Imperial College
London

MEng Individual Project

Imperial College London

Department of Computing

Verification Aware Elixir (Interim Report)

Author:
Matthew Neave

Supervisor:
Dr. Naranker Dulay

Second Marker:
TBD

January 2, 2024

Abstract

Your abstract goes here

Acknowledgements

Thanks mum!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Hello [?] Hello [?]

1.1 Objectives

1.2 Challenges

1.3 Contributions

Chapter 2

Background

2.1 Communicating Sequential Processes

2.2 Model Checking

Model checking is the process of determining if a finite-state machine (FSM) is correct under a provided specification. It typically involves enumerating all possible states of an FSM and ensuring the correctness of each state. 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.

$$\begin{aligned} \tau_1 &: \text{ acquire lock A} \\ \tau_2 &: \text{ acquire lock B} \\ \tau_1 &: \text{ acquire lock B} \\ \tau_2 &: \text{ acquire lock A} \end{aligned}$$

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

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

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

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

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

This is a simple example of a specification, that we can write in a specification language and use in tandem with our model to check the correctness of using a model checker.

2.2.1 A Comparison Of Model Checkers

Many model checkers have been invented for this reason, each with different focuses and specification languages. This section will comment on some of the more common model checkers and discuss their functionalities.

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.

```
mtype = { HELLO };
chan channel = [10] of { mtype };

init {
    channel ! HELLO;
}
```

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

2.3 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.

```
% Spawn a new process
spawn(fn -> 1 + 2 end)

% Create a new BEAM instance
Node.spawn(:"node1@localhost", MyModule, :start, [])
```

Listing 2.2: An example of spawn/1 and spawn/4 in Elixir for spawning a new lightweight process and a new Elixir node

2.3.1 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. Fig 2.2 shows a basic example of two processes that write to a shared in-memory array. Due to how often we see shared memory used in large-scale distributed systems, much work has been done in the verification of these systems using shared memory models. For example, Jon Mediero Iturrioz used Dafny [18] to prove the correctness of concurrent programs that implement shared memory [17].

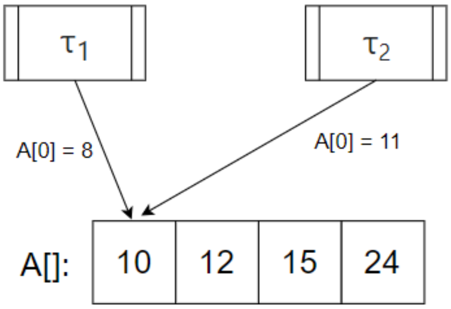

Figure 2.2: An example of two processes writing to a shared in-memory array

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. Fig 2.3 shows an example of how actors behave. The mailbox is not necessarily first in, first out (FIFO) but often implementations tend to be.

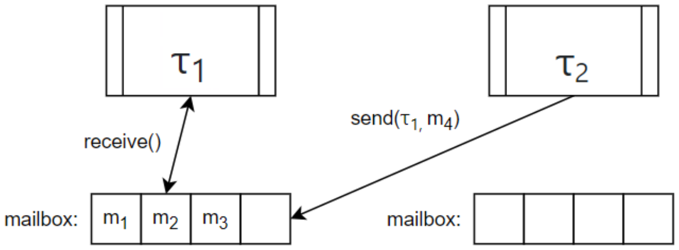

Figure 2.3: An example of actors sending and receiving messages under the actor model

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.

```
% Example send in Elixir
send self(), {:hello, "world"}

% Example receive block in Elixir
receive do
   {:hello, msg} -> IO.puts msg
end
```

Listing 2.3: An example of spawn/1 and spawn/4 in Elixir for spawning a new lightweight process and a new Elixir node

2.3.2 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 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. Fig.2.4 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 2.4: Elixir example of `quote/2` and `unquote/1`.

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

[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.

2.3.3 Metaprogramming

2.4 Existing Work

2.4.1 Lean

2.4.2 C Wolf

2.4.3 Dafny

2.4.4 Promela

2.4.5 Gomela

2.5 Modelling Elixir Programs in Promela

2.5.1 Basic Deadlock

2.5.2 Dining Philosophers

2.5.3 Preconditions and Postconditions

Chapter 3

PROJECT X

Chapter 4

Evaluation

Chapter 5

Conclusion

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] The Temporal Logic of Actions Available from: https://lamport.azurewebsites.net/pubs/lamport-actions.pdf.

[6] Specifying Concurrent Systems with TLA+ Available from: https://lamport.azurewebsites.net/tla/xmxx01-06-27.pdf.

[7] Model Checking TLA+ Specifications Available from: https://lamport.azurewebsites.net/pubs/yuanyu-model-checking.pdf.

[8] The PlusCal Algorithm Language 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] 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