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

December 31, 2023

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
this is a description of PAT
BLAST
PRISM
SPIN
TAPAAL
TLA+

2.3 Elixir

2.3.1 Shared Memory and Message Passing

2.3.2 Quote and Unquote

2.3.3 Metaprogramming

2.4 Existing Work

2.4.1 Dafny

2.4.2 Promela

2.4.3 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.