixir Programs

5.3.1 Sequential Execution

Selection

Functions

5.3.2 Concurrent Execution

5.3.3 Memory Model

Actors, Mailboxes and Message Passing

Dynamic Lists

5.3.4 The Elixir Standard Library

For Comprehension

Standard Modules

5.4 Simulation and Verification

5.4.1 Simulation

5.4.2 Verification

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.