# Improvements to Transitive-Closure-based Model Checking in Alloy

by

Sabria Farheen

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2018

© Sabria Farheen 2018

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

## Abstract

Model checking, which refers to the verification of temporal properties of a transition system, is a common formal method for verifying models. Transitive-closure-based model checking (TCMC), developed by Vakili *et al.*, is a symbolic representation of the semantics of computational tree logic with fairness constraints (CTLFC) for finite models in first-order logic with transitive closure (FOLTC). TCMC is an expression of the complete (*i.e.*, unbounded) model checking problem for CTLFC as a set of constraints in FOLTC without induction, iteration, or invariants. TCMC has been implemented in the Alloy Analyzer.

This thesis focuses on improving practical aspects of using TCMC in Alloy. We provide style guidelines for writing concise declarative models of transition systems for behavioural analysis in Alloy without any extensions to the Alloy language. We address the issue of spurious instances produced when generating instances at small scopes using the Alloy Analyzer by introducing *significance axioms*, which ensure the instance contains interesting behaviour. We define *scoped TCMC* for a state scope of $n$, where $n$ is less than the size of the reachable state space, as the model checking of all transition system instances of state size $n$ that satisfy the transition relation. By considering infinite and finite paths of a transition system separately, we can make useful deductions about the complete model checking problem from the results of scoped TCMC for certain categories of properties. The *significant scope*, derived from the significance axioms, provides a measure independent of computing resource limitations that a significant part of the state space has been verified, providing higher confidence in the deductions from scoped TCMC.

We present case studies that demonstrate the claims and results of this work. We also compare TCMC in Alloy to NuSMV and bounded model checking in terms of modelling practices, expressibility of temporal properties, model checking results, and performance.

**Acknowledgements**

First of all, I would like to thank my supervisor, Dr. Nancy A. Day, who taught me how to research, write academic literature, and think critically about formal methods. Her extensive knowledge and insight have been paramount in the development of this work. She has been truly supportive and inspirational throughout my time in Waterloo.

I would like to thank Dr. Amirhossein Vakili for joining many discussions throughout my research, and for explaining much of his work to me in great detail. I would like to thank Dr. Jeffrey Joyce for taking the time to assess my work, providing valuable industry feedback, and for allowing me the opportunity to contribute to his projects – it was a great learning experience.

I would like to thank Dr. Joanne Atlee and Dr. Derek Rayside for reviewing my thesis. I would like to thank my colleagues, Jose Serna and Ali Abbassi, who were very helpful in providing fresh perspectives and thoughts on many of my research topics.

I would also like to thank Dr. Alma L. Juarez Dominguez for speaking to me about her career and her experiences in formal methods, which was extremely helpful in developing my personal goals.

Finally, I would like to thank my family and friends for their continued love and support throughout all my endeavours.

# Table of Contents

**List of Tables** ix

**List of Figures** x

**1 Introduction** 1

1.1 Thesis Overview and Contributions . . . . . . . . . . . . . . . . . . . . . 3

1.2 Case Studies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

1.3 Thesis Organization . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

**2 Background** 7

2.1 Temporal Logic Model Checking . . . . . . . . . . . . . . . . . . . . . . 7

2.2 Transitive-Closure-based Model Checking (TCMC) . . . . . . . . . . . . . 9

2.3 Alloy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

2.4 TCMC in Alloy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

2.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

**3 Modelling a Transition System in Alloy** 16

3.1 Declaring the State Space . . . . . . . . . . . . . . . . . . . . . . . . . 18

3.1.1 State Equivalence . . . . . . . . . . . . . . . . . . . . . . . . . 19

3.1.2 No Invariants . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

3.2 Defining Initial States and Operations . . . . . . . . . . . . . . . . . . . 21

3.2.1 Pre- and Post-conditions . . . 21
3.2.2 Distinct Operations . . . 22
3.3 Model Definition . . . 23
3.3.1 Transition Relation: DisjMethod vs. ConjMethod . . . 24
3.4 Summary . . . 27

**4 Generating an Instance 28**
4.1 Scope . . . 28
4.2 Spurious Instance Problem . . . 29
4.3 Significance Axioms . . . 31
4.4 Significant Scope . . . 33
4.5 Summary . . . 33

**5 Scoped-TCMC Methodology 34**
5.1 Types of Properties . . . 34
5.2 Safety Properties . . . 37
5.3 Finite Liveness Properties . . . 38
5.4 Infinite Liveness Properties . . . 42
5.5 Existential Properties . . . 43
5.6 Summary . . . 45

**6 Case Studies 46**
6.1 Musical Chairs . . . 46
6.1.1 Style Guidelines . . . 46
6.1.2 Significance Axioms . . . 47
6.1.3 Scoped-TCMC Methodology . . . 48
6.2 Elevator System . . . 52
6.2.1 Style Guidelines . . . 52

6.2.2 Significance Axioms . . . . . . . . . . 54
6.2.3 Scoped-TCMC Methodology . . . . . . . . . . 55
6.3 Traffic Light Controller . . . . . . . . . . 58
6.3.1 Style Guidelines . . . . . . . . . . 58
6.3.2 Significance Axioms . . . . . . . . . . 60
6.3.3 Scoped-TCMC Methodology . . . . . . . . . . 60
6.4 Feature Interaction in a Telephone System . . . . . . . . . . 61
6.4.1 Style Guidelines . . . . . . . . . . 62
6.4.2 Significance Axioms . . . . . . . . . . 64
6.4.3 Scoped-TCMC Methodology . . . . . . . . . . 64
6.5 Scalability . . . . . . . . . . 65
6.6 Summary . . . . . . . . . . 66

**7 Comparison to NuSMV and BMC** **68**
7.1 NuSMV . . . . . . . . . . 68
7.2 BMC in Alloy . . . . . . . . . . 71
7.3 Summary . . . . . . . . . . 75

**8 Related Work** **76**

**9 Conclusion** **79**

**References** **82**

**APPENDICES** **87**

**A Alloy Models: TCMC Case Studies** **88**
A.1 Musical Chairs . . . . . . . . . . 88
A.2 Elevator System . . . . . . . . . . 92
A.3 Traffic Light Controller . . . . . . . . . . 97
A.4 Feature Interaction in a Telephone System . . . . . . . . . . 106

**B Non-Alloy Model for Comparison** **113**

B.1 Musical Chairs in NuSMV . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113

**C Alloy Models From Other Works for Comparison** **117**

C.1 Traffic Light by Vakili [42] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 117

C.2 Span Tree by Macedo *et al.* [29] . . . . . . . . . . . . . . . . . . . . . . . . . . 121

# List of Tables

5.1 Deducing Complete Model Checking Results from Scoped TCMC . . . . . 45

6.1 Performance Results of Case Studies. NS: Number of Signatures, NR: Number of Relations, SS: Scope Size, min: minutes, s: seconds . . . . . . . . . . . 66

7.1 Performance Results for BMC of a Safety Property (Figure 7.3 Line 9) in Musical Chairs. NS: Number of Signatures, NR: Number of Relations, SS: Scope Size, s: seconds . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72

7.2 Deducing Complete Model Checking Results in Alloy: Scoped TCMC vs. BMC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74

# List of Figures


2.1 Relationship Between CTLFC and CTL Connectives (figure from [42]) . . 9
2.2 TCMC implementation in Alloy . . . . . . . . . . . . . . . . . . . . . . 13
2.3 Template for use of TCMC . . . . . . . . . . . . . . . . . . . . . . . . 14
2.4 Multiple instances of a transition system for the constraint: "every state must reach a state that is reachable from itself" (figure from [42]) . . . . . 14

3.1 Predicate from Span Tree Alloy Model [29] . . . . . . . . . . . . . . . . 16
3.2 Applying Our Guidelines to Predicate from Span Tree Alloy Model [29] in Figure 3.1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.3 Musical Chairs Overview . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.4 Musical Chairs State Space . . . . . . . . . . . . . . . . . . . . . . . . 19
3.5 Part of an Arbitrary Model . . . . . . . . . . . . . . . . . . . . . . . . 20
3.6 Initial state constraints . . . . . . . . . . . . . . . . . . . . . . . . . . 21
3.7 Eliminate Loser Operation in Musical Chairs: Pre- and Post-Condition . . 21
3.8 Example of a Non-distinct Operation . . . . . . . . . . . . . . . . . . . 22
3.9 Operations in Musical Chairs: `music_starts` and `declare_winner` . . . . . 23
3.10 DisjMethod for Eliminate Loser Operation and Musical Chairs Model Definition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.11 ConjMethod Method for Defining `nextState` Relation . . . . . . . . . . . 25
3.12 Overlap in pre-conditions. Shows only transitions starting from $S1$. . . . . 26
3.13 Incomplete pre-conditions. Shows all transitions between $S5$ and $S6$. . . . 26

4.1 Musical Chairs: 3-State Spurious Instance . . . . . . . . . . . . . . . . . . . 29
4.2 Arbitrary Model: State Space and Instances. $I_n$: initial states, $Op_n$: user-defined operations. Intances A and B are Spurious. Instance C is Non-Spurious. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.3 Musical Chairs: Significance Axioms . . . . . . . . . . . . . . . . . . . . . 32

5.1 Modelling and checking transitions systems using TCMC in Alloy. $T_1$, $T_2$, and $T_3$ are transition systems from a non-unique NSR definition. . . . . . . 35
5.2 CTLFC Property Categories . . . . . . . . . . . . . . . . . . . . . . . . . 36
5.3 Safety Property: Scoped-TCMC Methodology . . . . . . . . . . . . . . . . . 37
5.4 Checking a safety property of Musical Chairs . . . . . . . . . . . . . . . . . 38
5.5 Finite Liveness Property: Scoped-TCMC Methodology . . . . . . . . . . . . 39
5.6 $AFp$: TCMC (*without* dead-loop) results in pass for both . . . . . . . . . . 40
5.7 $AFp$: TCMC *with* dead-loop results in pass for (a) and fail for (b) . . . . . 40
5.8 Checking a finite liveness property of Musical Chairs . . . . . . . . . . . . 41
5.9 Dead-loop Constraint in an Alloy Model . . . . . . . . . . . . . . . . . . . 41
5.10 Infinite Liveness Property: Scoped-TCMC Methodology . . . . . . . . . . . 42
5.11 Checking an infinite liveness property of Musical Chairs . . . . . . . . . . . 43
5.12 Existential Property: Scoped-TCMC Methodology . . . . . . . . . . . . . . 44
5.13 Checking an existential property of Musical Chairs . . . . . . . . . . . . . 44

6.1 Musical Chairs: 5-State Spurious Instance . . . . . . . . . . . . . . . . . . . 47
6.2 Musical Chairs: 8-State Non-Spurious Instance (sigma: Transition Relation) 48
6.3 Musical Chairs: Safety Property Counterexample with A Finite Path (sigma: Transition Relation) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
6.4 Musical Chairs: Infinite Liveness Property Counterexample with An Infinite Path (sigma: Transition Relation) . . . . . . . . . . . . . . . . . . . . . . . 51
6.5 Elevator System State Space . . . . . . . . . . . . . . . . . . . . . . . . . 52
6.6 Elevator System Operations . . . . . . . . . . . . . . . . . . . . . . . . . 53
6.7 Elevator System Operations . . . . . . . . . . . . . . . . . . . . . . . . . 54

6.8 Elevator System: 4-State Spurious Instance . . . . . . . . . . . . . . . . . . . 54
6.9 Elevator System: Full Significance Axioms . . . . . . . . . . . . . . . . . . . 55
6.10 Elevator System: 7-State Non-Spurious Instance (sigma: Transition Relation) 56
6.11 Elevator System: Safety Property . . . . . . . . . . . . . . . . . . . . . . . 56
6.12 Elevator System: Finite Liveness Property . . . . . . . . . . . . . . . . . . . 57
6.13 Elevator System: Infinite Liveness Property . . . . . . . . . . . . . . . . . . 57
6.14 Traffic Lights Control: State Space . . . . . . . . . . . . . . . . . . . . . . 58
6.15 Traffic Lights Control: Operations and Model Definition . . . . . . . . . . . 59
6.16 Traffic Lights Control: Fairness Constraints . . . . . . . . . . . . . . . . . . 60
6.17 Traffic Lights Control: Significance Axioms (full axioms in Appendix A.3) . 61
6.18 Traffic Lights Control: Safety Property . . . . . . . . . . . . . . . . . . . . 61
6.19 Telephone System: State Space . . . . . . . . . . . . . . . . . . . . . . . . 62
6.20 Telephone System: Operations and Transition Relation . . . . . . . . . . . 63
6.21 Telephone System: Significance Axioms (full axioms in Appendix A.4) . . 64
6.22 Telephone System: Safety Property . . . . . . . . . . . . . . . . . . . . . . 64

7.1 Part of Musical Chairs model in NuSMV . . . . . . . . . . . . . . . . . . . 69
7.2 Example Transition System . . . . . . . . . . . . . . . . . . . . . . . . . . 71
7.3 Musical Chairs: BMC in Alloy . . . . . . . . . . . . . . . . . . . . . . . . 73

# Chapter 1

# Introduction

Poor software quality can lead to severe consequences, especially for high-impact systems, such as, transportation vehicle software, where safety is paramount. A recent article in the popular magazine, *The Atlantic* [40], discusses several mishaps, ranging from alarming to fatal, all caused due to unexpected failures in software, including 911-emergency line outages and automobile accidents. As discussed in the same article, blame is often attributed to software engineers who lack insight into their system requirements and jump to code before verifying their intended designs. However, verifying system designs is often not a simple task, as it requires some form of exhaustive exploration of all possible behaviour a software system can exhibit.

Formal methods address precisely this concern: how to mathematically (and formally) prove that a system behaves as it should. When software is so complex that it handles every scenario of a vehicle's movement, for example, utilizing formal methods for verification is perhaps the only fail-safe way to ensure correctness.

When using formal verification, it is best to start early in the software development process, eliminating flaws before any code is written. To verify systems, it is necessary to create an abstract model of the system that can be analyzed. These abstract models are usually declarative, meaning that they are described using a set of constraints. Several tools and languages exist to create and analyze declarative models, such as, Alloy [23], B [1], Z [22], TLA+ [45], and ASMs [4]. These languages have many features to express abstract concepts (*e.g.*, sets, relations, and functions) without sacrificing precision.

These tools can be used to create *static models*, where the objective is to reason about a system's structure, or *behavioural models*, where the system's behaviour as it is executed over time is modelled. A behavioural model usually represents a *transition system*, which

contains a set of states, and transitions connecting those states. Specifications that make assertions about the execution, or behaviour, of the system, are often expressed in terms of a temporal logic, such as, linear temporal logic (LTL) [37] and computational tree logic (CTL) [10]. Behavioural models can be investigated for correctness using *model checking* [10], which analyzes a model exhaustively to test if it satisfies a temporal property.

There are two broad techniques for model checking: explicit-state model checking [19] and symbolic model checking [31]. Explicit-state model checking enumerates all possible reachable states, and ensures that each of these states satisfies the given property. The brute-force nature of this algorithm means that any application usually requires a considerable amount of computing resources when analyzing large models. Symbolic model checking refers to algorithms that traverse the state space more efficiently, considering multiple states at the same step by representing the set of states as a formula in logic with the help of data structures or solvers, such as, BDDs [6], SAT solvers and SMT solvers [2]. Applications of symbolic model checking methods are usually faster, and therefore, preferred over explicit-state model checking. Some symbolic model checking algorithms, such as IC3 [5], are iterative, that is, they involve multiple runs of the solver. Bounded model checking (BMC) [3] uses symbolic model checking to verify paths of a specified length.

Model checking, specifically of temporal properties, in abstract behavioural models can be performed using a number of available tools, each of which implements a version of the two model checking algorithms. TLA+ [45] (with the TLC model checker) creates and checks behavioural models for a subset of LTL properties using explicit-state model checking. ProB [27] is a tool for analyzing finite B machines against LTL specifications using explicit-state model checking. Iterative symbolic model checking algorithms (such as IC3) for checking B machines are implemented in [25]. None of these approaches use non-iterative symbolic model checking algorithms.

The SMV [30] family of tools, including NuSMV [9] and nuXmv [7], perform symbolic model checking of temporal properties, however, they are quite lacking in language support for expressing abstract models. Chang and Jackson [8] added finite relations and functions to a traditional state-based specification language (*i.e.*, the SMV language [30]), and developed a BDD-based model checker that analyzes these models for CTL specifications.

Del Castillo and Winter [13] provided model checking support for a transition system specified as an Abstract State Machine (ASM) [4], via the translation of a class of ASMs to SMV by restricting the range of functions to finite sets. Translation-based approaches usually unfold user-level abstractions and make understanding models and counterexamples difficult.

The Alloy Analyzer [24], which is implemented using the Kodkod [41] SAT solver, can

be used to perform symbolic model checking. It is fairly straightforward to specify a transition relation in Alloy for BMC: iterate the transition relation to check bounded duration temporal properties [24]. Cunha [12] describes how to perform BMC of LTL properties in Alloy. Electrum [29], an extension of Alloy, can also be used to model check LTL properties using BMC; additionally, Electrum provides a feature to translate the model and properties to nuXmv [7]. DynAlloy [18, 38], another extension of Alloy, focuses on describing models and transition relations using "actions". It does not support any temporal properties. None of these approaches allow checking a full set of temporal properties for the complete (unbounded) model checking problem in Alloy, without extensions or translations.

Transitive-closure-based model checking (TCMC) [17,42,43] encodes CTL with fairness constraints (CTLFC) in first-order logic with transitive closure (FOLTC) in a size linear to the model. It is an expression of the complete (unbounded) model checking problem for a transition system with a finite-state space for CTLFC as a set of constraints in FOLTC without induction, iteration, or invariants. TCMC is inspired by the previous work of Immerman and Vardi [21], which encodes the semantics of CTL and CTL* in FOLTC, but requires an exponential increase in the size of the model with respect to the size of the temporal logic formula. TCMC has been implemented as an Alloy module, making it possible to perform model checking of declarative models of transition systems described in Alloy without translation to another tool.

In this thesis, we focus on using existing tools to make model checking of abstract behavioural models more practical. The Alloy Analyzer is an established tool that provides valuable support for creating abstract models; TCMC is able to check a broad category of temporal (CTLFC) properties of behavioural systems using the Alloy Analyzer without any extension or translation, making it an appropriate candidate for our purpose. We attempt to increase TCMC's practical value by addressing some issues faced by users when modelling and model checking using TCMC in Alloy.

## 1.1 Thesis Overview and Contributions

We investigate the following aspects of using TCMC in Alloy:

### (a) Modelling a Transition System in Alloy

While studying some Alloy models for transition systems, for example those described in [29, 42], we found that they lacked structure and readability. The expected behaviour of

the model was often convoluted: pre- and post-conditions of the operations were unclear, and there was no precise method for separating operations from each other.

We developed guidelines for creating abstract declarative models of transition systems in Alloy that we believe promote structure, modularity, and consistency. These guidelines do not involve any extensions to Alloy. We discuss the implications of using two common styles for defining the transition relation, which we call the *ConjMethod* and the *DisjMethod*.

### (b) Generating an Instance

After creating a model, it is common to inspect an instance of the model to confirm that the expected behaviour is produced. This initial analysis can help catch any modelling errors before starting the model checking process. In Alloy, we can generate an instance of a model by executing the `run` command.

Since the total state space is rarely representable in Alloy due to the state-space explosion problem, we often generate instances using small scopes. When using an exact scope of $n$ states, the Alloy Analyzer produces a full subgraph that can consist of any subset of size $n$ of the state space of the complete transition system. This random subset of states is often not a useful one to inspect and analyze. The instance may not have an initial state or any interesting operations represented. It is unlikely that anything useful, such as, the presence of modelling errors, is deducible from such an instance. We refer to such instances as **spurious instances**.

To eliminate the spurious instance problem, and assist in the production of a useful instance, we created a set of axioms, which we call the **significance axioms**. These axioms prevent the production of spurious instances by ensuring the existence of some key states. To be satisfied, these significance axioms require the instance to be of a large enough size, which we call the **significant scope**.

### (c) Scoped-TCMC Methodology

**Scoped TCMC** for a state scope of $n$, where $n$ is less than the reachable state space, is the model checking of all transition system instances of size $n$ that satisfy the transition relation. The result from scoped TCMC of a property holds for all/some transition system instances of that size, but moreover, we can draw some conclusions about whether the property holds for any complete transition system also. In our scoped-TCMC methodology, we carefully describe the meaning of results from scoped TCMC with respect to the complete model checking problem (meaning over the entire state space), highlighting

distinctions for properties with respect to finite and infinite paths. During TCMC, the significant scope provides a measure independent of computing resource limitations that a significant part of the state space has been verified.

### (d) Comparison to NuSMV and BMC

We compare TCMC in Alloy to NuSMV and BMC in terms of abstract modelling practices, expressibility of temporal properties, model checking results, and performance.

From our investigations, we have the following thesis statement:

> **Thesis statement:** We developed a set of style guidelines with the goal of writing structured, modular, and consistent declarative behavioural models in Alloy. Generating instances of a model with our *significance axioms* produces transition system instances that are more representative of the user model. We define *scoped TCMC* for a state scope of $n$, where $n$ is less than the reachable state space, as the model checking of all transition system instances of size $n$ that satisfy the transition relation. By considering infinite and finite paths of a transition system separately, we can make useful deductions about the complete model checking problem from the results of scoped TCMC for certain categories of properties. The *significant scope*, derived from the significance axioms, provides a measure independent of computing resource limitations that a significant part of the state space has been verified, providing higher confidence in the deductions from scoped TCMC.

The following lists the contributions of this thesis:

- Establishes a set of style guidelines for modelling abstract behavioural systems in Alloy without extensions to Alloy.
- Introduces *significance axioms* and *significant scope* for transition systems, which address the spurious instances problem.
- Introduces *scoped TCMC* to apply TCMC on subgraphs of a complete transition system.

- Analyzes and documents the meanings of scoped TCMC results for different property categories with respect to the complete model checking problem.
- Presents case studies to demonstrate proposed claims and results.
- Investigates the scalability of TCMC in Alloy.
- Compares declarative modelling practices in Alloy to those in NuSMV.
- Compares expressibility of temporal properties, and model checking results of TCMC to those of BMC.

## 1.2 Case Studies

Including a running example of the game of Musical Chairs, we use four case studies, described in Chapter 6, to demonstrate our claims. The case studies illustrate the use of our style guidelines and the benefits associated with them. They show the utility of significance axioms when generating instances for inspection, how to use our proposed TCMC methodology to deduce useful results from scoped TCMC for the complete model checking problem, and how to gain higher confidence by model checking at the significant scope.

We show that these features can be used at scopes generally used for analyzing Alloy models in terms of scalability. We also studied comparable models of our case studies implemented and checked using NuSMV and BMC.

## 1.3 Thesis Organization

This thesis is organized as follows: In the next chapter, we provide brief background material on CTLFC model checking, the Alloy language, TCMC, and the implementation of TCMC in the Alloy Analyzer. Chapter 3 discusses our style guidelines for creating models of transition systems in Alloy. Chapter 4 presents scoped TCMC, the spurious instance problem, and our significance axioms as a solution. In Chapter 5, we describe our methodology for model checking via scoped TCMC and how to interpret the results with respect to the complete model checking problem. Chapter 6 discusses case studies, which demonstrate our claims, and investigates the scalability of TCMC through these case studies. Chapter 7 compares TCMC in Alloy to NuSMV and BMC. We discuss related work in Chapter 8, and conclude in Chapter 9.

# Chapter 2

# Background

In this section, we provide a brief overview on temporal logic model checking, Alloy, TCMC, and its implementation in Alloy.

## 2.1 Temporal Logic Model Checking

Temporal logic model checking is a technique for verifying whether a transition system satisfies a temporal logic property [10].

A transition system is a finite directed graph with vertices and edges. A vertex represents a state of a system, where each state is labelled with some attributes whose values are propositions. A labelling function is used to set the attribute values of the states. An edge between two vertices represents a transition from one state to another.

**Definition 1. Transition System:** The transition system $TS$ is a five tuple, $TS = (S, S_0, \sigma, P, l)$, where: $S$ is a finite set of states; $S_0$, the set of initial states, is a non-empty subset of $S$; $\sigma$, the transition relation, is a binary relation over $S$; $P$ is a finite set of atomic propositions; $l$, the labelling function, is a total function from $S$ to the power set of $P$.

A computation path starting at $s$ where $s \in S$ is a sequence of states, $s_0 \to s_1 \to \ldots$ such that $s_0 = s$ and $\forall i \geq 0 : \ \sigma(s_i, s_{i+1})$. If the transition relation is a total binary relation then all paths starting at each state are infinite computation paths.

A temporal logic property is a set of logical formulas that describe some desirable behaviour of a system over time [10]. A temporal logic, such as CTL or CTLFC [10], has

connectives for specifying properties over the computation paths of a transition system. Equation 2.1 represents the grammar for a complete fragment of CTL:

$$\varphi \ ::= \ p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\varphi \mid EG\varphi \mid \varphi EU\varphi \ , \text{ where } p \in P \tag{2.1}$$

The satisfiability relation for CTL, $\models$, is used to give meaning to formulas. The notation $TS, s \models \varphi$ denotes that the state $s$ of the transition system $TS$ satisfies the property $\varphi$ and $TS, s \not\models \varphi$ is used when $TS, s \models \varphi$ does not hold. The relation $\models$ is defined by structural induction on $\varphi$.

**Definition 2. Semantics of CTL:** For a transition system, $TS$, with a total transition relation, $\sigma$, the semantics of CTL formulas is as follows:

| | | |
|---|---|---|
| $TS, s \models p$ | iff | $p \in l(s)$ |
| $TS, s \models \neg\varphi$ | iff | $TS, s \not\models \varphi$ |
| $TS, s \models \varphi \vee \psi$ | iff | $TS, s \models \varphi$ or $TS, s \models \psi$ |
| $TS, s \models EX\varphi$ | iff | there exists $s' \in S$ such that $\sigma(s, s') \wedge TS, s' \models \varphi$ |
| $TS, s \models EG\varphi$ | iff | there exists a computation path $(s_0 \to s_1 \to \dots)$ starting at $s$ such that for all $i$ $TS, s_i \models \varphi$. |
| $TS, s \models \varphi EU\psi$ | iff | there exists a $j$ and a computation path $(s_0 \to s_1 \to \dots)$ starting at $s$ such that $TS, s_j \models \psi$ and for all $i$ less than $j$ $TS, s_i \models \varphi$. |

The transition system $TS$ satisfies the CTL formula $\varphi$, denoted by $TS \models \varphi$, if and only if for all $s_0 \in S_0$ we have $TS, s_0 \models \varphi$.

The syntax of a complete fragment of CTLFC is the same as Equation 2.1 with the addition of one connective, $E_cG$. In this connective, $c$ is a fairness constraint, which is used to define a *fair* computation path. A computation path $s_0 \to s_1 \to \dots$ is fair with respect to $c$ iff:

$$\{i \mid TS, s_i \models c\} \text{ is infinite.}$$

The semantics of **CTLFC** is the same as Definition 2 along with the semantics of $E_cG$:

| | | |
|---|---|---|
| $TS, s \models E_cG\varphi$ | iff | there exists a fair computation path starting at $s$, $s_0 \to s_1 \to \dots$, such that for all $i$'s $TS, s_i \models \varphi$. |

$$
\begin{array}{lcl}
E_c X\varphi & := & EX(\varphi \wedge (E_c Gtrue)) \\
E_c F\varphi & := & EF(\varphi \wedge (E_c Gtrue)) \\
\varphi_1 E_c U \varphi_2 & := & \varphi_1 EU(\varphi_2 \wedge (E_c Gtrue)) \\
A_c X\varphi & := & \neg E_c X \neg\varphi \\
A_c F\varphi & := & \neg E_c G \neg\varphi \\
A_c G\varphi & := & \neg E_c F \neg\varphi \\
\varphi_1 A_c U \varphi_2 & := & \neg(E_c G \neg\varphi_2) \wedge \neg(\neg\varphi_2 E_c U(\neg\varphi_1 \wedge \varphi_2))
\end{array}
$$

Figure 2.1: Relationship Between CTLFC and CTL Connectives (figure from [42])

Other CTLFC formulas with fairness constraints can be expressed in terms of CTL formulas and $E_c Gtrue$, as shown in Figure 2.1.

## 2.2 Transitive-Closure-based Model Checking (TCMC)

CTL and CTL* can be encoded in first-order logic with transitive closure (FOLTC) for finite models as presented by Immerman and Vardi [21]. Their translation of CTL* requires the introduction of Boolean variables into the model for every sub-formula, and as a result, the number of states in the transition system increases exponentially with respect to the size of the formula.

TCMC [17,42,43] presents a translation of CTLFC to FOLTC with a similar approach to that of Immerman and Vardi. The key difference is that TCMC only considers CTLFC properties. Therefore, each formula can be defined directly, which means that unlike CTL*, the encoding of CTLFC in FOLTC does not increase the size of the transition system. CTLFC is more expressive than CTL, and LTL model checking can be reduced to CTLFC model checking[1] [11].

The general idea of TCMC is to use the (reflexive) transitive closure operator to specify the necessary and sufficient conditions for the set of states that satisfy a CTLFC property. The closure operator is used to specify the reachability relation, which is not expressible in FOL. Similar to traditional representations of CTL model checking, an operator, $[\cdot]$, is defined that takes a formula as input, and outputs a symbolic representation of the set of states that satisfy the input formula, except in TCMC this operator is defined using transitive closure. The recursive definition for $[\cdot]$ is given in Definition 3, where if $X$ is a

[1] This translation increases the size of a transition system.

subset of $S$, then $\sigma_X$ denotes the transition relation $\sigma$ when its domain is restricted to $X$:

$$\sigma_X(s_1, s_2) \text{ iff } \sigma(s_1, s_2) \wedge s_1 \in X$$

Also, ˆ denotes the transitive closure operator; for example, $\hat{}\sigma_X$ is the transitive closure of the relation $\sigma_X$. The reflexive transitive closure operator is $*$.

**Definition 3. TCMC** Let $TS = (S, S_0, \sigma, P, l)$ be a transition system and $c$ be a fairness constraint. The operator $[\cdot]$ takes a CTLFC formula, and produces a subset of $S$:

1. $[p] = \{s \in S| \; p \in l(s)\}$
2. $[\neg\varphi] = \{s \in S| \; s \notin [\varphi]\}$
3. $[\varphi \vee \psi] = [\varphi] \cup [\psi]$
4. $[EX\varphi] = \{s \in S| \; \exists t \in [\varphi] : \; \sigma(s, t)\}$
5. $[EF\varphi] = \{s \in S| \; \exists t \in [\varphi] : \; *\sigma(s, t)\}$
6. $[\varphi EU\psi] = \{s \in S| \; \exists t \in [\psi] : \; *(\sigma_{[\varphi]})(s, t)\}$
7. $[EG\varphi] = \{s \in S| \; \exists t \in [\varphi] : \; *(\sigma_{[\varphi]})(s, t) \; \wedge \; \hat{}(\sigma_{[\varphi]})(t, t)\}$
8. $[E_cG\varphi] = \{s \in S| \; \exists t \in [\varphi] : \; *(\sigma_{[\varphi]})(s, t) \; \wedge \; \hat{}(\sigma_{[\varphi]})(t, t) \; \wedge \; t \in [c]\}$

The definition of $[E_cG\varphi]$ is based on the model checking algorithm of $E_cG$ that finds the strongly connected components (SCCs) in a transition system. The state $t$ in the definition of $[E_cG\varphi]$ is a state that belongs to a SCC and satisfies the fairness constraint, $c$.

Properties with multiple fairness constraints can be converted to an equivalent property with a single fairness constraint using the method described in [42], which is based on Vardi and Wolper's work [44]. Therefore, here, TCMC is described for a single fairness constraint.

**Theorem 1.** *Let $TS = (S, S_0, \sigma, P, l)$ be a transition system, $\varphi$ a CTLFC formula, and $[\cdot]$ the operator defined in Definition 3. We have:*

$$[\varphi] = \{s \in S \;\; | \;\; TS, s \models \varphi\}$$

The proof for Theorem 1 is in [42]. The following corollary of this theorem defines the use of TCMC for model checking a transition system:

**Corollary 1.** *Let* $TS = (S, S_0, \sigma, P, l)$ *be a transition system,* $\varphi$ *a CTLFC formula, and* $[\cdot]$ *the operator defined in Definition 3. We have:*

$$TS \models \varphi \text{ iff } S_0 \subseteq [\varphi]$$

## 2.3 Alloy

Alloy is a lightweight declarative relational modelling language [23]. All structures in Alloy are represented by sets and relations. Alloy supports first-order logic, including set operators, along with the transitive closure operator for modelling systems.

The Alloy Analyzer, which is the primary analysis tool for Alloy models, provides support for finite scope analysis where the user specifies constant sizes of the sets in the model. The Alloy Analyzer translates the model to a propositional CNF formula, which is handed to a SAT solver, called Kodkod [41], for consistency checking.

The Alloy Analyzer evaluates a model for the specified sizes of the defined sets for consistency using the `run` command. If the model is consistent, the `run` command produces instances of the model for user inspection. The Analyzer model checks against assertions at small specified set scopes using the `check` command.

An Alloy model consists of:

- Declarations, which specify the sets, relations, and functions in a model. These declarations dictate the state space of the model. The keyword `sig` is used to initiate a set structure. The sizes of these structures are not declared at this point – the sizes are specified when we generate instances of the model or check the model for properties.
- Constraints, which are logical formulas, that limit the instances of the model. The keyword `fact` creates blocks containing these formulas that must hold in the model. The keyword `pred` is used to declare named predicate formulas (these `pred`s can be thought of as macros).
- Assertions are formulas that should hold in all instances of the model, and can be checked by the Alloy Analyzer. The keyword `assert` is used to define properties that can be checked by the Analyzer. The `check` command, along with the desired scope

sizes of all declared sets, is used to check the validity of an assertion. If the assertion does not hold, then a counterexample instance of the model's sets and relations is shown.

## 2.4 TCMC in Alloy

TCMC has been implemented in the Alloy language in two modules, `ctlfc` and `ctl` [42]; `ctlfc` is used for modelling systems with fairness constraints, whereas, `ctl` is used when no fairness constraints need to be modelled. The implementation is shown in Figure 2.2. The `TS` (Lines 1–5) declares the sets and relations that are needed to describe a transition system, where `S0` refers to the initial states, `sigma` refers to the transition relation, and `FC` refers to the set of fair states if a fairness constraint is present. These are accessed using the functions on Lines 7–9.

TCMC (Definition 3) of CTLFC is implemented as Alloy functions as shown in Figure 2.2 Lines 16–36. It uses two helper functions, `domainRes` and `id`, implemented and explained in Lines 11–14. `domainRes[R,X]` is the subset of `R` with its domain restricted to `X`; `id[X]` is the identity relation over `X`. The functions take advantage of the Alloy join function, "`.`". For example, the `.S` on Line 25 extracts the domain from the relation produced in the rest of the expression. The `ecg` function (Lines 27–29) shows the implementation of a formula with a fairness constraint: `TS.FC` (Line 28) ensures that only states that reach a fair state infinitely often are considered. Other CTLFC formulas with fairness constraints are implemented in the full `ctlfc` module in terms of CTL formulas and $E_c Gtrue$ (the relationship is shown in Figure 2.1). Also included in the modules are the universal path quantifiers, `AX, AG, AU, AG`, defined in terms of the existential temporal operators (Lines 31–36).

The two modules, `ctl` and `ctlfc`, are separated because when there are no fairness constraints, it was found that using the `ctl` module without the fairness constraints leads to better performance. These modules are available on-line[2] [17].

An example template for developing a model to use with TCMC is shown in Figure 2.3. The `ctlfc` module is imported in the model file (Line 1). In the `modelDefinition`, on Line 6, the `initialState` function from the module is equated with the initial state constraints of the model. Similarly, the `nextState` relation, and the fairness constraint (`fc`), if any, are set up. Then the `ctlfc_mc` function is used (Lines 10-15) to perform model checking

[2] http://www.cs.uwaterloo.ca/~nday/models/TCMC-in-Alloy

```
1   private one sig TS{
2     S0: some S,    // initial states
3     sigma: S -> S, // next-state relation
4     FC: set S      // fair states
5   }
6   ----------------------- MODEL -----------------------------
7   fun initialState: S {TS.S0}        // initial state constraints
8   fun nextState: S -> S {TS.sigma}  // transition relation
9   fun fc: S {TS.FC}                  // fairness constraints
10  ------------------ HELPER FUNCTIONS _--------------------
11  // domainRes[R,X]={(x, y) in R | x in X}
12  private fun domainRes[R: S -> S, X: S]: S -> S {X <: R}
13  // id[X]={(x, x) | x in X}
14  private fun id[X:S]: S->S {domainRes[iden,X]}
15  ----------------- LOGICAL OPERATORS ----------------------
16  fun not_[phi: S]: S {S - phi}
17  fun and_[phi, si: S]: S {phi & si}
18  fun or_[phi, si: S]: S {phi + si}
19  fun imp_[phi, si: S]: S {not_[phi] + si}
20  ----------------- TEMPORAL OPERATORS ----------------------
21  fun ex[phi: S]: S {TS.sigma.phi}
22  fun ef[phi: S]: S {(*(TS.sigma)).phi }
23  fun eu[phi, si: S]: S {(*(domainRes[TS.sigma, phi])).si}
24  fun eg[phi: S]: S { let R= domainRes[TS.sigma,phi]|
25    *R.((^R & id[S]).S)
26  }
27  fun ecg[phi:S]:S { let R= domainRes[TS.sigma,phi]|
28      *R.((^R & id[S]).S & TS.FC)
29  }
30  ---------------DERIVED TEMPORAL OPERATORS -----------------
31  fun ax[phi:S]:S {not_[ex[not_[phi]]]}
32  fun af[phi: S]: S {not_[eg[not_[phi]]]}
33  fun au[phi, si: S]:S {
34    not_[or_[eg[not_[si]], eu[not_[si], not_[or_[phi, si]]]]]
35  }
36  fun ag[phi: S]: S {not_[ef[not_[phi]]]}
37  -------------------- MODEL CHECKING -----------------------
38  // used for model checking in user's model file
39  pred ctlfc_mc[phi: S] {TS.S0 in phi}
```

Figure 2.2: TCMC implementation in Alloy

```
1 open ctlfc[State] as ctlfc
2
3 sig State { ... }
4
5 fact modelDefinition {
6   all s:State | s in initialState iff ...
7   all s,s':State | s->s' in nextState iff ...
8   all s:State | s in fc iff ...
9 }
10 // universal TCMC
11 check {ctlfc_mc[ag[{s:State| <universal_property>}]]}
12   for exactly <scope>
13 // existential TCMC
14 run {ctlfc_mc[ef[{s:State| <existential_property>}]]}
15   for exactly <scope>
```

Figure 2.3: Template for use of TCMC

tasks. The template shows the use of the `ag` and `ef` temporal logic properties, but others can be used.

If the declarative model of a transition system is not uniquely defined for a set of states, there can be multiple transition system instances (TS instances) that satisfy its constraints. For example, the declarative specification "every state must reach a state that is reachable from itself" specifies more than one transition system for two states, as shown in Figure 2.4.

Figure 2.4: Multiple instances of a transition system for the constraint: "every state must reach a state that is reachable from itself" (figure from [42])

**Universal TCMC** checks whether the property is satisfied on all paths starting from all initial states in *all* TS instances of the model. To implement universal TCMC, we use `ctlfc_mc` with `check`, as shown in Figure 2.3, Line 11. Universal TCMC addresses the complete model checking problem only if all the states in the reachable state space are in the scope of the `check` command. Similar to other CTL model checkers, TCMC in Alloy calculates the set of states that satisfies the given property in a TS instance. If all initial states of the TS instance are in this satisfying set of states, meaning that the property

is satisfied in that TS instance, we get a `No counterexample found` result. If any initial state of the TS instance is not in the satisfying set of states, we get a counterexample – an inspectable transition system that is an instance of our model containing a path that violates the checked property. As with other CTL model checkers, a particular counterexample path has to be extracted or observed within the instance. Since instances in Alloy are generally small, we rely on observation of the TS instance to determine the bug in the model.

**Existential TCMC** checks if *some* TS instance of the model satisfies the property. For existential TCMC, we use `ctlfc_mc` with `run`, as shown in Figure 2.3, Line 13. If the model constraints are consistent with the temporal logic property, the Analyzer shows a transition system that is a valid instance of our model. Otherwise, we see an `Instance not found. Predicate may be inconsistent` message.

If a model defines a unique transition system, then there is only one TS instance that satisfies the model, which means that both, universal and existential TCMC, only check one TS instance (assuming that all states in the reachable state space are included), resulting in the same result. Therefore, universal and existential TCMC differ only if there are multiple TS instances that satisfy the model description.

## 2.5 Summary

A transition system is a finite directed graph with vertices and edges, where the vertices represent states, and the edges represent transitions, with a non-empty set of initial states. A temporal logic (*e.g.*, CTLFC) property is a set of logical formulas that describe some desirable behaviour of a system over time. Temporal logic model checking verifies whether a transition system satisfies a temporal logic property. Transitive-closure-based model checking (TCMC) defines CTLFC formulas for finite models in terms of first-order logic with transitive closure (FOLTC). TCMC has been implemented in Alloy, a language used to create abstract declarative models, which can be evaluated using the Alloy Analyzer.

# Chapter 3

# Modelling a Transition System in Alloy

While studying some existing Alloy models of transition systems, for example those described in [29, 42] (some of these models are included for comparison in Appendix C), we found that they lacked structure and readability. The expected behaviour of the model, such as the operation shown in Figure 3.1 (full model in Appendix C.2), was often convoluted: pre- and post-conditions of the operations were unclear, and there was no precise method for separating operations from each other.

We developed guidelines for writing Alloy models of transition systems that we believe

```
 1 pred Act[p : Process, t,t' : State] {
 2   no lvl.t[p]
 3   (p = Root) => {
 4     lvl.t'[p] = lo/first
 5     no parent.t'[p]
 6   } else {
 7     some adjProc: p.adj {
 8       some lvl.t[adjProc]
 9       lvl.t'[p] = lo/next[lvl.t[adjProc]]
10       parent.t'[p] = adjProc
11     }
12   }
13 }
```

Figure 3.1: Predicate from Span Tree Alloy Model [29]

```
1 pred pre_Act_Root[p : Process, t : State] {
2   no lvl.t[p]
3   p = Root
4 }
5 pred post_Act_Root[p : Process, t,t' : State] {
6   lvl.t'[p] = lo/first
7   no parent.t'[p]
8 }
9 pred pre_Act_NonRoot[p : Process, t : State] {
10   no lvl.t[p]
11   not (p = Root)
12 }
13 pred post_Act_NonRoot[p : Process, t,t' : State] {
14   some adjProc: p.adj {
15     some lvl.t[adjProc]
16     lvl.t'[p] = lo/next[lvl.t[adjProc]]
17     parent.t'[p] = adjProc
18   }
19 }
```

Figure 3.2: Applying Our Guidelines to Predicate from Span Tree Alloy Model [29] in Figure 3.1

promote structure, modularity, and consistency in the model: In this context, the term *structure* refers to the readability of the model, which we believe is improved by following a template, the term *modularity* refers to the ability to retain existing behaviour when an element/behaviour is added to the model, and the term *consistency* refers to the likelihood that at least one TS instance of the model exists. For example, we find the operation in Figure 3.2, where we applied our guidelines to the operation in Figure 3.1, more structured and modular. Our guidelines, which do not involve any extensions to Alloy, are presented via an example in this section.

We use the term **model** to refer to the user-written description of a system in Alloy, and the term **instance** to refer to a collection of the declared set elements and relations of the specified scope sizes produced by the Alloy Analyzer that represents the model. A **transition system instance** (TS instance) is an instance produced by the Analyzer that is a transition system, which means that it contains some state elements, at least one of which is an initial state, and a transition relation. We use the term **operation** to refer to a part of the Alloy model that defines a user-level grouping of some of the system's transitions, meaning that an operation defines a group of transitions in the TS instance that are all the same behaviour to the user.

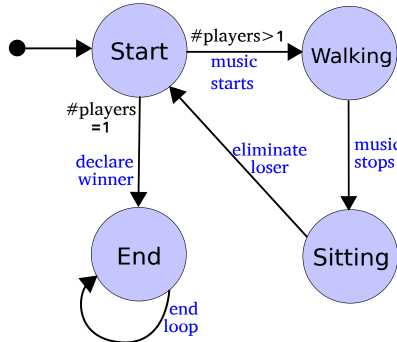



Figure 3.3: Musical Chairs Overview

We use the game of Musical Chairs to illustrate an Alloy model of a transition system. Our model is based on Nissanke's model of Musical Chairs [35]. As illustrated in Figure 3.3, each round of the game moves through the modes Start, Walking, Sitting and End. The number of rounds will depend on the number of players; we wish to write a flexible model description that can be used for any number of players, and choose the number of players by setting a finite scope when we generate an instance and analyze the model.

Our behavioural model for Musical Chairs in Alloy consists of three parts: 1) the declaration of the state space, 2) the initial state constraints, and 3) the constraints describing the operations. We combine the constraints describing the operations to create the transition relation in a standard way.

In the rest of this chapter, we describe how we create each of these pieces, highlighting the practices that promote structure, modularity, and consistency in the model. The complete Musical Chairs Alloy model, discussed throughout the rest of the thesis, can be found in Appendix A.1.

## 3.1 Declaring the State Space

First, we declare the state space for our system. The state-space declaration for Musical Chairs, as shown in Figure 3.4, consists of the uninterpreted sets [24] `Chair` and `Player`, and the four possible modes. The `State` set encapsulates the current set of players, chairs, mode, and chair occupancy by players, `occupied`, which is a relation from chairs to players. The use of uninterpreted sets, such as `Chair` and `Player`, plus the use of the relation

```
1 sig Chair, Player {}
2 abstract sig Mode {}
3 one sig start, walking, sitting, end extends Mode {}
4 sig State {
5   players: set Player,
6   chairs: set Chair,
7   occupied: set Chair -> set Player,
8   mode : set Mode
9 }
10 pred equality [s,s':State] {
11   (s.players=s'.players and
12    s.chairs=s'.chairs and
13    s.occupied=s'.occupied and
14    s.mode=s'.mode)
15      implies s = s'
16 }
```

Figure 3.4: Musical Chairs State Space

`occupied` are examples of the abstractions possible in declarative models, which make models concise, but precise.

### 3.1.1 State Equivalence

The encapsulation provided by the `State` set is convenient, but in Alloy such encapsulation is not a record, rather the elements of `State` are a distinct set, and the attributes are mappings from a `State` element to a set of players, chairs *etc.* Two elements with the same attribute values are treated as two distinct elements by default. To match our intuition that states with the same attributes are equivalent, we introduce an *equality predicate*, shown in Lines 11-17 in Figure 3.4. This predicate requires that, if all of the attributes, that is, `players`, `chairs`, *etc.*, of two states are exactly the same, then the two state elements are equal. (The predicate is added as a constraint in the model definition part of the model.) This guideline removes a source of inconsistency between the user's expectation of their model and the actual model.

### 3.1.2 No Invariants

In Alloy, all elements are modelled as sets or relations. Alloy provides keywords for multiplicity constraints, such as `lone` and `one`, to constrain relations (and the reachable state

space). These constraints are implicit invariants on the behaviour of the model. Therefore, using these keywords when declaring our state space can lead to inconsistencies in our model, because, it is possible to write contradictory constraints in another part of the model, such as, the initial state or operations constraints. For example, in the model in Figure 3.5, the state-space declaration, which requires `attribute` to contain only one element (Line 3), contradicts the initial state constraint, which requires the initial state's `attribute` to contain two elements (Line 7). Executing the `run` command in Line 10 produces a `No instance found. Predicate may be inconsistent` result. (Similar challenges of maintaining consistency exist in other modellings techniques as well, such as, UML diagrams, as described in [28].)

```
1 one sig P, Q, R extends Attribute {}
2 sig State {
3   attribute: one Attribute
4 }
5 // initial state constraints
6 pred init [s:State] {
7   (P+Q) in s.attribute
8 }
9 ...
10 run {} for 3
```

Figure 3.5: Part of an Arbitrary Model

Therefore, it is important to consider carefully how our model elements should behave in any given situation before writing invariants in the declarations. To avoid potential pitfalls, we recommend not declaring any state invariants in the model. Any constraints dictating the behaviour of the model should be defined as part of the initial state or operations constraints. If there are invariants that are expected to hold throughout the model, then these invariants should be model checked as properties to ensure modelling correctness.

In reflection of this guideline, in our state-space declaration, we use the `set` keyword in Lines 5–8 of Figure 3.4 to define all the attributes in `State`. All structures, including single-element structures, must be declared as sets in Alloy, whereas, many other languages allow declaring individual elements. While this guideline is not sufficient to guarantee that the model is consistent, it does remove a source of inconsistency in models of transition systems.

```
1 pred init [s:State] {
2   s.mode = start
3   #s.players > 1
4   #s.players = (#s.chairs).plus[1]
5 }
```

Figure 3.6: Initial state constraints

## 3.2 Defining Initial States and Operations

Next, we set up our initial state constraints and define our operations. The initial state constraints for Musical Chairs, shown in Figure 3.6, set the initial `mode` to `start` and constrain the number of players in the game to be greater than one. They also ensure that the number of players in the game initially exceeds the number of chairs by one.

There are five operations in our Musical Chairs model, as shown in Figure 3.3: `music_starts`, `music_stops`, `eliminate_loser`, `declare_winner`, and `end_loop`. Figure 3.7 shows the predicates that define the `eliminate_loser` operation in Alloy.

### 3.2.1 Pre- and Post-conditions

For the sake of structure in the model description, we separate the operation descriptions into separate predicates for the pre- and post-conditions. The pre-condition is a constraint on the previous state and the post-condition is a constraint on the previous and the next states.

In Figure 3.7, the pre-condition states that the eliminate-loser operation can only occur

```
1 pred pre_eliminate_loser [s: State] {
2   s.mode = sitting
3 }
4 pred post_eliminate_loser [s, s': State] {
5   s'.mode = start
6   // loser: player in game not in the range of occupied
7   s'.players = Chair.(s.occupied)
8   #s'.chairs = (#s.chairs).minus[1]
9 }
```

Figure 3.7: Eliminate Loser Operation in Musical Chairs: Pre- and Post-Condition

when the state is in the `sitting` mode (Line 2). The post-condition sets the mode after the operation to `start` (Line 5). Line 7 removes from the game the player not in the range of the `occupied` relation, who is the loser of the round – this is an example of the declarative nature of the model. By decrementing the number of chairs in Line 8, we remove a chair; this modelling structure is also declarative – the chair to be removed is not explicitly chosen. Thus, we create declarative constraints for operations separated as pre- and post-conditions.

### 3.2.2 Distinct Operations

We recommend separating operations according to the two following rules:

1. *A transition in a TS instance cannot belong to more than one operation*: Each operation defines a distinct set of transitions with no overlap. Equation 3.1 represents this rule, showing that the pre- and post-conditions of two operations cannot be satisfied by the same pair of states. In the equation, $op_1$ and $op_2$ are operations, $s$ and $s'$ are states, and $op_{1_{pre}}(s)$, $op_{1_{post}}(s, s')$, $op_{2_{pre}}(s)$, and $op_{2_{post}}(s, s')$ are pre- and post-conditions of $op_1$ and $op_2$, respectively.

$$\forall op_1, op_2 \;\cdot\; \forall s, s' \;\cdot\; \neg((op_{1_{pre}}(s) \wedge op_{1_{post}}(s, s')) \wedge (op_{2_{pre}}(s) \wedge op_{2_{post}}(s, s'))) \quad (3.1)$$

2. *No atomic formula (meaning a formula with no logical connective) in the post-condition depends only on the previous state*: Any atomic formula depending only on the previous state should be part of the pre-condition of the operation.

```
1 pred pre_from_start [s:State] {
2   s.mode = start
3 }
4 pred post_from_start [s,s':State] {
5   #s.players > 1 implies (s'.mode=walking and s'.occupied=none->none)
6   #s.players = 1 implies s'.mode = end
7   s'.players = s.players
8   s'.chairs = s.chairs
9 }
```

Figure 3.8: Example of a Non-distinct Operation

```
1 pred pre_music_starts [s: State] {
2   #s.players > 1
3   s.mode = start
4 }
5 pred post_music_starts [s, s': State] {
6   s'.players = s.players
7   s'.chairs = s.chairs
8   s'.occupied = none -> none
9   s'.mode= walking
10 }
11 pred pre_declare_winner [s: State] {
12   #s.players = 1
13   s.mode = start
14 }
15 pred post_declare_winner [s, s': State] {
16   s'.players = s.players
17   s'.chairs = s.chairs
18   s'.mode = end
19 }
```

Figure 3.9: Operations in Musical Chairs: `music_starts` and `declare_winner`

We refer to an operation that follows these two rules as a **distinct operation**. Creating distinct operations promotes modularity in our model (which also facilitates our significance axioms in the next section).

In Musical Chairs, instead of creating a single operation from the `start` mode, as shown in Figure 3.8, which breaks our second rule due to the use of atomic propositions that depend only on the previous state, `s.players > 1` (Line 5) and `s.players = 1` (Line 6), in the post-condition, we create separate operations, `music_starts` and `declare_winner` (Figure 3.9): `music_starts` occurs when the number of players is greater than 1 in the previous state (Line 2), and `declare_winner` occurs when the number of players is equal to 1 in the previous state (Line 12).

## 3.3 Model Definition

To add structure to the model, we create a `fact` block to compose the initial state constraints and operations into a transition system. We call this block the *model definition*. Lines 6–15 of Figure 3.10 show a model definition `fact` block: it matches the template of

Figure 2.3 (in Section 2.4) and begins to make use of the `ctl/ctlfc` modules. It equates the `initialState` and `nextState` functions from the `ctl/ctlfc` modules to the model-specific constraints. A state can be an initial state if and only if it satisfies the constraints set in the `init` fact, and a pair of states can be in the `nextState` relation if and only if it satisfies the constraints in one of the operations. Since we use the "if and only if" connective to define the transition relation (`iff` in Line 7), we define a single transition system in Alloy (if all the states in the reachable state space are in the scope), as is commonly done when describing models in other languages. The model definition `fact` also enforces the `equality` predicate described previously for all elements of `State`. Although it is often common to define a total transition relation, a model defined in this way does not guarantee a total transition relation.

### 3.3.1 Transition Relation: DisjMethod vs. ConjMethod

In this section, we explore two common but distinct methods for defining the transition relation of a declarative model, which we call the **disjunctive modelling method** (DisjMethod) and the **conjunctive modelling method** (ConjMethod). In the **DisjMethod**, (Musical Chairs implementation shown in Figure 3.10), each operation is the conjunction of its pre- and post-conditions (Lines 1–4), and the `nextState` relation, which is the transition relation for the system, is a disjunction of the definitions of each operation (Lines

```
 1 pred eliminate_loser [s, s': State] {
 2   pre_eliminate_loser[s]
 3   post_eliminate_loser[s,s']
 4 }
 5 fact modelDefinition{
 6   all s:State | s in initialState iff init[s]
 7   all s,s':State | s->s' in nextState iff
 8     (music_starts[s,s'] or
 9      music_stops[s,s'] or
10      eliminate_loser[s,s'] or
11      declare_winner[s,s'] or
12      end_loop[s,s'])
13   all s, s': State | equality[s,s']
14 }
```

Figure 3.10: DisjMethod for Eliminate Loser Operation and Musical Chairs Model Definition

8–13). In the **ConjMethod** (Musical Chairs implementation shown in Figure 3.11), we define each operation as an implication, where the pre-condition implies the post-condition (Lines 1–3), and conjunct the definitions of all the operations (similar to Dijkstra's guarded commands [14]) (Lines 5–9).

```
 1 pred eliminate_loser [s, s': State] {
 2   pre_eliminate_loser[s] implies post_eliminate_loser[s,s']
 3 }
 4 fact modelDefinition{ all s,s':State | s->s' in nextState iff
 5     (music_starts[s,s'] and
 6      music_stops[s,s'] and
 7      eliminate_loser[s,s'] and
 8      declare_winner[s,s'] and
 9      end_loop[s,s'])
10     ...   }
```

Figure 3.11: ConjMethod Method for Defining `nextState` Relation

For Musical Chairs, these two modelling methods yield equivalent transition relations, but this is not the case for all models. The two methods produce equivalent transition relations when the pre-conditions are mutually exclusive and complete (some pre-condition is satisfied in every state). Otherwise, the transition relations resulting from these two methods can differ.

If a state satisfies multiple operations' pre-conditions, that is, the pre-conditions are not mutually exclusive, then the transition relation from the DisjMethod can include more transitions than the ConjMethod. Figure 3.12 illustrates this case; the figure only shows transitions that *start* from $S1$, and each state is labelled with the pre- and post-conditions that hold in it. For the ConjMethod, all operations from a state that satisfies their pre-conditions ($S1$) must have their post-conditions satisfied in the next state ($S4$). This requires the next state to satisfy the post-conditions of multiple operations at the same time, thus there are fewer transitions. But for the DisjMethod, only one of the possible post-conditions from a state that satisfies their pre-conditions ($S1$) needs to hold in the next state ($S2, S3, S4$). So there is a higher number of transitions included in the transition relation for the DisjMethod.

The opposite happens when the pre-conditions of the operations are incomplete, that is, they do not cover all states. Figure 3.13 illustrates this case; the figure shows *all* transitions occurring between the two states. From a state that does not satisfy any pre-condition ($S6$), transitions to all other states ($S5, S6$) are included in the transition relation for the ConjMethod, because the antecedent of the implications in all the operations is false. So

Transition Relation Definition in DisjMethod (solid lines):
$\sigma(s, s') \Leftrightarrow$
$(pre_1(s) \wedge post_1(s, s')) \vee$
$(pre_2(s) \wedge post_2(s, s')))$

---

Transition Relation Definition in ConjMethod (dashed lines):
$\sigma(s, s') \Leftrightarrow$
$(pre_1(s) \Rightarrow post_1(s, s')) \wedge$
$(pre_2(s) \Rightarrow post_2(s, s')))$

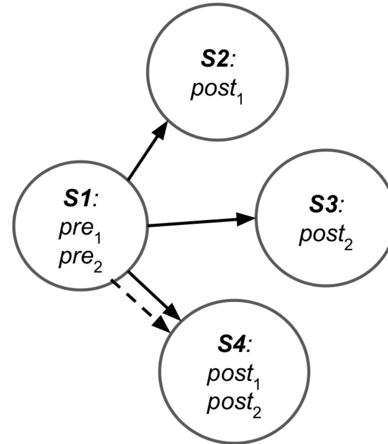



Figure 3.12: Overlap in pre-conditions. Shows only transitions starting from $S1$.

there are more transitions included in the transition relation for the ConjMethod than the DisjMethod in this scenario, although none of these extra transitions are likely ones the user is expecting.

While the modelling style is a matter of user preference, we prefer the DisjMethod because it is more modular and additive (*i.e.*, an added operation does not change the behaviour of the existing operations) in nature than the ConjMethod, and we believe it is

Transition Relation Definition in DisjMethod
(solid lines):
$\sigma(s, s') \Leftrightarrow$
$(pre_1(s) \wedge post_1(s, s')) \vee$
$(pre_2(s) \wedge post_2(s, s')))$

---

Transition Relation Definition in ConjMethod
(dashed lines):
$\sigma(s, s') \Leftrightarrow$
$(pre_1(s) \Rightarrow post_1(s, s')) \wedge$
$(pre_2(s) \Rightarrow post_2(s, s')))$

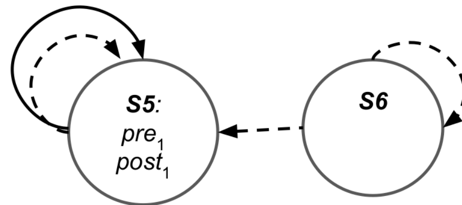



Figure 3.13: Incomplete pre-conditions. Shows all transitions between $S5$ and $S6$.

more likely to produce a transition relation that the modeller is expecting.

## 3.4 Summary

In summary, our style guidelines for modelling a transition system in Alloy consist of:

- *State Equivalence*: Use an equality constraint to force all states with the same attributes to be equivalent. This guideline promotes consistency.
- *No Invariants*: Do not write invariants to define attributes of the state. It is unclear whether operations maintain these constraints, or whether they are invariants to verify. This guideline promotes consistency.
- *Pre- and Post-Conditions*: Separate the description of each operation into separate facts for pre- and post- conditions. This guideline promotes structure.
- *Distinct Operations*: Operations should be defined such that each transition belongs to only one operation, and no atomic formula in the post-condition depends only on the previous state. This guideline promotes modularity.
- *DisjMethod vs. ConjMethod*: Build the transition relation using the DisjMethod (disjunct together the conjunctions of the pre- and post-conditions for each operation), instead of the ConjMethod. This guideline promotes structure and modularity.

Our style guidelines are useful for describing a transition system in Alloy in a structured, modular, and consistent manner, which may be analyzed via TCMC or BMC in the Alloy Analyzer.

# Chapter 4

# Generating an Instance

After creating a model, it is common to inspect an instance of the model to confirm that the expected behaviour is produced. This initial analysis can help catch any modelling errors before starting the model checking process.

In Alloy, we can execute the `run` command to produce an instance of a model with a finite number of states. This chapter discusses common methods for generating an instance of a model in Alloy, some problems involved, and potential solutions.

## 4.1 Scope

To generate an instance of our model, we use the Alloy Analyzer's finite model finding capabilities by executing the `run` command. As part of this command, we need to provide scope sizes for the sets declared in our state space. Since we force all elements with the same attributes to be equivalent (using our state equality predicate), if we consider finite sizes for our uninterpreted sets (`Players` and `Chairs` for Musical Chairs), the maximum size of all other sets (for example, the `State` set) can be determined, generating a total state space. For example, in Musical Chairs, if we consider a game starting with 5 players and 4 chairs, we can calculate the number of `State` elements in the total state space to be the product of the number of possible combinations of players ($2^5$, because the players currently in the game, `players`, is defined as a `set` of `Player`, meaning that any of the 5 `Player` elements can be present or absent from the set of `players`), the number of possible combinations of chairs ($2^4$), the number of possible combinations of the occupied relation ($2^9$), and the number of possible combinations of modes ($2^4$), which comes to 4,194,304. If we could set

the scope of the `State` set to this size when executing the **`run`** command, we would get a transition system instance containing all possible states (that is, states with all possible attribute value combinations, reachable and unreachable), and all transitions among them – we call such a transition system the **complete transition system**. However, this total state space is usually too large to produce in the Alloy Analyzer due to computing resource limitations. One potential strategy to decrease the state scope would be to attempt to limit the scope to the number of reachable states, however, this is also usually too big, and not possible to calculate *a priori*.

Instead, we follow Jackson's small scope hypothesis [24], and start generating instances at small scopes for the `State` set with the goal of producing a transition system instance that is useful for analyzing model correctness with the least amount of resources possible. (We also start at small scopes when model checking for properties to find bugs in the system, which we discuss in detail in the next chapter). We call generating instances of models at scopes smaller than the total-state-space size **scoped generation**. For a given exact state scope of $n$ in scoped generation, the Alloy Analyzer produces a *full subgraph* of size $n$ of a complete transition system for inspection. A full subgraph of a graph is a subset of the nodes, along with all edges between these nodes that are found in the original graph.

## 4.2 Spurious Instance Problem

When we generate an instance using an exact scope, $n$, in the Alloy Analyzer, a full subgraph that can consist of any subset of size $n$ of the total state space is produced. This random subset of states is often not a useful one to inspect and analyze. The instance may not have an initial state or any interesting operations represented. In Musical Chairs, for example, executing a command such as `run {} for exactly 3 Player, exactly 2`

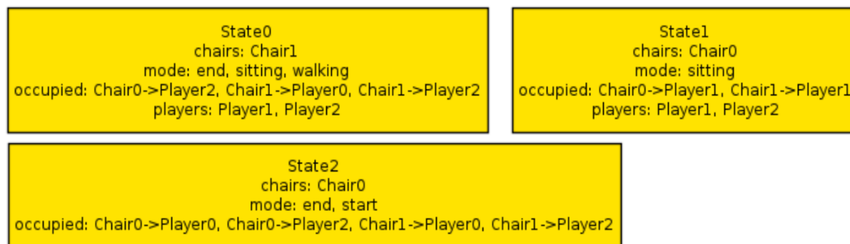



Figure 4.1: Musical Chairs: 3-State Spurious Instance

`Chair, exactly 3 State` for our current model produces a random, disconnected set of states shown in Figure 4.1. The `initialState` set and the `nextState` relation produced are empty, and two of the states have multiple modes, which are unreachable in our model. It is unlikely that anything useful, such as, the presence of modelling errors, is deducible from such an instance. We refer to such instances as **spurious instances**. In our context, a spurious instance exhibits one or more of the following characteristics:

- *No initial state present:* When no state is included that satisfies the initial state constraints, the instance does not have a starting point, making it less useful for inspection.
- *Includes unreachable states:* Unreachable states are generally not useful for inspection, therefore, it is important for the states included to be reachable.
- *Absence of reachable transitions representing all operations defined in the model:* The

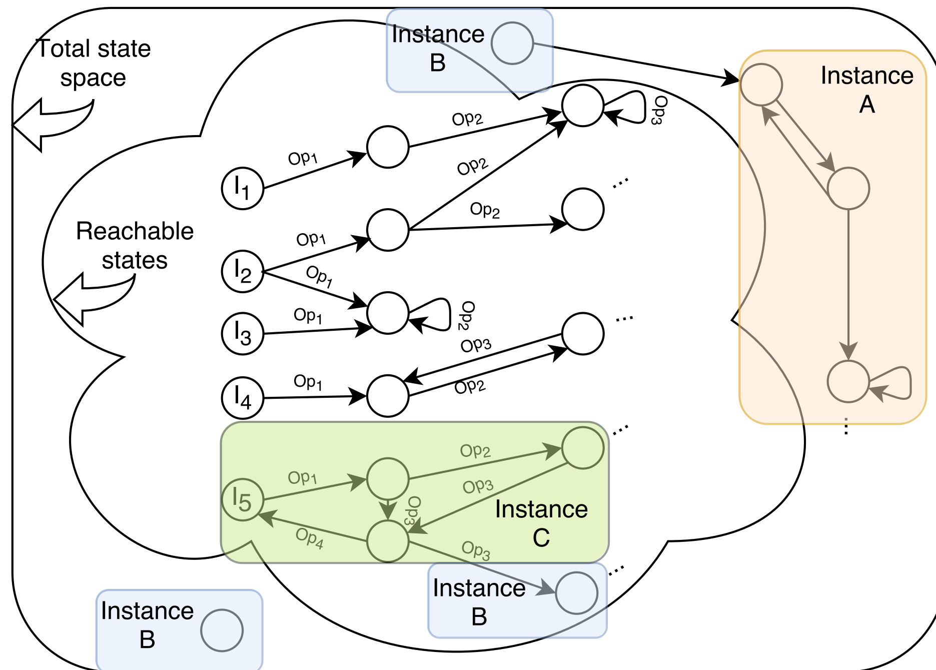



Figure 4.2: Arbitrary Model: State Space and Instances. $I_n$: initial states, $Op_n$: user-defined operations. Intances A and B are Spurious. Instance C is Non-Spurious.

user may wish to inspect reachable representatives of all operations in order to check for modelling errors.

In Figure 4.2, which shows some instances of an arbitrary model, instances A and B are spurious. They are both instances produced from a state scope set to 3. Instance B does not contain any transitions, and neither instance contains a valid initial state. Also, instance A has no reachable states, and instance B has only one. These two instances provide no useful feedback to the modeller about the correctness of the model.

We call the inability to produce a useful instance for inspection during scoped generation as the **spurious instance problem**.

## 4.3 Significance Axioms

To tackle the spurious instance problem, and to assist in the production of a useful instance during scoped generation, we propose a set of axioms, which we call the **significance axioms**. When included as facts in the model, these axioms address each of the deficiencies listed above to prevent the production of spurious instances by ensuring the existence of some key states. Our significance axioms are:

- *Reachability Axiom*: All states produced must be reachable from an initial state. This axiom also ensures that an initial state is included, and all transitions in the instance are reachable. Equation 4.1 represents this axiom, where $s$ and $s_i$ are states, $\sigma$ is the transition relation, and $S_0$ is the set of initial states (recall that $*$ is the reflexive transitive closure operator):

$$\forall s \cdot \exists s_i \cdot *\sigma(s_i, s) \land s_i \in S_0 \tag{4.1}$$

- *Operations Axiom*: At least one reachable transition that satisfies each operation must be produced. Equation 4.2 represents this axiom, where $s$ and $s'$ are states, $op$ is an operation, and $op(s, s')$ is a predicate where $(s, s')$ satisfies the operation $op$.

$$\forall op \cdot \exists s, s' \cdot op(s, s') \tag{4.2}$$

To be satisfied, these axioms, which are added as facts to the model, require a "big enough" state scope, because there is a minimum set of (reachable) states that need to be included if we wish each operation to be represented, as required by the Operations Axiom. These axioms work whether the transition relation is uniquely defined or not.

One can view our significance axioms as an example of Jackson's generator axioms [24] that are specific for transition systems. Jackson describes generator axioms as a means of forcing certain sets to be fully populated when verifying an assertion, so that instances that do not satisfy the assertion due to a scope limitation, that is, instances where increasing the scope would satisfy the assertion, are not produced as counterexamples. Jackson's generator axioms focus on the generation of states with all possible attribute values, which can cause the state scope to explode. Our significance axioms focus on generating a transition representing each possible operation. The scope only increases linearly to the finite number of operations defined (depending on the reachability of transitions representing the operations), which is usually smaller than that of all possible state attribute combinations.

The significance axioms for Musical Chairs are shown in Figure 4.3. The Reachability Axiom (Lines 1–3) is generic for all models. The Operations Axiom (Lines 4–10) ensures that each of the model's operations (in this case, `music_starts`, `music_stops`, *etc.*) are represented in the produced instance. Lines 16–17 shows the `run` command used to include the significance axioms in the model when generating an instance. These axioms ensure that we have an instance in which some player wins, but the transition system is not required to include a path for every player to win.

```
 1 pred reachabilityAxiom {
 2   all s:State | s in initialState.*nextState
 3 }
 4 pred operationsAxiom {
 5   some s,s':State | music_starts[s,s']
 6   some s,s':State | music_stops[s,s']
 7   some s,s':State | eliminate_loser[s,s']
 8   some s,s':State | declare_winner[s,s']
 9   some s,s':State | end_loop[s,s']
10 }
11 pred significanceAxioms {
12   reachabilityAxiom
13   operationsAxiom
14 }
15 run significanceAxioms for exactly 3 Player,
16   exactly 2 Chair, exactly 8 State
```

Figure 4.3: Musical Chairs: Significance Axioms

## 4.4 Significant Scope

As discussed previously, when generating an instance using the `run` command in the Alloy Analyzer, we specify a (usually small) state scope. After we add the significance axioms to our model, we may find that too small a scope makes our model inconsistent and no instance is found. As previously mentioned, a minimum set of states is required to satisfy the Operations Axiom, which means that a big enough state scope must be provided to generate an instance. Hence, we iteratively increment our state scope until we have a consistent model, and an instance is produced. We call the minimum scope at which the significance axioms are satisfied the **significant scope**.

For the abstract example of Figure 4.2, a scope of 4 states (Instance C) is needed to satisfy the significance axioms. Instance C contains only reachable states, an initial state, and a transition for each operation.

This significant scope is also useful when performing scoped TCMC for certain kinds of properties. The general idea for why this is useful in model checking is that checking at the significant scope ensures that some non-spurious part of the total state space has been verified, which is a measured method independent of computing resource limitations. This is discussed in detail in the next chapter.

## 4.5 Summary

Scoped generation is the process of generating instances of models at scopes smaller than the total-state-space size, and, scoped TCMC is the process of model checking using TCMC in Alloy at scopes smaller than the total-state-space size. These scoped processes are useful because the total state space is usually too large to be represented in Alloy. However, scoped generation can produce spurious instances, which are full subgraphs of the model that do not convey useful information for inspection and analysis. Thus, the generation of spurious instances poses a problem for modellers when trying to detect modelling errors. We created a set of axioms, called the *significance axioms*, that addresses this issue by ensuring that some key states and transitions are produced, and that the states produced are within the reachable state space. A minimum state scope, called the *significant scope* is required to satisfy the significance axioms. We can find this scope by iteratively incrementing the state scope until a valid instance is found by the Analyzer.

# Chapter 5

# Scoped-TCMC Methodology

**Scoped TCMC** for a state scope of $n$, where $n$ is less than the reachable state space, is the model checking of all TS instances of size $n$ of any transition system that satisfies the transition relation. An example of scoped TCMC is shown in Figure 5.1, where $T_1$, $T_2$, and $T_3$ represent possible transition systems from a non-unique transition relation definition, and the example TS instances are some of the checked TS instances of size $n$. Even with a uniquely-defined transition system, there are multiple TS instances of size $n$, thus, scoped universal TCMC, which checks whether a property is satisfied on all paths starting from all initial states in *all* TS instances of the model, and scoped existential TCMC, which checks if *some* TS instance of the model satisfies the property, return different results.

The result from scoped universal TCMC of a property holds for all transition system instances of that size, and that from scoped existential TCMC holds for at least one transition system instance of that size, but moreover, we can draw some conclusions about whether the property holds for any complete transition system. Recall that a complete transition system of a model is a transition system containing all possible states (that is, states with all possible attribute value combinations, reachable and unreachable), and all transitions among them. In this chapter, we propose a scoped-TCMC methodology that helps us make such deductions.

## 5.1 Types of Properties

Before introducing our proposed scoped-TCMC methodology, we establish some categories for classifying CTLFC properties. These categories, shown in Figure 5.2, help us compare

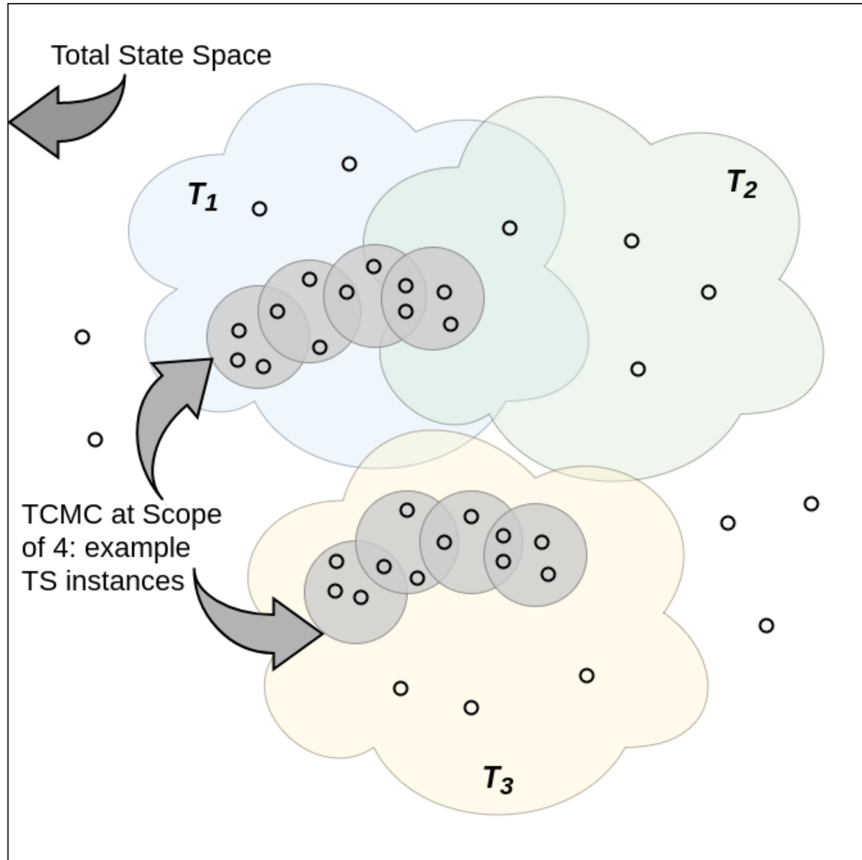



Figure 5.1: Modelling and checking transitions systems using TCMC in Alloy. $T_1$, $T_2$, and $T_3$ are transition systems from a non-unique NSR definition.

the model checking abilities of TCMC to those of BMC in Chapter 7. For our categorization, we convert any given property to *negation normal form*, so that negations are only applied to atomic propositions (similar to [10]). In our property examples, $p$ and $q$ represent atomic propositions.

The first distinction made is between universal and existential properties [10]. **Universal properties** are those where we reason about *all* paths of a transition system. These properties, also referred to as ACTL formulas [10], only contain universal quantifiers, $A$'s, when expressed in negation normal form. If a universal property does not hold, a counterexample, which is a path where the property is not satisfied, can be produced. $AGp$, $AFp$, $AFAGp$ and $AG(p \Rightarrow AFq)$ are all examples of universal properties. **Existential properties** are those where we reason about the existence of a satisfying path. Such

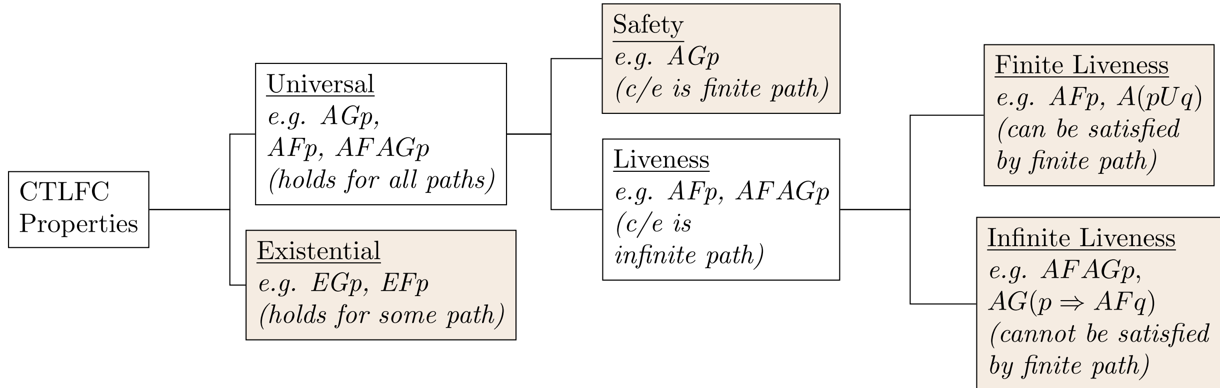



Figure 5.2: CTLFC Property Categories

properties (all non-ACTL formulas) contain one or more existential quantifier, $E$, when expressed such that negations are only applied to atomic propositions. If the property does not hold, no counterexample is produced, however, if the property holds, a witness, which is a path where the property holds, can be produced. $EGp$ and $EFp$ are examples of such properties.

Following traditional definitions, universal properties are categorized into safety and liveness properties [10]. **Safety properties** are properties that have finite paths as counterexamples. **Liveness properties** are those that have infinite paths as counterexamples. Since every path of a finite model has a finite representation, we define an **infinite path** as one *with* a loop at the end, which means that the last state appears earlier in the path, and a **finite path** as one *without* a loop at the end, meaning there is no repetition of the last state earlier in the path.

Liveness properties are further categorized based on whether they can be *satisfied* by a finite path or not. **Finite liveness properties** are those that *can* be satisfied by a finite path, that is, the prefix of an infinite path. Properties of the form $AFp$ or $A(pUq)$ are finite liveness properties. **Infinite liveness properties** are those that *cannot* be satisfied by finite paths. Examples of such properties are those of the form $AFAGp$ or $AG(p \Rightarrow AFq)$. Any universal property with a fairness constraint is categorized as an infinite liveness property because only infinite paths can satisfy these properties.

The rest of this chapter describes model checking methodologies and how to interpret results for the complete transition system for these different types of properties. In our figures, we use the word *real* to signify if a scoped TCMC pass or fail holds for the complete transition system of the model. We use the term *ambiguous* to refer to a scoped TCMC result that could potentially be reversed when considering the complete transition system.

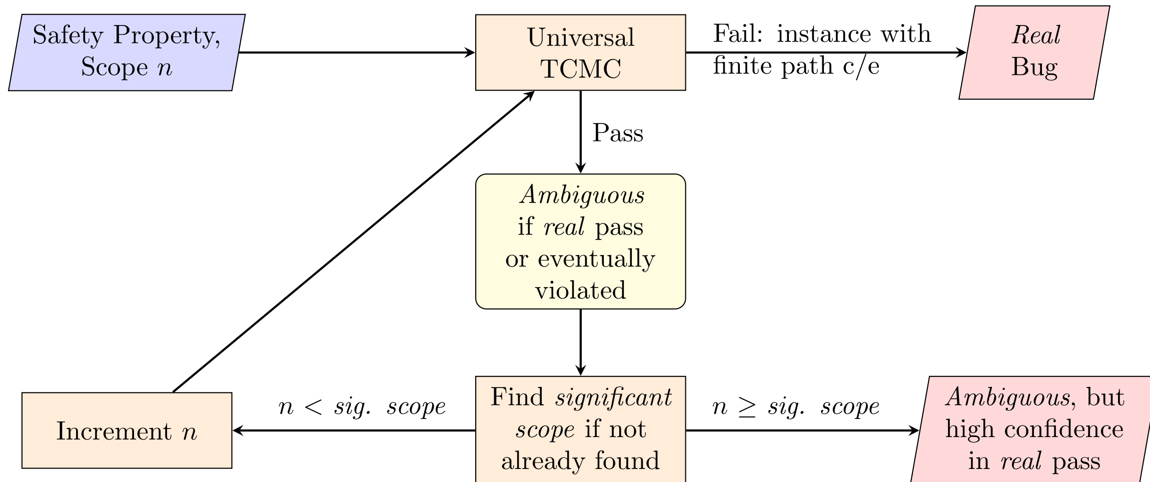



Figure 5.3: Safety Property: Scoped-TCMC Methodology

## 5.2 Safety Properties

Safety properties, that is, properties that have finite path counterexamples, are checked using the process outlined in Figure 5.3. We run universal TCMC as described in Section 2.4. If the check fails, we get a TS instance with a finite path containing a bug: this is a real bug in the complete transition system of the model.

If it passes, we can conclude that it passes in all transition systems of the specified scope, however, for the complete transition system, it is ambiguous if the pass holds or if a violating state would be encountered at a larger scope. At this stage, we recommend testing the model at least up to the significant scope, which is the minimum scope required to satisfy our significance axioms, as described in Section 4.3. We iteratively increment the state scope for our `check` and rerun universal TCMC until this significant scope is reached or a failure occurs. We increment iteratively instead of directly checking at the significant scope to take advantage of the better model checking performance at lower scopes. Model checking at least at the significant scope ensures that we check some non-spurious instances, which contain at least one transition representing each operation. This results in a higher confidence in our pass result. If computing resources allow it, then we continue to increment our state scope for model checking to continue to increase our confidence in the result.

Figure 5.4 shows an example of checking a safety property in our Musical Chairs model. Here we consider a game starting with 3 players and 2 chairs. We check that the number

```
1 assert safety {
2   // number of players is always 1 greater than number of chairs
3   ctl_mc[ ag [{s: State| #s.players = (#s.chairs).plus[1] }]]
4 }
5 check safety for exactly 3 Player, exactly 2 Chair,
6   exactly 8 State
```

Figure 5.4: Checking a safety property of Musical Chairs

of players is always one more than the number of chairs using the `ag` function from the `ctl` module. The model checking process is started at a low `State` scope of 4 to detect initial bugs since a lower scope yields better performance. When we get a pass result, we iteratively increment the `State` scope until we reach 8, which is the significant scope for the Musical Chairs model of 3 players and 2 chairs. A pass at this scope gives us some confidence that the property is satisfied in the complete transition system.

## 5.3 Finite Liveness Properties

Although transition systems are often thought of as having only infinite paths generated from a total next-state relation, when we perform scoped TCMC in Alloy, the transition systems checked contain a limited number of states, and thus may contain finite paths (*i.e.*, states that have no successor). Finite liveness properties are those that are violated only by infinite paths, but can be satisfied by finite paths. These properties can be checked using scoped TCMC in Alloy using the methodology illustrated in Figure 5.5.

When checking finite liveness properties, universal TCMC inherently only considers and checks infinite paths (see the use of `id[X]` in `eg`, and the derived functions `af` and `au`, in the TCMC implementation in Figure 2.2). Therefore, if the check fails (while considering only infinite paths), the culprit path in the counterexample instance is an infinite path, guaranteeing that a real bug has been uncovered in the complete transition system of the model.

If the check passes, it is ambiguous whether the property holds for the complete transition system, since paths that are finite at the specified scope have not been checked. However, it is useful to consider finite paths when checking finite liveness properties, since these properties can be satisfied by finite paths, which means that, at the given scope, if all paths, finite and infinite, satisfy a finite liveness property, then the property is satisfied for the complete transition system as well. For the transition system instances in

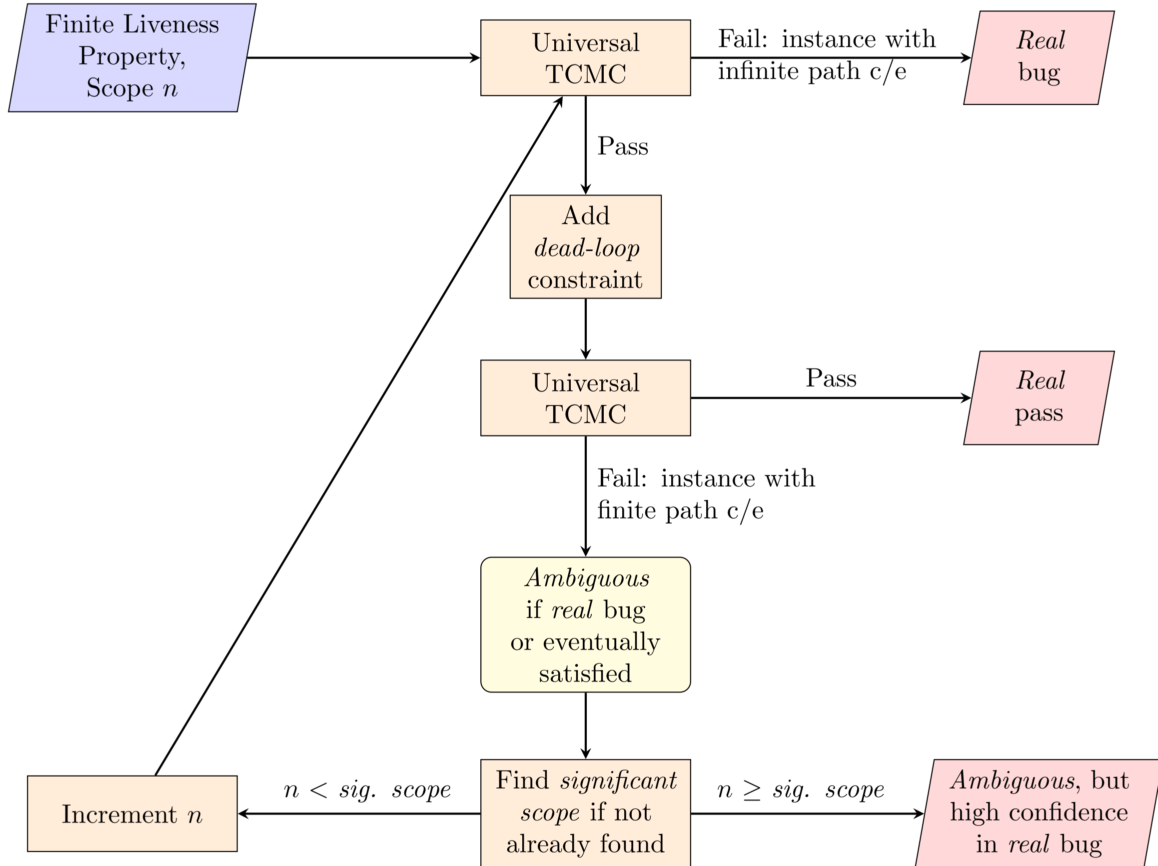
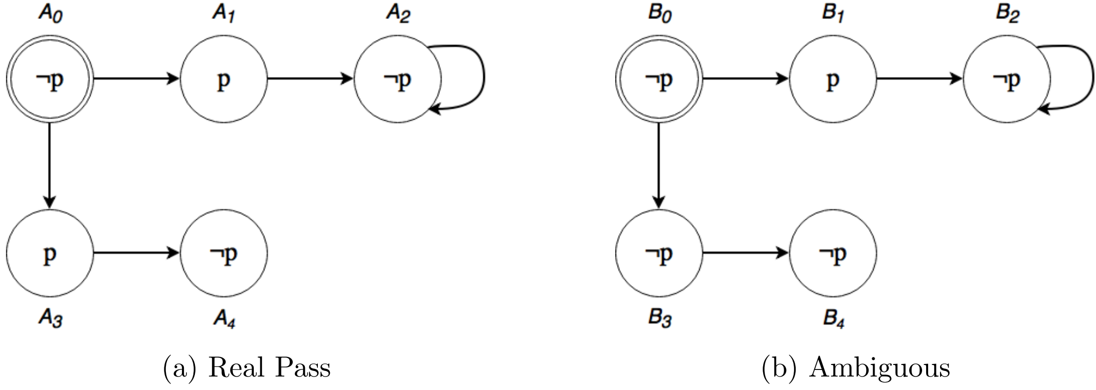



Figure 5.5: Finite Liveness Property: Scoped-TCMC Methodology

Figure 5.6, where $A_0$ and $B_0$ are the initial states, 5.6a shows an instance that should be a real pass for $AFp$ since all paths contain a state satisfying $p$, and 5.6b shows an instance that is ambiguous for the property since there is a finite path with no state that satisfies $p$. In both of these cases, TCMC of the $AFp$ property results in a pass because there is no infinite path without $p$ being true at some point: It cannot differentiate the real and ambiguous passes, because in 5.6a, the path $A_0 \rightarrow A_3 \rightarrow A_4$ is not infinite and, therefore, is not checked. We wish to distinguish between these two instances.

Therefore, at this point, we check if the given property holds on the finite paths of the transition system. We achieve this by adding a constraint to our model that creates a loop at any *dead* state, which is a reachable state with no successor, in the limited scope. We call this constraint the **dead-loop** constraint. Equation 5.1 represents such a constraint, where $s$, $s'$, and $s_n$ are states, $ops(s, s_n)$ is a predicate where $(s, s_n)$ satisfies any of the



(a) Real Pass (b) Ambiguous

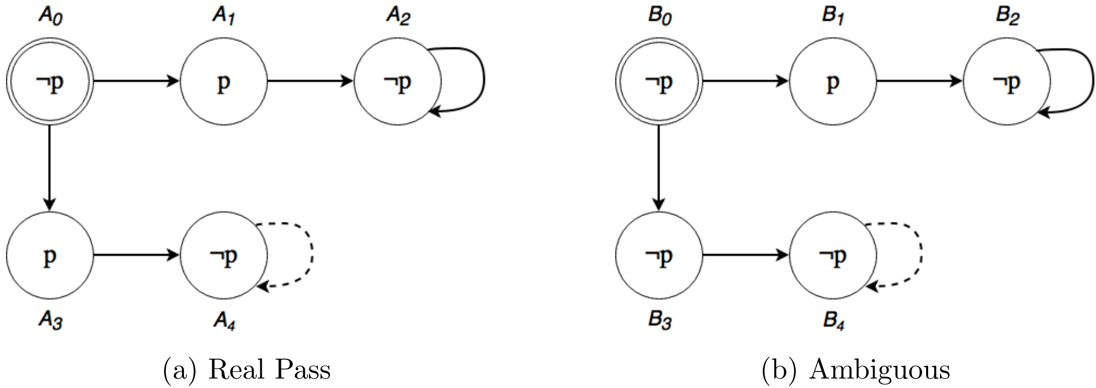
Figure 5.6: $AFp$: TCMC (*without* dead-loop) results in pass for both

model's operations, and $\sigma$ is the transition relation.

$$\forall s, s' \cdot (\neg(\exists s_n \cdot ops(s, s_n)) \land (s = s')) \Rightarrow \sigma(s, s') \tag{5.1}$$

Adding the dead-loop constraint forces all finite paths in an instance to be infinite by adding a transition from any reachable state without a successor back to itself. This enables us to check finite paths when checking for finite liveness properties using TCMC. Figure 5.7 shows the transition systems from Figure 5.6 with the additional transitions resulting from the dead-loop constraint with dashed arrows. These added transitions make all paths infinite and allow TCMC to distinguish between a real pass and an ambiguous pass.

When we perform scoped TCMC after adding the dead-loop constraint, a pass result



(a) Real Pass (b) Ambiguous

Figure 5.7: $AFp$: TCMC *with* dead-loop results in pass for (a) and fail for (b)

(Figure 5.7a) means that all paths originating from all initial states reach satisfying states, where the desired behaviour occurs, within the limited scope. So we can deduce that the property passes in the complete transition system as well, and we can stop our model checking process.

If the check fails (Figure 5.7b), it means that there is a violating finite path in the given scope. However, it is unknown whether the path represents a real bug in the complete transition system or if the finite path can eventually lead to a satisfying state, which makes the fail result ambiguous. To add some assurance to this result, as with safety properties, we model check up to the significant scope. A failure at the significant scope results in higher confidence that the finite liveness property is not satisfied in the complete transition system.

Figure 5.8 shows an example of checking a finite liveness property in our Musical Chairs model. Here, we check that a game of 3 players and 2 chairs always reaches a state with a `sitting` mode, ensuring the game's progress. We assert the property using the `af` function from the `ctl` module. When we perform TCMC at a scope of 2 `States`, the check passes, although vacuously, since no infinite paths exist for a scope of 2 for this model. Then we add our dead-loop constraint to the model, as shown in Figure 5.9, to consider finite paths as well. On executing the `check` command with a state scope of 2, the check fails. We

```
1 assert finiteLiveness {
2   ctl_mc[ af [{ s: State| s.mode=sitting }]]
3 }
4 check finiteLiveness for exactly 3 Player, exactly 2 Chair,
5   exactly 3 State
```

Figure 5.8: Checking a finite liveness property of Musical Chairs

```
1 // ops[s1,s2] is a disjunction of the model's operations
2 pred dead_loop [s,s':State] {
3   (no s_n:State | ops[s,s_n]) and s=s'
4 }
5 fact modelDefinition {
6   all s,s':State | s->s' in nextState iff
7     (ops[s,s'] or dead_loop[s,s'])
8   ...
9 }
```

Figure 5.9: Dead-loop Constraint in an Alloy Model

increase the scope to increase confidence since 2 is less than the significant scope. When we set the `State` scope to 3, we find that the property holds, which is a real pass for the complete transition system.

## 5.4 Infinite Liveness Properties

An infinite liveness property can only be satisfied and violated by infinite paths, therefore, we only need to consider and check infinite paths during scoped TCMC. Our proposed method for using TCMC to check infinite liveness properties is outlined in Figure 5.10.

If TCMC for an infinite liveness property fails, the counterexample produced represents a real bug in the complete transition system. TCMC inherently only considers infinite paths for these properties, meaning that only an instance with a culprit infinite path, thus, representing a real bug, can be produced as a counterexample.

If TCMC passes for such a property, then it is ambiguous whether the result represents a real pass in the complete transition system or a false positive. Longer paths may exist that have not been checked that violate the property. However, as before, model checking up to the significant scope gives us greater confidence in our pass result. There is no point

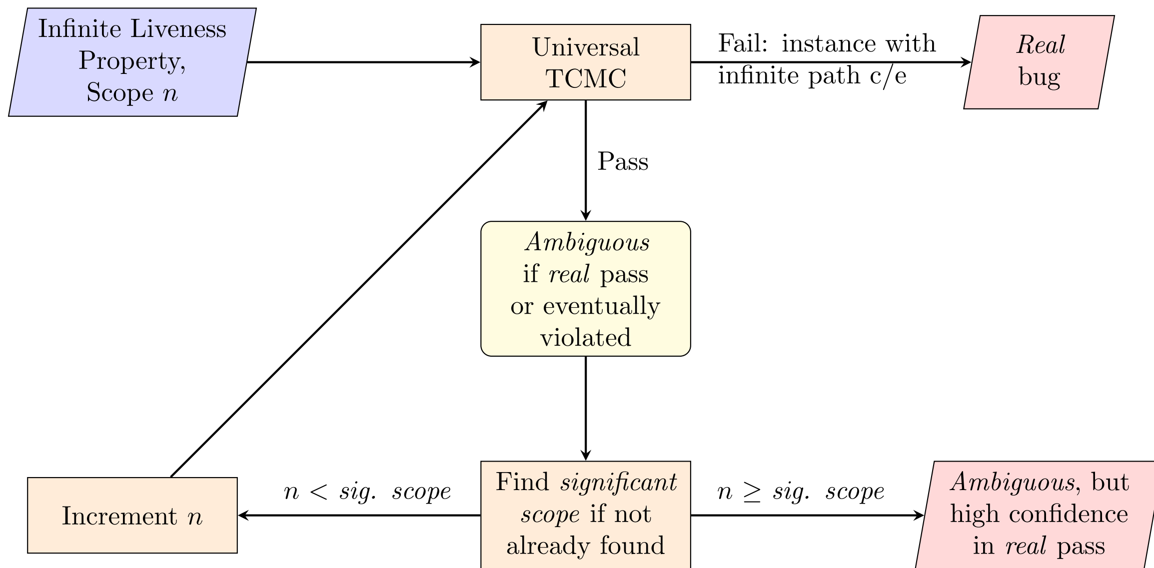



Figure 5.10: Infinite Liveness Property: Scoped-TCMC Methodology

```
1 assert infiniteLiveness {
2   // #players eventually always reaches and remains at 1
3   ctl_mc[ af [ ag [{s: State| #s.players=1}]]]
4 }
5 check infiniteLiveness for exactly 3 Player, exactly 2 Chair,
6   exactly 8 State
```

Figure 5.11: Checking an infinite liveness property of Musical Chairs

in adding our dead-loop constraint to check finite paths in this case, because, unlike finite liveness properties, infinite liveness properties cannot be satisfied by finite paths.

Our methodology for infinite liveness properties is similar to that for safety properties, however, there is a distinction between the kind of counterexample returned for each property on failure: TCMC of a safety property returns a counterexample with a finite culprit path, whereas, TCMC of an infinite liveness property returns a counterexample with an infinite culprit path.

Figure 5.11 shows an example of checking an infinite liveness property in our Musical Chairs model of the form $AFAGp$. We use the `af` and `ag` functions from the `ctl` module to check that we always eventually reach a point where the number of players is one and always remains at one at all further states on that path, in other words, the transition system converges. We start the model checking at a `State` scope of 4. We find that the check passes (although, from our knowledge about the model, we know that this pass occurs vacuously since no paths at this scope are infinite). We repeat the check until we reach a scope of 8, which is the significant scope. At this point, we are relatively confident of our pass result (and we know that the pass is not vacuously obtained since infinite paths exist at this scope).

## 5.5 Existential Properties

To check existential properties (including existential properties with fairness constraints) such as $EFp$ or $EGp$, in TCMC, we use *existential* TCMC as described in Section 2.4. Checking an existential property using *universal* TCMC would check if there is *some* path in *all* instances of the model that satisfies the property. This check is too strong since to satisfy an existential property, there only needs to be *some* path in *some* instance of the model that satisfies the property, which is what we accomplish with existential model checking.

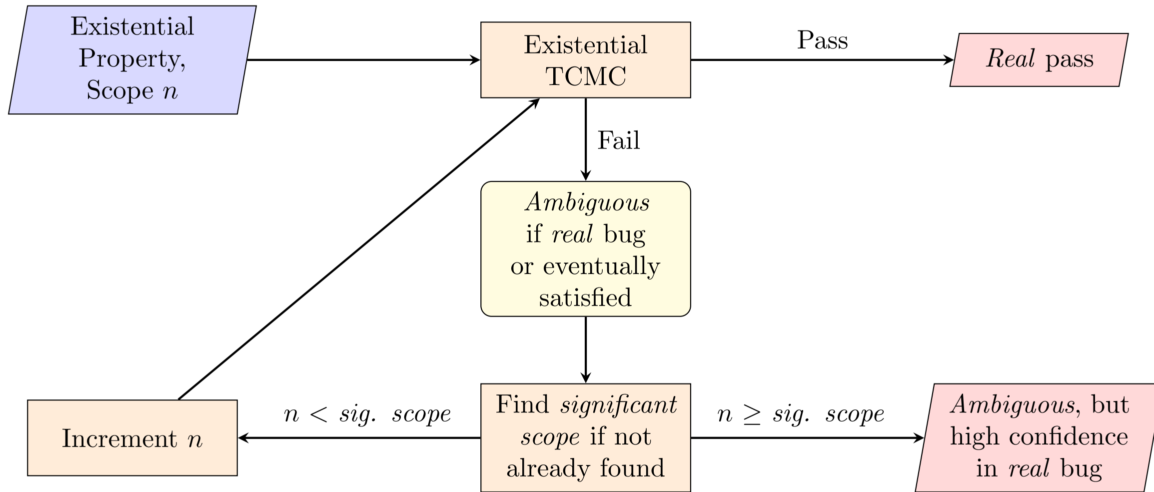



Figure 5.12: Existential Property: Scoped-TCMC Methodology

Our methodology for checking existential properties is shown in Figure 5.12. If an existential TCMC `run` produces a satisfying instance, then the property passes for the complete transition system of the model because a path (finite or infinite) exists in some instance that satisfies the property. If the `run` does not find an instance, it is ambiguous whether the property fails for the complete transition system of the model, because there may exist an instance larger than the specified scope that contains a path satisfying the property. However, as before, model checking up to the significant scope gives us greater confidence in our pass result.

Figure 5.13 shows an example of checking an existential property in our Musical Chairs model. In this example, we assert, using the `ef` function, that there is a player named Alice in the game, and there exists an instance where she eventually wins the game. When we start our model checking process at a `State` scope of 4, our property fails (since an `end`

```
1 one sig Alice extends Player{}
2 pred existential {
3   // Alice wins in some instance
4   ctl_mc[ ef [{s: State| s.mode=end and s.players=Alice}]]
5 }
6 run existential for exactly 3 Player, 2 Chair, exactly 8 State
```

Figure 5.13: Checking an existential property of Musical Chairs

state has not been reached). We increment the scope but get failures until we reach 8. At this point, we find the property is satisfied, which means it is satisfied for the complete transition system.

## 5.6 Summary

We can categorize CTLFC properties into existential and universal properties. Universal properties can be divided into safety (counterexample is a finite path) and liveness (counterexample is an infinite path) properties. Liveness properties can, in turn, be divided into infinite liveness properties and finite liveness properties, depending on whether finite paths can satisfy them or not. We can use scoped TCMC for each of these property categories, and then, deduce model checking results for the complete transition system from the scoped TCMC results, as shown in Table 5.1. By checking at the significant scope, we gain higher confidence in our scoped TCMC result with respect to the complete transition system. Our scoped-TCMC methodology works whether the model defines a unique transition system or not, because, scoped TCMC checks all TS instances of the given state-scope size across all possible transition systems.

Table 5.1: Deducing Complete Model Checking Results from Scoped TCMC

<table>
<tr><th colspan="2">Property</th><th>Scoped TCMC Pass</th><th>Scoped TCMC Fail</th></tr>
<tr><td colspan="2">Safety</td><td>Unknown</td><td>Real Bug</td></tr>
<tr><td rowspan="2">Finite Liveness</td><td>w/o dead-loop</td><td>Unknown</td><td>Real Bug</td></tr>
<tr><td>w/ dead-loop</td><td>Real Pass</td><td>Unknown</td></tr>
<tr><td colspan="2">Infinite Liveness</td><td>Unknown</td><td>Real Bug</td></tr>
<tr><td colspan="2">Existential</td><td>Real Pass</td><td>Unknown</td></tr>
</table>

# Chapter 6

# Case Studies

We developed four case studies, including our Musical Chairs example, to demonstrate the contributions made in this thesis. The complete Alloy models are included in Appendix A. In the first four sections of this chapter, we discuss our case studies highlighting how each utilizes the style guidelines discussed in Chapter 3, the significance axioms in Chapter 4, and the scoped-TCMC methodology in Chapter 5. In the last section of the chapter, we investigate the scalability of TCMC in Alloy in terms of these case studies.

## 6.1 Musical Chairs

As described throughout the previous chapters, we modelled the game of Musical Chairs based on [35] and checked its properties using TCMC. (The full model is provided in Appendix A.1.) This section describes the utility provided by our contributions to this process.

### 6.1.1 Style Guidelines

Our Musical Chairs model adheres to our style guidelines in Chapter 3 that we believe promote structure, modularity, and consistency when modelling. We use an equality constraint to ensure that states with equivalent attribute values are treated as the same state to avoid the occurrence of multiple `State` elements that do not represent distinct states of the system (Appendix A.1 Lines 125–129). The state-space declaration avoids enforcing invariants (such as using multiplicity constraints like `lone` or `one`). This rule helps the

modeller to define their model only through initial state constraints and operations, so that any expected invariants can be model checked to verify modelling correctness, promoting consistency when modelling. The operations defined in the model are separated into pre- and post-conditions, and into distinct operations (Section 3.2.2), to support structure and modularity. This treatment of operations also helps when using significance axioms to generate a non-spurious instance (Section 4.2). When constraining our model's transition relation, we use the DisjMethod (as opposed to the ConjMethod) discussed in Section 3.3.1, which results in a transition system with the expected behaviour.

### 6.1.2 Significance Axioms

Generating an instance of our Musical Chairs model using the `run` command in the Alloy Analyzer produces a spurious instance, which is a random set of states that provides no value to the modeller for inspection or analysis. Figure 6.1 shows such an instance produced using the command `run {} for exactly 3 Player, exactly 2 Chair, exactly 5 State`. The instance has no initial state, no transitions, and include unreachable states (*e.g.* the states with multiple modes).

Using the significance axioms shown previously (Figure 4.3) to generate an instance with the command `run significanceAxioms for exactly 3 Player, exactly 2 Chair, exactly 8 State` produces a more useful instance, as shown in Figure 6.2. The significant

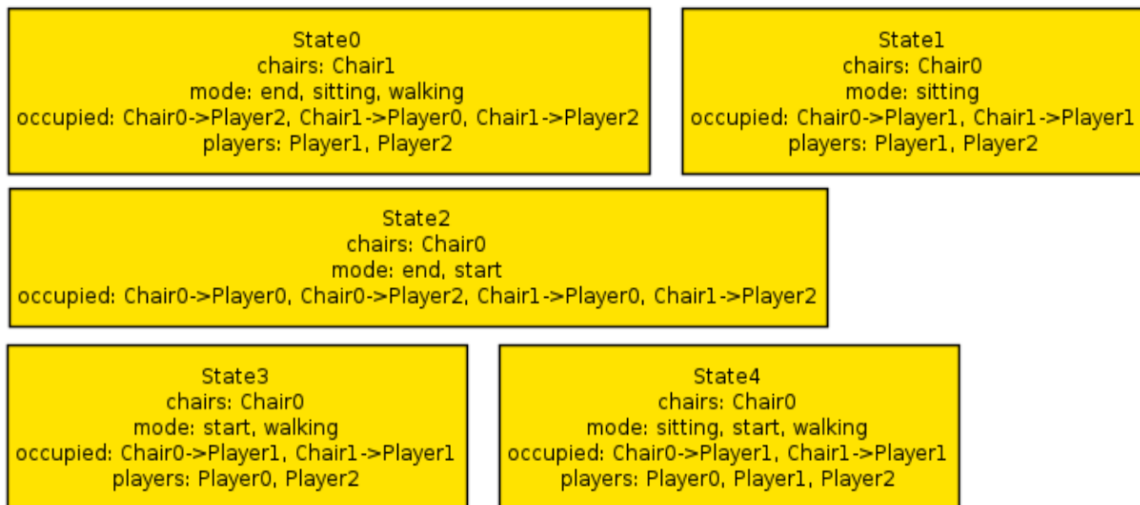



Figure 6.1: Musical Chairs: 5-State Spurious Instance

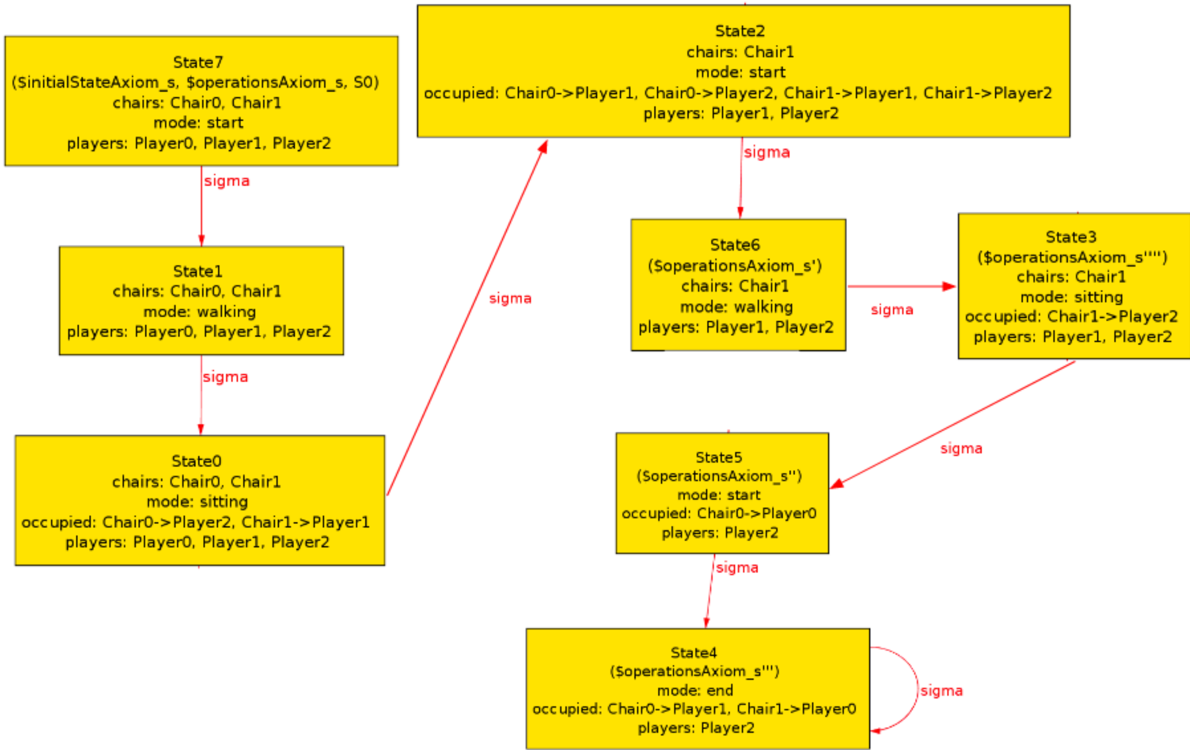



Figure 6.2: Musical Chairs: 8-State Non-Spurious Instance (sigma: Transition Relation)

scope for this model for 3 players and 2 chairs is 8 states, which means a minimum of 8 states must be present to satisfy the significance axioms. The instance produced has an initial state (`State7`), and shows at least one reachable transition for each operation. As modellers, we are able to inspect the pre- and post-conditions of each operation, and the initial state constraints in this instance. This establishes the utility of significance axioms when generating instances of models for inspection.

### 6.1.3 Scoped-TCMC Methodology

In Musical Chairs, we check a property in each of the four different categories using the scoped-TCMC methodology proposed in this thesis.

As a **safety property**, we check if the number of players in the game is always 1 greater than the number of chairs: $AG(\#players = \#chairs{+}1)$. This property holds in the model as described previously (and in Appendix A.1). However, if a bug is introduced by removing

the constraint `#s'.chairs = (#s.chairs).minus[1]` (Line 75 in Appendix A.1) from the `post_eliminate_loser` predicate that enforces a decremented number of chairs in the next state, and we execute the command `check safety for exactly 3 Player, exactly 2 Chair, exactly 4 State`, we discover the property is not satisfied, which means the bug has been discovered. The counterexample shown in Figure 6.3 is produced by the Analyzer. From our scoped-TCMC methodology, we know that this is a real bug, that is, a bug in the complete transition system.

After fixing the bug, we find the significant state scope as outlined in Chapter 4. We execute the `check` command, and since it returns a pass result, we iteratively increment the scope until the significant scope, which is 8 in this case, is reached. By performing TCMC at this scope, we can have some confidence that this pass result holds for the complete model checking problem of the entire state space. Inspection of an instance generated

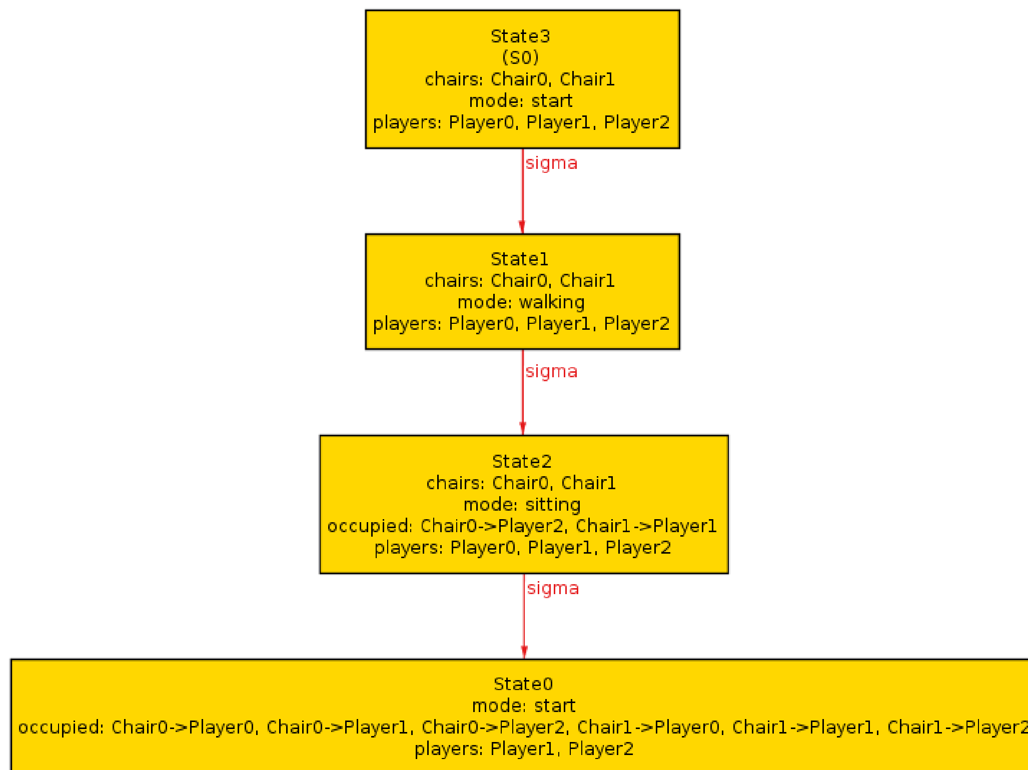



Figure 6.3: Musical Chairs: Safety Property Counterexample with A Finite Path (sigma: Transition Relation)

using the significance axioms (*e.g.*, the one in Figure 6.2), corroborates this deduction.

As a **finite liveness** property, we check if a state with a `sitting` mode is always eventually reached: $AF(mode = sitting)$. The property checks if a significant part of the game is reached where the loser of the round is eliminated. We use the `ctl` module's `af` function to express this property. At the significant scope of 8 states (for 3 players), the check passes, meaning the property holds on all infinite paths present at this scope. If we introduce a bug, for example by replacing all occurrences of `sitting` in the model with an arbitrary mode, then the check produces a counterexample with an infinite path showing a real bug in the complete transition system.

Without the bug, if we now add our dead-loop constraint (Section 5.3) to check all finite paths as well, and execute TCMC at a state scope of 8, we find that the check passes again. This pass represents a real pass for the complete model checking problem, since all paths, finite and infinite, reach a `sitting` mode at the given scope.

As an **infinite liveness** property, we check if the number of players eventually reaches and remains at 1: $AFAG(\#players = 1)$. Performing TCMC at a state scope of 8, results in a pass for the property. This pass means that all infinite paths at this scope satisfy the property, but since it is unknown whether other infinite paths exist in the complete transition system that potentially violate the property, the result is ambiguous in terms of the complete model checking problem.

If we introduce a bug in our model by letting the `declare_winner` operation occur when the number of players reaches 2 instead of 1, TCMC fails, and produces a counterexample with an infinite path (Figure 6.4). This counterexample represents a real bug in the complete transition system.

To check an **existential property**, we first assert that there is a player in the system called Alice, and write a property to check that there exists a path where Alice wins, which happens when a state with the `end` mode is reached and Alice is the only remaining player: $EF(mode = end \wedge players = \{Alice\})$. If we start existential TCMC for the property at a state scope of 4, we find that the property fails, and no instance is found. From our knowledge of the model, we know the fail occurs because at this scope, a winner has not been declared yet. As we increase the state scope to the significant scope of 8, we find that TCMC for this property holds, and an instance showing Alice winning is produced. Since this is an existential property, as long as there is one path present in the system that satisfies the property, the property holds for the complete transition system. The opposite, however, is not true: if an existential property does not hold for scoped TCMC, it is ambiguous whether it holds in the complete transition system, since there could be a path present when considering the entire state space that satisfies the property.

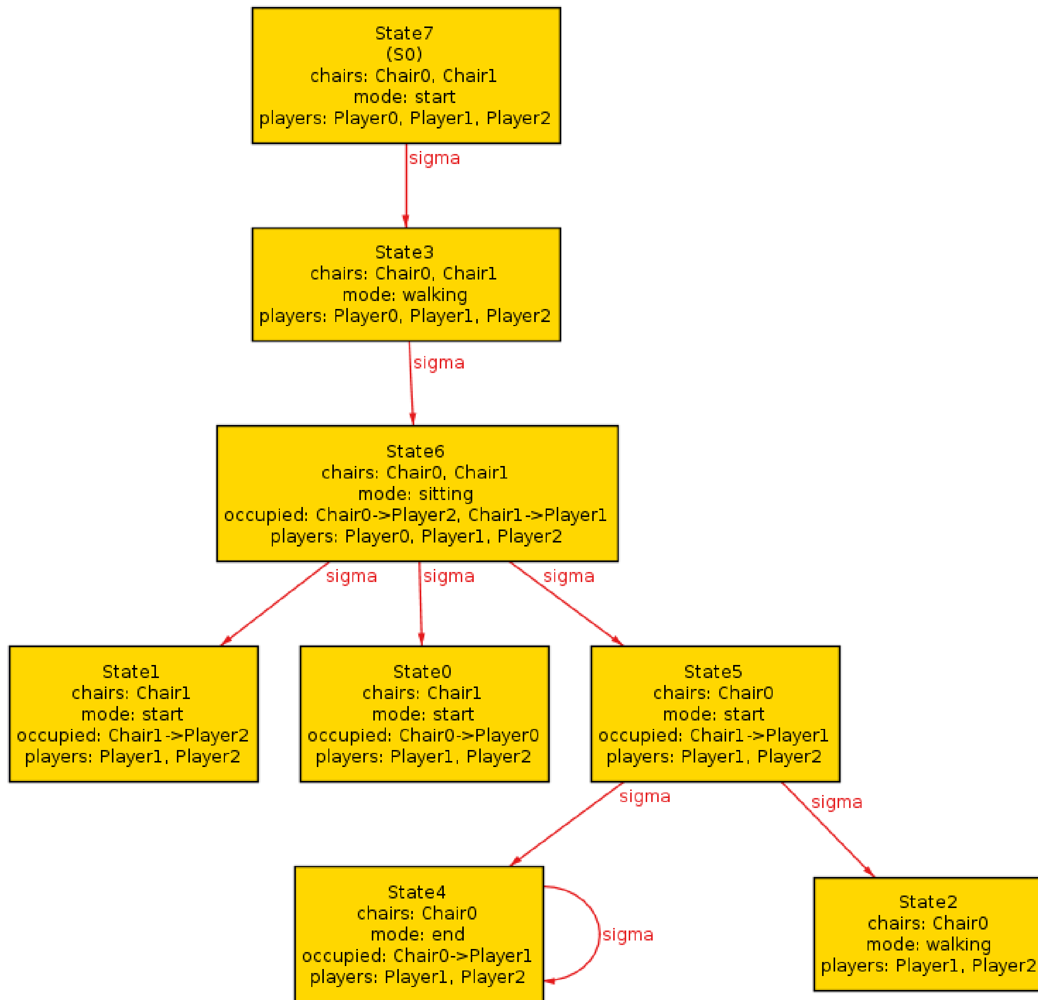



Figure 6.4: Musical Chairs: Infinite Liveness Property Counterexample with An Infinite Path (sigma: Transition Relation)

## 6.2 Elevator System

We developed a model (Appendix A.2) of a simple Elevator System in Alloy inspired by [36] and [29]. The elevator can be called by any floor; it moves up and down, stopping at the closest called floor in its current direction, and switches directions when no more called floors exist in the current direction. The elevator also requires a maintenance check after it has changed direction a certain number of times. Every state of the Elevator System contains attributes pertaining to the `current` floor, the set of `called` floors, the current direction of the elevator's movement (`goingUp`), and the number of times the direction has been changed since the last maintenance (`maintenance`).

The rest of this section discusses how our style guidelines, significance axioms, and scoped-TCMC methodology contribute to the modelling and model checking process of this system.

### 6.2.1 Style Guidelines

Our Elevator System model follows the style guidelines discussed in Chapter 3 of this thesis. An equality constraint, shown in Figure 6.5 Lines 9-15, is used to require states with equivalent attributes to be the same state. No invariants are used in the state space declaration. For example, since the `current` attribute can only contain one `Floor` element

```
 1 sig Floor {}
 2 one sig Up {}
 3 sig State {
 4   current: set Floor,
 5   goingUp: set Up,
 6   called: set Floor,
 7   maintenance: Int
 8 }
 9 pred equality [s,s':State] {
10   (s'.current = s.current and
11   s'.maintenance = s.maintenance and
12   s'.goingUp = s.goingUp and
13   s'.called = s.called)
14     implies s = s'
15 }
```

Figure 6.5: Elevator System State Space

at a time, we could have declared it as `current: one Floor`. However, to maintain a consistent model, this invariant should hold throughout the reachable states of the model because of its initial state and operation constraints. Therefore, instead of declaring the invariant as an explicit constraint in the model, we verify it through model checking.

```
1 pred pre_moveUp[s:State] {
2   some s.called
3   some s.goingUp
4   some nexts[s.current] & s.called
5 }
6 pred post_moveUp[s,s':State] {
7   s'.current = min[nexts[s.current] & s.called]
8   s'.current not in s'.called
9   s'.maint = s.maint
10  s'.goingUp = s.goingUp
11  (s.called - s'.current) in s'.called
12 }
13
14 pred pre_moveDown[s:State] {
15  some s.called
16  no s.goingUp
17  some prevs[s.current] & s.called
18 }
19 pred post_moveDown[s,s':State] {
20  s'.current = max[prevs[s.current] & s.called]
21  s'.current not in s'.called
22  s'.maint = s.maint
23  s'.goingUp = s.goingUp
24  (s.called - s'.current) in s'.called
25 }
```

Figure 6.6: Elevator System Operations

When defining the model's operation constraints, we ensure that each operation is distinct, as discussed in Section 3.2.2. For example, we create separate operations to represent the elevator moving up and down (`moveUp` and `moveDown`), as shown in Figure 6.6, since each of these operations have distinct pre- and post-conditions. Also, for the sake of structure and modularity, we create separate predicates for the pre- and post-conditions within each operation, as shown in Figure 6.6. These practices in modularity also help when creating our significance axioms to ensure that in every instance produced, we are shown at least one transition from each defined operation.

The model definition block, which adds structure to our model, uses the DisjMethod to

```
1 pred moveUp[s,s':State] {
2   pre_moveUp[s]
3   post_moveUp[s,s']
4 }
5 ...
6 fact modelDefinition {
7   all s:State | s in initialState iff init[s]
8   all s,s':State | s->s' in nextState iff
9     changeDir[s,s'] or
10    moveUp[s,s'] or
11    moveDown[s,s'] or
12    defaultToGround[s,s'] or
13    idle[s,s'] or
14    maintain[s,s']
15  all s, s': State | equality[s,s']
16 }
```

Figure 6.7: Elevator System Operations

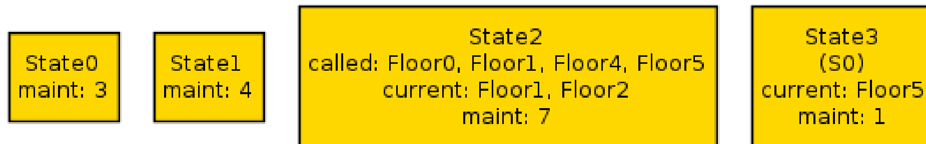



Figure 6.8: Elevator System: 4-State Spurious Instance

constrain our transition relation as shown in Figure 6.7. This method of conjuncting the pre- and post-condition of the operations (Lines 1-4), and disjuncting each operation for the transition relation (Lines 8-14) models the behaviour that we expect for this system.

### 6.2.2 Significance Axioms

When we generate an instance of our Elevator System model in the Alloy Analyzer using the command `run {} for exactly 6 Floor, exactly 4 State`, we get an unhelpful spurious instance like the one shown in Figure 6.8. Therefore, we create significance axioms as described in Section 4.3. The significance axioms for this model are shown in Figure 6.9. We find that a significant scope of 7 states is required to satisfy the axioms for 6 floors. When we generate an instance of the model using the command `run significanceAxioms for exactly 6 Floor, exactly 7 State`, an instance such as the one shown in Figure 6.10

```
1 pred reachabilityAxiom {
2   all s:State | s in State.(initialState <: *nextState)
3 }
4 pred operationsAxiom {
5   some s,s':State | changeDirToDown[s,s']
6   some s,s':State | changeDirToUp[s,s']
7   some s,s':State | moveUp[s,s']
8   some s,s':State | moveDown[s,s']
9   some s,s':State | defaultToGround[s,s']
10  some s,s':State | idle[s,s']
11  some s,s':State | maintain[s,s']
12 }
13 pred significanceAxioms {
14  reachabilityAxiom
15  operationsAxiom
16 }
```

Figure 6.9: Elevator System: Full Significance Axioms

is produced. This instance contains an initial state, and reachable transitions representing each of our defined operations. We can inspect this instance to verify that it exhibits the behaviour expected from the constraints declared in the model.

### 6.2.3 Scoped-TCMC Methodology

In our Elevator System model, we check a property from each of the three categories of universal properties. We follow the scoped-TCMC methodology described in Chapter 5 for each of these properties.

For **safety**, we check model correctness by verifying that there is only one floor associated with `current` for every state. We use the `ag` function from the `ctl` module as shown in Figure 6.11. TCMC of the property passes at the given significant scope, which means that although it is ambiguous whether the property holds for the entire state space, we gain confidence in the pass by checking at least at the significant scope.

As a **finite liveness** property, we check that the elevator always reaches a maintenance state. Maintenance is required after the elevator changes direction a specified number of times. The maintenance state is represented by a `0` value for the `maintenance` attribute, which tracks the number of times the elevator changed directions since the last time the maintenance check was 0. We use the `af` function to write the property as shown in

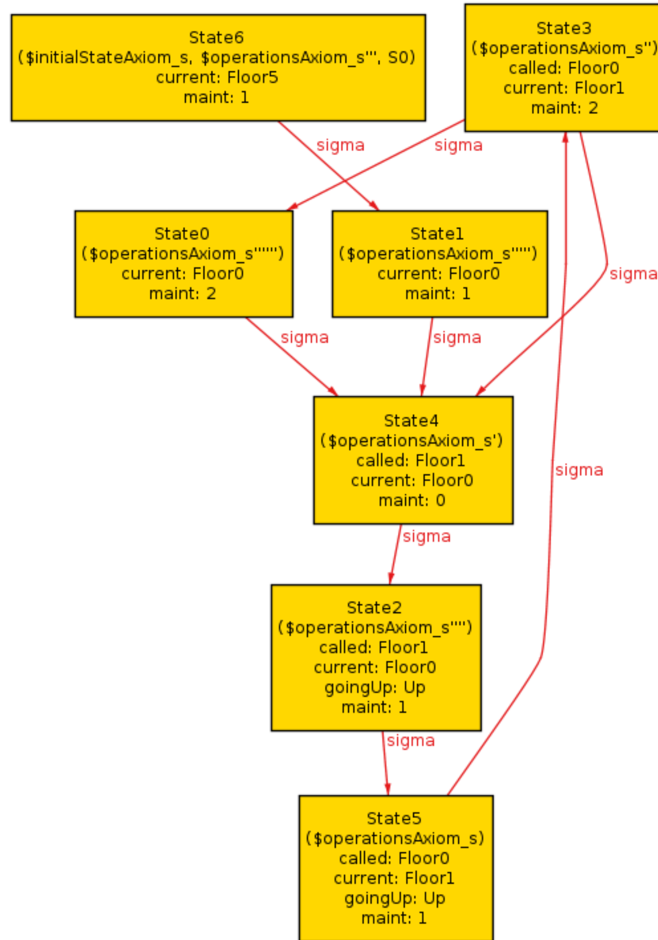



Figure 6.10: Elevator System: 7-State Non-Spurious Instance (sigma: Transition Relation)

```
1 assert safety {
2   // there is only one current floor
3   ctl_mc[ ag [{s: State| one s.current}]]
4 }
5 check safety for exactly 6 Floor, exactly 7 State
```

Figure 6.11: Elevator System: Safety Property

```
1 assert finiteLiveness {
2   // always reaches maintenance state
3   ctl_mc[ af [{s: State| s.maint = 0}]]
4 }
5 check finiteLiveness for exactly 6 Floor, exactly 7 State
```

Figure 6.12: Elevator System: Finite Liveness Property

Figure 6.12 Line 3. This property passes for 6 floors and 7 states. Therefore, no bugs are found when considering only infinite paths. Next, we add our dead-loop constraint (Figure 5.9) to consider finite paths as well. This time, TCMC fails with an ambiguous counterexample with a violating finite path, which means that it is uncertain whether the property holds in the complete transition system. We check up to the significant scope, and the check fails, thus, we have a higher confidence in the failure since we checked up to the significant scope.

As an **infinite liveness** property, we check that whenever a floor is called, that floor is always eventually reached as a current floor. The property is expressed in Alloy for TCMC as shown in Figure 6.13 Line 4. We use the `ag` and `af` temporal logic functions from the `ctl` module, along with the `imp_` (implies) logical operator function to create the property. We perform TCMC up to the significant scope, and find that the property holds. Although it is ambiguous from our scoped TCMC result if the property holds for the complete model checking problem, we have gained confidence in the pass result by checking at least at the significant scope.

```
1 assert infiniteLiveness {
2   // a floor called is always eventually reached as current
3   // AG(floorCalled => AF (floorCurrent) )
4   all f:Floor | ctl_mc[ ag [ imp_[called.f, af [current.f]] ] ]
5 }
6 check infiniteLiveness for exactly 6 Floor, 7 State
```

Figure 6.13: Elevator System: Infinite Liveness Property

## 6.3 Traffic Light Controller

We developed a Traffic Light Controller model by applying our modelling guidelines to the model developed by Vakili [42] (Appendix C.1 for comparison), which was originally inspired by [30]. We represented the three fairness constraints in the original model using one equivalent fairness constraint. Our full model is in Appendix A.3. The system models the behaviour of a three-way traffic intersection. Each side of the intersection has a sensor to detect if there are vehicles present that request the green light from that direction. On such a request, the green light is eventually provided to that direction. The value of our contributions for this model is discussed in the rest of this section.

### 6.3.1 Style Guidelines

In the state-space declaration, we avoid enforcing invariants (which promotes consistency), and include an equality constraint to consider all states with equivalent attributes as the

```
 1 abstract sig Counter{}
 2 one sig f0, f1, f2, f3 extends Counter{}
 3 abstract sig Sense{}
 4 one sig N_Sense, S_Sense, E_Sense extends Sense{}
 5 abstract sig Go{}
 6 one sig N_Go, S_Go, E_Go extends Go{}
 7 abstract sig Request{}
 8 one sig N_Req, S_Req, E_Req extends Request{}
 9 sig State{
10   sensors: set Sense,
11   goes: set Go,
12   req: set Request,
13   NS_Lock: set Bool,
14   counter: set Counter
15 }
16 pred equality [s,s':State] {
17   (s.sensors = s'.sensors and
18   s.goes = s'.goes and
19   s.req = s'.req and
20   s.NS_Lock = s'.NS_Lock)
21     implies s = s'
22 }
```

Figure 6.14: Traffic Lights Control: State Space

```
1 pred pre_N_Go[s:State]{
2   N_Go_True[s]
3 }
4 pred post_N_Go[s':State] {
5   N_Go in s'.goes
6 }
7 pred N_Go_[s,s':State]{
8   pre_N_Go[s]
9   post_N_Go[s']
10 }
11 ...
12 fact modelDefinition{
13   all s:State| initial[s] iff (s in initialState)
14   all s,s':State| s->s' in nextState iff (
15     N_Go_[s,s'] or
16     S_Go_[s,s'] or
17     E_Go_[s,s'] or
18     ...
19   )
20 }
```

Figure 6.15: Traffic Lights Control: Operations and Model Definition

same state, as shown in Figure 6.14. When declaring our operations, we create separate predicates for the pre- and post-conditions, promoting structure and modularity, and use the DisjMethod in the model definition to constrain the transition relation, as shown in Figure 6.15. The original Alloy model by Vakili [42] is provided for comparison in Appendix C.1. We found that applying our style guidelines promotes structure, modularity, and consistency in the model.

This case study uses the `ctlfc` module and shows an example of the use of fairness constraints in TCMC. It also shows an application (by hand) of the method described in [42], which is based on [44], to convert multiple fairness constraints to one. Our model has three fairness constraints that ensure all directions at the three-way traffic light intersection (North, South and East) receive adequate green light time. The fair states satisfying each of these three constraints are described by the functions implemented in Lines 3–5 in Figure 6.16. The `fact fairness { ... }` in Lines 8–15 dictates the update of a counter attribute in `State` whenever a new type of fair state is encountered, and the counter is reset when all three types of fair states have occurred. The predicate `fair` (Lines 16–18) is true whenever a member from each of the three fair state sets has been encountered. We equate the function representing the set of accepted fair states in the `ctlfc` module,

```
1 open ctlfc[State]
2 ...
3 fun N_fair[]: State {State - (sensors.N_Sense & goes.N_Go)}
4 fun S_fair[]: State {State - (sensors.S_Sense & goes.S_Go)}
5 fun E_fair[]: State {State - (sensors.E_Sense & goes.E_Go)}
6 // combines 3 fcs into 1 fc by checking that
7 // all 3 fcs occur infinitely often using a counter
8 fact fairness {
9   all s,s':State | s->s' in nextState implies (
10     (s in N_fair[] and s.counter=f0) implies s'.counter=f1 else
11     (s in S_fair[] and s.counter=f1) implies s'.counter=f2 else
12     (s in E_fair[] and s.counter=f2) implies s'.counter=f3 else
13     s.counter=f3 implies s'.counter=f0 else
14     s'.counter=s.counter)
15 }
16 pred fair[s:State] {
17   s.counter = f3
18 }
19 ...
20 fact modelDefinition{
21   ...
22   all s:State | s in fc iff fair[s]
23 }
```

Figure 6.16: Traffic Lights Control: Fairness Constraints

`fc`, to the set of states satisfying `fair` (Line 21). Therefore, when performing TCMC, the `ctlfc` module ensures that the `fair` predicate holds infinitely often in checked instances, thus, satisfying all three fairness constraints of the model.

### 6.3.2 Significance Axioms

We use our significance axioms, shown in Figure 6.17, to generate a non-spurious instance by executing the command `run significanceAxioms for exactly 17 State`. The significant scope for this model is found to be 17 states.

### 6.3.3 Scoped-TCMC Methodology

Using TCMC, we check a safety property in this model that traffic from cross directions never receive the green light at the same time, as shown in Figure 6.18. We use the `ag`

```
1   pred reachablityAxiom {
2     all s:State | s in State.(initialState <: *nextState)
3   }
4   pred operationsAxiom {
5     some s,s':State | N_Go_[s,s']
6     some s,s':State | N_Not_Go[s,s']
7     some s,s':State | N_Go_Unchanged[s,s']
8     ...
9   }
10  pred significanceAxioms {
11    reachablityAxiom
12    operationsAxiom
13  }
```

Figure 6.17: Traffic Lights Control: Significance Axioms (full axioms in Appendix A.3)

```
1 assert safety{
2   // light in cross directions never on at same time
3   // AG !(E_Go & (N_Go | S_Go))
4   ctlfc_mc[ ag [not_[goes.E_Go & goes.(N_Go + S_Go)]]]
5 }
6 check safety for exactly 17 State
```

Figure 6.18: Traffic Lights Control: Safety Property

function and the `not_` function, representing the negation logical operator, from the `ctlfc` module to express the property. We find that the property holds for the specified scope. According to our methodology, this pass is ambiguous in terms of the complete model checking problem, however, we gain confidence in the result by performing TCMC at least at the significant scope of 17 states.

## 6.4 Feature Interaction in a Telephone System

We created a model (Appendix A.4) for Feature Interaction in a Telephone System, which was previously modelled by Vakili [42]. This example models two features of a telephone system, call waiting and call forwarding, and investigates any interference caused by each feature for the other. We applied our contributions to this model, and discuss our modelling and model checking process in the rest of this section.

### 6.4.1 Style Guidelines

Similar to our previous case studies, we avoid invariants, including multiplicity constraints, in our state-space declaration, which promotes consistency, and use an equality constraint, as shown in Figure 6.19. For our operations, we separate the pre- and post-conditions, which adds structure and modularity, and ensure that distinct operations are defined to represent different kinds of transitions. For example, we create separate operations called `calling_talkingTo` and `calling_busy`, instead of aggregating two different pre- and post-conditions in one operation, as shown in Figure 6.20 Lines 1–19. We use the DisjMethod to constrain the transition relation, as shown in Figure 6.20 Lines 20–33, which produces the transitions we expect in the system.

```
 1 abstract sig Feature{}
 2 one sig CW,CF extends Feature{}
 3 sig PhoneNumber{
 4   feature: set Feature,
 5   fw: set PhoneNumber
 6 }
 7 sig State{
 8   idle: set PhoneNumber,
 9   busy: PhoneNumber -> PhoneNumber,
10   calling: PhoneNumber -> PhoneNumber,
11   talkingTo: PhoneNumber -> PhoneNumber,
12   waitingFor: PhoneNumber -> PhoneNumber,
13   forwardedTo: PhoneNumber -> PhoneNumber
14 }
15 pred equality [s,s':State] {
16   (s.idle = s'.idle and s.calling = s'.calling and
17   s.talkingTo = s'.talkingTo and s.busy = s'.busy and
18   s.waitingFor = s'.waitingFor and
19   s.forwardedTo = s'.forwardedTo)
20     implies s = s'
21 }
```

Figure 6.19: Telephone System: State Space

```
1 pred pre_calling_busy[s:State]{
2   some n,n':PhoneNumber |
3   (n->n' in s.calling) and (n' not in s.idle)
4 }
5 pred post_calling_busy[s,s':State]{
6   some n,n':PhoneNumber |
7   (s'.calling = s.calling - (n->n')) and (s'.busy = s.busy + (n->n'))
8   ...
9 }
10 pred pre_calling_talkingTo[s:State]{
11   some n,n':PhoneNumber |
12   (n->n' in s.calling) and (n' in s.idle)
13 }
14 pred post_calling_talkingTo[s,s':State]{
15   some n,n':PhoneNumber |
16   (s'.idle = s.idle - n') and (s'.calling = s.calling - (n -> n')) and
17   (s'.talkingTo = s.talkingTo + (n -> n'))
18   ...
19 }
20 pred calling_busy[s,s':State]{
21   pre_calling_busy[s]
22   post_calling_busy[s,s']
23 }
24 fact modelDefinition {
25   all s,s': State | (s->s' in nextState) iff
26     (idle_calling[s,s'] or calling_talkingTo[s,s'] or
27     talkingTo_idle[s,s' or calling_busy[s,s'] or
28     busy_waitingFor[s,s'] or busy_forwardedTo[s,s'] or
29     busy_idle[s,s'] or waitingFor_idle[s,s'] or
30     waitingFor_talkingTo[s,s'] or
31     forwardedTo_calling[s,s'])
32   ...
33 }
```

Figure 6.20: Telephone System: Operations and Transition Relation

### 6.4.2 Significance Axioms

We use our significance axioms, shown in Figure 6.21, to generate a non-spurious instance of the model by executing the command `run significanceAxioms for exactly 6 State, exactly 4 PhoneNumber`. The significant scope in this model for 4 phone numbers is 6 states.

```
 1 pred reachablityAxiom {
 2  all s:State | s in State.(initialState <: *nextState)
 3 }
 4 pred operationsAxiom {
 5  some s,s':State | idle_calling[s,s']
 6  some s,s':State | calling_talkingTo[s,s']
 7  ...
 8 }
 9 pred significanceAxioms {
10  reachablityAxiom
11  operationsAxiom
12 }
```

Figure 6.21: Telephone System: Significance Axioms (full axioms in Appendix A.4)

### 6.4.3 Scoped-TCMC Methodology

In this case study, we check the safety property that no phone number is being waited for and forwarded to at the same time, as shown in Figure 6.22. TCMC passes for the property at the specified scope, which is the significant scope. It is ambiguous whether the property holds for the entire state space, however, we gain confidence in our pass result by checking at least at the significant scope.

```
1 pred ap_safety [s:State] {
2   no s.waitingFor.PhoneNumber & s.forwardedTo.PhoneNumber
3 }
4 assert safety { ctl_mc[ ag [{s:State | ap_safety[s]}]] }
5 check safety for exactly 6 State, exactly 4 PhoneNumber
```

Figure 6.22: Telephone System: Safety Property

## 6.5 Scalability

This section explores the performance of TCMC with respect to these case studies. For this analysis, each property checked was satisfied by the model, which represents the worst-case scenario in terms of performance for universal properties, since all instances need to be checked in this case. For TCMC of our models, we used the Alloy Analyzer 4.2 with the MiniSat SAT-solver [16]. The experiments were run on an Intel(R) Xeon(R) CPU E3-1240 v5 @ 3.50GHzx8 machine running Linux version 4.4.0-92-generic with up to 64GB of user-space memory.

We show our performance results in the three sub-tables in Table 6.1: 6.1a gives performance data for our four properties (one from each property category) from the Musical Chairs example, 6.1b shows the performance data for the three universal properties checked in the Elevator System model, and 6.1c shows the performance results for model checking safety properties for the Traffic Light Controller and the Feature Interation in Telephone System case studies. The scope size (SS) denotes the sum of scopes of all uninterpreted sets.

With respect to scalability, although our contributions do not improve performance of the TCMC process, we found that they do not deteriorate performance compared to the TCMC process described in [42]. Therefore, CTLFC specifications can still be analyzed using TCMC up to the scopes that non-temporal specifications are often analyzed in Alloy, as previously shown in [42].

The models checked in Alloy are not as large as those that can be checked using a model checker such as NuSMV [9], however, the declarative and relational aspects of Alloy have significant advantages for creating concise, abstract behavioural models. TCMC adds to Alloy the ability to check complex temporal logic specifications directly on small scopes of these models, and this thesis provides a methodology to make useful deductions about the complete model checking problem as well.

| Musical Chairs. NS: 8, NR: 4 | | | | |
|---|---|---|---|---|
| SS | Safety | Existential | Finite Liveness | Infinite Liveness |
| 8 | 0.041 s | 0.011 s | 0.015 s | 0.132 s |
| 10 | 1.037 s | 0.076 s | 0.025 s | 0.379 s |
| 13 | 8.547 s | 0.377 s | 0.050 s | 4.726 s |
| 15 | 11 min 51 s | 0.488 s | 0.096 s | 6 min 29 s |
| 18 | >1 hour | 4.386 s | 0.134 s | >1 hour |

(a)

| Elevator System. NS: 3, NR: 4 | | | |
|---|---|---|---|
| SS | Safety | Finite Liveness | Infinite Liveness |
| 12 | 0.626 s | 1.815 s | 2.197 s |
| 13 | 1.934 s | 16.111 s | 18.676 s |
| 14 | 22.621 s | 1 min 24 s | 4 min 4 s |
| 15 | 3 min 11 s | 9 min 38 s | >1 hour |

(b)

| Feature Interaction. NS:5, NR:6 | | Traffic Light Controller. NS:18, NR:5 | |
|---|---|---|---|
| SS | Safety | SS | Safety |
| 9 | 2.54 s | 16 | 0.711 s |
| 10 | 18.40 s | 17 | 3.815 s |
| 11 | 9 min 25 s | 18 | 11 min 55 s |
| 12 | > 1 hour | 19 | > 1 hour |

(c)

Table 6.1: Performance Results of Case Studies. NS: Number of Signatures, NR: Number of Relations, SS: Scope Size, min: minutes, s: seconds

## 6.6 Summary

We developed four case studies to demonstrate the contributions of this thesis, as described in Chapters 3, 4, and 5. We believe that each of our models benefited in terms of structure, modularity, and consistency by using our proposed modelling style guidelines for abstract behavioural models. When generating instances of our models in the Alloy Analyzer, we were able to produce more useful transition systems for inspection by using the significance

axioms – these instances contained at least one initial state, and a reachable transition representing each defined operation. Using our proposed scoped-TCMC methodology in these case studies, we were able to reach useful deductions about properties for the entire state space from our results for scoped TCMC. We also showed an example of using multiple fairness constraints with TCMC using the `ctlfc` module. Through scalability experiments, we found that using TCMC for our case studies is feasible from a performance perspective.

# Chapter 7

# Comparison to NuSMV and BMC

As previously discussed, many different languages and tool-sets are available to create models and perform model checking tasks. NuSMV [9] is a tool from the SMV system [30] family that can be used for model checking. Alloy can be used to perform model checking using bounded model checking (BMC) [3]. In this chapter, we compare TCMC in Alloy to these two alternatives.

## 7.1 NuSMV

NuSMV [9] is a tool for symbolic model checking that uses the SMV system [30]. The tool only provides a rudimentary set of modelling constructs, with no support for first-order logic or sets/relations When modelling a transition system in NuSMV, it is only possible to define a unique transition relation, and models are written for fixed-sized collection structures. With sufficient computational resources, the tool checks the entire state space for desired properties.

To compare NuSMV to TCMC in Alloy, we modelled our Musical Chairs example in NuSMV. A part of this model is shown in Figure 7.1; the full model can be found in Appendix B.1. We focused primarily on investigating how to write a declarative behavioural model in NuSMV. The rest of this section compares the authoring experience of Musical Chairs in Alloy to that in NuSMV.

In Alloy, we model all our collection structures as sets and relations; however, in NuSMV, we have to use arrays, since sets and relations like Alloy's are not supported as native constructs. When declaring the state space for Musical Chairs in NuSMV, we

```
1 DEFINE
2   numPlayers := 3;
3   numChairs := numPlayers - 1;
4 VAR
5   players : array 1..numPlayers of boolean;
6   chairs : array 1..numChairs of boolean;
7   occupied : array 1..numChairs of 0..numPlayers;
8   ...
9 TRANS
10   case
11     mode = walking : next(mode) = sitting &
12     occupied[1] = 0 &
13     occupied[2] = 0 &
14     next(players[1]) = players[1] &
15     next(players[2]) = players[2] &
16     next(players[3]) = players[3] &
17     next(chairs[1]) = chairs[1] &
18     next(chairs[2]) = chairs[2] ;
19
20     mode = sitting : next(mode) = start &
21     -- occupied only has current chairs and players
22     (occupied[1]!=0 -> (players[occupied[1]] <-> chairs[1])) &
23     (occupied[2]!=0 -> (players[occupied[2]] <-> chairs[2])) &
24     -- eliminate player if player doesn't occupy any chairs
25     ((occupied[1]!=1 & occupied[2]!=1) ?
26       !next(players[1]) : next(players[1])=players[1]) &
27     ((occupied[1]!=2 & occupied[2]!=2) ?
28       !next(players[2]) : next(players[2])=players[2]) &
29     ((occupied[1]!=3 & occupied[2]!=3) ?
30       !next(players[3]) : next(players[3])=players[3]) &
31     -- leave chair outside game if already outside
32     ((!chairs[1]) -> next(chairs[1])=FALSE) &
33     ((!chairs[2]) -> next(chairs[2])=FALSE) &
34     -- eliminate 1 chair for next round
35     count(chairs[1],chairs[2]) =
36       next(count(chairs[1],chairs[2])) + 1 &
37     ...
```

Figure 7.1: Part of Musical Chairs model in NuSMV

use arrays of Booleans to represent our players and chairs as shown in Lines 5–6 of Figure 7.1. The `occupied` structure is modelled as an array of integers (Line 7), where each index represents a chair, and its value represents the player occupying that chair. A value

of `0` is used to indicate that the chair at that index is unoccupied. We found representing structures and attributes as sets and relations in Alloy more intuitive and concise than modelling them as arrays in NuSMV.

We cannot change the sizes of the collection structures in our model in NuSMV as easily as in Alloy. In Alloy, we can create our whole model without ever hard-coding the sizes of our sets, such as the number of player and chairs, into the constraints. We only specify the sizes when we generate an instance of the model or perform model checking tasks. Therefore, changing these sizes is trivial – we just change the scope of our `run` or `check` command. In NuSMV, however, the sizes of the arrays used to represent our structures must be specified at the very beginning (Figure 7.1 Lines 2–3). The size specified also dictates many of the model's constraints, which means that if we wish to change the size of any of these structures, such as the number of players, we need to make non-trivial changes to our model to reflect it. Lines 14–18 show the NuSMV model's constraints to ensure that the same players and chairs remain in the game in the next state – a similar constraint needs to be included for every additional player or chair. This kind of addition must be made to other more complex constraints as well, such as, the ones in Lines 22–23, which constrain the values of the `occupied` structure. Since the sizes of the structures are deeply ingrained in the model, we found NuSMV more inflexible than Alloy when modelling and model checking our Musical Chairs example for different numbers of players and chairs.

Alloy supports non-determinism in a first-class way by providing an abstract declarative language, whereas, creating non-determinism in NuSMV is not intuitive because of the lack of necessary language support. In the `eliminate_loser` operation in Alloy, we only need one constraint to ensure that the loser of the round is removed from the game: `s'.players = Chair.(s.occupied)`. In NuSMV, this non-deterministic behaviour, described in Lines 25–30 of Figure 7.1, requires a separate constraint for each player (which needs to be extended if we wish to increase the number of players). Additional constraints (Lines 32–33) are also needed to keep track of previously eliminated chairs. We found this kind of description of constraints in NuSMV tedious to write when compared to Alloy's concise constraints.

In summary, although NuSMV is faster than TCMC in Alloy, we found that our Musical Chairs model was more easily and clearly implemented in Alloy than in NuSMV. Modules in NuSMV may make the model's description less verbose, but it is clear that the abstractions provided by Alloy are substantially better for writing declarative models.

## 7.2 BMC in Alloy

Bounded model checking (BMC) [3] uses symbolic model checking to verify temporal (generally LTL) properties along paths up to a certain length. It is different from scoped TCMC in that, when model checking, BMC limits the path length, whereas, scoped TCMC limits the number of states. In scoped universal TCMC of scope $n$, we check all TS instances of size $n$ of any transition system that satisfies the transition relation. For each of these TS instances, TCMC calculates the set of states that satisfies the property, and checks if each initial state for that TS instance is in it. In BMC, all paths of the given length of any transition system that satisfies the transition relation are checked for the property.

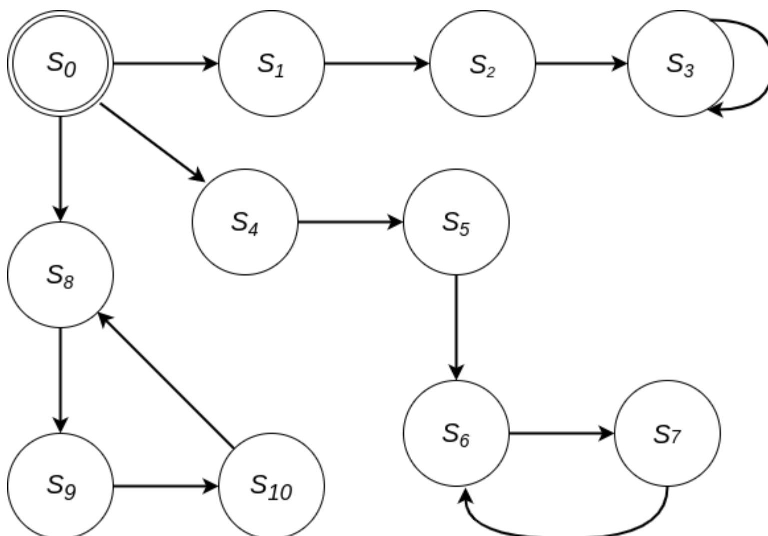



Figure 7.2: Example Transition System

If we consider the example transition system shown in Figure 7.2, where $S_0$ is the initial state, we can compare the paths (and instances) considered during model checking as follows: For a bound of 3, BMC looks at the following paths:

- $S_0 \to S_1 \to S_2 \to S_3$
- $S_0 \to S_4 \to S_5 \to S_6$
- $S_0 \to S_8 \to S_9 \to S_{10}$

For a state scope of 4 (which is comparable to a BMC bound of 3 since one more state than the path length may be used in BMC), TCMC in Alloy considers transition system

Table 7.1: Performance Results for BMC of a Safety Property (Figure 7.3 Line 9) in Musical Chairs. NS: Number of Signatures, NR: Number of Relations, SS: Scope Size, s: seconds

| Musical Chairs. NS: 8, NR: 1 | |
|---|---|
| SS | BMC of Safety Property |
| 8 | 0.001 s |
| 10 | 0.001 s |
| 13 | 0.002 s |
| 15 | 0.004 s |
| 18 | 0.007 s |

instances (and paths) such as the following, as well as all other instances with 4 states (all transitions between these states are included in the instance):

- Instance 1: $S_0, S_1, S_2, S_3$
  - $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_3 \rightarrow$ ... (infinite path)
- Instance 2: $S_0, S_4, S_5, S_6$
  - $S_0 \rightarrow S_4 \rightarrow S_5 \rightarrow S_6$
- Instance 3: $S_0, S_8, S_9, S_{10}$
  - $S_0 \rightarrow S_8 \rightarrow S_9 \rightarrow S_{10} \rightarrow S_8 \rightarrow S_9 \rightarrow S_{10} \rightarrow$ ... (infinite path)
- Instance 4: $S_0, S_1, S_2, S_4$
  - $S_0 \rightarrow S_1 \rightarrow S_2$
  - $S_0 \rightarrow S_4$

In TCMC, paths are not limited to the scope size, and can be infinite. In BMC, all paths are finite, and of the length specified; infinite paths can be represented using repeated states, but usually additional constraints (separate from the transition relation constraints) are necessary to designate loops, and for the model checking task to consider the paths infinite (for example, the `trace` module [12]).

In Alloy, we can perform BMC by utilizing Jackson's `ordering` module [24]. We can use the module to define the transition relation, and to express certain temporal properties for

```
1 open util/ordering [State] as StateOrder
2 ...
3 fact modelDefinition {
4   init [first]
5   all s: State-last | let s' = s.next | ops[s,s']
6 }
7 assert bmc {
8   // G(#players=#chairs+1)
9   all s:State | #s.players = (#s.chairs).plus[1]
10   // F(mode=sitting)
11   all s:State | init[s] implies ( some s':State |
12     (s' in nexts[s]+s and s'.mode=sitting) )
13 }
14 check bmc for exactly 3 Player, exactly 2 Chair, exactly 6 State
```

Figure 7.3: Musical Chairs: BMC in Alloy

model checking. The `ordering` module does not allow repeated states in a path, therefore, it is impossible to represent infinite paths. To compare TCMC in Alloy with BMC in Alloy, we implemented our Musical Chairs model using the `ordering` module, and model checked a safety property and a finite liveness property using BMC. Figure 7.3 shows the model definition block created using the `ordering` module (Lines 3–6), and the properties checked (Lines 7–13).

BMC performance is shown in Table 7.1. BMC is faster than TCMC because TCMC checks more instances (and paths) than BMC. Aside from performance, TCMC has several advantages over BMC in Alloy:

- The counterexamples produced by TCMC for liveness are real bugs in the complete transition system. In BMC, it is not possible to limit the search to infinite paths, and therefore ambiguous counterexamples, *i.e.*, instances with violating finite paths that would satisfy the liveness property if extended, are possible. It is possible to represent infinite paths for BMC in Alloy using the method proposed in [12] (requires extra constraints to represent loops in paths and to consider only infinite paths), which would prevent ambiguous counterexamples.
- Checking up to the significant scope in TCMC provides a measure of confidence in the scoped result independent of computing resources (which limit BMC).
- BMC can only check LTL properties, which means that we cannot express or check

CTLFC's existential properties using BMC. Existential TCMC allows us to check existential properties in Alloy.

- When performing BMC in Alloy using the `ordering` module, no repeated states are allowed in the paths, and therefore, it is only possible to consider finite paths. This means that there is no way to express and model check infinite liveness properties, which includes all properties with fairness constraints. TCMC can check all CTLFC properties. Without the `ordering` module it is possible that the paths may have repeated states.

- When a property does not hold, universal TCMC returns an instance that is a transition system, in which there is no path that satisfies the property. A transition system instance provides more inspectable information than a path; a violating (likely small) instance may include multiple paths that violate the property uncovering multiple bugs.

We also compared the deductions we can make from scoped TCMC about the entire state space, as described in Chapter 5, to those from BMC. Table 7.2 summarizes our findings. For safety properties, we found that TCMC and BMC achieve similar conclusions. For finite liveness properties, TCMC produces real bugs, whereas, a failure in BMC using the `ordering` module represents an ambiguous result for the complete model checking problem. On the other hand, a pass result from BMC indicates a real pass, whereas a pass from TCMC is ambiguous since it only checks infinite paths. However, in TCMC, we can add our dead-loop constraint (Section 5.3) to check finite paths as well, yielding

Table 7.2: Deducing Complete Model Checking Results in Alloy: Scoped TCMC vs. BMC

<table>
<tr><th colspan="2"></th><th colspan="2">Scoped TCMC</th><th colspan="2">BMC using <code>ordering</code></th></tr>
<tr><th colspan="2">Property</th><th>Pass</th><th>Fail</th><th>Pass</th><th>Fail</th></tr>
<tr><td colspan="2">Safety</td><td>Ambiguous</td><td>Real Bug</td><td>Ambiguous</td><td>Real Bug</td></tr>
<tr><td rowspan="2">Finite Liveness</td><td>w/o dead-loop</td><td>Ambiguous</td><td>Real Bug</td><td rowspan="2">Real Pass</td><td rowspan="2">Ambiguous</td></tr>
<tr><td>w/ dead-loop</td><td>Real Pass</td><td>Ambiguous</td></tr>
<tr><td colspan="2">Infinite Liveness</td><td>Ambiguous</td><td>Real Bug</td><td colspan="2">Cannot Express</td></tr>
<tr><td colspan="2">Existential</td><td>Real Pass</td><td>Ambiguous</td><td colspan="2">Cannot Express</td></tr>
</table>

results similar to BMC for finite liveness properties. We cannot express infinite liveness and existential properties for BMC using the `ordering` module.

In conclusion, while TCMC is currently slower than BMC, TCMC produces more conclusive results for the complete model checking problem, and can check a larger range of properties than BMC.

## 7.3 Summary

In this chapter, we compared techniques for developing a model in NuSMV to those in Alloy. We found Alloy to be a more suitable language for creating declarative behavioural models. We investigated the use of BMC in Alloy using the `ordering` module, which exhibits several deficiencies when compared to TCMC, including the inability to check infinite paths and lack of support for all CTLFC properties. TCMC checks more instances (and paths) than BMC, which makes TCMC slower than BMC, but TCMC provides more conclusive results for a comparable scope.

# Chapter 8

# Related Work

In this chapter, we discuss work related to this thesis.

The SMV system [30] family provides efficient tools, such as NuSMV [9], for symbolic model checking of finite-state transition systems for temporal properties. However, these tools only provide a primitive set of modelling constructs, which excludes first-order logic and sets/relations. The lack of support in model development, as elaborated in Section 7.1, often makes such tools unsuitable for users.

Chang and Jackson [8] propose a modelling language for transition systems focusing on the integration of operators used in programming languages (such as loops, function calls, and integer arithmetic) in temporal properties and behavioural constraints, which can be expressed imperatively or declaratively, augmenting the traditional languages of model checkers with sets, relations and declarative constructs to specify a transition system. They introduce a BDD-based model checker for temporal properties that has only been verified with small simple models. TCMC in Alloy and our proposed methodologies provide more robust support for model checking of temporal properties, and for generating interesting instances for inspection.

The B [1] modelling language provides many features that are similar to Alloy, such as sets and relations. ProB [27] can be used to model check finite B *machines*, the name given to B models, for LTL properties. However, ProB employs an explicit-state-search based model checking technique which requires a considerable amount of computing resources. Several implementations of symbolic model checking algorithms (BMC, k-induction, IC3) for B machines are provided in [25], however, they cannot check all CTLFC properties, and suffer from solver performance constraints (as does our TCMC in Alloy methodologies). Some of these algorithms are iterative (k-induction, IC3), meaning that they involve

multiple runs of the solver, hindering efficiency.

The Abstract State Machine (ASM) method [4] is generally used to model high-level system designs as infinite transition systems. The transition systems can be analyzed using techniques such as, theorem proving [15, 39], and model checking [13], which involves translating ASMs to SMV models with fixed sized structures. Translation-based approaches usually unfold user-level abstractions and make understanding models and counterexamples difficult.

TLA+ [45] (with the TLC model checker) checks behavioural models for temporal properties. TLC supports model checking of a subset of LTL formulas using explicit-state model checking. Our work using TCMC in Alloy supports all CTLFC properties and involves a symbolic approach to model checking.

The `ordering` module of Alloy can be used for simple bounded model checking (BMC) [3]. Cunha [12] proposes a module called `trace`, which is based on the `ordering` module, to model infinite paths, and for bounded model checking of LTL properties. TCMC, which is available as the `ctlfc` and `ctl` modules in Alloy, and our model checking methodology, support more sophisticated temporal properties and provide some advantages over BMC as discussed in Section 7.2.

Electrum [29] is an extension of Alloy that incorporates features from both Alloy and TLA+. It introduces syntax for signifying mutable variables, and focuses on the ability to model check both structural and behavioural properties. It supports finite-state bounded model checking of LTL properties using methods similar to those proposed in [12]. It also provides a feature to translate the model and properties to nuXmv [7], which is a faster extension of NuSMV, for unbounded model checking. TCMC and our model checking methodology focus on using only Alloy without extensions, to perform model checking tasks of a larger set of temporal properties (CTLFC), and producing interesting transition system instances for inspection.

DynAlloy [18], along with the DynAlloy Analyzer [38], is a set of extensions to Alloy for describing and analyzing dynamic properties of systems, that is, properties that reason about execution traces as opposed to structural properties. They introduce syntax for the use of actions to describe operations of the system when defining a transition relation. This kind of description requires the separation of pre- and post-conditions of actions. However, DynAlloy does not support the model checking of any temporal logic properties. The work in [33] proposes an extension similar to DynAlloy, which provides support for imperative operators, and adds syntax to denote actions and mutable variables. Our proposed Alloy modelling style guidelines provide guidance for creating behavioural models for model checking of temporal properties, such as CTLFC, without the use of any extensions or

extraneous keywords.

Giannakopoulos *et al.* [20] offers some ideas on how to use Alloy to model state-based systems. The authors ask users to separate constraints that are state invariants from initial state and operation constraints. We suggest avoiding constraining the state space with expected invariants altogether, and instead model the system using only initial state and operation constraints, which allows us to check for modelling errors by model checking for the invariant. Alchemy [26] is another extension to Alloy that interprets Alloy operation constraints and specifications imperatively. It mainly focuses on interpreting systems modelled using Alloy as relational database systems.

Generator axioms, described by Jackson in [24], provide a technique to consider all elements in a set when checking assertions that may be affected by the scope size specified for such a set. Our significance axioms are similar to generator axioms, but are motivated by our interest in producing non-spurious instances for inspection, as well as for model checking. Jackson's generator axioms require the generation of states with all possible attribute values, which causes the state scope to explode. Our significance axioms focus on generating a transition representing each possible operation. The scope only increases linearly to the finite number of operations defined (depending on the reachability of transitions representing the operations), which is usually smaller than that of all possible state attribute combinations.

# Chapter 9

# Conclusion

In this thesis, we focused on using existing tools to make model checking of abstract behavioural models more practical. Transitive-closure-based model checking (TCMC) is able to check a broad category of temporal (CTLFC) properties of behavioural systems using the Alloy Analyzer, a tool that provides valuable support for creating abstract models, without any extension or translation. We attempted to increase TCMC's practical value by addressing some issues faced by users when modelling and model checking using TCMC in Alloy.

We investigated three distinct aspects of using TCMC in Alloy:

1. *Modelling a Transition System in Alloy:* We developed some guidelines for Alloy that we believe promote structure, modularity, and consistency in the model. These guidelines do not involve any extensions to Alloy. We recognized two common styles for defining the transition relation, which we call the *ConjMethod* and the *DisjMethod*, and discussed their implications.

2. *Generating an Instance:* We proposed a set of axioms, called the *significance axioms*, that aid in the generation of useful instances when using the `run` command in the Alloy Analyzer. These axioms ensure that the instance produced contain only reachable states, an initial state, and reachable transitions representing each user-defined operation. As a modeller, we found these instances more useful to inspect for modelling correctness.

3. *Scoped-TCMC Methodology:* We carefully described the meaning of results from TCMC at scopes smaller than that of the total state space, with respect to the

complete model checking problem (meaning over the entire state space), highlighting distinctions for properties with respect to finite and infinite paths. We discussed how, during TCMC, the *significant scope*, the minimum scope at which our significance axioms are satisfied, provides a measure independent of computing resource limitations that a significant part of the state space has been verified.

We utilized our guidelines and methodologies to develop four case studies, which demonstrated our claims and results. We also used these case studies to compare TCMC in Alloy to other tools and model checking methods. Although NuSMV analysis is faster, we found Alloy to be a more suitable language for creating abstract declarative models than the NuSMV language. BMC in Alloy, although currently faster than TCMC in Alloy, does not support checking all CTLFC properties. Counterexamples produced by TCMC always contain real, useful bugs because of its ability to consider finite and infinite paths separately, whereas, counterexamples produced by BMC for liveness properties can represent false negatives.

The following lists the contributions of this thesis:

- Establishes a set of style guidelines for modelling abstract behavioural systems in Alloy without extensions to Alloy.
- Introduces *significance axioms* and *significant scope* for transition systems, which address the spurious instances problem.
- Introduces *scoped TCMC* to apply TCMC on subgraphs of a complete transition system.
- Analyzes and documents the meanings of scoped TCMC results for different property categories with respect to the complete model checking problem.
- Presents case studies to demonstrate proposed claims and results.
- Investigates the scalability of TCMC in Alloy.
- Compares declarative modelling practices in Alloy to those in NuSMV.
- Compares expressibility of temporal properties, and model checking results of TCMC to those of BMC.

The work in this thesis could be supplemented in the future by looking into categorizing existential properties further to provide more useful deductions from scoped existential TCMC. It may be helpful to investigate if our Operations Axiom can be modularized to make writing it simpler and more flexible. It may also be useful to explore the use of TCMC for declarative models that define more than one transition system. Research on the extraction of a particular path from counterexample TS instances produced by the Alloy Analyzer would be helpful. TCMC could be implemented in Alloy* [32], an extension of the Alloy Analyzer, to take advantage of its higher-order quantification features. Further investigation into improving the scalability of TCMC could also increase the method's utility.

Model checking of software has become more relevant in engineering industries because of the rise of easily adoptable tools, as described in [34]. We believe that TCMC shows promise in the world of temporal model checking, therefore, this thesis attempts to make TCMC's implementation in Alloy more practical. We believe research into the accessibility of existing and upcoming tools and methods is immensely important in order to encourage the usage of formal methods in the wider community of software engineering, because it has the potential to prevent many critical software disasters.

# References


[1] Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. *Satisfiability Modulo Theories*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. 2009.

[3] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. volume 58 of *Advances in Computers*, pages 117 – 148. Elsevier, 2003.

[4] Egon Börger. The ASM Method for System Design and Analysis. A Tutorial Introduction. In *Frontiers of Combining Systems*, volume 3717 of *Lecture Notes In Computer Science*, pages 264–283. Springer, 2005.

[5] Aaron R. Bradley. SAT-Based Model Checking without Unrolling. In *Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.

[6] Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24:293–318, 1992.

[7] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model checker. In *Computer-Aided Verification*, pages 334–342. Springer, 2014.

[8] Felix Sheng-Ho Chang and Daniel Jackson. Symbolic Model Checking of Declarative Relational Models. In *International Conference on Software Engineering*, pages 312–320. ACM, 2006.

[9] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Computer Aided Verification*, volume 2404 of *Lecture Notes In Computer Science*, pages 241–268. Springer, 2002.

[10] Edmund Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[11] Edmund M. Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another Look at LTL Model Checking. *Formal Methods in System Design*, 10:47–71, 1997.

[12] Alcino Cunha. Bounded Model Checking of Temporal Formulas with Alloy. In *International Conference on Abstract State Machines, Alloy, B, VDM, and Z*, pages 303–308. Springer, 2014.

[13] Giuseppe Del Castillo and Kirsten Winter. Model Checking Support for the ASM High-Level Language. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes In Computer Science*, pages 331–346. Springer, 2000.

[14] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[15] A. Dold. A Formal Representation of Abstract State Machines Using PVS. Verifix Technical Report Ulm/6.2, Universität Ulm, 1998.

[16] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes In Computer Science*, pages 333–336. Springer, 2004.

[17] Sabria Farheen, Nancy A. Day, Amirhossein Vakili, and Ali Abbassi. Transitive-closure-based model checking (TCMC) in Alloy. Manuscript submitted for publication, 2017.

[18] Marcelo F. Frias, Juan P. Galeotti, Carlos G. López Pombo, and Nazareno M. Aguirre. DynAlloy: Upgrading Alloy with Actions. In *International Conference on Software Engineering*, pages 442–451. ACM, 2005.

[19] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. *Simple On-the-fly Automatic Verification of Linear Temporal Logic*, pages 3–18. Springer, 1996.

[20] Theophilos Giannakopoulos, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Towards an Operational Semantics for Alloy. In *FM 2009: Formal Methods: Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, pages 483–498. Springer, 2009.

[21] Neil Immerman and Moshe Vardi. Model Checking and Transitive-Closure Logic. In *Computer-Aided Verification*, volume 1254 of *Lecture Notes In Computer Science*, pages 291–302. Springer, 1997.

[22] *Information Technology Z Formal Specification Notation Syntax, Type System and Semantics*, 2000.

[23] Daniel Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.

[24] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006.

[25] Sebastian Krings and Michael Leuschel. Proof assisted bounded and unbounded symbolic model checking of software and system models. *Science of Computer Programming*, 2017.

[26] Shriram Krishnamurthi, Kathi Fisler, Daniel J. Dougherty, and Daniel Yoo. Alchemy: Transmuting Base Alloy Specifications into Implementations. In *Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 158–169. ACM, 2008.

[27] Michael Leuschel and Michael Butler. ProB : an automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer*, 10:185–203, 2008.

[28] Xianhong Liu. Identification and check of inconsistencies between UML diagrams. In *International Conference on Computer Sciences and Applications*, pages 487–490. IEEE, 2013.

[29] Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. Lightweight Specification and Analysis of Dynamic Systems with Rich Configurations. In *Foundations of Software Engineering*, pages 373–383. ACM, 2016.

[30] Kenneth L. McMillan. The SMV system, 1992.

[31] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[32] Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang, and Daniel Jackson. Alloy*: A general-purpose higher-order relational constraint solver. In *International Conference on Software Engineering*, volume 1, pages 609–619. IEEE, 2015.

[33] Joseph P. Near and Daniel Jackson. An Imperative Extension to Alloy. In *International Conference on Abstract State Machines, Alloy, B, VDM, and Z*, pages 118–131. Springer, 2010.

[34] Chris Newcombe. Why Amazon Chose TLA+. In *International Conference on Abstract State Machines, Alloy, B, VDM, and Z*, pages 25–39. Springer, 2014.

[35] Nimal Nissanke. *Formal Specification: Techniques and Applications*. Springer Verlag, 1999.

[36] Malte Plath and Mark Ryan. Feature integration using a feature construct. *Science of Computer Programming*, 41(1):53–84, 2001.

[37] Amir Pnueli. The Temporal Logic of Programs. In *Foundations of Computer Science*, pages 46–57. IEEE Computer Society, 1977.

[38] Germán Regis, César Cornejo, Simón Gutiérrez Brida, Mariano Politano, Fernando Raverta, Pablo Ponzio, Nazareno Aguirre, Juan Pablo Galeotti, and Marcelo Frias. DynAlloy Analyzer: A Tool for the Specification and Analysis of Alloy Models with Dynamic Behaviour. In *Foundations of Software Engineering*, pages 969–973. ACM, 2017.

[39] G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science*, 3(4):377–413, 1997.

[40] James Somers. The Coming Software Apocalypse. *The Atlantic*. [Online; posted 26-September-2017].

[41] Emina Torlak and Daniel Jackson. Kodkod: A Relational Model Finder. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647. Springer, 2007.

[42] Amirhossein Vakili. *Temporal Logic Model Checking as Automated Theorem Proving*. PhD thesis, University of Waterloo, David R. Cheriton School of Computer Science, 2016.

[43] Amirhossein Vakili and Nancy A. Day. Temporal Model Checking in Alloy. In *International Conference on Abstract State Machines, Alloy, B, VDM, and Z*, pages 150–163. Springer, 2012.

[44] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115:1–37, 1994.

[45] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. In *IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 54–66. Springer Verlag, 1999.

# APPENDICES

# Appendix A

# Alloy Models: TCMC Case Studies

## A.1 Musical Chairs

```
1  open ctl[State]
2  open util/integer
3
4  //**********************STATE SPACE************************//
5  sig Chair, Player {}
6  abstract sig Mode {}
7  one sig start, walking, sitting, end extends Mode {}
8
9  sig State {
10   // current players
11   players: set Player,
12   //current chairs
13   chairs: set Chair,
14   // current chair player relation
15   occupied: set Chair -> set Player,
16   // current state of game, should always be 1
17   mode : set Mode
18 }
19
20 //****************INITIAL STATE CONSTRAINTS*******************//
21
22 pred init [s:State] {
23   s.mode = start
24   #s.players > 1
25   #s.players = (#s.chairs).plus[1]
26   // force all Chair and Player to be included
```

```
27    s.players = Player
28    s.chairs = Chair
29    s.occupied = none -> none
30  }
31
32  //******************TRANSITION CONSTRAINTS*****************//
33  pred pre_music_starts [s: State] {
34    #s.players > 1
35    s.mode = start
36  }
37  pred post_music_starts [s, s': State] {
38    s'.players = s.players
39    s'.chairs = s.chairs
40    // no one is sitting after music starts
41    s'.occupied = none -> none
42    s'.mode= walking
43  }
44  pred music_starts [s, s': State] {
45    pre_music_starts[s]
46    post_music_starts[s,s']
47  }
48
49  pred pre_music_stops [s: State] {
50    s.mode = walking
51  }
52  pred post_music_stops [s, s': State] {
53    s'.players = s.players
54    s'.chairs = s.chairs
55    // no other chair/player than chairs/players
56    s'.occupied in s'.chairs -> s'.players
57    // forcing occupied to be total and
58    //each chair mapped to only one player
59    all c:s'.chairs | one c.(s'.occupied)
60    // each "occupying" player is sitting on one chair
61    all p:Chair.(s'.occupied) | one s'.occupied.p
62    s'.mode = sitting
63  }
64  pred music_stops [s, s': State] {
65    pre_music_stops[s]
66    post_music_stops[s,s']
67  }
68
69  pred pre_eliminate_loser [s: State] {
70    s.mode = sitting
71  }
```

```
72  pred post_eliminate_loser [s, s': State] {
73    // loser is the player in the game not in the range of occupied
74    s'.players = Chair.(s.occupied)
75    #s'.chairs = (#s.chairs).minus[1]
76    s'.mode = start
77  }
78  pred eliminate_loser [s, s': State] {
79    pre_eliminate_loser[s]
80    post_eliminate_loser[s,s']
81  }
82
83  pred pre_declare_winner [s: State] {
84    #s.players = 1
85    s.mode = start
86  }
87  pred post_declare_winner [s, s': State] {
88    s'.players = s.players
89    s'.chairs = s.chairs
90    s'.mode = end
91  }
92  pred declare_winner [s, s': State] {
93    pre_declare_winner[s]
94    post_declare_winner[s,s']
95  }
96
97  pred pre_end_loop [s: State] {
98    s.mode = end
99  }
100 pred post_end_loop [s, s': State] {
101   s'.mode = end
102   s'.players = s.players
103   s'.chairs = s.chairs
104   s'.occupied = s.occupied
105 }
106 pred end_loop [s, s': State] {
107   pre_end_loop[s]
108   post_end_loop[s,s']
109 }
110
111 // helper to define valid transitions
112 pred ops [s,s': State] {
113   music_starts[s,s'] or
114   music_stops[s,s'] or
115   eliminate_loser[s,s'] or
116   declare_winner[s,s'] or
```

```
117    end_loop[s,s']
118  }
119
120  //**********************MODEL DEFINITION********************//
121  fact {
122    all s:State | s in initialState iff init[s]
123    all s,s':State | s->s' in nextState iff ops[s,s']
124    // equality pred: two states with the same features are equivalent
125    all s, s': State | s.players=s'.players and
126      s.chairs=s'.chairs and
127      s.occupied=s'.occupied and
128      s.mode=s'.mode
129      implies s = s'
130  }
131
132  //*********************SIGNIFICANCE AXIOMS********************//
133  pred reachablityAxiom {
134    all s:State | s in State.(initialState <: *nextState)
135  }
136  pred operationsAxiom {
137    some s,s':State | music_starts[s,s']
138    some s,s':State | music_stops[s,s']
139    some s,s':State | eliminate_loser[s,s']
140    some s,s':State | declare_winner[s,s']
141    some s,s':State | end_loop[s,s']
142  }
143  pred significanceAxioms {
144    reachablityAxiom
145    operationsAxiom
146  }
147  run significanceAxioms for exactly 3 Player, exactly 2 Chair,
148   exactly 8 State
149
150  //*********************PROPERTIES********************//
151  //*********************SAFETY**********************//
152  assert safety {
153    // number of players is walways 1 greater than number of chairs
154    ctl_mc[ag[{s: State| #s.players = (#s.chairs).plus[1] }]]
155  }
156  check safety for exactly 3 Player, exactly 2 Chair,
157    exactly 8 State
158  //*********************EXISTENTIAL**********************//
159  one sig Alice extends Player{}
160  pred existential {
161    // Alice wins in some instance
```

```
162     ctl_mc[ef[{s: State| s.mode=end and s.players=Alice}]]
163   }
164   run existential for exactly 3 Player, exactly 2 Chair,
165     exactly 8 State
166   //**********************FINITE LIVENESS*******************//
167   assert finiteLiveness {
168     ctl_mc[ af [{ s: State| s.mode=sitting }]]
169   }
170   check finiteLiveness for exactly 3 Player, exactly 2 Chair,
171     exactly 8 State
172   //*********************INFINITE LIVENESS********************//
173   assert infiniteLiveness {
174     // number of players eventually always reaches and remains at 1
175     ctl_mc[ af [ ag [{s: State| #s.players=1}]]]
176   }
177   check infiniteLiveness for exactly 3 Player, exactly 2 Chair,
178     exactly 8 State
```

## A.2 Elevator System

```
 1   module Elevator
 2   open util/integer
 3   open util/ordering[Floor]
 4   open ctl[State]
 5
 6   sig Floor {}
 7   one sig Up {}
 8   sig State {
 9     current: set Floor,
10     goingUp: set Up,
11     called: set Floor,
12     maint: Int
13   }
14
15   //****************INITIAL STATE CONSTRAINTS*******************//
16   pred init [s:State] {
17     #s.called = 0
18     s.maint = 1
19     no s.goingUp
20     s.current = max[Floor]
21   }
22
```

```
23  //*********************OPERATIONS**********************//
24  pred pre_maintain[s:State] {
25    s.maint = 2
26  }
27  pred post_maintain[s,s':State] {
28    s'.current = min[Floor]
29    no s'.goingUp
30
31    s'.maint = 0
32    s'.goingUp = s.goingUp
33    (s.called - s'.current) in s'.called
34    s'.current not in s'.called
35  }
36  pred maintain[s,s':State] {
37    pre_maintain[s]
38    post_maintain[s,s']
39  }
40
41  pred pre_changeDirToDown[s:State] {
42    some s.called
43    s.maint < 2
44    some s.goingUp
45    no nexts[s.current] & s.called
46  }
47  pred post_changeDirToDown[s,s':State] {
48    no s'.goingUp
49    s'.maint = s.maint.plus[1]
50
51    s'.current = s.current
52    (s.called - s'.current) in s'.called
53    s'.current not in s'.called
54
55  }
56  pred changeDirToDown[s,s':State] {
57    pre_changeDirToDown[s]
58    post_changeDirToDown[s,s']
59  }
60
61  pred pre_changeDirToUp[s:State] {
62    some s.called
63    s.maint < 2
64    no s.goingUp
65    no prevs[s.current] & s.called
66  }
67  pred post_changeDirToUp[s,s':State] {
```

```
68    some s'.goingUp
69    s'.maint = s.maint.plus[1]
70
71    s'.current = s.current
72    (s.called - s'.current) in s'.called
73    s'.current not in s'.called
74
75  }
76  pred changeDirToUp[s,s':State] {
77    pre_changeDirToUp[s]
78    post_changeDirToUp[s,s']
79  }
80
81  pred pre_moveUp[s:State] {
82    some s.called
83    some s.goingUp
84    some nexts[s.current] & s.called
85  }
86  pred post_moveUp[s,s':State] {
87    s'.current = min[nexts[s.current] & s.called]
88    s'.current not in s'.called
89
90    s'.maint = s.maint
91    s'.goingUp = s.goingUp
92    (s.called - s'.current) in s'.called
93  }
94  pred moveUp[s,s':State] {
95    pre_moveUp[s]
96    post_moveUp[s,s']
97  }
98
99  pred pre_moveDown[s:State] {
100   some s.called
101   no s.goingUp
102   some prevs[s.current] & s.called
103 }
104 pred post_moveDown[s,s':State] {
105   s'.current = max[prevs[s.current] & s.called]
106
107   s'.current not in s'.called
108   s'.maint = s.maint
109   s'.goingUp = s.goingUp
110   (s.called - s'.current) in s'.called
111 }
112 pred moveDown[s,s':State] {
```

```
113     pre_moveDown[s]
114     post_moveDown[s,s']
115   }
116
117   pred pre_defaultToGround[s:State] {
118     no s.called
119     min[Floor] not in s.current
120   }
121   pred post_defaultToGround[s,s':State] {
122     s'.current = min[Floor]
123     no s'.goingUp
124     s'.maint = s.maint
125     (s.called - s'.current) in s'.called
126     s'.current not in s'.called
127   }
128   pred defaultToGround[s,s':State] {
129     pre_defaultToGround[s]
130     post_defaultToGround[s,s']
131   }
132
133
134   pred pre_idle[s:State] {
135     no s.called
136     s.current = min[Floor]
137   }
138   pred post_idle[s,s':State] {
139     s'.maint = 0
140     s'.current = s.current
141     s'.goingUp = s.goingUp
142     (s.called - s'.current) in s'.called
143     s'.current not in s'.called
144   }
145   pred idle[s,s':State] {
146     pre_idle[s]
147     post_idle[s,s']
148   }
149
150   // helper to define valid transitions
151   pred ops [s,s': State] {
152     changeDirToDown[s,s'] or
153     changeDirToUp[s,s'] or
154     moveUp[s,s'] or
155     moveDown[s,s'] or
156     defaultToGround[s,s'] or
157     idle[s,s'] or
```

```
158     maintain[s,s']
159   }
160
161   //*********************MODEL DEFINITION******************//
162   fact {
163     all s:State | s in initialState iff init[s]
164     all s,s':State | s->s' in nextState iff ops[s,s']
165     // equality pred: two states with the same features are equivalent
166     all s, s': State |
167       s'.current = s.current and
168       s'.maint = s.maint and
169       s'.goingUp = s.goingUp and
170       s'.called = s.called
171       implies s = s'
172   }
173
174   //*********************SIGNIFICANCE AXIOMS*****************//
175   pred reachablityAxiom {
176     all s:State | s in State.(initialState <: *nextState)
177   }
178   pred operationsAxiom {
179     some s,s':State | changeDirToDown[s,s']
180     some s,s':State | changeDirToUp[s,s']
181     some s,s':State | moveUp[s,s']
182     some s,s':State | moveDown[s,s']
183     some s,s':State | defaultToGround[s,s']
184     some s,s':State | idle[s,s']
185     some s,s':State | maintain[s,s']
186   }
187   pred significanceAxioms {
188     reachablityAxiom
189     operationsAxiom
190   }
191   run significanceAxioms for exactly 6 Floor, exactly 7 State
192
193   //*********************PROPERTIES********************//
194   //*********************SAFETY**********************//
195   assert safety {
196     // current is only one floor
197     ctl_mc[ag[{s: State| one s.current}]]
198   }
199   check safety for exactly 6 Floor, exactly 7 State
200   //*****************FINITE LIVENESS****************//
201   assert finiteLiveness {
202     // eventually reaches a maintenance state
```

```
203     ctl_mc[af[{s: State| s.maint = 0}]]
204   }
205   check finiteLiveness for exactly 6 Floor, exactly 7 State
206   //***************INFINITE LIVENESS******************//
207   assert infiniteLiveness {
208     // a floor called is always eventually reached as current
209     // AG(floorCalled => AF (floorCurrent) )
210     all f:Floor | ctl_mc[ ag [ imp_[called.f, af [current.f]] ] ]
211   }
212   check infiniteLiveness for exactly 6 Floor, 8 State
```

## A.3 Traffic Light Controller

```
 1   open util/integer
 2   open util/boolean
 3   open ctlfc[State]
 4
 5   //**********************STATE SPACE************************//
 6   abstract sig Counter{}
 7   one sig f0, f1, f2, f3 extends Counter{}
 8
 9   abstract sig Sense{}
10   one sig N_Sense, S_Sense, E_Sense extends Sense{}
11
12   // Go is for modeling which direction is allowed to go
13   abstract sig Go{}
14   one sig N_Go, S_Go, E_Go extends Go{}
15
16   // Request is to latch the traffic sensors
17   abstract sig Request{}
18   one sig N_Req, S_Req, E_Req extends Request{}
19
20   sig State{
21     sensors: set Sense,
22     goes: set Go,
23     req: set Request,
24     NS_Lock: set Bool, // NS_Lock is true iff East is not allowed to go
25     // counter for fairness
26     counter: set Counter
27   }
28
29   //***************INITIAL STATE CONSTRAINT***************//
```

```
30  pred initial[s:State]{
31    !no s.sensors
32    no s.goes
33    no s.req
34    s.NS_Lock = False
35    s.counter = f0
36  }
37  //*************TRANSITION CONSTRAINTS/OPERATIONS************//
38  // Predicates for N_Go
39  pred N_Go_True[s:State]{
40    N_Req in s.req
41    N_Go !in s.goes
42    E_Req !in s.req
43  }
44  pred N_Go_False[s:State]{
45    N_Go in s.goes
46    N_Sense !in s.sensors
47  }
48
49  pred pre_N_Go[s:State]{
50    N_Go_True[s]
51  }
52  pred pre_N_Not_Go[s:State]{
53    !N_Go_True[s] and N_Go_False[s]
54  }
55  pred pre_N_Go_Unchanged[s:State]{
56    !N_Go_True[s] and !N_Go_False[s]
57  }
58
59  pred post_N_Go_[s':State]{
60    N_Go in s'.goes
61  }
62  pred post_N_Not_Go[s':State]{
63    N_Go !in s'.goes
64  }
65  pred post_N_Go_Unchanged[s,s':State]{
66    N_Go in s'.goes iff N_Go in s.goes
67  }
68
69  pred N_Go_[s,s':State]{
70    pre_N_Go[s]
71    post_N_Go_[s']
72  }
73  pred N_Not_Go[s,s':State]{
74    pre_N_Not_Go[s]
```

```
75    post_N_Not_Go[s']
76  }
77  pred N_Go_Unchanged[s,s':State]{
78    pre_N_Go_Unchanged[s]
79    post_N_Go_Unchanged[s,s']
80  }
81
82
83  // Predicates for S_Go
84  pred S_Go_True[s:State]{
85    S_Req in s.req
86    S_Go !in s.goes
87    E_Req !in s.req
88  }
89  pred S_Go_False[s:State]{
90    S_Go in s.goes
91    S_Sense !in s.sensors
92  }
93
94  pred pre_S_Go[s:State]{
95    S_Go_True[s]
96  }
97  pred pre_S_Not_Go[s:State]{
98    !S_Go_True[s] and S_Go_False[s]
99  }
100 pred pre_S_Go_Unchanged[s:State]{
101   !S_Go_True[s] and !S_Go_False[s]
102 }
103
104 pred S_Go_[s,s':State]{
105   pre_S_Go[s]
106   S_Go in s'.goes
107 }
108 pred S_Not_Go[s,s':State]{
109   pre_S_Not_Go[s]
110   S_Go !in s'.goes
111 }
112 pred S_Go_Unchanged[s,s':State]{
113   pre_S_Go_Unchanged[s]
114   S_Go in s'.goes iff S_Go in s.goes
115 }
116
117
118 // Predicates for E_Go
119 pred E_Go_True[s:State]{
```

```
120    E_Req in s.req
121    E_Go !in s.goes
122    s.NS_Lock = False
123  }
124  pred E_Go_False[s:State]{
125    E_Go in s.goes
126    E_Sense !in s.sensors
127  }
128
129  pred pre_E_Go[s:State]{
130    E_Go_True[s]
131  }
132  pred pre_E_Not_Go[s:State]{
133    !E_Go_True[s] and E_Go_False[s]
134  }
135  pred pre_E_Go_Unchanged[s:State]{
136    !E_Go_True[s] and !E_Go_False[s]
137  }
138
139  pred E_Go_[s,s':State]{
140    pre_E_Go[s]
141    E_Go in s'.goes
142  }
143  pred E_Not_Go[s,s':State]{
144    pre_E_Not_Go[s]
145    E_Go !in s'.goes
146  }
147  pred E_Go_Unchanged[s,s':State]{
148    pre_E_Go_Unchanged[s]
149    E_Go in s'.goes iff E_Go in s.goes
150  }
151
152  // Predicates for N_Req
153  pred N_Req_True[s:State]{
154    N_Sense in s.sensors
155  }
156  pred N_Req_False[s:State]{
157    N_Go_False[s]
158  }
159
160  pred pre_N_Req[s:State]{
161    N_Req_True[s]
162  }
163  pred pre_N_Not_Req[s:State]{
164    !N_Req_True[s] and N_Req_False[s]
```

```
165   }
166   pred pre_N_Req_Unchanged[s:State]{
167     !N_Req_True[s] and !N_Req_False[s]
168   }
169
170   pred N_Req_[s,s':State]{
171     pre_N_Req[s]
172     N_Req in s'.req
173   }
174   pred N_Not_Req[s,s':State]{
175     pre_N_Not_Req[s]
176     N_Req !in s'.req
177   }
178   pred N_Req_Unchanged[s,s':State]{
179     pre_N_Req_Unchanged[s]
180     N_Req in s'.req iff N_Req in s.req
181   }
182
183
184   // Predicates for S_Req
185   pred S_Req_True[s:State]{
186     S_Sense in s.sensors
187   }
188   pred S_Req_False[s:State]{
189     S_Go_False[s]
190   }
191
192   pred pre_S_Req[s:State]{
193     S_Req_True[s]
194   }
195   pred pre_S_Not_Req[s:State]{
196     !S_Req_True[s] and S_Req_False[s]
197   }
198   pred pre_S_Req_Unchanged[s:State]{
199     !S_Req_True[s] and !S_Req_False[s]
200   }
201
202   pred S_Req_[s,s':State]{
203     pre_S_Req[s]
204     S_Req in s'.req
205   }
206   pred S_Not_Req[s,s':State]{
207     pre_S_Not_Req[s]
208     S_Req !in s'.req
209   }
```

```
210  pred S_Req_Unchanged[s,s':State]{
211    pre_S_Req_Unchanged[s]
212    S_Req in s'.req iff S_Req in s.req
213  }
214
215  // Predicates for E_Req
216  pred E_Req_True[s:State]{
217    E_Sense in s.sensors
218  }
219  pred E_Req_False[s:State]{
220    E_Go_False[s]
221  }
222
223  pred pre_E_Req[s:State]{
224    E_Req_True[s]
225  }
226  pred pre_E_Not_Req[s:State]{
227    !E_Req_True[s] and E_Req_False[s]
228  }
229  pred pre_E_Req_Unchanged[s:State]{
230    !E_Req_True[s] and !E_Req_False[s]
231  }
232
233  pred E_Req_[s,s':State]{
234    pre_E_Req[s]
235    E_Req in s'.req
236  }
237  pred E_Not_Req[s,s':State]{
238    pre_E_Not_Req[s]
239    E_Req !in s'.req
240  }
241  pred E_Req_Unchanged[s,s':State]{
242    pre_E_Req_Unchanged[s]
243    E_Req in s'.req iff E_Req in s.req
244  }
245
246  // Predicates for NS_Lock
247  pred NS_Lock_True[s:State]{
248    N_Go_True[s] or S_Go_True[s]
249  }
250  pred NS_Lock_False[s:State]{
251    (N_Go_False [s] and S_Go !in s.goes) or (S_Go_False [s] and N_Go !
         in s.goes)
252  }
253
```

```
254 pred pre_NS_Lock[s:State]{
255   NS_Lock_True[s]
256 }
257 pred pre_NS_Not_Lock[s:State]{
258   !NS_Lock_True[s] and NS_Lock_False[s]
259 }
260 pred pre_NS_Lock_Unchanged[s:State]{
261   !NS_Lock_True[s] and !NS_Lock_False[s]
262 }
263
264 pred NS_Lock_[s,s':State]{
265   pre_NS_Lock[s]
266   s'.NS_Lock = True
267 }
268 pred NS_Not_Lock[s,s':State]{
269   pre_NS_Not_Lock[s]
270   s'.NS_Lock = False
271 }
272 pred NS_Lock_Unchanged[s,s':State]{
273   pre_NS_Lock_Unchanged[s]
274   s'.NS_Lock=s.NS_Lock
275 }
276
277
278 //**********************FAIRNESS CONSTRAINTS
       ************************//
279 // Modeling fairness constraints
280 fun N_fair[]:State{
281   State - (sensors.N_Sense & goes.N_Go)
282 }
283 fun S_fair[]:State{
284   State - (sensors.S_Sense & goes.S_Go)
285 }
286 fun E_fair[]:State{
287   State - (sensors.E_Sense & goes.E_Go)
288 }
289
290 // combines 3 fcs into 1 fc by checking that all 3 fcs occur
       infinitely often thru a counter
291 fact fairness {
292   all s,s':State | s->s' in nextState implies (  (s in N_fair[] and s
         .counter=f0) implies s'.counter=f1 else
293     (s in S_fair[] and s.counter=f1) implies s'.counter=f2 else
294     (s in E_fair[] and s.counter=f2) implies s'.counter=f3 else
295     s.counter=f3 implies s'.counter=f0 else
```

```
296       s'.counter=s.counter)
297     // don't have to keep track of immediate next state fcs when
            counter=f3
298     // because it doesn't matter in the context of infinitely often
299   }
300   pred fair[s:State] {
301     s.counter = f3
302   }
303
304   //**********************MODEL DEFINITION************************//
305   fact modelDefinition{
306     // init state constraints
307     all s:State| initial[s] iff (s in initialState)
308     // transition constraints
309     all s,s':State| s->s' in nextState iff (
310       N_Go_[s,s'] or
311       N_Not_Go[s,s'] or
312       N_Go_Unchanged[s,s'] or
313       S_Go_[s,s'] or
314       S_Not_Go[s,s'] or
315       S_Go_Unchanged[s,s'] or
316       E_Go_[s,s'] or
317       E_Not_Go[s,s'] or
318       E_Go_Unchanged[s,s'] or
319
320       N_Req_[s,s'] or
321       N_Not_Req[s,s'] or
322       N_Req_Unchanged[s,s'] or
323       S_Req_[s,s'] or
324       S_Not_Req[s,s'] or
325       S_Req_Unchanged[s,s'] or
326       E_Req_[s,s'] or
327       E_Not_Req[s,s'] or
328       E_Req_Unchanged[s,s'] or
329
330       NS_Lock_[s,s'] or
331       NS_Not_Lock[s,s'] or
332       NS_Lock_Unchanged[s,s']
333     )
334     // fairness constraints
335     all s:State | s in fc iff fair[s]
336     // equality predicate: states are records
337     all s,s':State| (s.sensors = s'.sensors and s.goes = s'.goes and s.
            req = s'.req and s.NS_Lock = s'.NS_Lock) implies s = s'
338   }
```

```
339
340
341   //****************SIGNIFICANCE AXIOMS*******************//
342   pred reachablityAxiom {
343     all s:State | s in State.(initialState <: *nextState)
344   }
345   pred operationsAxiom {
346     some s,s':State | N_Go_[s,s']
347     some s,s':State | N_Not_Go[s,s']
348     some s,s':State | N_Go_Unchanged[s,s']
349     some s,s':State | S_Go_[s,s']
350     some s,s':State | S_Not_Go[s,s']
351     some s,s':State | S_Go_Unchanged[s,s']
352     some s,s':State | E_Go_[s,s']
353     some s,s':State | E_Not_Go[s,s']
354     some s,s':State | E_Go_Unchanged[s,s']
355
356     some s,s':State | N_Req_[s,s']
357     some s,s':State | N_Not_Req[s,s']
358     some s,s':State | N_Req_Unchanged[s,s']
359     some s,s':State | S_Req_[s,s']
360     some s,s':State | S_Not_Req[s,s']
361     some s,s':State | S_Req_Unchanged[s,s']
362     some s,s':State | E_Req_[s,s']
363     some s,s':State | E_Not_Req[s,s']
364     some s,s':State | E_Req_Unchanged[s,s']
365
366     some s,s':State | NS_Lock_[s,s']
367     some s,s':State | NS_Not_Lock[s,s']
368     some s,s':State | NS_Lock_Unchanged[s,s']
369   }
370   pred significanceAxioms {
371     reachablityAxiom
372     operationsAxiom
373   }
374   --run significanceAxioms for exactly 17 State
375
376   //**********************PROPERTIES***********************//
377   // safety property
378   assert MC{
379     // light in cross directions never on at same time
380     ctlfc_mc[ag[not_[goes.E_Go & goes.(N_Go + S_Go)]]]
381   }
382   check MC for exactly 17 State
```

## A.4 Feature Interaction in a Telephone System

```
1  open ctl[State]
2  open util/boolean
3
4  //**********************STATE SPACE************************//
5  // Feature={CW,CF} is the set of features.
6  abstract sig Feature{}
7  one sig CW,CF extends Feature{}
8
9  // Each phone number can have some features.
10 //If a number has call-forwarding (CF), fw points to forwarded number
     .
11 sig PhoneNumber{
12   feature: set Feature,
13   fw: set PhoneNumber
14 }
15 fact { // facts about types (PhoneNumber)
16   // any PN can only have 0 or 1 PN as its fw number
17   all n:PhoneNumber| lone n.fw
18   // CF is a feature of PN only if the PN has a fw number set
19   all n:PhoneNumber| CF in n.feature iff some n.fw
20   // no number is forwarded to itself thru other numbers
21   no (iden & (^fw))
22 }
23
24 // Used to model the global states.
25 sig State{
26   // Numbers that are idle,
27   idle: set PhoneNumber,
28   // (a->b) in busy iff a wants to talk to b, but b is not idle
29   busy: PhoneNumber -> PhoneNumber,
30   // (a->b) in calling iff a is trying to call b
31   calling: PhoneNumber -> PhoneNumber,
32   // (a->b) in talking iff a is talking to b
33   talkingTo: PhoneNumber -> PhoneNumber,
34   // (a->b) in waitingFor iff a is waiting for b
35   waitingFor: PhoneNumber -> PhoneNumber,
36   // (a->b) in forwardedTo iff a is forwarded to b
37   forwardedTo: PhoneNumber -> PhoneNumber
38 }
39
40 //****************INITIAL STATE CONSTRAINTS******************//
41 pred initial[s:State]{
42   s.idle = PhoneNumber
```

```
43    no s.calling
44    no s.talkingTo
45    no s.busy
46    no s.waitingFor
47    no s.forwardedTo
48  }
49
50  //****************TRANSITION CONSTRAINTS/OPERATIONS
       *******************//
51
52  pred pre_idle_calling[s: State]{
53    some n,n':PhoneNumber | n in s.idle and n != n'
54  }
55  pred post_idle_calling[s,s': State]{
56    some n,n':PhoneNumber |
57    ((s'.idle = ((s.idle) - n)) and
58    (s'.calling = s.calling + (n->n')))
59
60    s'.talkingTo = s.talkingTo
61    s'.busy = s.busy
62    s'.waitingFor = s.waitingFor
63    s'.forwardedTo = s.forwardedTo
64  }
65  pred idle_calling[s,s': State]{
66    pre_idle_calling[s]
67    post_idle_calling[s,s']
68  }
69
70  pred pre_calling_talkingTo[s:State]{
71    some n,n':PhoneNumber |  n->n' in s.calling and  n' in s.idle
72  }
73  pred post_calling_talkingTo[s,s':State]{
74    some n,n':PhoneNumber |
75    (s'.idle = s.idle - n') and
76    (s'.calling = s.calling - (n -> n')) and
77    (s'.talkingTo = s.talkingTo + (n -> n'))
78
79    s'.busy = s.busy
80    s'.waitingFor = s.waitingFor
81    s'.forwardedTo = s.forwardedTo
82  }
83  pred calling_talkingTo[s,s':State]{
84    pre_calling_talkingTo[s]
85    post_calling_talkingTo[s,s']
86  }
```

```
87
88  pred pre_talkingTo_idle[s:State]{
89    some n,n':PhoneNumber | n -> n' in s.talkingTo
90  }
91  pred post_talkingTo_idle[s,s':State]{
92    some n,n':PhoneNumber |
93    (s'.talkingTo = s.talkingTo - (n->n')) and
94    (s'.idle = s.idle + (n + n'))
95
96    s'.busy = s.busy
97    s'.calling = s.calling
98    s'.waitingFor = s.waitingFor
99    s'.forwardedTo = s.forwardedTo
100 }
101 pred talkingTo_idle[s,s':State]{
102   pre_talkingTo_idle[s]
103   post_talkingTo_idle[s,s']
104 }
105
106 pred pre_calling_busy[s:State]{
107   some n,n':PhoneNumber | n->n' in s.calling and n' not in s.idle
108 }
109 pred post_calling_busy[s,s':State]{
110   some n,n':PhoneNumber |
111   (s'.calling = s.calling - (n->n')) and
112   (s'.busy = s.busy + (n->n'))
113
114   s'.idle = s.idle
115   s'.talkingTo = s.talkingTo
116   s'.waitingFor = s.waitingFor
117   s'.forwardedTo = s.forwardedTo
118 }
119 pred calling_busy[s,s':State]{
120   pre_calling_busy[s]
121   post_calling_busy[s,s']
122 }
123
124 pred pre_busy_waitingFor[s:State]{
125   some n,n':PhoneNumber |
126   (n->n') in s.busy and
127   CW in n'.feature and
128   n' not in PhoneNumber.(s.waitingFor)
129   // PN is not already being waited for, i.e.,
130   // can have only one call in CW queue, otherwise stay busy
131 }
```

```
132  pred post_busy_waitingFor[s,s':State]{
133    some n,n':PhoneNumber |
134    (s'.busy = s.busy - (n->n')) and
135    (s'.waitingFor = s.waitingFor + (n->n'))
136
137    s'.forwardedTo = s.forwardedTo
138    s'.idle = s.idle
139    s'.calling = s.calling
140    s'.talkingTo = s.talkingTo
141  }
142  pred busy_waitingFor[s,s':State]{
143    pre_busy_waitingFor[s]
144    post_busy_waitingFor[s,s']
145  }
146
147  // caller on CW hangs up
148  pred pre_waitingFor_idle[s:State]{
149    some n,n':PhoneNumber | n -> n' in s.waitingFor
150  }
151  pred post_waitingFor_idle[s,s':State]{
152    some n,n':PhoneNumber |
153    (s'.waitingFor = s.waitingFor - (n -> n')) and
154    (s'.idle = s.idle + n)
155
156    s'.calling = s.calling
157    s'.talkingTo = s.talkingTo
158    s'.busy = s.busy
159    s'.forwardedTo = s.forwardedTo
160  }
161  pred waitingFor_idle[s,s':State]{
162    pre_waitingFor_idle[s]
163    post_waitingFor_idle[s,s']
164  }
165
166  pred pre_waitingFor_talkingTo[s:State]{
167    some n,n':PhoneNumber | n -> n' in s.waitingFor
168  }
169  pred post_waitingFor_talkingTo[s,s':State]{
170    some n,n':PhoneNumber |
171    (s'.waitingFor = s.waitingFor - (n -> n')) and
172    (s'.talkingTo = s.talkingTo + (n -> n'))
173
174    s'.idle = s.idle
175    s.busy = s'.busy
176    s.forwardedTo = s'.forwardedTo
```

```
177   s.calling = s'.calling
178 }
179 pred waitingFor_talkingTo[s,s':State]{
180   pre_waitingFor_talkingTo[s]
181   post_waitingFor_talkingTo[s,s']
182 }
183
184 pred pre_busy_forwardedTo[s:State]{
185   some n,n':PhoneNumber | n -> n' in s.busy and CF in n'.feature
186 }
187 pred post_busy_forwardedTo[s,s':State]{
188   some n,n':PhoneNumber |
189   (s'.busy = s.busy - (n -> n')) and
190   (s'.forwardedTo = s.forwardedTo + (n -> n'.fw))
191
192   s'.idle = s.idle
193   s'.talkingTo = s.talkingTo
194   s'.calling = s.calling
195   s'.waitingFor = s.waitingFor
196 }
197 pred busy_forwardedTo[s,s':State]{
198   pre_busy_forwardedTo[s]
199   post_busy_forwardedTo[s,s']
200 }
201
202 pred pre_forwardedTo_calling[s:State]{
203   some n,n':PhoneNumber | n -> n' in s.forwardedTo
204 }
205 pred post_forwardedTo_calling[s,s':State]{
206   some n,n':PhoneNumber |
207   (s'.forwardedTo = s.forwardedTo - (n->n')) and
208   (s'.calling = s.calling + (n -> n'))
209
210   s'.idle = s.idle
211   s'.busy = s.busy
212   s'.talkingTo = s.talkingTo
213   s'.waitingFor = s.waitingFor
214 }
215 pred forwardedTo_calling[s,s':State]{
216   pre_forwardedTo_calling[s]
217   post_forwardedTo_calling[s,s']
218 }
219
220 pred pre_busy_idle[s:State]{
221   some n,n':PhoneNumber | n -> n' in s.busy and no n'.feature
```

```
222 }
223 pred post_busy_idle[s,s':State]{
224   some n,n':PhoneNumber |
225   (s'.busy = s.busy - (n -> n')) and
226   (s'.idle = s.idle + n)
227 
228   s.talkingTo = s'.talkingTo
229   s.waitingFor = s'.waitingFor
230   s.forwardedTo = s'.forwardedTo
231   s.calling = s'.calling
232 }
233 pred busy_idle[s,s':State]{
234   pre_busy_idle[s]
235   post_busy_idle[s,s']
236 }
237 
238 //****************MODEL DEFINITION*******************//
239 
240 fact md{
241   // init state constraint
242   all s:State | s in initialState iff initial[s]
243   // transition constraints
244   all s,s': State|
245     s->s' in nextState iff
246       (idle_calling[s,s'] or calling_talkingTo[s,s'] or
          talkingTo_idle[s,s'] or
247       calling_busy[s,s'] or busy_waitingFor[s,s'] or busy_forwardedTo
          [s,s'] or
248       busy_idle[s,s'] or waitingFor_idle[s,s'] or
          waitingFor_talkingTo[s,s'] or
249       forwardedTo_calling[s,s'])
250   // equality predicate: states are records
251   all s,s':State|(
252     ((s.idle = s'.idle) and (s.calling = s'.calling) and
253     (s.talkingTo = s'.talkingTo) and (s.busy = s'.busy) and
254     (s.waitingFor = s'.waitingFor) and (s.forwardedTo = s'.
         forwardedTo)) implies (s =s'))
255 }
256 
257 //****************SIGNIFICANCE AXIOMS******************//
258 pred reachablityAxiom {
259   all s:State | s in State.(initialState <: *nextState)
260 }
261 pred operationsAxiom {
262   some s,s':State | idle_calling[s,s']
```

```
263     some s,s':State | calling_talkingTo[s,s']
264     some s,s':State | talkingTo_idle[s,s']
265     some s,s':State | calling_busy[s,s']
266     some s,s':State | busy_waitingFor[s,s']
267     some s,s':State | busy_forwardedTo[s,s']
268     some s,s':State | busy_idle[s,s']
269     some s,s':State | waitingFor_idle[s,s']
270     some s,s':State | waitingFor_talkingTo[s,s']
271     some s,s':State | forwardedTo_calling[s,s']
272   }
273   pred significanceAxioms {
274     reachablityAxiom
275     operationsAxiom
276   }
277   run significanceAxioms for exactly 6 State, exactly 4 PhoneNumber
278
279   //****************PROPERTIES/CHECK*******************//
280   pred ap_safety [s:State] {
281     // no PN is both being waited for and being forwarded to
282     no s.waitingFor.PhoneNumber & s.forwardedTo.PhoneNumber
283   }
284   assert safety { ctl_mc[ ag [{s:State | ap_safety[s]}]] }
285   check safety for exactly 6 State, exactly 4 PhoneNumber
```

# Appendix B

# Non-Alloy Model for Comparison

## B.1 Musical Chairs in NuSMV

```
1   MODULE main
2
3   DEFINE
4     numPlayers := 3;
5     numChairs := numPlayers - 1;
6
7   -------------------- STATE VARIABLES ------------------------
8
9   VAR
10
11    mode : {start, walking, sitting, end};
12
13    -- bool represents whether player is still in the game.
14    -- 0 represents no players in chairs
15    players : array 1..numPlayers of boolean; -- don't need 0 here
16
17    -- bool represents whether chair is still in the game
18    chairs : array 1..numChairs of boolean;
19
20    -- mapping of chairs to players
21    occupied : array 1..numChairs of 0..numPlayers;
22
23
24  ASSIGN
25
26  --------------------- INIT STATE ----------------------
```

```
27
28    init(mode) := start;
29
30    -- needs to be as many init(players) as numPlayers
31    init(players[1]) := TRUE;
32    init(players[2]) := TRUE;
33    init(players[3]) := TRUE;
34
35    -- needs to be as many init(chairs) as numChairs
36    init(chairs[1]) := TRUE;
37    init(chairs[2]) := TRUE;
38
39  --------------- TRANSITION CONSTRAINTS ------------------------
40
41  TRANS
42    case
43    mode = start & count(players[1],players[2],players[3])>1:
44      next(mode) = walking &
45
46      next(players[1]) = players[1] &
47      next(players[2]) = players[2] &
48      next(players[3]) = players[3] &
49      next(chairs[1]) = chairs[1] &
50      next(chairs[2]) = chairs[2] ;
51
52    mode = walking :
53      next(mode) = sitting &
54
55      -- no one is sitting in walking state
56      occupied[1] = 0 &
57      occupied[2] = 0 &
58
59      next(players[1]) = players[1] &
60      next(players[2]) = players[2] &
61      next(players[3]) = players[3] &
62      next(chairs[1]) = chairs[1] &
63      next(chairs[2]) = chairs[2] ;
64
65    mode = sitting :
66      next(mode) = start &
67
68      (chairs[1] -> (occupied[1]!=0)) &
69      (chairs[2] -> (occupied[2]!=0)) &
70
```

```
71      -- in sitting mode, only chairs in game are occupied by players
           in game
72      -- occupiers of chairs currently in game are players who are
           currently in the game
73      (occupied[1]!=0 -> (players[occupied[1]] <-> chairs[1])) &
74      (occupied[2]!=0 -> (players[occupied[2]] <-> chairs[2])) &
75
76      -- chairs cannot be occupied by the same player except null (0)
           in sitting mode
77      ((occupied[1]!=0 & occupied[2]!=0) -> (occupied[1]!=occupied[2]))
            &
78
79      -- eliminate player if player doesn't occupy any chairs
80      ((occupied[1]!=1 & occupied[2]!=1) ? !next(players[1]) : next(
           players[1])=players[1]) &
81      ((occupied[1]!=2 & occupied[2]!=2) ? !next(players[2]) : next(
           players[2])=players[2]) &
82      ((occupied[1]!=3 & occupied[2]!=3) ? !next(players[3]) : next(
           players[3])=players[3]) &
83
84      -- leave chair outside game if already outside
85      ((!chairs[1]) -> next(chairs[1])=FALSE) &
86      ((!chairs[2]) -> next(chairs[2])=FALSE) &
87      -- eliminate 1 chair: count of number of chairs in current round
           is
88      -- 1 more than count of chairs in next round
89      count(chairs[1],chairs[2]) = next(count(chairs[1],chairs[2])) +
           1;
90
91    mode = start & count(players[1],players[2],players[3])=1:
92      next(mode) = end &
93
94      next(players[1]) = players[1] &
95      next(players[2]) = players[2] &
96      next(players[3]) = players[3] &
97      next(chairs[1]) = chairs[1] &
98      next(chairs[2]) = chairs[2] ;
99
100   TRUE:
101     next(mode) = mode &
102     next(players[1]) = players[1] &
103     next(players[2]) = players[2] &
104     next(players[3]) = players[3] &
105     next(chairs[1]) = chairs[1] &
106     next(chairs[2]) = chairs[2] ;
```

```
107
108     esac &
109
110  ---------------------- SPECS TO CHECK -------------------------------
111
112  SPEC
113    -- chair in game is always occupied in sitting mode
114    AG( (mode=sitting -> chairs[1] -> occupied[1]!=0) &
115      (mode=sitting -> chairs[2] -> occupied[2]!=0) )
116  SPEC
117    -- players in game always > 0
118    AG(count(players[1],players[2],players[3])>0)
119  SPEC
120    -- end mode only has 1 player and 0 chairs
121    AG(mode=end -> (count(players[1],players[2],players[3])=1 & count(
         chairs[1],chairs[2])=0))
122  SPEC
123    -- there can be 1 player only in start or end games
124    AG(count(players[1],players[2],players[3])=1 -> (mode=start | mode=
         end))
125  SPEC
126    -- players = chairs + 1
127    AX(AG(count(players[1],players[2],players[3]) = count(chairs[1],
         chairs[2]) + 1))
```

# Appendix C

# Alloy Models From Other Works for Comparison

## C.1 Traffic Light by Vakili [42]

```
1  module TrafficLightController
2
3  open util/boolean
4  open temporal_logics/ctlfc[State]
5
6  // There are 3 sensors
7  abstract sig Sense{}
8  one sig N_Sense, S_Sense, E_Sense extends Sense{}
9
10 // Go is for modeling which direction is allowed to go
11 abstract sig Go{}
12 one sig N_Go, S_Go, E_Go extends Go{}
13
14 // Request is to latch the traffic sensors input.
15 abstract sig Request{}
16 one sig N_Req, S_Req, E_Req extends Request{}
17
18 sig State{
19   input: set Sense,
20   output: set Go,
21
22   req: set Request,
23   NS_Lock: Bool // NS_Lock is true iff East is not allowed to go
```

```
24  }
25
26  pred initial[s:State]{
27    no s.output
28    no s.req
29    s.NS_Lock = False
30  }
31
32  // setting the initial states
33  fact{ all s:State| initial[s] iff (s in initialState)}
34
35
36  // Predicates for N_Go
37  pred N_Go_True[s:State]{
38    N_Req in s.req
39    N_Go !in s.output
40    E_Req !in s.req
41  }
42
43  pred N_Go_False[s:State]{
44    N_Go in s.output
45    N_Sense !in s.input
46  }
47
48  pred N_Go_[s,s':State]{
49    N_Go_True[s] implies N_Go in s'.output else (N_Go_False[s] implies
        N_Go !in s'.output else (N_Go in s.output iff N_Go in s'.output
        ))
50  }
51
52  // Predicates for S_Go
53  pred S_Go_True[s:State]{
54    S_Req in s.req
55    S_Go !in s.output
56    E_Req !in s.req
57  }
58
59  pred S_Go_False[s:State]{
60    S_Go in s.output
61    S_Sense !in s.input
62  }
63
64  pred S_Go_[s,s':State]{
65    S_Go_True[s] implies S_Go in s'.output else (S_Go_False[s] implies
        S_Go !in s'.output else (S_Go in s.output iff S_Go in s'.output
```

```
        ))
66  }
67
68  // Predicates for E_Go
69  pred E_Go_True[s:State]{
70    E_Req in s.req
71    E_Go !in s.output
72    s.NS_Lock = False
73  }
74
75  pred E_Go_False[s:State]{
76    E_Go in s.output
77    E_Sense !in s.input
78  }
79
80  pred E_Go_[s,s':State]{
81    E_Go_True[s] implies E_Go in s'.output else (E_Go_False[s] implies
        E_Go !in s'.output else (E_Go in s.output iff E_Go in s'.output
        ))
82  }
83
84  // Predicates for N_Req
85  pred N_Req_True[s:State]{
86    N_Sense in s.input
87  }
88
89  pred N_Req_False[s:State]{
90    N_Go_False[s]
91  }
92
93  pred N_Req_[s,s':State]{
94    N_Req_True[s] implies N_Req in s'.req else (N_Req_False[s] implies
        N_Req !in s'.req else  (N_Req in s.req  iff N_Req in s'.req))
95  }
96
97  // Predicates for S_Req
98  pred S_Req_True[s:State]{
99    S_Sense in s.input
100 }
101
102 pred S_Req_False[s:State]{
103   S_Go_False[s]
104 }
105
106 pred S_Req_[s,s':State]{
```

```
107    S_Req_True[s] implies S_Req in s'.req else (S_Req_False[s] implies
           S_Req !in s'.req else  (S_Req in s.req  iff S_Req in s'.req))
108  }
109
110  // Predicates for E_Req
111  pred E_Req_True[s:State]{
112    E_Sense in s.input
113  }
114
115  pred E_Req_False[s:State]{
116    E_Go_False[s]
117  }
118
119  pred E_Req_[s,s':State]{
120    E_Req_True[s] implies E_Req in s'.req else (E_Req_False[s] implies
           E_Req !in s'.req else  (E_Req in s.req  iff E_Req in s'.req))
121  }
122
123  // Predicates for NS_Lock
124  pred NS_Lock_True[s:State]{
125    N_Go_True[s] or S_Go_True[s]
126  }
127
128  pred NS_Lock_False[s:State]{
129    (N_Go_False [s] and S_Go !in s.output) or (S_Go_False [s] and N_Go
           !in s.output)
130  }
131
132  pred NS_Lock_[s,s':State]{
133    NS_Lock_True[s] implies s'.NS_Lock = True else (NS_Lock_False[s]
           implies s'.NS_Lock = False else s.NS_Lock=s'.NS_Lock)
134  }
135
136  fact TransitionRelation{
137  //  all s,s':State| (s.input = s'.input and s.output = s'.output and
         s.req = s'.req and s.NS_Lock = s'.NS_Lock) implies s = s'
138    all s,s':State| s' in nextState[s] iff (N_Go_[s,s'] and S_Go_[s,s']
            and E_Go_[s,s'] and N_Req_[s,s'] and S_Req_[s,s'] and E_Req_[s
           ,s'] and NS_Lock_[s,s'])
139  }
140
141  // Modeling fairness constraints:
142
143  fun N_fair[]:State{
144    State - (input.N_Sense & output.N_Go)
```

```
145 }
146
147 fun S_fair[]:State{
148   State - (input.S_Sense & output.S_Go)
149 }
150
151 fun E_fair[]:State{
152   State - (input.E_Sense & output.E_Go)
153 }
154
155 fact{
156   fc1 = N_fair
157   fc2 = S_fair
158   fc3 = E_fair
159 }
160
161 fun bound[R:State->State,X:State]
162 :State->State{
163   X <: R
164 }
165
166 fun id[X:State]
167 :State->State{
168   bound[iden,X]
169 }
170
171 fun loop[R: State->State]
172 :State{
173   State.(^R & id[State])
174 }
175
176 assert MC{
177   CTLFC_MC[not_ctlfc[ECF[output.E_Go & output.(N_Go + S_Go)]]]
178 }
179 check MC for 7 State
```

## C.2 Span Tree by Macedo *et al.* [29]

```
1 module examples/algorithms/opt_spantree
2
3 open util/ordering[Lvl] as lo
4 open util/ordering[State]
```

```
5   open util/graph[Process] as graph
6
7   sig State {
8     Next : one State
9   }
10  lone sig Loop in State {}
11
12  fact {
13    Next = next + last -> Loop
14  }
15
16  sig Process {
17    adj : set Process,
18    lvl: Lvl lone -> State,
19    parent: Process lone -> State,
20  }
21
22  one sig Root extends Process {}
23
24  sig Lvl {}
25
26  fact processGraph {
27    graph/noSelfLoops[adj]
28    graph/undirected[adj]
29    Process in Root.*adj
30  }
31
32  pred Init[t:State] {
33    no lvl.t
34    no parent.t
35  }
36
37  pred Nop[t,t': State] {
38    lvl.t = lvl.t'
39    parent.t = parent.t'
40  }
41
42  pred MayAct[p : Process, t : State] {
43    no lvl.t[p]
44    (p = Root || some lvl.t[p.adj])
45  }
46
47  pred Act[p : Process, t,t' : State] {
48    no lvl.t[p]
49    (p = Root) => {
```

```
50lvl.t'[p] = lo/first
51no parent.t'[p]
52    } else {
53some adjProc: p.adj {
54    some lvl.t[adjProc]
55    lvl.t'[p] = lo/next[lvl.t[adjProc]]
56    parent.t'[p] = adjProc
57    }
58  }
59    all p1 : Process-p | lvl.t[p1] = lvl.t'[p1] and parent.t[p1] =
        parent.t'[p1]
60  }
61
62  pred Fairness {
63    all t : *Next[ordering/first] | ((some p : Process | MayAct[p,t])
        =>
64      (some t1 : t.*Next, p : Process | Act[p,t1,Next[t1]]))
65  }
66
67  fact Trace {
68    Init[first]
69    all t : *Next[ordering/first] | (some p : Process | Act[p, t, Next[
        t]]) || Nop[t,Next[t]]
70  }
71
72  pred IsSpanTree[t : State] {
73    Process in Root.*~(parent.t)
74    graph/dag[~(parent.t)]
75  }
76
77  pred SuccessfulRun {
78    some t : *Next[ordering/first] | IsSpanTree[t]
79  }
80
81  pred Liveness {
82    some Loop => some t : *Next[ordering/first] | IsSpanTree[t]
83  }
84
85  pred Safety {
86    all t : *Next[ordering/first] | no p : Process | p in p.^(parent.t)
87  }
88
89  assert BadLiveness {
90    Liveness
91  }
```

```
92
93  assert GoodLiveness {
94    Fairness => Liveness
95  }
96
97  assert GoodSafety {
98    Safety
99  }
100
101 // Span (1) scenario
102 check BadLiveness for 3 but 10 State
103 // Span (2) scenario
104 check GoodLiveness for 3 but 10 State
105 // Span (3) scenario
106 check GoodSafety for 3 but 10 State
```