Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

# Automatic Verification of Actor-Based Systems

*Author:*
Matthew Neave

*Supervisor:*
Dr. Naranker Dulay

*Second Marker:*
TBD

May 18, 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, $\alpha P$ 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 $\alpha S$.

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

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 modelled 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 TODO

### 2.2.1 Temporal Logic

**Predicate Logic**

**Linear Temporal Logic**

**Computation Tree Logic and Alternating-time Temporal Logic**

### 2.2.2 Safety and Liveness

## 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 $\varphi$, if no state of M violates $\varphi$, then we can say M satisfies $\varphi$. 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, $\tau_1$ and $\tau_2$.

| | |
|---|---|
| $\tau_1$ : acquire lock A | $\tau_2$ : acquire lock B |
| $\tau_1$ : acquire lock B | $\tau_2$ : acquire lock A |
| $\tau_1$ : release locks | $\tau_2$ : release locks |

An interleaving such as ($\tau_1$, $\tau_2$, $\tau_1$, $\tau_2$, ...) results in $\tau_1$ blocking until it can acquire lock B, and $\tau_2$ 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.

## 2.4 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.

```
method Add(a: int, b: int) returns (c: int)
    ensures c == a + b;
{
    if  a < 0 {
        c := -1;
    } else {
        c := a + b;
    }
}
```

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.

```
procedure add(a: int, b: int) returns (c: int)
    ensures c == a + b;
{
    if (a < 0)
    {
        c := -1;
    } else {
        c := a + b;
```

```
    }
}
```

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.

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

#### 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.

```
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.

### 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.

```
mtype = { 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.

```
chan global_broadcast = [10] of { mtype, int };
```

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

```
global_broadcast ! BROADCAST, 1;
```

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

```
int x;
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].

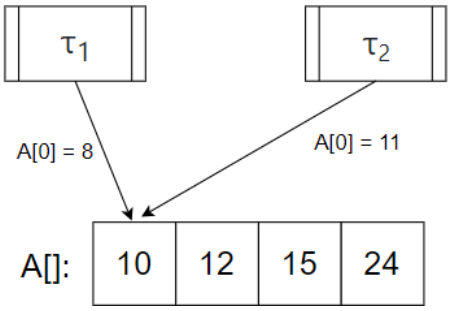


Figure 3.1: An example of two processes writing to a shared in-memory array

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.

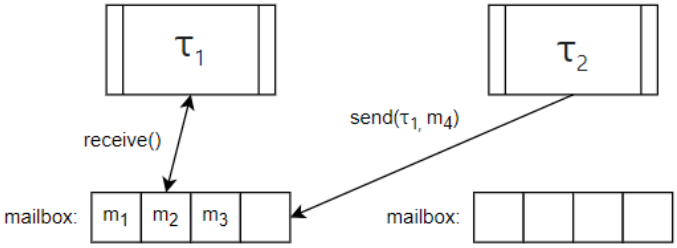


Figure 3.2: An example of actors sending and receiving messages under the actor model

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.

```
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.

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

We want to verify that eventually, the number of alive clients equals the number the server created, we can write this using the formula $<>(\mathit{alive\_clients} == \mathit{client\_n})$. 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 ($<>(\mathit{alive\_clients} == 1)$). 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.

### 4.2.3 Pre and Post Conditions

### 4.2.4 parameterized Systems

# 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

# Chapter 8

# Project Plan

This chapter will discuss what needs to be done for the project to be successful, the paths that can be taken, areas that can be explored as an extension and fall-back positions in the limit of time.

## 8.1 The Artifact

The key aim of the project is to produce a code artifact that can verify the correctness of Elixir programs. "Verify" is being used as an umbrella term for three potential components of the artifact. In no particular order:

- Determining if a program is deadlock-free.
- Allowing users to write specifications about functions that are inline and verifiable.
- Verifying liveness properties.

It would be difficult/infeasible to design a tool that can do all three from scratch and similarly, I have yet to find a tool that can do all three. Hence, my current path forward is to use a combination of existing tools where necessary to achieve these verification feats. The most appropriate tools I have identified for each case respectively are the SPIN model-checker, the Boogie IVL for theorem-proving and the TLC model-checker.

Given these three tools, the plan would be to develop a command-line tool, that takes an Elixir project as an input, parses the Elixir code, and extracts the underlying model from the project to create an internal representation that can be used to generate models in the three target output grammers.

The main difficulty of this project will lie in the design of Elixir. Elixir focuses on the concurrent execution of programs, using the actor model for communication between sequential processes executing in parallel. None of the target output tools have support for message passing and Boogie does not support concurrency. That means the main challenge of the project will be designing a framework for modelling programming languages that use message-passing as a first-class solution to communication. If the framework is well designed, this may not be limited to Elixir, the intermediate representation could be a target grammar that any message-passing oriented language can be translated into (such as Rust). However, for the scope of this project, modelling Elixir will be sufficient.

Once a model has been extracted from an Elixir program, the next step will be code generation for the relevant tools, which can then be ran to determine the correctness of the program.

## 8.2 Timeline and Milestones

This section is a brief overview of what needs to happen and what has already began.

#### Manual Translation (started)

The first step involved understanding what an Elixir program looks like in the target representations. I spent time taking some basic Elixir programs and translating them into Promela (the

specification language used by SPIN) to gather an understanding of how they can begin to be translated. I modelled four programs, an entirely sequentially executed program, a program that introduces a deadlock, a program that introduces a livelock and a program based on a deadlocking dining philosophers algorithm. While doing this, I made notes of how various components can be modelled in Promela as well as what is difficult to model. A brief summary of some findings:

- In the basic deadlock model, a deadlock was detected.
- In the dining philosophers model (a more complex example of an Elixir program) a deadlock was detected.
- In the basic livelock model, the model-checker ran forever and did not detect the livelock. In theory, Promela should be detecting livelocks so more investigation needs to be done into how models need to be bound to allow for the successful detection.
- The key primitives unique to the actor model were able to be modelled with fair success in Promela.

As well as translating Elixir programs to Promela models, I spent some important time translating quoted expressions (equivalent to ASTs) to models, as Elixir provides the functionality to easily access them.

#### Parsing (started)

Parsing Elixir programs so they can be stored in an intermediate representation is important. Because of the access to quoted expressions, there is no need to parse Elixir grammar directly, instead parsing quoted expressions is easier. They are guaranteed to be well-formed, and when parsed you are left with an AST by nature. Work has begun here using a parser combinator library (pest) in Rust. The Elixir developers do not provide exhaustive documentation of the grammar of quoted expressions, so instead they have to be derived from examples and trial-and-error.

#### Model Extraction (started)

The main challenge of the project is model extraction. It introduces many questions that need answering:

- What does a sequential execution of statements look like?
- How can functions be represented to be modelled in specifications that don't support functions (Promela)?
- How can Elixir recursion be modelled in specifications that don't support recursion (Promela)?
- How can concurrency processes be modelled as a sequential process (Boogie)?
- How can message-passing be modelled, in particular when specifications don't support shared memory?

Unfortunately, the list goes on the more you look into the Elixir language. A starting point will be modelling sequential programs.

#### Promela code-gen (begin in term 2)

The first model checker I aim to target is Promela. Although it doesn't natively support function calls or definitions it does support concurrency, therefore I deem it an easier milestone to reach. Once code-gen for Promela is implemented, it should be possible to begin proving Elixir programs are deadlock-free, which is a massive step towards being a verification-aware language.

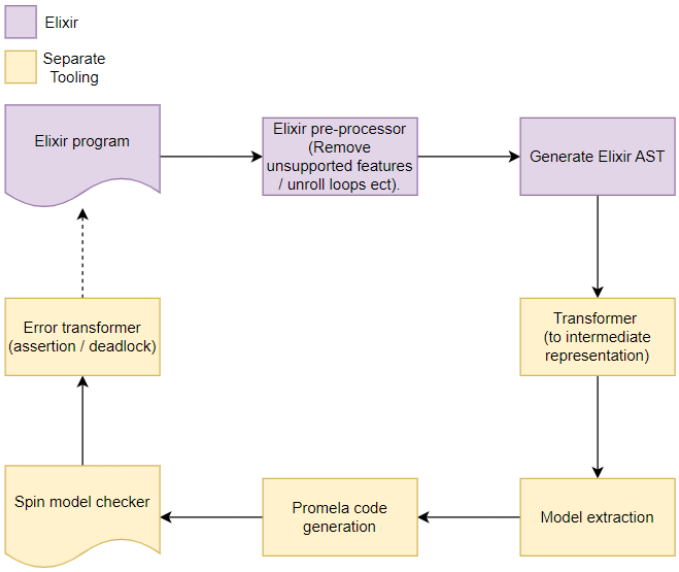


Figure 8.1: v0 system design for deadlock detection.

### Extending Elixir with Metaprogramming (begin in term 3)

It will be difficult within the scope of the project to allow any Elixir program without modification to be verified. Using metaprogramming, steps can be taken to introduce an Elixir library that allows developers to write code that is easier to verify. For example, developers can introduce bounds to mailboxes and recursive calls that make model extracting a lot more direct. Anything unbounded won't be verifiable, so I want to put the burden on the developer writing specifications instead of the artifact to approximate bounds.

Metaprogramming can also be used to introduce pre- and post-conditions to Elixir to extend the support for verification.

### Boogie code-gen (term 3 / future work)

After extending the Elixir language to allow the introduction of pre- and post-conditions, code-gen for Boogie programs can take place that allows stronger verification support for Elixir. I deem this an important part of the project but it depends on a lot of prior work being finished, so it's hard to determine if it fits on the timeline.

### Liveness (future work)

It is unlikely that liveness will be implemented as part of the verification toolkit unless it is discovered one of the existing downstream tools supports it. I believe code-gen for a new tool will be required (such as TLA+) to achieve this, which is likely to be infeasible in the scope of the project, but on my radar.

# Chapter 9

# Evaluation Plan

This section will discuss what a successful project looks like and some methods that can be used to evaluate the success of the project.

## 9.1 Manual Translation

The first and easiest method for evaluating the performance of the tool will be comparing the results of the tool to manually designed models. It will be easy to introduce deadlocks into small programs, reason that there is an execution path that leads to the deadlock, and then use the verifier to confirm the program introduces a deadlock. This will form a basis for evaluating the correctness of the produced tool. It will be harder to evaluate cases where a deadlock is present in the program that the tool fails to identify. With a large test suite, hopefully, this will not often be the case however, these bugs may occasionally occur.

Evaluating the effectiveness of other verification techniques, such as liveness and the introduction of pre- and post-conditions is more straightforward. Depending on the expressions that will be supported in pre- and post-conditions, it is likely these expressions will be constructed recursively where the base cases can be exhaustively tested (for example, the use of the addition operator in a post-condition can be shown to be correct with a unit test).

## 9.2 True Positives Vs False Positives

An important metric to track when evaluating the produced tool will be how many programs produce false positives. This has been shown as a relevant metric in related work. Determining if a deadlock, condition or liveness property violation are true or false positives will be difficult to reason about for larger systems. As a starting point, the tool can be run on programs that are known to violate a specification in a state and compare its performance before applying the tool to unknown programs that have been assumed to be correct under a specification.

## 9.3 Open Source Projects

There are some open source Elixir projects that can be used as real-world examples of determining if programs follow specifications. For example, Discord released Elixir libraries for various distributed network tasks that could be verified with a verifier. Finding errors in these programs would be a strong indication of the effectiveness of the tool. Likely, the verifier will not be able to run directly on these programs without modification, for example introducing bounds on execution in various places as many real-world applications of Elixir may be in long-lived processes.

## 9.4 Runtime

The runtime of the produced tool is not a focus of the project. However, with many modern verification tools being used on production-level code, it will be important to measure how the tool

performs on small and large-scale applications to allow for comparison between other verifiers for other programming languages (i.e. Dafny or C).

# 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.