

DEPARTMENT OF COMPUTER SCIENCE
AND
ARTIFICIAL INTELLIGENCE

University of Malta



MODEL CHECKING GAMES

ANDREW CALLEJA

September 2007

*Submitted in partial fulfillment of the requirements
for the degree of B.Sc.I.T. (Hons.)*

Supervisors: Dr. Gordon Pace, Mr. Sandro Spina

Board of Studies for IT

Declaration

Plagiarism is defined as “the unacknowledged use, as one’s own work, of work of another person, whether or not such work has been published” (Regulations Governing Conduct at Examinations, 1997, Regulation 1 (viii), University of Malta).

I, the undersigned, declare that the Final Year Project report submitted is my work, except where acknowledged and referenced.

I understand that the penalties for making a false declaration may include, but are not limited to, loss of marks, cancellation of examination results, enforced suspension of studies, or expulsion from the degree programme.

ANDREW CALLEJA

Student Name

Signature

CSA3400

Course Code

Model Checking Games

Title of work submitted

14rd September 2007

Date

Dedicated to ...

Raymond and Frances

Nazzareno and Carmena

Joseph and Mary Anne

... whom I can never thank enough.

Acknowledgements

First and foremost I would like to thank my tutors Dr. Gordon Pace and Mr. Sandro Spina. Dr. Pace's insights, perseverance and good advice (not to mention his boundless patience) is what kept me going and helped me overcome the hurdles I encountered during the course of this lengthy work. Mr Spina and our (mostly) unscheduled meetings have helped me many times when I stumbled upon hard problems and his recommendations always proved fruitful.

I would also like to thank Mr. Joseph Cordina, for helping me compile the μ cke model checker using an old version of the gcc C/C++ compiler. Without his intervention I would probably have had to forfeit a substantial amount of work I meant to do.

My deepest and heartfelt thanks go to my family, especially my parents, Raymond and Frances, my sister Justine and my grandparents for their support throughout this project and my studies at UoM. This thesis is dedicated to them.

I would also like to thank my course mates and close friends for the good times (and not-so-good times) we had together during the length of this course. They made the experience a memorable one.

Last, but not least, I would like thank Annaliz for being always there for me during the entire length of this project and for keep up with my several ups and downs. Her positive advice and common sense have often gotten me out of sour moods when sometimes things started to looked bleak.

Abstract

Games have been an area of study for computer science and artificial intelligence ever since the first computers were created. They not only provide us with a means of challenging our mental prowess alone and against other players but have also been successfully used as case studies for how computers may be taught to emulate human thought and intelligence. By studying games, and proving properties about them, computer scientists have been able to develop theories and techniques which have been subsequently used for other types of systems.

Another, highly studied field of computer science is system verification. System verification goes beyond the normal validation techniques which prove only that a system is fit for its job. Verification makes sure that the system at hand satisfies its requirements fully by making sure that there are no hidden faults which validation techniques such as testing and simulation may not detect. To do so verification techniques make a thorough and complete search of the system and ensure that is bullet-proof as far as its requirements demand and can hence handle any situation it is required to without breaking down.

These two fields of study have been sometimes combined together so as to prove a number of interesting, and sometimes surprising, properties about games. Since verification is a complete way to check a system, analysing games using its techniques proves properties about the latter which cannot be contested since they follow a rigorous proof-based approach. In our study we are interested in analysing how an automatic and relatively recently developed verification technique called *Model Checking* fares when attempting to verify properties about games as has been done before with other more established verification techniques.

Our study will first focus on model checking itself: its steps, the structures utilised such as Kripke structures and binary decision diagrams, the logics involved such as *CTL* and μ -calculus temporal logics and the algorithms employed for automatic verification. The next part of the study will then focus on how the theory of model checking may be put to use for games themselves. Finally, we will consider two simple but important case studies: Tictactoe and Connect Four and use these to show how model checking can prove properties about these games by verifying them by means of two model checkers: SMV and μ cke. In parallel to this we will see which of the two mentioned temporal logics is more suitable for describing game properties and also obtain an indication of model checking's ability to scale up for games with large board sizes.

CONTENTS

1	Introduction	1
1.1	Overview	1
1.2	Aims and Objectives	2
1.3	Document Structure	2
2	Literature Review	4
2.1	Overview	4
2.2	Introduction	5
2.3	Types of Verification Techniques	6
2.3.1	Comparison of Model Checking with the other Verification Techniques	7
2.4	Advantages	9
2.5	Limitations and the State Space Explosion Problem	10
2.6	Model Checking Process	10
2.7	Types of Model Checking	12
2.7.1	Enumerative Model Checking	12
2.7.2	Symbolic Model Checking	14
2.7.3	Other forms of Model Checking	16
2.8	Conclusion	16

3	Modelling	17
3.1	Overview	17
3.2	Introduction	18
3.3	Some Notation	18
3.4	Modelling using an Enumerative Representation	19
3.5	Modelling using a Symbolic Representation	24
3.6	Conclusion	32
4	Specification	33
4.1	Overview	33
4.2	Introduction	33
4.3	Temporal Logic and Temporal Logic Operators	34
4.4	Linear-time and Branching-time Temporal Logics	35
4.5	Temporal Logic Properties	38
4.6	Temporal Logic Languages	40
4.7	Computation Tree Logic— <i>CTL*</i>	41
4.7.1	Introduction	41
4.7.2	Syntax	41
4.7.3	Semantics	43
4.7.4	Minimal Set of Operators	44
4.7.5	Examples	44
4.8	<i>CTL*</i> Branching-time Temporal Logic— <i>CTL</i>	44
4.8.1	Introduction	44
4.8.2	Syntax	46
4.8.3	Semantics	47
4.8.4	Examples	47
4.9	<i>CTL*</i> Linear-time Temporal Logic— <i>LTL</i>	48
4.9.1	Introduction	48
4.9.2	Syntax	48
4.9.3	Semantics	48
4.9.4	Example	48

4.10	μ -Calculus	49
4.10.1	Introduction	49
4.10.2	Fixpoint Representation	50
4.10.3	Syntax	53
4.10.4	Semantics	53
4.10.5	Fixpoint characterisation of <i>CTL</i> Formulas	54
4.11	Other Temporal Logics	55
4.12	Conclusion	56
5	Verification	57
5.1	Overview	57
5.2	Introduction	57
5.3	Enumerative Model Checking	58
5.4	Symbolic Model Checking	58
5.5	Counterexamples and Witnesses	59
5.6	Algorithms for Model Checking	59
5.7	Model Checking Tools	59
5.7.1	Introduction	59
5.7.2	SPIN	60
5.7.3	Mocha	60
5.7.4	SMV	61
5.7.5	μ cke	64
5.8	Conclusion	66
6	Games	68
6.1	Overview	68
6.2	Introduction	68
6.3	Modelling Game Systems	69
6.4	Specifying Game Systems	71
6.5	A very simple game: Tictac	73
6.5.1	Modelling Tictac	74

6.5.2	Specifying Tictac	76
6.5.3	Verifying Tictac	78
6.6	Conclusion	79
7	Case Study 1: Tictactoe	80
7.1	Overview	80
7.2	Game Rules	80
7.3	Modelling Tictactoe	82
7.3.1	Game State	82
7.3.2	Game Model	84
7.3.3	End-Game Definitions	86
7.3.4	Game Computation Tree	87
7.3.5	Game Strategy	88
7.3.6	SMV Model	88
7.3.7	μ cke Model	89
7.4	Specifying Tictactoe	89
7.4.1	CTL Properties	90
7.4.2	μ -calculus Properties	95
7.5	Verifying Tictactoe	101
7.5.1	Results	101
7.6	Conclusion	103
8	Case Study 2: Connect Four	104
8.1	Overview	104
8.2	Game Rules	104
8.2.1	Introduction	104
8.3	Modelling Connect Four	106
8.3.1	Game State	107
8.3.2	Game Model	109
8.3.3	End-Game Definitions	111
8.3.4	Game Computation Tree	112

8.3.5	Game Strategy	112
8.4	Specifying Connect Four	115
8.4.1	CTL Properties	115
8.4.2	μ -calculus Properties	118
8.5	Verifying Connect Four	123
8.6	Results	123
8.7	Conclusion	125
9	Evaluation & Related Work	126
9.1	Overview	126
9.2	Evaluation of Results	126
9.3	Related Work	131
9.4	Conclusion	134
10	Future Work & Conclusion	135
10.1	Overview	135
10.2	Future Work	135
10.3	Conclusion	136
A	Game Model Generator Tool	138
A.1	Overview	138
A.2	Aim	138
A.3	Options	139
A.4	User Manual	142
A.4.1	Generating a new Game Model	142
A.5	Contents of CD-ROM	143

LIST OF FIGURES

2.1	Model Checking Process	12
2.2	Example of a Kripke Structure modelling a system	13
2.3	Example of OBDDs modelling a system	15
3.1	An Enumerative Model (Kripke Structure) of a simple system	21
3.2	Binary Decision Tree of the example formula $(a \vee b) \wedge (c \rightarrow d)$	26
3.3	Distinct d node sub-trees	26
3.4	OBDD Reduction Techniques	28
3.5	Binary Decision Diagram of the example formula $(a \vee b) \wedge (c \rightarrow d)$	28
3.6	OBDD Variable Ordering	29
3.7	A Symbolic Model (OBDD) of a simple system	32
4.1	Finite State Transition Graph and Computation Tree	42
4.2	Sequence of approximations for $\mathbf{EF}p$	52
6.1	Board for Tictac	73
6.2	A complete game of Tictac. Circle Player wins horizontally.	74
6.3	Computation Tree for Player 1 in Tictac	76
6.4	Tictac First Moves	76
6.5	Minimised Kripke Structure for Tictac	77

7.1	A standard 3×3 tictactoe board	81
7.2	Example moves in a 3×3 tictactoe board	81
7.3	Example winning boards in a 3×3 tictactoe board	81
7.4	Example draw boards in a 3×3 tictactoe board	82
7.5	The positioning of <i>locs</i> variables in a 3×3 tictactoe board	83
7.6	Example tictactoe board state represented by a state formula	83
7.7	Some illegal board configurations in a 3×3 tictactoe board	85
7.8	Initial part of computation tree of a 3×3 tictactoe board	87
7.9	Symmetrical view of a 3×3 tictactoe board	89
8.1	A standard 7×6 connect four board	105
8.2	Example moves in a 7×6 connect four board	105
8.3	Example winning boards in a 7×6 connect four board	106
8.4	Example draw boards in a 7×6 connect four board	106
8.5	The positioning of <i>locs</i> variables in a 4×4 connect four board	108
8.6	Example connect four board state represented by a state formula	108
8.7	Some illegal board configurations in a 4×4 connect four board	109
8.8	Initial part of the computation tree of a 4×4 connect four board where Player 1 (black) makes the first move.	113
8.9	Grouping of the board locations of a 4×4 connect four board.	114
8.10	Possible values of dual locations and their codes.	114
8.11	Board configuration used to verify Property 9 on a 4×4 connect four	117
A.1	Game Model Generator Tool Main Window	139
A.2	Game Model Generator Information Panel History Tab Window	141
A.3	Game Model Generator Information Panel Errors Tab Window	142

LIST OF TABLES

3.1	Truth Table of Boolean function $(a \vee b) \wedge (c \rightarrow d)$	25
4.1	Summary of Temporal Logic Operators	36
7.1	Tictactoe Verification Results for the <i>CTL</i> properties verified using SMV	102
7.2	Tictactoe Verification Times for the <i>CTL</i> properties verified using SMV	102
7.3	Tictactoe Counterexample Trace Results for the <i>CTL</i> properties verified using SMV .	102
7.4	Tictactoe State Variable Counts for the <i>CTL</i> properties verified using SMV	102
7.5	Tictactoe Verification Results for the μ -calculus properties verified using μ cke	103
7.6	Tictactoe Verification Times for the μ -calculus properties verified using μ cke	103
8.1	Connect Four Verification Results for the <i>CTL</i> properties verified using SMV	124
8.2	Connect Four Verification Times for the <i>CTL</i> properties verified using SMV	124
8.3	Connect Four Counterexample Trace Results for the <i>CTL</i> properties verified using SMV	124
8.4	Connect Four State Variable Counts for the <i>CTL</i> properties verified using SMV . . .	124
8.5	Connect Four Verification Results for the μ -calculus properties verified using μ cke .	125
8.6	Connect Four Verification Times for the μ -calculus properties verified using μ cke . .	125

INTRODUCTION

A journey of a thousand miles begins with a single step.

Lao-tzu

1.1 Overview

Games have been an area of study for computer science and artificial intelligence ever since the first computers were created. They not only provide us with a means of challenging our mental prowess alone and against other players but have also been successfully used as case studies for how computers may be taught to emulate human thought and intelligence. By studying games, and proving properties about them, computer scientists have been able to develop theories and techniques which have been subsequently used for other types of systems.

Another, highly studied field of computer science is system verification. System verification goes beyond the normal validation techniques which prove only that a system is fit for its job. Verification makes sure that the system at hand satisfies its requirements fully by making sure that there are no hidden faults which validation techniques such as testing and simulation may not detect. To do so verification techniques make a thorough and complete search of the system and ensure that is bullet-proof as far as its requirements demand and can hence handle any situation it is required to without breaking down.

These two fields of study have been sometimes combined together so as to prove a number of interesting, and sometimes surprising, properties about games. Since verification is a complete

way to check a system, analysing games using its techniques proves properties about the latter which cannot be contested since they follow a rigorous proof-based approach. In our study we are interested in analysing how an automatic and relatively recently developed verification technique called Model Checking fares when attempting to verify properties about games as has been done before with other more established verification techniques.

1.2 Aims and Objectives

As just mentioned, our main aim is to use model checking with games and hence utilise this verification technique to see if we can prove properties about them. To do so we will first see how the model checking steps may be applied to games. This involves three stages each of which demanding the application of various techniques to game systems, as we shall see in the next chapter. An integral part of the above aim is the use of what are known as temporal logics. Since these logics come in different flavours we have selected two likely candidates called *CTL* and μ -calculus which we will make use of. We wish to compare and contrast these two logics and see which of these is more suited for model checking games.

After having seen how model checking may be adapted for games, we will make use of two games as case studies: Tictactoe and Connect Four. These simple games should allow us to have an insight on how model checking works with games without us being lost into the games' intricacies. As a final aim we will also create a model generator which generates game models for increasing board sizes of these two games. The code generated will allow us to model check tictactoe and connect four with two model checking tools called SMV and μ cke. Moreover by model checking these games for different board sizes we will get an indication of model checking's ability to scale up to games with larger boards.

1.3 Document Structure

This document is divided into ten chapters and one appendix. Chapter 2 gives an overview on model checking and reviews the literature which served as a source to our knowledge on model checking. Chapters 3, 4 and 5 are theoretical and consist of the background theory required to

understand how model checking works in detail by going through each of its steps and discussing any structures, logics, techniques and tools used. Chapter 6 explains how game systems may be verified using the techniques mentioned in the earlier chapters. We will see how a game may be viewed as a system which requires verification and how this view allows us to write and prove properties about games. Chapters 7 and 8 are case studies for the aforementioned tictactoe and connect four games respectively. Here we show all the steps required to model check these two games and what properties were used to verify them. We also present the verification results we obtained for different board sizes of these two games in their respective chapters. We also propose a more compact encoding for connect four which allows larger board to be verified. In Chapter 9 we discuss and evaluate the use of model checking for games in the light of the results we obtained by modelling tictactoe and connect four. Furthermore we relate our work on games by discussing various other techniques which have been used to verify properties about games. Chapter 10 discusses some future research prospects which may emerge from our work and gives our final conclusions. Finally, Appendix A is the user manual for the model generating tool used to obtain different board-sized models of the two games.

LITERATURE REVIEW

He who controls the present, controls the past.

He who controls the past, controls the future.

George Orwell

2.1 Overview

Model Checking is a type of formal automatic system verification technique whereby a system is considered to be correct if it satisfies a set of requirements. To achieve this, a formal *model* based on the system is constructed and it is checked to see whether it meets the required specification criteria to make it correct, subsequently proving or disproving the system itself. The *checking* part, where the specification criteria are checked on the model, is done by a number of algorithms which automatically check the requirements on the model of the system to see if they are satisfied.

In this chapter we present the main principles behind model checking and also to provide a brief description of the model checking process itself and its three stages: *modelling*, *specification* and *verification*. We will discuss what each step entails in the overall process, introduce two types of model checking, known as *enumerative model checking* and *symbolic model checking* and explain the relationship between them.

2.2 Introduction

Computer hardware and software form an integral part of our daily lives. They are put to countless uses: networking, air traffic control systems, medical equipment, e-commerce, military equipment and so on. Most of these systems require that both the hardware and software components of the system are reliable and that if possible they never fail to produce the required and expected behaviour they were developed to carry out. Moreover, reliability becomes immensely critical when human lives are involved. Unfortunately, we have witnessed many cases where either a software or a hardware failure has led to a substantial financial loss or even loss of lives.

Problems in both the hardware and software of the Therac-25 Radiation Therapy Machine has resulted in six cases of radiation overdose and subsequently five patients dying. The electron-beam had no hardware interlocks to stop it from operating in high-energy mode without targets in place. Also, a race condition occurred due to synchronisation issues between the equipment control task and the operator interface task. [35]

A technical report [42] by the United States General Accounting Office discussing one of the MIM-104 Patriot Missile's first deployments in February 1991 at Dhahran, Saudi Arabia states that the system had a software error in its clock which caused it to fail to stop a Scud from hitting a barracks. The system's radar sub-system had correctly detected the Scud but due to the clock error made an erroneous prediction on the next location of the missile. Since it did not find the missile as it expected it no longer tried to intercept it. The end-result of this bug was that 28 soldiers were killed.

The Ariane 5 Rocket has had several of problems which resulted in the loss of specialised equipment worth millions. According to [28], Ariane 5 self-destructed during its test flight about 40 seconds after launch due to a problem in the control software which consisted of a simple mistake in conversion from a 64-bit floating point to 16-bit unsigned integer value. Neither the main computer, nor the backup computer contained code to protect this conversion and this has led to incorrect data reaching the on-board computer causing the rocket to fail. Gérard Le Lann in [34] states that the fault was, amongst others, due to the failure of the "*capture of the overall Ariane 5 application/environment requirements*" [p.339].

It is discouraging that despite the amount of checks and testing the systems had been subjected

to some errors still went undetected. Its even more alarming that the errors in the systems were discovered under such unfortunate circumstances.

2.3 Types of Verification Techniques

McMillan [37] and Clarke et al. [20] discuss the different kind of verification techniques which exist and show why the more traditional approaches of *simulation* and *testing* used to detect and remove faults from systems are not exhaustive enough to ensure a bug-free system.

Simulation makes use of an abstraction of the system by creating a model which represents it. A set of inputs is provided to the model and the latter generates a corresponding set of outputs. The output of the model is evaluated with anticipated results of the expected system. This method is often quite efficient in detecting problems in the system itself. However, as McMillan points out in [37] the system engineer can seldomly create a complete input set which is able to detect all the possible errors in the system. Another problem met is that since simulation is performed on the design of a system, rather than the actual system, it is hard to compare a model's reaction to the inputs with that of the actual system since the latter has often not been implemented at this stage.

Testing, on the other hand, is undertaken on the actual implemented system. The system is provided with a set of inputs and is evaluated according to whether the output obtained is in fact the expected one. Again here the problem of providing a complete set of inputs arises. Moreover, an error detected at this stage is very expensive to adjust.

Thus a more complete approach is required which detects all the potential faults in a system as early as possible. To this end formal verification was introduced. Compared to simulation and testing, formal verification makes sure that all possible computation paths are considered thus fully validating that the system is completely reliable according to its specifications. It also ensures that the system is verified prior implementation. Formal verification is of two main types: *deductive verification* and *model checking*.

Deductive verification makes use of axioms and proof rules in order to prove the correctness of a system. This is usually done either manually by a verification expert or by means of a specialised software tool such as a theorem prover (which still however requires human intervention). If a system is proven by means of deductive verification its reliability can be guaranteed to be com-

plete. Despite this however, both manual and automated methods have some drawbacks. Manual proof construction of a system is expensive, tedious and requires a considerable amount of time. Also, in [19], Clark et al. tell us that automated theorem provers fail “*due to the inherent complexity of testing validity for even the simplest logics*” [p.245].

Model checking, as its name implies, makes use of a *model* of the system which represents the system’s reachable states. This can be achieved through an enumerative representation such as by listing all the states and the transitions from one to the other directly, or by a representation which encodes them symbolically. This model is then automatically *checked* by the use of a set of algorithms which search the states it contains to see if they satisfy a specification requirement of the system. As in the case of deductive verification, model checking performs an exhaustive search of the system’s possible behaviour thus ensuring that once a system has been verified by model checking it is reliable and meets its requirements.

2.3.1 Comparison of Model Checking with the other Verification Techniques

The use of model checking for the purpose of verification benefits the user with various advantages over other verification techniques mentioned earlier.

Testing and Model Checking

In [20] Amir Pnueli discusses how testing (and simulation) are both very effective when a design is full of bugs but as less and less bugs are present this effectiveness decreases considerably. So much so that is often impossible to make sure that a system is completely bug-free through these methods. Hence the most important advantage of model checking over testing is its exhaustive nature. Testing attempts to do what model checking does, i.e. cover all computation paths to check that they are consistent with the system’s requirements. Although it is technically possible for testing to achieve this it requires a lot of insight from the testing party’s side and a complete critical analysis of the system. However, it is easy to overlook some computation paths or to be biased in giving certain paths higher priority over others. Moreover, testing becomes increasingly unattainable as the system grows since its number of possible states grows at high rate with every state variable introduced. [20]

Testing may be at least partially automated by providing a test environment which checks the inputs provided with the expected outputs. However real automation is achieved by providing an environment which attempts all the possible paths as indicated earlier. This is usually not possible with testing. Model checking was created to perform this automatically by means of appropriate algorithms. Model checking is thus fully automatic compared to testing. [20]

Testing allows easier debugging of faults in a system by providing an output trace. This output trace aids testers to find where the problem has occurred. Model checking acts in a similar fashion by providing a trace of the system which shows how the system's states changed before reaching the faulty state. This trace allows an easier way to pinpoint where the fault has occurred and what is required to mitigate it.

A final advantage of model checking over testing is that testing can be done only after the system has been implemented, or partially implemented. Model checking can be done at the design stage of a system's creation life-cycle, i.e. before implementation has even started. Faults discovered during design are much less expensive to fix than those found later on. If the system has already been implemented and the mistake found is large enough to require a re-design of the system itself a lot of time, effort and money would have been spent in vain. Model checking prevents this from occurring by acting in the design stage itself, where a re-design might be expensive, but at least less so.

Simulation and Model Checking

Model checking offers more or less the same advantages over simulation as it does over testing with the only major difference being in the fact that like model checking, simulation occurs during the design stage, pre-implementation. Thus like testing, simulation suffers from the lack of adequate coverage of all computation paths and may not usually be fully automated by algorithms. However both simulation and model checking make use of a model of the system before it is implemented thus allowing errors to be detected much earlier on than in testing. Unfortunately, translation of a design of system into a model might incur errors in the model which are not actually found in the design. Care must be taken therefore in both these techniques to ensure that the model is an exact representation of the system. This is somewhat attainable if the system engineer has complete understanding of the system being modelled.

Deductive Verification and Model Checking

Deductive verification has an advantage over modelling and simulation because like model checking it provides complete coverage of the computation paths of the system. As stated earlier, this means that once a system is verified through deductive verification it is completely in-line with its requirements. [20]

Deductive verification suffers a lot when one tries to automate it [13, 19]. Attempts at automation still require quite a considerable amount of human interaction and insight, making this process unfeasible to mechanise. On the other hand, in model checking human interaction is only required at the very end to check if the verification is either complete or changes need to be made to the design and model due to any discovered faults. Thus in contrast to model checking's full possibility of automation, deductive verification can only be partially automated.

The ability required to verify a system using deductive verification is very high and usually is at the level of an expert in the field of logic or mathematics [19, 20]. This is of course, apart from the knowledge of the system itself being verified. Model checking does require some knowledge to model a system according to a model verification tool's syntax and its specification language however this is usually quite straight forward to learn and use.

One final advantage of model checking over deductive verification is that the latter, like testing and simulation provides no means of an error trace. Finding an error in the design depends on the software engineer's knowledge of the system, which once again must be very comprehensive.

2.4 Advantages

We will now summarise the advantages of model checking as mentioned in the earlier sections. Model checking's main advantage is its exhaustive nature which allows it to be a complete form of system verification. Apart from this it is fully automatic as it can be implemented by means of various algorithms which perform the verification part without the need of user interaction. These algorithms can also be complemented with ones which produce error traces when mistakes are found in the system at hand. Training in the use of model checking is minimal as it requires little specialised knowledge. Since verification is performed on a model and not the actual imple-

mented system, it can provide a means of ensuring the soundness of a system before it is actually implemented. This can sometimes significantly reduce costs as it removes bugs earlier on.

Model checking, like the other verification techniques, is often used to check an entire system at one go but it can be applied as well to modules of such system separately. Apart from checking part of the model it is also possible to use a partial specification if required. Finally, any specification can be written with the limit imposed only by the kind of temporal logic which the model checker to be used accepts. [20]

2.5 Limitations and the State Space Explosion Problem

The disadvantages of model checking are in fact its few limitations. The main limitation which nearly made model checking non-practical for industrial use was the *State Space Explosion* Problem. The problem arises from the reason for which model checking was introduced in the first place: to perform an exhaustive search of all possible system behaviour. The more the number of state variables required to represent a state, the more the number of states required to represent the system (the state space) grows. The growth is in fact exponential in size since all possible combinations of the state variables must be considered. Fortunately, as mentioned earlier, a considerable amount of research was undertaken in this area and the number of states a system may contain has grown considerably by the use of techniques such as symbolic representation, partial ordering, abstraction, symmetry and induction. Refer to [20] for an indication of where to find more on these subjects.

Another minor limitation is that model checking can be used only on systems which have a finite number of states. This however still makes it a worthwhile verification technique as many systems are in fact finite-state. [20]

2.6 Model Checking Process

As previously stated, model checking consists of three main stages. These are: modelling, specification and verification. Clark et al. [20] give us a brief summary of each stage:

Modelling — In this stage we create a design of a system we would like to verify. Once the design is completed it is converted into a formal model which can be used by a model checking tool, called a model checker, to verify the design of the system. Thus usually modelling a system refers to the process of creating a representation of the states and transitions between states of the system in the language which the model checking tool accepts. Modelling should be done with care as it introduces mistakes if for example the design and model don't concur, i.e. a mistake in the model should first be verified with the design to see if the mistake was in fact in the model and not in the design. If the mistake is in fact in the design, the design and subsequently the model must be updated before continuing with the checking process. An example of a model is a model of a 3x3 tictactoe game or a model of a 7x4 connect four game.

Specification — Here a specification is created which contains a number of properties. These properties are the ones which the design must be able to satisfy in order for the system to be correct. Usually this involves the use of a logical formalism such as temporal logic. A temporal logic formula consists of a logic formula with added operators used for quantifying over paths and time. Temporal logics exist in many different flavours and not all of them can specify the same kind of properties about the system. The kind employed to specify a system depends on the system itself. Some systems are better specified with some logics than others because of the respective operators allowed and their semantics and syntax. Also the choice of model checking tool employed in the verification stage depends on the latter's support for the temporal logic chosen. For example, the original SMV Model Checker by McMillan can support a temporal logic called *CTL* [37] while Mucke by Biere can support μ -calculus [7, 8]. An example of a specification could be as simple as whether a game of tictactoe always ends with a player winning or a draw, or more complex such as whether a tictactoe player can always win in 3 moves no matter what moves his opponent makes.

Verification — This step of the process combines the model and the specifications and, as mentioned earlier, verifies them using an appropriate model checking tool called a model checker. If the design is correct, this stage ends the model checking process. Otherwise, if the tool reports that the specifications are not fully observed, it produces an error trace in the form of either a counterexample (negative trace) or a witness (positive trace). This trace can be used to amend the design and reapply the Model Checking Process until the design is verified. Many different model checkers exist and as mentioned earlier they support different kinds of temporal logics.

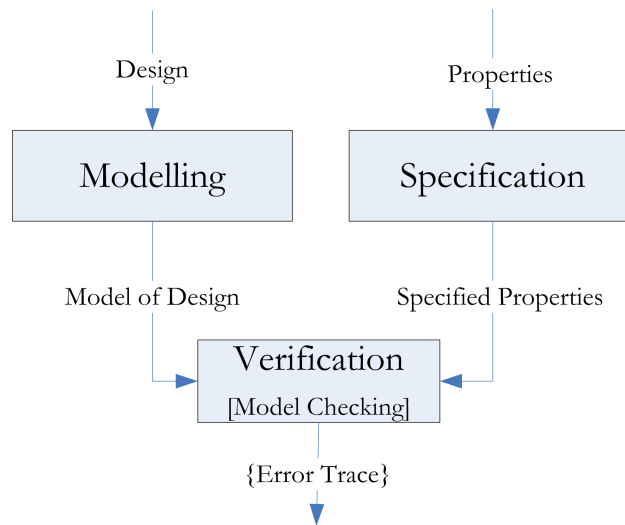


Figure 2.1: Model Checking Process

Figure 2.1 summarises the above stages and their interactions.

2.7 Types of Model Checking

Model Checking exists in two main forms: enumerative and symbolic. Symbolic model checking emerged from enumerative model checking due to the need to mitigate the state space explosion problem mentioned in Section 2.5.

2.7.1 Enumerative Model Checking

Enumerative model checking was first developed independently by two separate teams consisting of Clarke and Emerson [21], and Queille and Sifakis [40]. It is called enumerative because in this form of model checking, each state and transition of the state space is included in the model explicitly. To achieve this we make use of a finite-state graph called a Kripke structure. In these graphs, each node of the graph is a state and each transition between two nodes in the graph represents a transition between two states of the system. As an example of a Kripke structure representing a system see Figure 2.2. Here we have a simple system consisting of four states with four possible transitions. The initial state of the system is shown by the state with the incoming arrow which does not originate from another state. Not also how each state is labelled with the

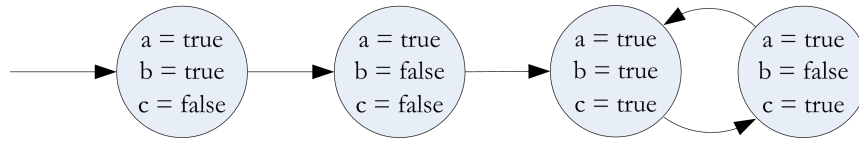


Figure 2.2: Example of a Kripke Structure modelling a system

values of the variables which are true or false in the respective state. For three variables there are $2^3 = 8$ possible states in the state space but only four are considered correct in our system.

Specification in enumerative model checking may be done in any formalism which allows us to reason about how systems change with time. Temporal logic is a form of logic which allows this without introducing time explicitly in the notation [20]. Formulas in temporal logic consist of the usual Boolean operators used in propositional logic such as \wedge , \vee , \neg , \Rightarrow and \Leftrightarrow . However in order to reason about how the states of a model representing the system changes with time other operators must be introduced. Such operators would allow us to specify things about the system such as whether a particular state will be eventually or finally reached or that an error state in the state space is never reached at all [20]. There are two kinds of these operators in temporal logic: path quantifiers and temporal operators [20]. Path quantifiers, as their name implies, quantify over paths (sequences of states). Clarke et al. tell us in [20] that they are usually of two forms:

- *Existential* path quantifier, (**E**) — signifies that a kind of path does exist
- *Universal* path quantifier, (**A**) — signifies that the formula at hand is true for all the paths starting at a particular state.

Temporal operators describe the paths themselves. There are various such operators depending on the temporal logic employed. These include:

- *Next* temporal operator, (**X**) — a property holds in the state after the current one on the path.
- *Finally* temporal operator, (**F**) — also known as *eventually* and *in the future*, this operator means that a property will hold on some state on the path.
- *Globally* temporal operator, (**G**) — also known as *always*, this operator asserts that a property holds on all the states on the path.

Other operators exist depending on the logic employed. Using the above operators we can write formulas such as **AF** p or **EG** $p \vee q$ which mean that for all paths, eventually p is true and there

exists a path where p or q are always true, respectively. Example of temporal logics are *CTL*, *LTL* and *CTL** [20].

Another important aspect of enumerative model checking is how verification is achieved. As stated earlier, a formula in temporal logic may be adequately used to represent a specification property which a model should satisfy. To verify this property on the model using enumerative model checking, we usually make use of an iterative approach. In this approach we break down the temporal logic formula we require into simpler, nested formulas, which we model check one at a time, working progressively outwards. Model checking a formula involves finding which states of the Kripke structure satisfy it. Once the outer formula is reached we have checked the actual formula we had set out for in the first place [20]. Consider again our former example $\mathbf{EG} \ p \vee q$. Verification of this formula may be achieved by first considering the states satisfying p , then the states satisfying q , then the states satisfying their disjunction and so on, until $\mathbf{EG} \ p \vee q$ itself is verified.

2.7.2 Symbolic Model Checking

Initially only models with at most 10^4 to 10^8 states could be checked for errors [19] due to the state explosion problem. Various improvements however have been suggested to upgrade this limiting factor. One of the first and most successful was symbolic model checking.

Symbolic model checking differs from enumerative model checking since in the former we don't represent the model of the system explicitly like in the latter. Instead we use Boolean formulas to represent and manipulate the states and transitions. The main idea behind this approach is that a single Boolean formula has the ability to refer to a set of states at a time and can thus provide a more compact representation than listing all the states and transitions themselves as done previously. However, in order to achieve an efficient use of symbolic model checking using Boolean formulas we require an efficient representation of such formulas. One such efficient representation is Bryant's Ordered Binary Decisions Diagrams (OBDDs) [4, 11] which were first suggested and used by McMillan in his study on symbolic model checking and his implementation of the SMV Model Checker [37]. As an example of how a system may be modelled using OBDDs see Figure 2.3. In this figure we represent the same system shown in Figure 3.1. The OBDD on the left-hand side represents the initial state while that on the right-hand side represents the transitions for the

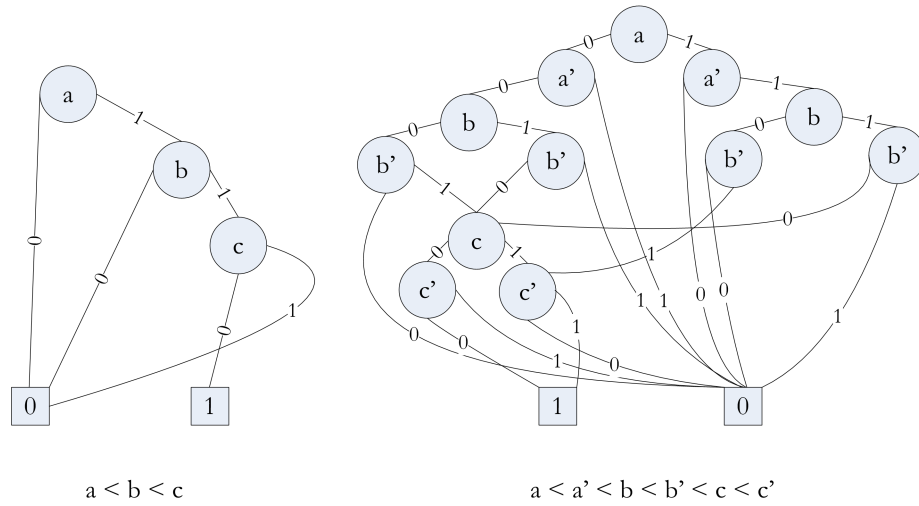


Figure 2.3: Example of OBDDs modelling a system

system. For the trivial system we have chosen a Kripke structure suffices. However as systems grow, while Kripke structure blow up in size, OBDDs tend to be less expansive in their growth. In its worst case scenario symbolic model checking provides no advantage over enumerative state model checking. In fact, in order to achieve a better representation certain measures must be taken to choose an adequate *ordering* of the variables of the binary decision diagram. The ordering used in our graphs is shown beneath the two diagrams.

Specification of properties in symbolic model checking can be done in the same temporal logics as in enumerative model checking. However more often then not, these are expressed using fixpoints by means of appropriate operators such as those available with the temporal logic called μ -calculus [5, 9, 10]. Hence, if logics such as μ -calculus are not employed it is often possible to represent the other languages such as *CTL* and *CTL** by translating the temporal and path operators into fixpoint representations [20, 21, 27]. As an example consider the μ -calculus fixpoint representation of the *CTL* formula $\mathbf{AF} p$ we explained earlier, which may be written as: $\mu Z. p \vee \mathbf{AX} Z$.

The main reason for the above stems from the fact that symbolic model checking differs from enumerative model checking also in how the verification part is achieved. Fixpoints allow us to use a temporal logic formula to recursively define the set of states required to satisfy specification we require which may be represented using OBDDs. This translates into a different verification algorithm then the one used for enumerative model checking [20]. As stated in [13] by Burch et al. using OBDD-based symbolic model checking has made it possible for systems with more than

10^{20} states to be verified using symbolic model checking.

2.7.3 Other forms of Model Checking

Clarke et al. state in [20] that further forms of model checking exist. These make use of techniques such as partial ordering, abstraction, symmetry and induction. (We refer the reader to [20] for an indication where to find more material on these techniques.) Such techniques have greatly extended the size of systems which can be verified by model checking. For example in [18, 22, 23] Clarke, Grumberg and Long claim that the abstraction technique has allowed them to verify a system with over 10^{1300} states.

2.8 Conclusion

In this chapter we have introduced and discussed model checking as a system verification tool. As we have seen model checking was born from the necessity to verify systems in a complete manner automatically without the need of writing formal proofs. Verification of this manner compares adequately with other forms of more common techniques used currently and provides its user with many advantages. As a technique it consists of three stages which complement one another. We have also discussed how model checking's initial explicit representation limited the size of the systems which could be verified due to the state explosion problem. However this problem was soon overcome by the use of many techniques, of most note to us being the symbolic representation employed in symbolic model checking. In the following chapters we will explain into more detail the modelling, specification and verification stages which we have introduced here by examining amongst others: the structures, the techniques, the logics and the algorithms required.

MODELLING

*We fortify in paper and in figures,
Using the names of men instead of men,
Like one that draws the model of an house
Beyond his power to build*

William Shakespeare

3.1 Overview

In this section we explore in detail the first stage of model checking which concerns itself with the modelling of systems. As its name implies, this stage's main aim is the conversion of a system into a model which can be then be later on verified to meet a particular specification. In enumerative state model checking such a model can be created by the use of a kind of graph known as a Kripke structure. While a Kripke structure achieves the required aim of modelling a system it is often inadequate for the modelling of larger systems due to what is known as the state space explosion problem. This arises from the fact that the Kripke structure represents each state of the system explicitly and as the system's state space grows the graph itself must grow to accommodate it. Needless to say the structure soon becomes unmanageable and heavy on system resources. Hence, a more compact, implicit representation is required. Ordered Binary Decision Diagrams (OBDDs) are employed in symbolic model checking to try to achieved this effect and thus allow us to model much larger systems than attainable before.

3.2 Introduction

The modelling of a system is an important aspect of model checking. A model of a system must represent the system itself whilst taking care to ensure that the correct level of abstraction is used as sometimes it is unnecessary to model the entire system's details. In model checking we are only interested in modelling the system's aspects which allow us to ensure that it is correct. Moreover, introducing unnecessary details only makes the model unnecessarily larger and harder to understand and reason about.

A model of a system can be achieved by modelling its components, i.e. its *states* and the *transitions* from one allowed state to the next. Representing all the states of a system and the possible transitions, allows us to model the system in its entirety. A state of a system describes the values of its variables at a specific point in time. A transition, on the other hand, describes how the variables change their value from a point in time to the next, i.e. from one state to the other. Thus a transition can be thought as a pair of states, one containing the variables' values before and one containing the variables' values after a transition has occurred. An infinite sequence of states of the system constitute a *computation* or *path*. Each state in this sequence is arrived to from the previous one by means of a valid transition.

3.3 Some Notation

In this section we will introduce the notation which we will use throughout this body of work. We will use the notation by Clarke et al. in [20].

In order to represent a state we require a set $V = \{v_1, \dots, v_n\}$ where each set member represents a variable of the system. Each of these variables ranges over another finite set D , or domain. Finally we require a function which associates a value in D for every value v in V . A state can thus be a valuation of this function, i.e. of $s : V \rightarrow D$. For example, given $V = \{a, b, c\}$ where a , b and c range over $D = \{2, 4, 6, 8\}$, a state x is can be represented by the valuation $\langle a \leftarrow 4, b \leftarrow 2, c \leftarrow 8 \rangle$.

Since a transition is in fact a pair of states, all we require to represent one is two valuations of the set V . We use V to represent the variables' values in the current state, and V' to represent the variables' values in next state, after the transition has occurred. For example, consider the state

x above. A next state, x' can be $\langle a \leftarrow 4, b \leftarrow 2, c \leftarrow 6 \rangle$. As can be seen the transition has caused variable c to change from 8 to 6 while the other variables have remained the same.

Atomic propositions allow us to reason about the system and later on describe the system's specifications. They constitute a set AP where the elements have the form

$$v = d$$

where $v \in V$ and $d \in D$. Using the function s introduced above, a proposition $v = d$ is true in a state if $s(v) \rightarrow d$. Thus the example state x above can be represented using the atomic propositions $(a = 4)$, $(b = 2)$ and $(c = 8)$. Using first order representation we obtain the formula $(a = 4) \wedge (b = 2) \wedge (c = 8)$. (Note that using first order formulas we can represent a number of states at the same time. The first order formula, $(a = 4) \wedge (b = 2)$ encompasses, amongst other states, both the states x and x' above).

3.4 Modelling using an Enumerative Representation

The first representation one might use to model a system is a direct one where all the states and transitions are represented explicitly into the model. A Kripke structure is a form of state transition graph which allows us to create such a kind of model. The nodes of a Kripke structure represent the states themselves, while the directed arcs between any pair of nodes (including the same node) represent the transitions from one state of the system to the next.

In order to know what the value of the variables in a particular state-node is we use an appropriate labelling function. This function when given a state returns the values of the variables true in that particular state. Moreover, a path or computation can be represented in a Kripke structure by a sequence of nodes whereby each node is reachable from the previous one by an existing directed arc.

Kripke structures have been described formally for modelling by various authors. These descriptions are congruent to one another and some of them may be found in [19–21, 37]. A *Kripke structure* M over AP is a four tuple

$$M = (S, S_0, R, L)$$

where,

- S is a finite set of states representing all the possible states of the system.
- $S_0 \subseteq S$ is the set of initial states which is subset of the set of possible states S .
- $R \subseteq S \times S$ is a transition relation describing the transitions of the system. This relation must be total i.e. there is a next state s' element of S for every state s element of S such that $R(s, s')$:
 $[\forall s \in S, \exists s' \in S. (s, s') \in R]$.
- $L : S \rightarrow 2^{AP}$ is a function that labels each state of the graph with the set of atomic propositions which are true in that state.

A path π in the structure M starting from state s can be defined as an infinite sequence of states $\pi = s_0 s_1 s_2 \dots$ such that $s_0 = s$ and for every pair of states on the path R holds.

In order to model a system using a Kripke structure just described we require two things, namely a formula for the initial states S_0 and a formula for the transition relation, \mathcal{R} . The set of states S is in fact the state space, i.e. all the possible valuations of V , some of which may not in fact be in the system we wish to model. The set of initial states S_0 is the set of all valuations for V which are true for the formula S_0 . This set of states forms our starting point from which we will obtain the reachable set of valid states of our system by means of the transition relation. The transition relation for two states s and s' holds if the formula \mathcal{R} evaluates to true with each valuation $v \in V$ for s and each $v' \in V$ for s' . One last item we require is the labelling function L . If $v \in L(s)$ then $s(v) = \text{true}$ else $v \notin L(s)$ and $s(v) = \text{false}$. Using thus the two formulas S_0 and \mathcal{R} we can describe our model's reachable (valid) states and use the Kripke structure to model these states explicitly.

Consider as example the system modelled by the Kripke structure shown in Figure 3.1. The system has three variables a , b and c . Each of these variables ranges over the binary domain $D = \{\text{true}, \text{false}\}$ (alternatively $D = \{1, 0\}$). Thus the 3-tuple $(d_1, d_2, d_3) \in D \times D \times D$ provides a valuation for the three variables, where d_1 is the value for a , d_2 is the value for b and d_3 is the value of c . The system will start at the state in which $a = \text{true}$, $b = \text{false}$ and $c = \text{false}$. Also, if a , b and c represent the values of the variables at the current state and a' , b' and c' represent the values of the variables at the next state the transitions of the system can be described as:

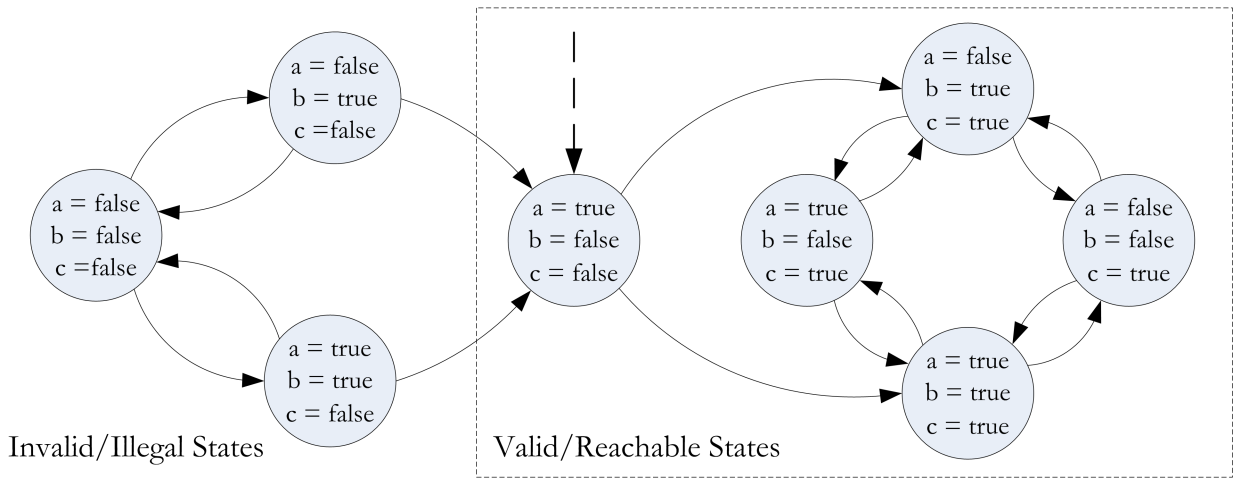


Figure 3.1: An Enumerative Model (Kripke Structure) of a simple system

$$a' :\approx a, \neg a$$

$$b' := \neg b$$

$$c' := (a \wedge b') \vee c$$

Note that:

“:=” represents a normal assignment

“:≈” represents a non-deterministic assignment

The two first order formulas \mathcal{S}_0 and \mathcal{R} can be used to fully characterise the system. \mathcal{S}_0 , the set of initial states is represented by:

$$\mathcal{S}_0(a, b, c) \equiv (a = \text{true}) \wedge (b = \text{false}) \wedge (c = \text{false}).$$

The set of transitions of the system, \mathcal{R} is represented by:

$$\mathcal{R}(a, b, c, a', b', c') \equiv$$

$$(((a = \text{true}) \wedge (b = \text{false}) \wedge (c = \text{false})) \wedge ((a' = \text{false}) \wedge (b' = \text{true}) \wedge (c' = \text{true}))) \vee$$

$$(((a = \text{true}) \wedge (b = \text{false}) \wedge (c = \text{false})) \wedge ((a' = \text{true}) \wedge (b' = \text{true}) \wedge (c' = \text{true}))) \vee$$

$$(((a = \text{true}) \wedge (b = \text{true}) \wedge (c = \text{true})) \wedge ((a' = \text{true}) \wedge (b' = \text{false}) \wedge (c' = \text{true}))) \vee$$

$$(((a = \text{true}) \wedge (b = \text{true}) \wedge (c = \text{true})) \wedge ((a' = \text{false}) \wedge (b' = \text{false}) \wedge (c' = \text{true}))) \vee$$

$$(((a = \text{false}) \wedge (b = \text{false}) \wedge (c = \text{true})) \wedge ((a' = \text{true}) \wedge (b' = \text{true}) \wedge (c' = \text{true}))) \vee$$

$$\begin{aligned}
& (((a = \text{true}) \wedge (b = \text{false}) \wedge (c = \text{true})) \wedge ((a' = \text{true}) \wedge (b' = \text{true}) \wedge (c' = \text{true}))) \vee \\
& (((a = \text{true}) \wedge (b = \text{false}) \wedge (c = \text{true})) \wedge ((a' = \text{false}) \wedge (b' = \text{true}) \wedge (c' = \text{true}))) \vee \\
& (((a = \text{false}) \wedge (b = \text{true}) \wedge (c = \text{true})) \wedge ((a' = \text{true}) \wedge (b' = \text{false}) \wedge (c' = \text{true}))) \vee \\
& (((a = \text{false}) \wedge (b = \text{false}) \wedge (c = \text{true})) \wedge ((a' = \text{false}) \wedge (b' = \text{true}) \wedge (c' = \text{true}))) \vee \\
& (((a = \text{false}) \wedge (b = \text{true}) \wedge (c = \text{true})) \wedge ((a' = \text{false}) \wedge (b' = \text{false}) \wedge (c' = \text{true}))) \vee \\
& (((a = \text{true}) \wedge (b = \text{true}) \wedge (c = \text{false})) \wedge ((a' = \text{true}) \wedge (b' = \text{false}) \wedge (c' = \text{false}))) \vee \\
& (((a = \text{false}) \wedge (b = \text{true}) \wedge (c = \text{false})) \wedge ((a' = \text{true}) \wedge (b' = \text{false}) \wedge (c' = \text{false}))) \vee \\
& (((a = \text{true}) \wedge (b = \text{true}) \wedge (c = \text{false})) \wedge ((a' = \text{false}) \wedge (b' = \text{false}) \wedge (c' = \text{false}))) \vee \\
& (((a = \text{false}) \wedge (b = \text{false}) \wedge (c = \text{false})) \wedge ((a' = \text{true}) \wedge (b' = \text{true}) \wedge (c' = \text{false}))) \vee \\
& (((a = \text{false}) \wedge (b = \text{false}) \wedge (c = \text{false})) \wedge ((a' = \text{false}) \wedge (b' = \text{true}) \wedge (c' = \text{false}))) \vee \\
& (((a = \text{false}) \wedge (b = \text{true}) \wedge (c = \text{false})) \wedge ((a' = \text{false}) \wedge (b' = \text{false}) \wedge (c' = \text{false}))).
\end{aligned}$$

Or, more compactly, using the system transitions directly:

$\mathcal{R} \equiv \mathcal{R}_0, \mathcal{R}_1$ where

$$\mathcal{R}_0(a, b, c, a', b', c') \equiv (a' = a) \wedge (b' = \neg b) \wedge (c' = (a \wedge b') \vee c).$$

$$\mathcal{R}_1(a, b, c, a', b', c') \equiv (a' = \neg a) \wedge (b' = \neg b) \wedge (c' = (a \wedge b') \vee c).$$

Now if we consider Figure 3.1 once more, as would be expected with three variables which range over a binary domain, the number of possible states of the system is eight. The transitions which satisfy \mathcal{R} are also shown as directed arches (sixteen in all). There is also one start state marked by an in-going directed arc which satisfies \mathcal{S}_0 . The states and transitions which satisfy both \mathcal{S}_0 and \mathcal{R} are enclosed by a dotted rectangle. These states and transitions compose the actual model of our system. Starting from the start state (itself a valid state) we can reach four other valid states (five states of the state space in all) by means of the transitions. In this way we obtain the computation paths which are valid according to our system. The rest of the states are considered invalid even though they are part of the possible state space. This is due to the fact that starting from the start state they can never be reached. Moreover the transition relation defines some transitions which

are never used since they exist between states that do not lie on paths which originate from the start state.

The Kripke structure $M = (S, S_0, R, L)$ in our example can be defined formally as:

- $S = D \times D \times D$.
- $S_0 = \{(true, false, false)\}$.
- $R = \{((true, false, false), (false, true, true)), ((true, false, false), (true, true, true)), ((true, true, true), (true, false, true)), ((true, true, true), (false, false, true)), ((false, false, true), (true, true, true)), ((true, false, true), (true, true, true)), ((true, false, true), (false, true, true)), ((false, true, true), (true, false, true)), ((false, false, true), (false, true, true)), ((false, true, true), (false, false, true)), ((true, true, false), (true, false, false)), ((false, true, false), (true, false, false)), ((true, true, false), (false, false, false)), ((false, false, false), (true, true, false)), ((false, false, false), (false, true, false)), ((true, false, true), (false, false, false))\}$.
- $L((true, false, false)) = \{a = false, b = true, c = true\}$,
 $L((true, false, false)) = \{a = true, b = true, c = true\}$,
 $L((true, true, true)) = \{a = true, b = false, c = true\}$,
 $L((true, true, true)) = \{a = false, b = false, c = true\}$,
 $L((false, false, true)) = \{a = true, b = true, c = true\}$,
 $L((true, false, true)) = \{a = true, b = true, c = true\}$,
 $L((true, false, true)) = \{a = false, b = true, c = true\}$,
 $L((false, true, true)) = \{a = true, b = false, c = true\}$,
 $L((false, false, true)) = \{a = false, b = true, c = true\}$,
 $L((false, true, true)) = \{a = false, b = false, c = true\}$,
 $L((true, true, false)) = \{a = true, b = false, c = false\}$,
 $L((false, true, false)) = \{a = true, b = false, c = false\}$,
 $L((true, true, false)) = \{a = false, b = false, c = false\}$,
 $L((false, false, false)) = \{a = true, b = true, c = false\}$,
 $L((false, false, false)) = \{a = false, b = true, c = false\}$,
 $L((false, true, false)) = \{a = false, b = false, c = false\}$.

and the model consists of a number of valid computation paths, some of which are:

- $(true, false, false), (false, true, true), (true, false, true), (true, true, true), (true, false, true), (false, true, true), \dots$
- $(true, false, false), (false, true, true), (false, false, true), (true, true, true), (true, false, true), (false, true, true), \dots$
- $(true, false, false), (true, true, true), (false, false, true), (false, true, true), (true, false, true), (true, true, true), \dots$

3.5 Modelling using a Symbolic Representation

Enumerative model checking using Kripke structures achieves the goal of modelling a system for model checking. However since it uses an explicit enumeration of the system's states, with every new system state introduced the model must grow accordingly. Due to the state explosion problem mentioned earlier some systems soon become too large to model using Kripke structures. In order to mitigate this problem, symbolic model checking was introduced whereby the states and transitions of the systems are represented more compactly by their associated Boolean formulas instead of explicitly. The set of states of the system can be thus represented by a formula which such states satisfy. The formula at hand can be characterised using various representations. One direct but inefficient symbolic representation is a truth-table. These tables adequately represent formulas but suffer from exponential blow-up as well since for every variable introduced the table grows twice in size. In this section we consider Ordered Binary Decision Diagrams as a much more efficient symbolic alternative which produces more compact models [7, 8, 13, 20, 37]. OBDDs were created as a structure by Bryant and discussed in various of his papers amongst which are [11, 12].

OBDDs are based on *binary decision trees* [4, 7, 20, 37]. A binary decision tree is a directed, rooted tree which has two kinds of vertices: nonterminal vertices and terminal vertices. Nonterminal vertices are labelled by $var(v)$ and have two outgoing branches, one for when v is assigned the value 1 and one for when v is assigned the value 0. These lead to two successor vertices (or child vertices), $high(v)$ and $low(v)$ respectively, which are themselves vertices (may be terminal or non-terminal depending on the case). Nonterminal vertices are the point of decision: where we decide

a	b	c	d	$a \vee b$	$c \rightarrow d$	$(a \vee b) \wedge (c \rightarrow d)$
0	0	0	0	0	1	0
0	0	0	1	0	1	0
0	0	1	0	0	0	0
0	0	1	1	0	1	0
0	1	0	0	1	1	1
0	1	0	1	1	1	1
0	1	1	0	1	0	0
0	1	1	1	1	1	1
1	0	0	0	1	1	1
1	0	0	1	1	1	1
1	0	1	0	1	0	0
1	0	1	1	1	1	1
1	1	0	0	1	1	1
1	1	0	1	1	1	1
1	1	1	0	1	0	0
1	1	1	1	1	1	1

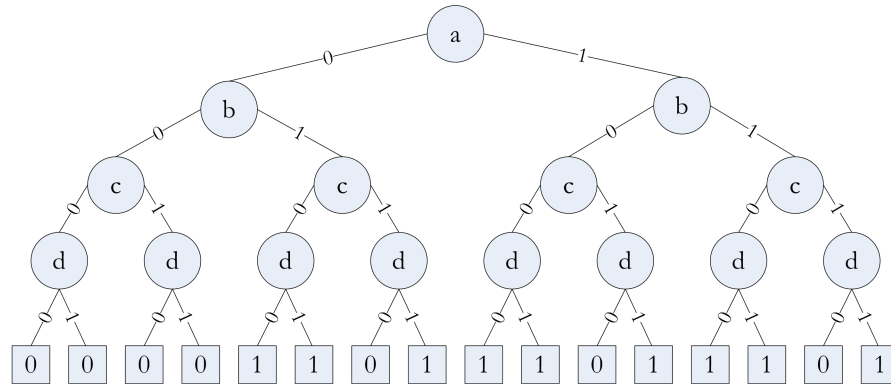
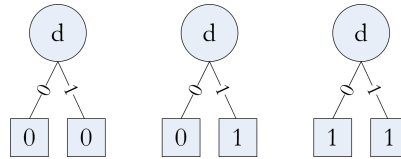
Table 3.1: Truth Table of Boolean function $(a \vee b) \wedge (c \rightarrow d)$

whether the Boolean variable is assigned 0 or 1 in the function. Depending on the assignments we require for the Boolean variables, the tree guides us to an appropriate terminal vertex. Terminal vertices are labelled by $value(v)$ which is either 0 or 1. These vertices signify the final value of a Boolean function and are reached by traversing the tree from the root variable non-terminal node by following the path according to the variable assignments required.

Consider as example the Boolean formula $(a \vee b) \wedge (c \rightarrow d)$. The truth table for this formula is shown in Table 3.1. The equivalent binary decision tree is shown in Figure 3.2. The circle nodes are the nonterminals while the square nodes are the terminals. As can be noted however, there is no real advantage in representing a formula using a binary decision tree over a truth table.

If one views Figure 3.2 one notices that there are many similar, repeated sub-trees; a lot of redundancy. Consider, for example, the sub-trees at the level of the variable d . There are in fact many repeated terminal node pairs as children of these d nodes as can be seen in Figure 3.3. If we identify and remove such repetition and redundancy cases in the tree without changing the formula represented by the graph, we obtain what is known as *binary decision diagram (BDD)* from the binary decision tree. The tree thus becomes a directed acyclic graph (DAG). [4, 7, 11, 20, 37]

Structure-wise a BDD is very similar to a binary decision tree. It consists of two types of nodes, ter-

Figure 3.2: Binary Decision Tree of the example formula $(a \vee b) \wedge (c \rightarrow d)$ Figure 3.3: Distinct d node sub-trees

minimals and nonterminals. Both terminal and non-terminal nodes are similar to the ones of binary decision trees described earlier.

Every sub-tree rooted at a non-terminal v represents a function f_v . If the variable represented by the node v is x_i then the function represented by the subtree rooted at v is:

$$f_v = (\neg x_i \wedge f_{low(v)}) \vee (x_i \wedge f_{high(v)})$$

If by using a particular assignment of variables, tree traversal leads to a terminal node, $value(v) = 1$ then the function's result is 1 else if $value(v) = 0$ then the function's result is 0.

The representation we require to model our system symbolically is based on a form of BDD called *ordered binary decision diagram* (OBDD). In [11] Bryant showed how BDDs can be ordered and reduced so that they provide a compact canonical representation of a Boolean function. Since they are able to represent a Boolean function, OBDDs may be use to represent models of system as described by McMillan in [37].

Ordering a BDD means that the variables in the graph are given a fixed order from root to any of the terminal nodes. This can be done by a total ordering $<$ of the variables by ordering the non-

terminal nodes which are labelled by them. This implies that if for example, $a < b$, in the graph a is always eventually a parent node of b if we go up the path from the latter, while b is always eventually a child node of a if we go down the path from the latter. In other words, b may be found in the sub-tree of a but not vice versa.

Reduction is achieved by considering isomorphic sub-trees and redundant nodes. Isomorphic subtrees are two subtrees where each of terminal and non-terminal nodes in the first subtree have corresponding nodes in the second subtree. Isomorphic sub-trees should be removed by removing duplicate copies and leaving a single sub-tree. All the arcs to the removed sub-trees are then pointed to the remaining isomorphic sub-tree. Moreover, redundant nodes such as those where a parent node points to a child node with both its *low* and *high* arcs should be replaced with the child node. To achieve this, three transformation rules should be applied on the graph repeatedly. These transformations don't alter the function represented. They are:

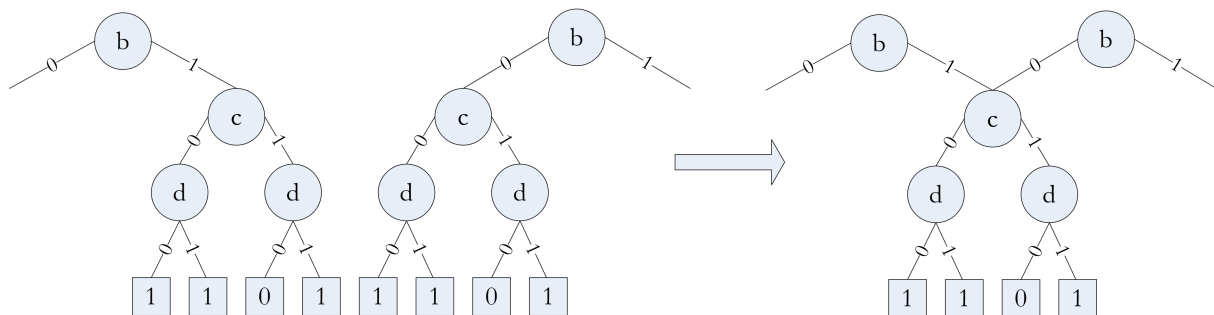
Remove duplicate terminals — remove all duplicated terminals leaving only one copy. All the directed arcs to these removed terminals should be redirected to the remaining terminal node representing them.

Remove duplicate nonterminals — if two nonterminals x and y represent the same variable (i.e. $var(x) = var(y)$) and their low and high arcs point to nonterminals that represent the same variable (i.e. $low(x) = low(y)$ and $high(x) = high(y)$), then either x or y can be eliminated and all the arcs going into the removed node can be directed to the remaining one. An example of this is shown in Figure 3.4.

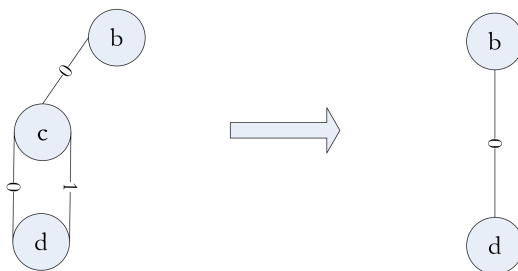
Remove redundant tests — if a non-terminal x has both its low and high arcs pointing to the same node (i.e. $low(x) = high(x)$), remove node x and point all the arcs going into x to $low(x)$. Refer once more to Figure 3.4 for an example of this reduction technique.

Using a fixed variable ordering and applying the above rules until the graph no longer reduces in size provides a canonical representation. The graph obtained is also known as an OBDD. As an example of an OBDD consider the ordering and reduction of the formula shown earlier $(a \vee b) \wedge (c \rightarrow d)$ in Figure 3.5 shown as a binary decision tree earlier in Figure 3.2.

In [11], Bryant describes various algorithms which allow us to construct, manipulate and reason about OBDDs. One of these is of course the algorithm *Reduce* which provides a canonical OBDD



Removal of duplicate nonterminals



Removal of redundant tests

Figure 3.4: OBDD Reduction Techniques: (Above) Removal of duplicate nonterminals, (Below) Removal of redundant tests

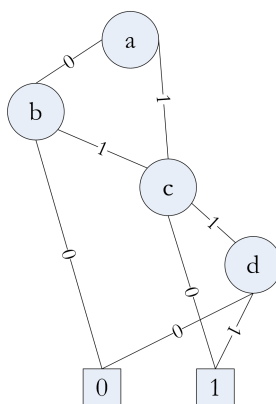


Figure 3.5: Binary Decision Diagram of the example formula $(a \vee b) \wedge (c \rightarrow d)$

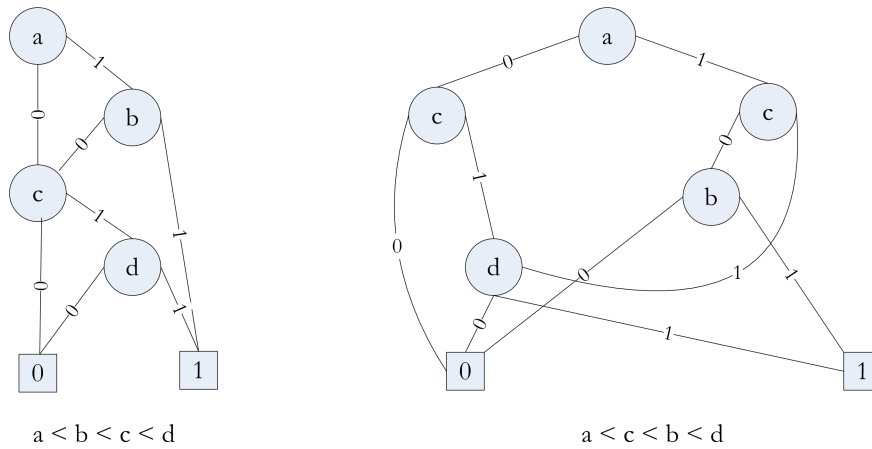


Figure 3.6: OBDD variable ordering for the formula $(a \wedge b) \vee (c \wedge d)$: (Left) $a < b < c < d$, (Right) $a < c < b < d$

from a function graph using the reduction rules just described. Also of note is the algorithm *Apply* which takes graphs which represent two functions and one of the sixteen binary operators and produces a reduced graph representing a new function which combines the functions using the operator.

It is important to know that different orderings of the variables leads to OBDDs of different sizes. For example consider the OBDD representation of the formula $(a \wedge b) \vee (c \wedge d)$. Figure 3.6 shows how different variable orderings effect the size of this OBDD. The OBDD on the left-hand side is smaller than the one on the right by one node. This example is trivial, however, as a formula grows variable ordering may have a great impact on the size of the OBDD. Since we are going to use OBDDs to create our model and in model checking the size of the model is an important factor on what sizes of systems can be modelled, finding the variable ordering which leads to an optimal ordering is essential. Clarke et al. in [20] however tell us that Bryant discovered that finding the optimal ordering however is infeasible. They continue to tell us that to mitigate this problem various heuristics were later developed.

Now that we have discussed OBDDs we show how to employ them for use in modelling for use in symbolic model checking. Modelling can be achieved in two ways, either by representing the system's Kripke structure by means of OBDDs or by constructing the OBDD directly from a high level description of the system such as the first order formulas, \mathcal{S}_0 and \mathcal{R} used in the earlier example to construct a Kripke structure. Needless to say it is often considered better to construct the

OBDD directly, as the intermediate Kripke structure might be very large even if the OBDD is not. [20]

In both of the above cases, in order to model a system using OBDDs we must represent the transition relation and, when required, the set of initial states which is also in fact a relation. (Here again we use the notation due to Clarke et al. in [20].) Both these relations may sometimes range over a binary domain such as $\{0, 1\}$ or $\{false, true\}$. In such a case to represent them using an OBDD we use the characteristic function of the relation. Generally, if Q is an n -ary relation over a domain $\{0, 1\}$ then we can represent it using its characteristic function f_Q , such that:

$$f_Q(x_1, \dots, x_n) = 1 \text{ iff } Q(x_1, \dots, x_n)$$

If the relation however does not range over a binary domain but over a larger finite one, we need a form of mapping which maps an encoding which consists of a string of digits (a vector) from the set $\{0, 1\}$ to the elements of this domain. We assume the size of the domain is 2^m where m tells us how many binary digits we require to map the entire domain. Note that this causes, sometimes, the number of binary patterns allowable with m bits to be larger than the domain. This however is a minor setback, as these bit patterns are just unused or considered invalid. Formally, if Q is an n -ary relation over a finite domain D with 2^m elements we use the bijection $\phi : \{0, 1\}^m \rightarrow D$ that maps a vector of size m to an element of D . Now that we have an adequate mapping, we require a Boolean relation that is the same as the original one in meaning but which instead of taking as input values of the domain D , it uses the mappings just created. Thus a relation of $m \times n$ arity must be created. We denote the relation by \hat{Q} and define it as:

$$\hat{Q}(\bar{x}_1, \dots, \bar{x}_n) = Q(\phi(\bar{x}_1), \dots, \phi(\bar{x}_n))$$

where \bar{x}_i is a vector of m Boolean variables that encodes the variable x_i . As can be seen bijection ϕ is being used to convert the encoding into a value in the domain D before passing it to the original relation Q . Q can thus be represented by an OBDD of the characteristic function $f_{\hat{Q}}$ of the newly defined relation \hat{Q} .

Finally, it is also possible to create the characteristic of a relation whose domain consists of the Cartesian products of different domains, $D_1 \times D_2 \times \dots \times D_n$, where n is the relation's arity. To achieve this all we require is different bijections for the different domains, $\phi_1, \phi_2, \dots, \phi_n$, respectively. We then define the new relation as:

$$\hat{Q}'(\bar{x}_1, \dots, \bar{x}_n) = Q(\phi_1(\bar{x}_1), \dots, \phi_n(\bar{x}_n))$$

Using the characteristic function of this relation enables us to represent it using an OBDD.

Clarke et al. [20] tell us that using this encoding it is possible to represent a Kripke structure $M = (S, S_0, R, L)$ using OBDDs by representing its components: the sets of states S and S_0 , the relation R and the mapping L . Representing S is quite straightforward. We use a bijection similar to the one just described which maps a vector of bits to each state. Each encoding is used to describe a state of the state space. Since S is in fact the entire state space we end up with the OBDD for 1. The set of initial states can be represented similarly to the encoding used for S . For the transition relation R we use the same state-encoding and as in enumerative model checking two sets of Boolean variables, one to represent the state before transition and one to represent the state after transition. If we use the binary relation \hat{R} to represent the required transition relation R as described above we can create the OBDD for it by using its characteristic function. Mapping L in this case will be considered to map an atomic proposition to a set of states, i.e. a subset of S where it is true. This set of states can also be encoded using the same encoding used to represent S (similar to how S_0 is also represented) and hence can be represented with an OBDD. Each of the atomic propositions can be represented this way. Thus we have shown how OBDDs can be used to represent a Kripke structure and hence model our system.

Instead of representing the Kripke structure itself we can employ another method to model our system symbolically. To do so we use a high level description of the system such as the first order formulas of the transition relation \mathcal{R} and the set of initial states S_0 in our earlier example. However, instead of creating their Kripke structure as in enumerative model checking we simply create their OBDDs by first obtaining the relations' characteristic functions. Using Bryant's *Apply* algorithm mentioned earlier we can easily build the OBDD representation sub-function at a time until we obtain the OBDDs of the characteristic functions which model the states and transitions of the system. As an example consider the OBDDs for the characteristic functions of S_0 (ordering: $a < b < c$) and \mathcal{R} (ordering: $a < a' < b < b' < c < c'$) which we have defined earlier. These OBDDs are shown in Figure 3.7 and they model the same exact system shown earlier in Figure 3.1.

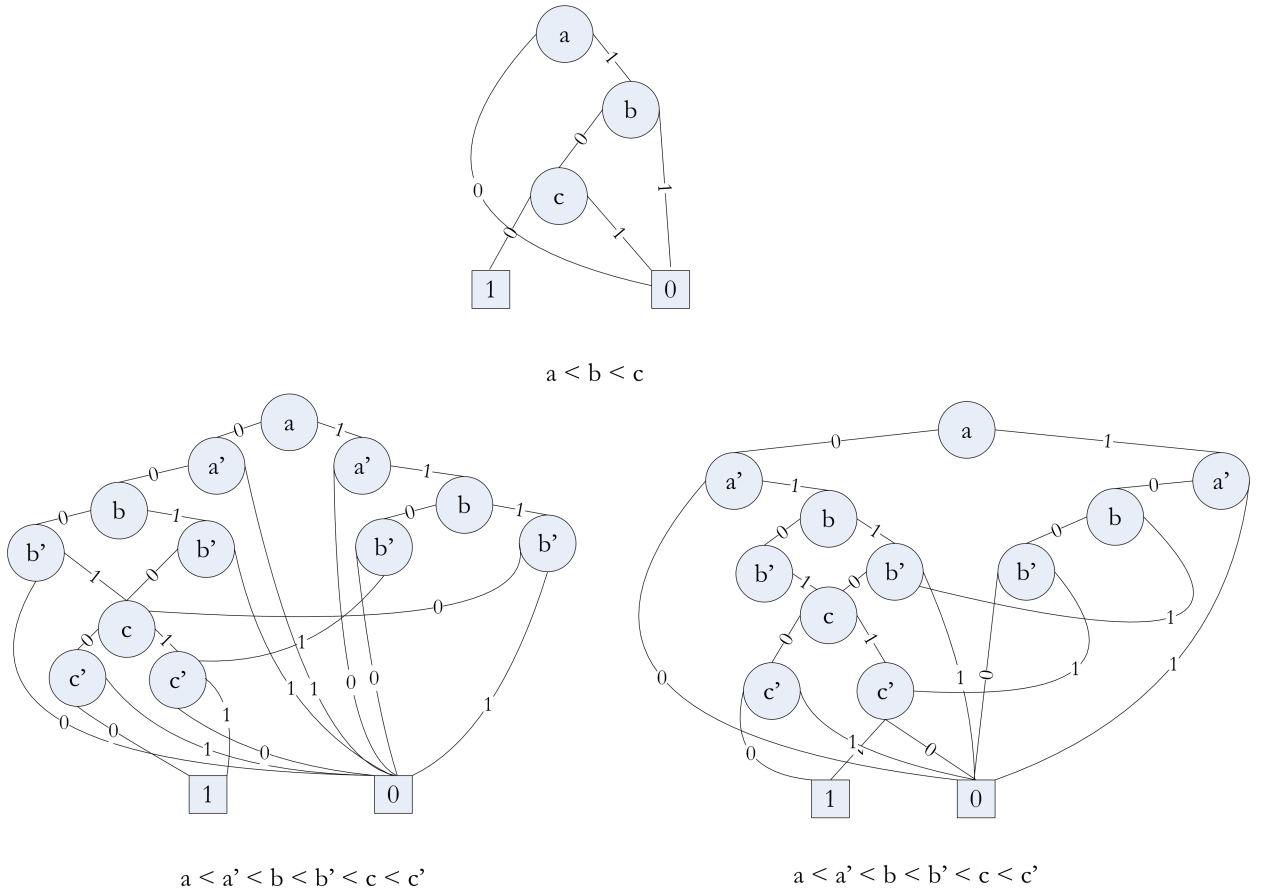


Figure 3.7: A Symbolic Model (OBDD) of a simple system: (Top) OBDD of the characteristic function of $S_0(a, b, c)$, (Bottom Left) OBDD of the characteristic function of $R_0(a, b, c, a', b', c')$ representing the first part of transition relation \mathcal{R} , (Bottom Right) OBDD of the characteristic function of $R_1(a, b, c, a', b', c')$ representing the second part of transition relation \mathcal{R} .

3.6 Conclusion

In this chapter we have seen how a system may be modelled enumeratively using a Kripke structure and a symbolically by means of OBDDs. In next chapter we will introduce the next stage of model checking, that is, specification. Specification concerns itself with the formulation of properties which are then used to verify the systems we have modelled. In order to achieve this we make use of what are known as temporal logics which we introduce shortly.

SPECIFICATION

*It is easier to change the specification
to fit the program than vice versa.*

Alan Perlis

4.1 Overview

This chapter is dedicated to the specification step of the model checking process. As highlighted earlier specification entails the translation of properties, which we wish to validate on the system at hand, into a form of logic which allows us to reason about the states of the system. The logic utilised is often *temporal logic*. It is of various forms, each of which has different expressiveness properties. In this section we will discuss various such forms and provide their syntax and semantics.

4.2 Introduction

The specification of a system, which is obtained by a careful analysis of its requirements, consists of the properties which the system must have at all costs for it to be considered valid. When we create a system we try to abide by these specifications to ensure that the system is correct. However when designing the system based on these specifications, sometimes there are inconsistencies or bugs which may crop up no matter the effort done to produce the correct design. A model of a

system is the translation of the design into a representation. If mistakes are present in the design, translation of the design into the model obviously leads to a faulty model. In model checking we check models for such faults by making an exhaustive search in the reachable states of the system. To perform this task the properties are converted into temporal logic formulas and by means of model checking algorithms these properties are verified on the model.

4.3 Temporal Logic and Temporal Logic Operators

Temporal logic is a form of logic which allows us to reason about sequences of transitions between states of a system without introducing time explicitly in the notation [20]. Formulas in temporal logic consist of the usual Boolean operators used in propositional logic such as \wedge , \vee , \neg , \Rightarrow and \Leftrightarrow . However in order to reason about the states of a model representing the system other operators must be introduced. Such operators would allow us to specify things about the system such as whether a particular state will be eventually or finally reached or that an error state in the state space is never reached at all [20].

There are two kinds of these operators in temporal logic: path quantifiers and temporal operators [20]. Path quantifiers, as their name implies, quantify over paths (sequences of states). Clarke et al. tell us in [20] that they are usually of two forms:

- *Existential* path quantifier, (**E**) — signifies that a kind of path does exist, i.e. starting from a certain state at least one path exists where the formula at hand is true.
- *Universal* path quantifier, (**A**) — signifies that the formula at hand is true for all the paths starting at a particular state.

Temporal operators describe the paths themselves. There are various such operators depending on the temporal logic employed. Some of the most common according to Clarke et al. [20] are:

- *Next* temporal operator, (**X**) — a property holds in the state after the current one on the path.
- *Finally* temporal operator, (**F**) — also known as *eventually* and *in the future*, this operator means that a property will hold on some state on the path.

- *Globally* temporal operator, (**G**) — also known as *always*, this operator asserts that a property holds on all the states on the path.
- *Until* temporal operator, (**U**) — this operator means that a second property holds at this state or in a future state and that in all the states leading to such a state the first property holds. Also, once the second property holds the first property does not need to hold.
- *Release* temporal operator, (**R**) — along the path, the second property holds. If along said path the first property is true, then the first property releases the second property, i.e. in a state (both are true), and in the rest of the path the second property is false. Otherwise, there is the possibility that the first property does not eventually hold.

The temporal logic operators just described are summarised in Table 4.1.

4.4 Linear-time and Branching-time Temporal Logics

Ever since the advent of the first temporal logics, temporal logics have been divided into many different categorisations. In our study the main categorisation we are interested in divides temporal logics into two main families: linear-time or branching-time. The difference between these families of temporal logics lies in how time is viewed. Linear-time temporal logics view time as a linear sequence of events while branching-time temporal logics consider time to be able to branch into different pathways. Various papers exist which discuss the differences between these kinds of logics: what they can express, how they are related to one another, whether one can discard one for the other and so on. Some of these papers are [6, 19, 21, 25–27, 33]. We will now summarise the contents of such papers so as to compare and contrast linear-time and branching-time temporal logic and at the same time introduce various temporal logic languages.

One of the first thorough studies which concerns linear-time temporal logics according to Lamport [33] was done by Pnueli. Lamport in his study compares linear temporal logic with branching time temporal logic and argues that linear temporal logic is better for specifying concurrent systems while branching-time is better for non-determinism due to the possibility of a state branching into different future states. He also argues that these logics are not equivalent in their expressiveness. The operators used by Lamport in his paper are the “always” (\Box) and “sometime” (\rightarrow) which

Name	Symbol	Arity	Type	Timeline
Next	N	Unary	Temporal Operator	
Finally	F	Unary	Temporal Operator	
Globally	G	Unary	Temporal Operator	
Until	U	Binary	Temporal Operator	
Release	R	Binary	Temporal Operator	

Table 4.1: Summary of Temporal Logic Operators. The graphs showing the timelines are partially reproduced from [38]

are similar to the **G** and **F** described above respectively. He also introduced some further logic operators to increase the expressiveness of his temporal logic language.

Ben-Ari et al. in [6] discuss linear-time and branching-time logic and combine them into a single language which they name *UB* for “the unified system of branching time” [p.165]. This language is based on time which is branch-like in structure. However, they argue that by providing symmetrical sets of temporal operators it is possible to write both linear and branching temporal logic properties. The temporal operators used in *UB* consist of a path quantifier (**A** and **E**) followed by one of these temporal operators: **G**, **F** and **X**. Apart from this they also tell us that whichever type of temporal logic is used in a program depends on the nature of the type of system one wishes to formalise.

In the first work to make use of model checking as it is known today, Clarke and Emerson [21] form the language *CTL*. *CTL* or computation tree logic is a branching-time temporal logic which extends *UB* by adding the until (**U**) temporal operator, i.e. by adding two combinations of path quantifier—temporal operator pair (as in Ben-Ari et al.’s work): **EU** and **AU**. Continuing on this work we find a study by Emerson and Halpern [26]. By altering the syntax of both *CTL* and *UB* in various ways they produce a number of languages which they compare for expressiveness.

Emerson and Halpern [25] continue on the work by both Lamport and Ben-Ari et al. by first providing criticism about some mistakes in former’s approach. Lamport stated that some linear properties cannot be expressed in branching temporal logic thus making the former superior to the latter when it comes to certain systems, namely concurrent ones. They argue that by introducing more operators branching temporal logic can become more expressive. They proceed in a similar way to Ben-Ari et al. by providing a temporal language which like *UB* allows a better comparison of linear-time and branching-time temporal logics which they call *CTL**.

Clarke, Emerson and Sistla provide in [19] a model checking algorithm for the branching-time temporal logic *CTL* which is of linear complexity in the temporal logic specification and the size of the state-space of the system. They also show how as a branching-time temporal logic *CTL* is able to handle fairness properties by using a simple change in its semantics. Fairness properties are properties where something must occur “sufficiently often”. This is an important aspect in most systems. Finally they consider the complexity of the model checking algorithms of various other temporal logic languages: *BT**, *CTL*⁺ and *CTL**. *CTL** is of interest since it gives more freedom in

how temporal operators and path quantifiers are used allowing both linear-time and branching-time temporal logic formulas to be expressed.

Emerson and Lei in [27] again compare linear-time and branching-time temporal logic and question which one of the two logics is better for concurrent systems where non-determinism exists. They say that in linear-time temporal logic although the universal quantifier is not used it is implied since in this logic only one future is concerned and to achieve this quantification over all possible futures must be done. Branching-time temporal logic on the other hand allows the possibilities of existential quantification as well. Thus while branching-time temporal logics often allow **AX**, **EX**, **AF**, **EF**, **AG**, **EG**, **AU** and **EU**, linear-time temporal logics often allow **(A)X**, **(A)F**, **(A)G** and **(A)U**. Apart from this difference Emerson and Lei state that unlike branching-time temporal logic, linear-time temporal logic is able to handle fairness without any change being required. Referencing work by Pnueli and Lichtenstein (see [27] for an indication of the paper at hand), they say that a linear-time temporal logic model checking exists which is quadratic in the model and exponential in the formula length. They also argue that CTL^* is more expressive than any linear-time temporal logic language due to the fact that it allows the use of **A** or **E** to be followed by any formula of linear-time temporal logic which is not restricted in any way. Thus they show that restricting oneself to linear-time temporal logic is unnecessary as it proves of no advantage over the equivalent branching-time temporal logic language. One final interesting point raised by Emerson and Lei is the one also found in [21]: instead of viewing CTL as a sublanguage of CTL^* it can be viewed as a sub-language of the propositional μ -calculus. The μ -calculus allows us to represent temporal operators using a recursive definition by means of the fixpoint operators μ and ν [27].

4.5 Temporal Logic Properties

The properties expressed by the various temporal logics may be categorised based on the kind features the properties wish to verify on the system at hand. During the development of model checking various categories have emerged. Some of the most prominent ones are:

Safety Properties — these kind of properties have been described by Lamport [33] as asserting that “*something bad never happens*”. The properties are often characterised by the *globally* temporal operator (**G**) since it allows us to assert that a property always holds on a path: i.e. that something

bad never happens to cause the property at hand to fail. These properties ensure that the system never reaches a state which although in the state-space, is invalid for the system. A simple example in the case of our domain of study is a tictactoe board where each board location is marked with a cross. This board state is invalid since it cannot be reached by any gameplay which follows the game's rules. A safety property can be written which allows us to check that such a state is never reached, i.e. that all board states at hand are valid ones. If we can define what a valid board state is and refer to this definition by *valid_state* we can write the safety property $\mathbf{AG} \text{valid_state}$ which means that for all computation paths, and all the states on such paths, the board state is a valid one.

Liveness Properties — again by Lamport, these properties assert that “*something good must eventually happen*”. To achieve these properties we need to express the concept of eventuality. As one would expect the temporal operator *finally* (\mathbf{F}) is often used as part of the property to achieve this. This is due to the fact that when it is used in a property it expresses the need for the property to be eventually true along a path, i.e. that the property will be eventually/finally true along the path. These properties allow us to show that the system at hand does reach a certain kind of states. In our case an example of liveness property would be one that verifies that both players may win eventually the game: i.e. that the game is not biased to allow only one player to win. If we can define what it entails for either of the two players to win by the designation $(\text{player1_win} \vee \text{player2_win})$ we can write the liveness property required as $\mathbf{AF}(\text{player1_win} \vee \text{player2_win})$ which means that on all paths there is always finally a state where either player one or player two wins. An even better example of liveness properties is another property based on $\mathbf{AF}(\text{player1_win} \vee \text{player2_win})$ which is $\mathbf{AG}(\text{empty_board} \rightarrow \mathbf{AF}(\text{player1_win} \vee \text{player2_win}))$. Using the definition *empty_board* which refers to the start state (this is usually an empty board in our case), in this property we are saying that in all states of all the computation paths of the system, it is true that starting from the empty board all computation paths lead to a state where either of the two player wins.

An important subcategory of liveness properties are *fairness properties*. A fairness property is one which is “*assumed to hold infinitely often along all computations paths*” [37]. These properties are often used to verify other properties which are typically liveness properties. These other properties are verified not directly on the model itself but on the fair paths designated by the fairness properties. Fairness properties usually require the use of both the \mathbf{G} and \mathbf{F} directly combined. This fact makes

certain languages unable to express fairness properties. An example of this was mentioned in Section 4.4 when we discussed *CTL* briefly. *CTL* requires a change in its semantics to allow such properties to be expressed. Other languages such as *CTL** and the latter's derived linear temporal logic language *LTL* are able to express fairness constraint directly since **G** and **F** may be combined in such logics.

4.6 Temporal Logic Languages

In following sections we will introduce some temporal logic languages which are of interest to our study. These are the aforementioned languages:

- Computation Tree Temporal Logic — *CTL**
- *CTL** Branching-time Temporal Logic — *CTL*
- *CTL** Linear-time Temporal Logic — *LTL*
- μ -calculus

LTL and *CTL* are sublogics of *CTL** which are restricted to allow the specification of pure linear-time temporal logic properties and of pure branching-time temporal logic properties respectively. Since *LTL* is a pure linear-time temporal logic and *CTL* is a pure branching-time temporal logic there are some properties in former which are not expressible in latter and vice versa [20, 25, 33]. As simple examples consider the example formulas given by Clarke et al. in [20]: *LTL* is unable to express the *CTL* formula **AG**(**EF** p) while *CTL* is unable to express the *LTL* formula **A**(**FG** p). More information on what these formulas mean and how they are derived will be given later on. Moreover if we combine these formulas by a disjunction we obtain a *CTL** formula which neither of them can express—this also implies that *CTL** is more expressive than the two languages combined.

The propositional μ -calculus may be used to express temporal logic formulas. By means of its fixpoint operators we are able to use a recursive definition which represents a specific set of states of the state-space we wish to verify. This set of states contains the states where the μ -calculus formula is satisfied. Moreover it is possible to represent the temporal operators described above

using the equivalent fixpoint translation. This practice has been often discussed in various works: [7, 13, 20, 21, 27, 37]. Emerson and Lei [27] give two adequate examples of a CTL formula and a CTL^* formula written as μ -calculus formulas which we reproduce here:

- CTL example: $EFp = \mu Z_1. p \vee EXZ_1$
- CTL^* example: $EGFp = \nu Z_1. \mu Z_2. EX[(p \wedge Z_1) \vee Z_2]$

where Z_1 and Z_2 are atomic proposition variables ranging over a set of states. The CTL formula expresses a simple liveness property, i.e. there exists a path starting for this state where p is true. The property expressed by the CTL^* formula is one which states that starting from this state there exists a path where p is true infinitely often. This is an example of a fairness property. These formulas are just quick examples and, as state earlier, the following sections explain in more detail how they may be formed and what they mean by providing their respective language syntax and semantics.

4.7 Computation Tree Logic— CTL^*

4.7.1 Introduction

CTL^* was first introduced by Clarke, Emerson and Sistla in [19]. It is further discussed in [18, 20, 24, 25, 27]. CTL^* is based on the concept of computation trees. While the state transition graph of a system is finite in size, if we pick one of the states to be the start state, follow the transitions possible at each state and record our paths we obtain a tree of infinite size [20]. As an example consider Figure 4.1. The figure contains a Kripke structure depicting a simple model of a system and the corresponding infinite computation tree obtained from it. We will now present the syntax and semantics of CTL^* based on Clarke et al. in [20].

4.7.2 Syntax

CTL^* properties are made of a path quantifier (**A** or **E**) followed by a formula made up of combinations of the temporal operators **G**, **F**, **X**, **U** and **R** described in Section 4.3. Emerson and Halpern [25], and Emerson and Lei [27] also include two other operators called infinitary state quantifiers which are abbreviations of commonly used operator combinations. These are:

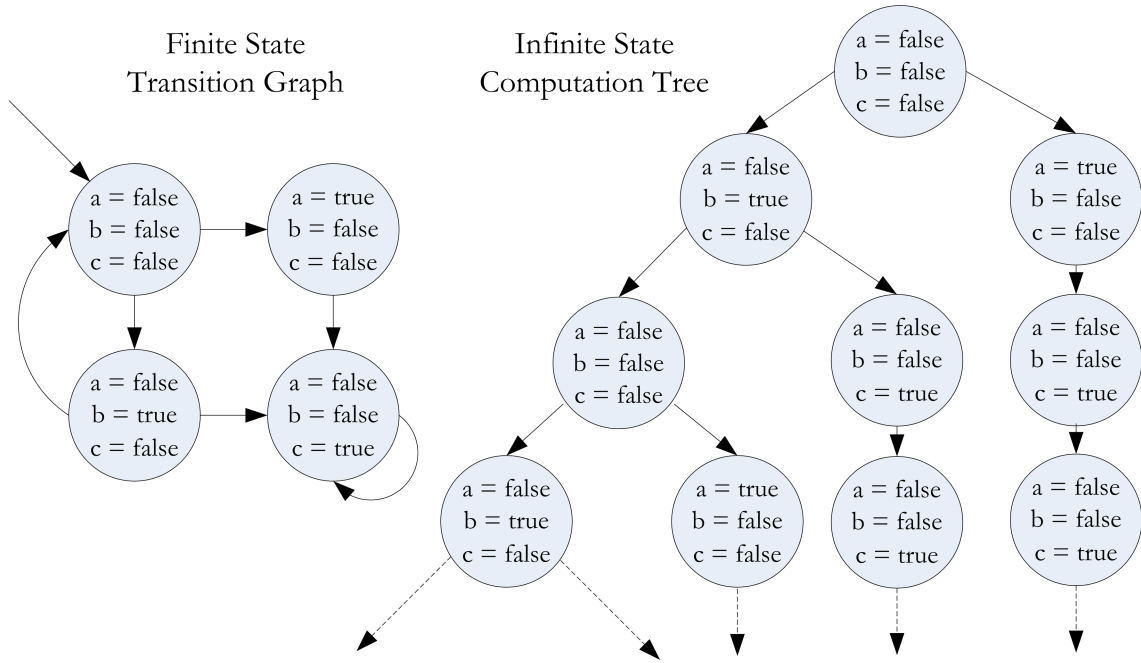


Figure 4.1: The finite state transition graph depicted and the computation tree obtained from it

- $\tilde{\mathbf{F}}p$ — “infinitely often” or $\mathbf{GF}p$
- $\tilde{\mathbf{G}}p$ — “almost everywhere” or $\mathbf{FG}p$

[25]

The syntax of CTL* divides the language’s formulas into two types [19, 20, 25]:

- *state formulas* — these are formulas which are true in a specific state.

The syntax of state formulas follows the three rules below:

- If $p \in AP$, then it is a state formula. (Rem: AP is the set of atomic propositions.)
- If s and t are state formulas, then $\neg s$, $s \wedge t$, $s \vee t$ are state formulas.
- If s is a path formula, then $\mathbf{E}s$ and $\mathbf{A}s$ are state formulas.

- *path formulas* — these are formulas which are true along a specific path.

The syntax of path formulas follows the two rules below:

- If s is a state formula, then s is also a path formula.
- If s and t are path formulas, then $\neg s$, $s \wedge t$, $s \vee t$, $\mathbf{X}s$, $\mathbf{F}s$, $\mathbf{G}s$, $s\mathbf{U}t$, $s\mathbf{R}t$, $\tilde{\mathbf{F}}s$, $\tilde{\mathbf{G}}s$ are path formulas.

Using the above five rules allows us to generate all of the formulas for CTL* .

4.7.3 Semantics

Various authors [18–20, 24, 25] define the semantics of CTL^* with respect to a Kripke structure. Note that:

- π^i is used to denote the part, or more accurately suffix of π starting at s_i .
- $M, s \models s$, where s is a state formula means that the latter holds at state s in the Kripke structure M .
- $M, \pi \models t$, where t is a path formula means that the latter holds along path π in the Kripke structure M .

The \models relation is inductively defined using a number of rules. Assuming s_1 and s_2 are state formulas and t_1 and t_2 are path formulas:

1. $M, s \models p \Leftrightarrow p \in L(s)$
2. $M, s \models \neg s_1 \Leftrightarrow M, s \not\models s_1$
3. $M, s \models s_1 \vee s_2 \Leftrightarrow M, s \models s_1 \text{ or } M, s \models s_2$
4. $M, s \models s_1 \wedge s_2 \Leftrightarrow M, s \models s_1 \text{ and } M, s \models s_2$
5. $M, s \models \mathbf{E}t_1 \Leftrightarrow$ there is a path π from s such that $M, \pi \models t_1$
6. $M, s \models \mathbf{A}t_1 \Leftrightarrow$ for every path π starting from s , $M, \pi \models t_1$
7. $M, \pi \models s_1 \Leftrightarrow s$ is the first state of π and $M, s \models s_1$
8. $M, \pi \models \neg t_1 \Leftrightarrow M, \pi \not\models t_1$
9. $M, \pi \models t_1 \vee t_2 \Leftrightarrow M, \pi \models t_1 \text{ or } M, \pi \models t_2$
10. $M, \pi \models t_1 \wedge t_2 \Leftrightarrow M, \pi \models t_1 \text{ and } M, \pi \models t_2$
11. $M, \pi \models \mathbf{X}t_1 \Leftrightarrow M, \pi^1 \models t_1$
12. $M, \pi \models \mathbf{F}t_1 \Leftrightarrow$ there exists a $k \geq 0$ such that $M, \pi^k \models t_1$
13. $M, \pi \models \mathbf{G}t_1 \Leftrightarrow$ for all $i \geq 0$, $M, \pi^i \models t_1$
14. $M, \pi \models t_1 \mathbf{U}t_2 \Leftrightarrow$ there exists a $k \geq 0$ such that $M, \pi^k \models t_2$ and for all $0 \leq j < k$, $M, \pi^j \models t_1$
15. $M, \pi \models t_1 \mathbf{R}t_2 \Leftrightarrow$ for all $j \geq 0$, if for every $i < j$ $M, \pi^i \not\models t_1$ then $M, \pi^j \models t_2$
16. $M, \pi \models \mathbf{\bar{F}}t_1 \Leftrightarrow$ for infinitely many distinct i , $\pi^i \models t_1$
17. $M, \pi \models \mathbf{\bar{G}}t_1 \Leftrightarrow$ for almost every distinct i , $\pi^i \models t_1$

4.7.4 Minimal Set of Operators

The operators \vee, \neg, X, U, E are enough to express all the possible CTL^* formulas. [18–20, 24, 25, 27]. Using the following equivalences we can simplify the syntax and semantics of CTL^* at the cost of decreasing readability. Assuming f_1 and f_2 are CTL^* formulas:

- $f_1 \wedge f_2 \equiv \neg(\neg f_1 \vee \neg f_2)$
- $f_1 R f_2 \equiv \neg(\neg f_1 U \neg f_2)$
- $Ff_1 \equiv True U f_1$
- $Gf_1 \equiv \neg F \neg f_1$
- $\tilde{F}f_1 \equiv GFf_1$
- $\tilde{G}f_1 \equiv \neg \tilde{F} \neg f_1$
- $A(f_1) \equiv \neg E(\neg f_1)$

4.7.5 Examples

Some example CTL^* formulas are as follows:

- $AF p$ — in all the paths of the system, p eventually holds.
- $E(AX p)$ — there exists states where for all the next states p is true.
- $AG (AF p \vee E(AX p))$ — the disjunction of the above formulas is true for all the states on all the paths of the system.

4.8 CTL^* Branching-time Temporal Logic— CTL

4.8.1 Introduction

As mentioned earlier, CTL is the branching time sub-logic of CTL^* [20]. Clarke and Emerson introduced CTL and CTL model checking in [21]. More papers which refer to CTL are [13, 18, 19, 26, 27, 37]. In CTL we restrict CTL^* by imposing a rule on temporal operators. A path quantifier E and A must lie in front of every temporal operator F, G, R, U and X at all times. Needless to say in CTL

\tilde{F} and \tilde{G} are not allowed due to this restriction. Hence in order to obtain CTL syntax from CTL^* syntax we must change the following rule:

- If s and t are path formulas, then $\neg s, s \wedge t, s \vee t, \mathbf{X} s, \mathbf{F} s, \mathbf{G} s, s \mathbf{U} t, s \mathbf{R} t, \tilde{F} s, \tilde{G} s$ are path formulas.

into

- If s and t are path formulas, then $\neg s, s \wedge t, s \vee t, \mathbf{X} s, \mathbf{F} s, \mathbf{G} s, s \mathbf{U} t, s \mathbf{R} t$ are path formulas.

and add this rule to the path formula rules:

- If s and t are state formulas, then $\mathbf{X} s, \mathbf{F} s, \mathbf{G} s, s \mathbf{U} t$ and $s \mathbf{R} t$ are path formulas. [20]

Moreover, the semantics of CTL^* must be altered by removing the last two inductive rules concerning \tilde{F} and \tilde{G} .

The final result of these restrictions is that in CTL formulas are composed of the path quantifier—temporal operator pairs: $\mathbf{AX}, \mathbf{EX}, \mathbf{AF}, \mathbf{EF}, \mathbf{AG}, \mathbf{EG}, \mathbf{AR}, \mathbf{ER}, \mathbf{AU}$ and \mathbf{EU} . The meaning of these operator pairs [13, 19, 21] is as follows:

- $\mathbf{AX}f_1$ — the formula f_1 holds in every state following the current state.
- $\mathbf{EX}f_1$ — the formula f_1 holds in one or more states following the current state.
- $\mathbf{AF}f_1$ — the formula f_1 eventually holds in all the paths originating from the current state.
- $\mathbf{EF}f_1$ — the formula f_1 eventually holds in one or more paths originating from the current state.
- $\mathbf{AG}f_1$ — the formula f_1 holds on every state of every path starting from the current state.
- $\mathbf{EG}f_1$ — the formula f_1 holds on every state of one or more paths starting from the current state.
- $\mathbf{A}[f_1 \mathbf{R} f_2]$ — for all paths starting from the current state, f_2 is true in the states of the path. It is possible that at a state on the path f_1 is also true. If this occurs f_1 , “releases” f_2 and f_2 no longer remains true in the subsequent states.

- $E[f_1 R f_2]$ — for one and more paths starting from the current state, f_2 is true in the states of the path. It is possible that at a state on the path f_1 is also true. If this occurs f_1 , “releases” f_2 and f_2 no longer remains true in the subsequent states.
- $A[f_1 U f_2]$ — for all paths starting from the current state, f_2 is true in the path following an initial prefix of states in which f_1 is true.
- $E[f_1 U f_2]$ — for one or more paths starting from the current state, f_2 is true in the path following an initial prefix of states in which f_1 is true.

4.8.2 Syntax

The syntax of CTL may be described in terms of CTL^* syntax as shown above. However it is possible to obtain a more compact syntax by using just three of the possible path quantifier—temporal operator pairs (**EX**, **EG** and **EU**) and then represent the rest of the pairs (**AX**, **AG**, **AU**, **EF**, **AF**, **ER** and **AR**) using a number of identities [13, 37]. The compact syntax is as follows:

- Every atomic proposition $p \in AP$ is a CTL formula.
- If f_1 and f_2 are CTL formulas, then $\neg f_1$, $f_1 \wedge f_2$, **EX** f_1 , $E[f_1 U f_2]$ and **EG** f_1 are CTL formulas as well.

Assuming f_1 and f_2 are CTL formulas, the rest of the operator pairs can be expressed in terms of **EX**, **EG** and **EU** as follows [13, 20, 37]:

- $AXf_1 = \neg EX(\neg f_1)$
- $EFf_1 = E[True U f_1]$
- $AGf_1 = \neg EF(\neg f_1)$
- $AFf_1 = \neg EG(\neg f_1)$
- $A[f_1 U f_2] \equiv \neg E[\neg f_2 U (\neg f_1 \wedge \neg f_2)] \wedge \neg EG\neg f_2$
- $A[f_1 R f_2] \equiv \neg E[\neg f_1 U \neg f_2]$
- $E[f_1 R f_2] \equiv \neg A[\neg f_1 U \neg f_2]$

4.8.3 Semantics

In this section we give the semantics of CTL formulas for the more compact syntax presented in the previous section. This is done in terms of a Kripke structure as was done previously with CTL^* formulas. We use the notation: $M, s \models f$, which means that the CTL formula f holds at state s in the Kripke structure M . Assuming f_1 and f_2 are CTL formulas, the \models relation is inductively defined as follows [19]:

1. $M, s \models p \iff p \in L(s)$
2. $M, s \models \neg f_1 \iff M, s \not\models f_1$
3. $M, s \models f_1 \vee f_2 \iff M, s \models f_1 \text{ or } M, s \models f_2$
4. $M, s \models f_1 \wedge f_2 \iff M, s \models f_1 \text{ and } M, s \models f_2$
5. $M, s \models \mathbf{EX} f_1 \iff \text{for some state } t \text{ such that } (s, t) \in R, M, t \models f_1$
6. $M, s \models \mathbf{EG} f_1 \iff \text{for one or more paths of the form } (s_0(=s), s_1, s_2, \dots),$
 $\forall i[i \geq 0 \wedge M, s_i \models f_1]$
7. $M, s \models \mathbf{E}[f_1 \mathbf{U} f_2] \iff \text{for one or more paths of the form } (s_0(=s), s_1, s_2, \dots),$
 $\exists i[i \geq 0 \wedge M, s_i \models f_2 \wedge \forall j[0 \leq j < i \rightarrow M, s_j \models f_1]]$

4.8.4 Examples

Some example CTL formulas are as follows:

- $\mathbf{EF} p$ — there exists a path where eventually p is true.
- $\mathbf{AG} (p \Rightarrow \mathbf{AF} q)$ — it is always the case that if p is true, q is always eventually true.
- $\mathbf{EX} (\mathbf{AX} p)$ — there exists a state following the current one where in all the next states p is true.

4.9 CTL^* Linear-time Temporal Logic— LTL

4.9.1 Introduction

Clarke et al. [20] tell us that LTL as a temporal logic was first introduced by Pnueli and that it can be derived as a sub-language from CTL^* by restricting the former's rules to quantify over all paths of the system. Due to this restriction, LTL formulas cannot use the existential path quantifier E . Instead they allow only the universal path quantifier A which is sometimes just implied and not written as part of the formula [27]. Unlike CTL , LTL allows us to combine temporal operators together (such as for example AGF) and hence it cannot be defined just in terms of operator pairs like CTL . This latter fact LTL shares with its parent language CTL^* .

4.9.2 Syntax

We define the syntax of LTL in terms of the syntax of CTL^* as per Clarke et al. in [20]. LTL formulas take the form Af where f is a restricted path formula. The restriction imposed on this formula is that its state subformulas are only atomic propositions, i.e.:

- If $p \in AP$, then p is a path formula.
- If s and t are path formulas, then $\neg s$, $s \vee t$, $s \wedge t$, Xs , Fs , Gs , $sU t$, and $sR t$ are path formulas as well [20].

4.9.3 Semantics

The semantics of LTL formulas are once more given in terms of a Kripke structure. The relation \models can be defined inductively using the same CTL^* semantic rules 8–17 shown in Section 4.7 and changing rule 7 as follows:

$$7. \quad M, \pi \models p \quad \Leftrightarrow \quad s \text{ is the first state of } \pi \text{ and } p \in L(s)$$

4.9.4 Example

Some example LTL formulas are as follows:

- $\mathbf{AX} p$ — for all next states p is true.
- $\mathbf{A}(p \mathbf{U} q)$ — for all paths, p is true until q is true.
- $\mathbf{A}(\mathbf{FG} p)$ — for all paths, eventually there is a state where p is true and from then onwards p is true.

4.10 μ -Calculus

4.10.1 Introduction

The μ -calculus was introduced by Kozen as a general purpose logic system which extended propositional modal logic¹ by the least fixpoint operator μ and consequently the greatest fixpoint operator ν by duality [5, 20]. By means of the fixpoints of μ -calculus we are able to reason about a set of states at a time. The μ -calculus was later on applied to model checking in various ways:

- Various temporal logics used in model checking can be encoded in μ -calculus [20].
- Efficient model checking algorithms exist for μ -calculus [20].
- The μ -calculus can be itself viewed as a temporal logic which subsumes other previously defined temporal logics (see references found in [5]).
- Since temporal logics can be translated into μ -calculus and model checking algorithms for μ -calculus are very efficient it is possible to perform efficient automatic verification of temporal logics by first translating them into μ -calculus [20].
- The μ -calculus aids in symbolic model checking making use of OBDD representation. As mentioned in previous sections, an OBDD represents the set of states which are true for the particular relation. It is thus extremely useful in symbolic model checking to reason about such sets of states using μ -calculus [20].

Many various forms of μ -calculus exist [20]. Two such calculi are Park's and Kozen's used in [13, 37] and [5, 9, 10, 20] respectively. Both of these have been applied to model checking in most of the papers just mentioned. In the following sections we will provide the syntax and semantics

¹Modal logic is the study of "necessity" and "possibility". Temporal logic is often considered to be modal logic applied to the notion of time.

of Kozen's mu-calculus as given by Clarke et al. [20] which is very similar to that found in [5]. The notation used in [9, 10] differs slightly but the syntax and semantic provided in these papers should result in the same temporal logic language.

The syntax and semantics of μ -calculus are given in terms of a transition system such as a Kripke structure. However, the definition of the Kripke structure given earlier is changed from $M = (S, S_0, R, L)$ to $M = (S, T, L)$. Instead of the transition relation R we will use a set of transition relations T . Each element in T , referred to by a , is a transition. Formally M is defined as:

- S is a nonempty set of states.
- T is a set of transitions, such that every transition $a \in T, a \subseteq S \times S$.
- $L : S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions which are true in that state. [20]

Apart from the Kripke structure we also require a set of relational variables $RV = \{Q_1, Q_2, Q_3, \dots\}$. Each relation variable is assigned a subset of S [20].

4.10.2 Fixpoint Representation

Before we present the syntax and semantics of μ -calculus we describe what the fixpoint operators μ and ν represent in terms of the Kripke structure $M = (S, R, L)$ described earlier, or more importantly in terms of the set of states S of the system. As we mentioned earlier we are interested in reasoning about sets of states which are true for a particular temporal logic formula as this allows us to use said formula to verify properties about a system. One should also remember that a set of states can be represented by a formula which is satisfied by such states.

The building of the required set of states which satisfies a temporal logic formula is not done in one step. Instead an iterative method is used to build it by using a recursive definition. Initially, the recursive definition provides us with a predicate which represents a rough approximation of what the required set of states is. Building on this rough approximation, we refine the predicate (through the recursive definition) and subsequently add or remove states to the current set of states so as to create a new set which satisfies the new refined formula. In this way our approximation becomes increasingly more accurate and in line to the actual required temporal logic formula. When the

approximations no longer cause a change in the set of states, the required set of states has been obtained. The lack of change in the set of states due to the approximations is known as a fixpoint.

When building a fixpoint we usually start from the empty set or from the whole set S . Depending on the case we then respectively add states to or remove states from the set. The type of fixpoint reached depends on whether we start from the empty set or the set S . Starting from the empty set we reach the least fixpoint μ . Conversely, starting from the set of states we reach the greatest fixpoint ν .

As an example we will show how the CTL formula $\mathbf{EF}p$ can be expressed using fixpoints and show the approximations required to derive it. In terms of fixpoints $\mathbf{EF}p$ corresponds to the least fixpoint of $f_1 \vee \mathbf{EX}Z$, i.e.

$$\mathbf{EF}p = \mu Z. p \vee \mathbf{EX}Z$$

Since we are dealing with the case of an eventuality, and not something which is global it makes sense to start from something smaller, i.e. from the empty set. That will be our first approximation. In the case of $\mathbf{EG}p$ it would make more sense to start with the full set of states and then remove the unnecessary states. Returning to our $\mathbf{EF}p$ example, the approximations required are as follows:

- | | |
|--------------------|-------------------------------------------------------------------------------------------------------|
| 1st Approximation: | $\mathbf{EF}p \simeq \text{False}$ |
| 2nd Approximation: | $\mathbf{EF}p \simeq p$ |
| 3rd Approximation: | $\mathbf{EF}p \simeq p \vee \mathbf{EX}p$ |
| 4th Approximation: | $\mathbf{EF}p \simeq p \vee \mathbf{EX}(p \vee \mathbf{EX}p)$ |
| 5th Approximation: | $\mathbf{EF}p \simeq p \vee \mathbf{EX}(p \vee \mathbf{EX}(p \vee \mathbf{EX}p))$ |
| 6th Approximation: | $\mathbf{EF}p \simeq p \vee \mathbf{EX}(p \vee \mathbf{EX}(p \vee \mathbf{EX}(p \vee \mathbf{EX}p)))$ |
| 7th Approximation: | \dots |

The number of approximations required depends solely on whether the number of states in the set represented by the predicate have changed from one approximation to the next. Diagrammatically the sequence of approximations required to verify $\mathbf{EF}p$ on an example Kripke structure is shown in Figure 4.2.

Each step of our iterations can be viewed as a function which maps a member in the power set of S , $\mathcal{P}(S)$, to another member in $\mathcal{P}(S)$ or more formally, $\mathcal{T} : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$ [20]. This function is known as a predicate transformer since by changing an input set of states into another set of states

$$\mathcal{T}(\cap_i P_i) = \cap_i \mathcal{T}(P_i).$$

Due to the above requirements, Clarke et al. [20] tell us that by a result of Tarski², \mathcal{T} has:

- A least fixpoint, $\mu Z. \mathcal{T}(Z)$, since when \mathcal{T} is monotonic $\mu Z. \mathcal{T}(Z) = \cap \{Z | \mathcal{T}(Z) \subseteq Z\}$ and when \mathcal{T} is \cup -continuous $\mu Z. \mathcal{T}(Z) = \cup_i \mathcal{T}^i(\text{False})$.
- A greatest fixpoint, $\nu Z. \mathcal{T}(Z)$, since when \mathcal{T} is monotonic $\nu Z. \mathcal{T}(Z) = \cup \{Z | \mathcal{T}(Z) \supseteq Z\}$ and when \mathcal{T} is \cap -continuous $\nu Z. \mathcal{T}(Z) = \cap_i \mathcal{T}^i(\text{True})$. [20]

This provides us with a function which has a least and greatest fixpoint as required. Note that $\mathcal{T}^i(Z)$ denotes i applications of \mathcal{T} to Z [20].

4.10.3 Syntax

The syntax which generates μ -calculus formulas is as follows:

- If an atomic proposition $p \in AP$, then p is a formula.
- If a relation variable $Q \in RV$, then Q is a formula.
- If f_1 and f_2 are formulas, then $\neg f_1$, $f_1 \wedge f_2$ and $f_1 \vee f_2$ are formulas.
- If f_1 is a formula, and $a \in T$, then $[a]f_1$ and $\langle a \rangle f_1$ are formulas.
- If $Q \in RV$ and f_1 is a formula, then $\mu Q. f_1$ and $\nu Q. f_1$ are formulas. [5, 20]

The last syntactical rule is restricted by the fact that f_1 has to be syntactically monotone³[5, 20]. The formula $\langle a \rangle f_1$ means that it is possible to make a transition using a from the current state to a state where f holds. The formula $[a]$ means that all transitions using a from the current state lead to states where f holds. The operators μ and ν represent the least and greatest fixpoints, respectively.

4.10.4 Semantics

A formula f_1 generated by the rules above corresponds to a set of states written as $\llbracket f_1 \rrbracket_{Me}$, where M is a transition system and $e : RV \rightarrow 2^S$ is an environment. By [20] the recursive definition of $\llbracket f_1 \rrbracket_{Me}$ is:

²Refer to Clarke et al. [20] for further reading on Tarski's result.

³This means that Q occurs under an even number of negations in f_1 .

1. $\llbracket p \rrbracket_{Me} = \{s \mid p \in L(s)\}$
2. $\llbracket Q \rrbracket_{Me} = e(Q)$
3. $\llbracket \neg f_1 \rrbracket_{Me} = S \setminus \llbracket f_1 \rrbracket_{Me}$
4. $\llbracket f_1 \wedge f_2 \rrbracket_{Me} = \llbracket f_1 \rrbracket_{Me} \cap \llbracket f_2 \rrbracket_{Me}$
5. $\llbracket f_1 \vee f_2 \rrbracket_{Me} = \llbracket f_1 \rrbracket_{Me} \cup \llbracket f_2 \rrbracket_{Me}$
6. $\llbracket \langle a \rangle f_1 \rrbracket_{Me} = \{s \mid \exists t [(s, t) \in a \text{ and } t \in \llbracket f_1 \rrbracket_{Me}]\}$
7. $\llbracket [a] f_1 \rrbracket_{Me} = \{s \mid \forall t [(s, t) \in a \text{ implies } t \in \llbracket f_1 \rrbracket_{Me}]\}$
8. $\llbracket \mu Q.f_1 \rrbracket_{Me}$ is the least fixpoint of the predicate transformer $\mathcal{T}(W) = \llbracket f_1 \rrbracket_{Me}[Q \leftarrow W]$,
 $\llbracket \mu Q.f_1 \rrbracket_{Me} = \bigcup_i \mathcal{T}^i(False)$.
9. $\llbracket \nu Q.f_1 \rrbracket_{Me}$ is the greatest fixpoint of the predicate transformer $\mathcal{T}(W) = \llbracket f_1 \rrbracket_{Me}[Q \leftarrow W]$,
 $\llbracket \nu Q.f_1 \rrbracket_{Me} = \bigcap_i \mathcal{T}^i(True)$.

Where $e[Q \leftarrow W]$ is a new environment that is the same as e except that in we substitute Q by W and $\mathcal{T}^i(Q)$ is defined recursively as $\mathcal{T}^0(Q) = Q$ and $\mathcal{T}^{i+1}(Q) = \mathcal{T}(\mathcal{T}^i(Q))$ [20]. Since all logical connectives are monotonic apart from negation, the latter is restricted in use. In fact negation is used by pushing this operator down to the level of atomic propositions by means of a number of laws and equivalences:

- De Morgan's Laws
- $\neg[a]f \equiv \langle a \rangle \neg f$
- $\neg \langle a \rangle f \equiv [a] \neg f$
- $\neg \mu Q.f(Q) \equiv \nu Q. \neg f(\neg Q)$
- $\neg \nu Q.f(Q) \equiv \mu Q. \neg f(\neg Q)$

The above laws and equivalences help to guarantee the existence of fixpoints (see [20] for more information on this subject and where to find further reading) and may also be used to produce a more compact syntax and semantics for μ -calculus [5].

4.10.5 Fixpoint characterisation of CTL Formulas

As mentioned earlier in Section 4.10.2 μ -calculus formulas can be used to characterise the formulas of various other temporal logics formulas by means of fixpoints [9, 10, 20]. For example,

μ -calculus can provide the fixpoint characterisation of the operators of *CTL* [20, 21, 37] allowing *CTL* to be model checked using μ -calculus model checking. Since it subsumes other logics, it is able to characterise other temporal logic languages, however in our study we are mostly interested in the characterisation of *CTL* formulas which is achieved as follows:

- $\mathbf{AF}f_1 = \mu Z.f_1 \vee \mathbf{AX}Z$
- $\mathbf{EF}f_1 = \mu Z.f_1 \vee \mathbf{EX}Z$
- $\mathbf{AG}f_1 = \nu Z.f_1 \wedge \mathbf{AX}Z$
- $\mathbf{EG}f_1 = \nu Z.f_1 \wedge \mathbf{EX}Z$
- $\mathbf{A}[f_1 \mathbf{U} f_2] = \mu Z.f_2 \vee (f_1 \wedge \mathbf{AX}Z)$
- $\mathbf{E}[f_1 \mathbf{U} f_2] = \mu Z.f_2 \vee (f_1 \wedge \mathbf{EX}Z)$
- $\mathbf{A}[f_1 \mathbf{R} f_2] = \nu Z.f_2 \wedge (f_1 \vee \mathbf{AX}Z)$
- $\mathbf{E}[f_1 \mathbf{R} f_2] = \nu Z.f_2 \wedge (f_1 \vee \mathbf{EX}Z)$

The operator pairs **AF**, **EF**, **AU** and **EU** deal with eventualities and so are characterised using a least fixpoint. Greatest fixpoints on the other hand are used in the case of operators that cause a property to hold forever such as **AG**, **EG**, **AR** and **ER** [20].

4.11 Other Temporal Logics

Various other temporal logics exist apart from the ones we discussed here. Some of these are derived from *CTL** and *CTL* such as *ACTL** and *ACTL*, respectively. Others such as alternating-time temporal logic (*ATL*) differ from other logics by their very nature.

- *ACTL** — *ACTL** is *CTL** restricted to universal path quantifiers. An example of an *ACTL** formula is **AGF** p .
- *ACTL* — *ACTL* is *CTL* restricted to universal path quantifiers. An example of an *ACTL* formula is **AF AG** p .

- *ATL* — Alternating-time temporal logic is of special interest to us even though it was not included in our study. As a logic it “offers selective quantification over those paths that are possible outcomes of games” [2][p.672] and can be viewed as the generalisation of existential and universal branching time logic [2]. *ATL* considers concurrent games in which at each state all players make their moves and the successor state is determined by combining all the players’ moves. Special cases of such concurrent games are: turn-based synchronous, turn-based asynchronous and Moore synchronous [2] of which the most interesting to us are turn-based synchronous since in these at each state one player has a choice of moves and the player at hand is determined by the state itself. An interesting application of this *ATL* model type would be turn-based board games which we explore in our study.

4.12 Conclusion

This chapter has introduced various temporal logics which may be used for the specification of properties. The temporal logics we discussed may be divided into branching-time (example *CTL*) and linear-time (example *LTL*) with the exception of *ATL* which introduces another form of logic called alternating-time logic. Apart from this we also introduced μ -calculus which we know supersedes *CTL* and *LTL* in expressiveness. Since our study compares *CTL* and μ -calculus we have focused on these two logics. *CTL** and *LTL* were also introduced due to their close relationship to *CTL*.

As we shall see in the following chapter the properties written using a temporal logic may be automatically verified on a model of a system if the appropriate algorithms are used. The following chapter discusses these algorithms and also introduces a number of model checkers which implement such algorithms.

VERIFICATION

*The meaning of a proposition is
the method of its verification.*

Moritz Schlick

5.1 Overview

We will now briefly consider the last stage of model checking, i.e. verification. In this stage an algorithm is used to automatically verify a specified property on a modelled system by considering all possible computation paths. If the property is satisfied by the model, model checking will indicate this fact. Otherwise using appropriate algorithms, if possible, counterexamples or witnesses are produced accordingly. If an error trace is provided, this can help locate the source of error in the model and hence the actual system. In this way the errors found in the system may be resolved. This procedure can be carried out until the required specification formulas are satisfied and hence the system is verified through the model.

5.2 Introduction

For model checking-based verification to be automatic we need an algorithm which given a structure (the model) representing a system and a temporal logic formula expressing a property of the structure, is able to find the part of the structure (more specifically the set of states in the

structure) which satisfies the temporal logic formula. More formally, given the Kripke structure $M = (S, S_0, R, L)$ and the temporal logic formula f an algorithm is required which computes the set $\{s \in S \mid M, s \models f\}$ [20].

5.3 Enumerative Model Checking

In model checking algorithms, the structure upon which the algorithm works is often a graph. In enumerative model checking the graph is one where the nodes represent the states of the Kripke structure while directed arcs going from one node to another node represent the transitions of the Kripke structure. Moreover, the labelling function L of the Kripke structure is represented in the graph by labels on the nodes which reflect this function. The algorithm for enumerative model checking works by labelling the nodes in the graph which satisfy the temporal logic formula, hence allowing us to obtain the required set of states. The approach used is iterative and breaks down the temporal logic formula we require into simpler, nested formulas, which we model check one at a time, working progressively outwards. Once the outer formula is reached we have checked the actual formula we had set out for in the first place[20]. For example consider $\mathbf{AF}(p \rightarrow (q \wedge r))$. This formula is checked by first finding the states which satisfy p , q and r , then $q \wedge r$, then $(p \rightarrow (q \wedge r))$ onwards until the actual formula is model checked.

5.4 Symbolic Model Checking

For symbolic model checking using OBDDs the structure is also a graph. As described earlier the graph represents the transitions of the system. Each node represents a variable of the characteristic function representing the transition relation. The left and right outgoing directed arcs from each node represent the node's variable assignment to 0 and 1 respectively. The OBDD graph satisfying the temporal logic formula f is the result we wish to obtain from a symbolic model checking algorithm. Such an OBDD would represent the required set of states which satisfy our temporal logic formula. Symbolic model checking achieves this through a algorithm which uses fixpoints to recursively obtain an OBDD representing the set of satisfied states.

5.5 Counterexamples and Witnesses

Model checking has the ability to provide us, when possible, a counterexample or witness trace. A counterexample trace can be obtained when a property involving universal quantification is false. The trace contains states of the model visited before reaching a state which does not fit the specification (an error state), i.e. it proves that the negation of the property is true and hence the property is not true for all computation paths as required by universal quantification. A witness trace is obtainable when a property involving existential quantification is true. It provides a computation path where the property is true, i.e. it proves that the property is true by an example where this is so. [20, 24]

5.6 Algorithms for Model Checking

Algorithms have been created to model check models specified using various temporal logics. These have been later on implemented in various model checkers. We refer the reader to a list of papers discussing possible model checking algorithms for various temporal logics:

- *CTL* Model Checking: Refer to [13, 18–21, 24, 26, 37]
- *LTL* Model Checking: Refer to [20, 24]
- *CTL** Model Checking: Refer to [20]
- μ -calculus Model Checking: Refer to [7, 8, 13, 20, 37]

5.7 Model Checking Tools

5.7.1 Introduction

In this section we will introduce and provide a brief overview of a few model checkers we have considered for our research. These are SPIN, Mocha, SMV and μ cke. Of these model checkers we will focus more on the latter two: SMV and μ cke as they provide us with the temporal logics we require for our study, *CTL* and μ -calculus respectively.

5.7.2 SPIN

SPIN is a model checker whose modelling language is based on another language called PROMELA and which proves properties of such models using specifications written in the linear temporal logic *LTL* [31]. As a model checking program it consists of various intercommunicating modules which parse a model written in the input modelling and specification language, verify it and provide, when possible, counterexample traces. SPIN is not based directly on work by Clarke and Emerson and by Sifakis and Queille mentioned earlier but on an extension of such work by Vardi and Wolper which makes use of Büchi automata (refer to [20, 31] for more information on where to follow-up on this work). The end result is still a complete check of the states of the global reachability graph hence resulting in performance issues if the state space is large. SPIN makes use of various optimisations to mitigate this including in the nested depth-first search used to perform the exhaustive search in the state space, efficient translation of *LTL* formulas to Büchi automata, a technique called partial order reduction which attempts to reduce the reachability graph and efficient memory management. We again refer the reader to [20, 31] for more information on these subjects.

5.7.3 Mocha

Mocha differs from other model checkers by the fact that it achieves its modelling by means of reactive modules instead of the traditional state-transition diagram. Reactive modules are modules which contain both states and transitions in them, which react to one another [3][p.2]. The reactive modules used to model a system are specified in a machine readable variant input language of the modules. The specification language on the other hand is *ATL* which we described in Section 4.11. Two verification methodologies are used: compositional verification and hierarchical verification [3]. A few of the functionalities provided by Mocha are: simulation, enumerative and symbolic invariant model checking, counterexample generation, compositional refinement checking, *ATL* model checking and reachability analysis of real-time systems. For more information regarding these topics please refer to [3].

5.7.4 SMV

Introduction

SMV is a model checker which comes in many different versions. It was originally developed by McMillan for his thesis on OBDD-based symbolic model checking [20, 37]. This version of SMV uses its own language for modelling and allows specifications using *CTL*. McMillan then produced another version while working at Cadence Berkley Labs which he named Cadence SMV. Cadence SMV adds some features to the original SMV while still retaining backwards compatibility. One of the most important of these features is *LTL* model checking [38, 39]. Cadence SMV has its own C-like language which differs from the one of the original SMV. Another version of SMV is NuSMV, developed by a collaboration between ITC-IRST, Carnegie Mellon University, the University of Genova and the University of Trento. As a model checker NuSMV contains various new additional functionalities to SMV such as SAT-based model-checking and allows specifications to be written in various languages, amongst which are *LTL* and *CTL*. Its modelling language is very similar to the one of the original SMV. Refer to [14–17] for more technical details. In the following sections we will focus on the Cadence version of SMV as this is the model checker we used for our study.

Techniques

SMV implements a number of algorithms [37]:

- Algorithms for the computation of the reachable set of states in one step from a set of states.
- Algorithms that avoid the construction of the global transition relation such as incremental transition relation generation, partitioning and modified breath-first search.
- Algorithms to speed up the computation such as forward analysis and frontier set simplification.
- Algorithms for counterexample generation.

Features

Cadence SMV has a number of interesting features. Some of the more prominent ones are [20, 39]:

- **Modules** — The model of the system may be broken down into separate modules which can have multiple instances at any one time. Such modules are able to see each other's variables according to standard hierarchical visibility rules. Modules may have state variables, input variables, output variables, transition relation definitions, *CTL* and *LTL* formula definitions and so on.
- ***CTL* and *LTL* model checking** — Cadence SMV allows the specification of both *LTL* and *CTL* properties since it implements the algorithms required to model check systems specified using these languages.
- **Synchronous composition** — SMV assumes synchronous component transition, that is, that all components of the system change state together in a single step.
- **OBDD modelling** — Cadence SMV employs symbolic model checking using OBDDs.
- **Interleaved composition** — Using an appropriate keyword it is possible to stop synchronous composition and force interleaving of processes, whereby only one component of the system changes its state in a transition.
- **Deterministic and non-deterministic transitions** — The state transitions may be modelled both deterministically and non-deterministically depending on the need. Non-deterministic choices are useful when there is the need to model cases where more than one possible transition can occur starting from a certain system state.
- **Transition relations** — Transition relations can be specified both explicitly or implicitly. Explicitly defined transitions are written in terms of Boolean relations on the current and next state values of the components of the state. Implicit ones are written using assignment statements which occur in parallel and which define the values of next state of the system in terms of the value of the current state.
- **Set expressions** — SMV has a set expression which can be used to build sets according to need by means of a compact definition. This is especially useful when building a set which represents the possible non-deterministic choices based on the values of the state. This feature is only available in Cadence SMV and not in the original or NuSMV.

Language

Cadence SMV's ability to be backwards compatible with the original SMV made it our choice language with which to write our SMV models. Not only it allows *CTL* formulas to be written (which is a main interest to us), but it also allows us to use the set expression which we found most useful for our models. In this section we introduce some of the most important semantics of the expressions available in SMV without going much into the syntax of the language. For features which SMV shares with other languages and for a more thorough explanation of the syntax and semantics of Cadence SMV we refer the reader to [38, 39].

- **Data types and type declarations** — SMV's base type is the Boolean or the set $\{0, 1\}$ representing *False* and *True*, respectively. Using this type it is possible to construct more complex data types which are known as enumerated and ranged types. Enumerated types consist of sets of symbolic values while ranged types consists of ranges of integers. These types are internally represented as combinations of Boolean variables. Arrays and defined types (structs) similar to those found in C are also possible.
- **Expressions** — an expression consists of the required combination of the state variables and number of operator connectives such as: Boolean operators, conditional operators, arithmetic operators, comparison operators, set operators and so on.
- **Assignments** — allow the value of an expression to be assigned to a signal (state, input, output) variable. Assignments in SMV are carried out in parallel, that is they are all executed together in one step of the system unless otherwise specified appropriately.
- **Delayed Assignments** — the value of an expression to be assigned to the delayed (next) signal, that is the signal in the state following this one can be achieved using a specific operator, the "*next*" operator. In this way we can define the transitions of the state values for our model.
- **Non-deterministic Assignments** — are assignments which allow more than one value to be selected as the value of the signal at hand. The value is selected arbitrary from a set of values. This allows us to define models where the next state can take a number of equally likely

values. The model checker attempts the assignment of all such non-deterministic values during verification.

- **Modules** — these are definitions which group together related type declarations and assignments that, as explained before, can be reused more than once. Modules also have input and output parameter signals which allow it, when instantiated, to be plugged into the system as a complete stand alone component (much like a black-box).
- **Set Expressions** — these expressions are important as they allow us to compactly define sets which in turn may be used to allow the definition of non-deterministic assignments. A set expression may be either written as a list of elements or iteratively built using the construction $\{f(i) : i = x..y\}$ where $f(i)$ is an expression containing the parameter i and $x..y$ is an expression which represents a series of integers from x to y . The model checker then expands the construction into the set $\{f(x), \dots, f(y)\}$. For example: $\{i \times 2 : i = 1..5\}$ expands to the set $\{2, 4, 6, 8, 10\}$.
- **CTL Expression** — as their name implies, these expressions consist of *CTL* properties which we use to specify the model we wish to verify.

5.7.5 μ cke

Introduction

Armin Biere developed μ cke as a product of his dissertation on efficient μ -calculus model checking using OBDDs. According to [8], Biere developed μ cke as a general framework based on work by Burch et al. in [13]. Using such a framework it would be possible to describe properties which are not expressible in temporal logics such as *LTL* and *CTL*.

Techniques

The μ cke model checker has a number of techniques which were first utilised for model checkers using other temporal logics. These have been then adapted for μ -calculus model checking and implemented in μ cke. A few of these were mentioned earlier for the SMV Model Checker. Apart

from these μ cke adds algorithms for the automatic ordering of BDD variables. See [7, 8] for more information on these techniques and how they were integrated in μ cke.

Features

Some of the features included in the μ cke model checker are:

- **Class** — Synonymous to an SMV Module, the class contains the definition of a component of the system.
- **μ -calculus model checking** — The μ -calculus allows the specification of μ -calculus properties since it implements the algorithms required to model check systems specified using this temporal logic.
- **OBDD modelling** — The μ cke model checker employs symbolic model checking using OBDDs.
- **Initial state definition** — Initial states can be specified explicitly using a Boolean relation.
- **Transition relation definition** — The transition relation may be written explicitly using a Boolean relation in terms of the current and next state.

Language

The μ cke model checker has an input language which is very similar to that of C and C++. Again, as with SMV, we will present some of the possible language expressions allowed in μ cke without presenting the full syntax and semantics. For more information refer to [7, 8]:

- **Data types and type declarations** — μ cke's base type is the Boolean or the set $\{0, 1\}$ representing *False* and *True*, respectively. The base type allows us to construct more complex data types such as enumerated and ranged types. These are similar to the ones available for SMV. Arrays and defined types (structs) similar to those found in C are also possible.
- **Boolean Relations** — these allow us to define a set of states of the system such as for example the set of initial states.

- **Transition Relations** — these allow us to define the transitions of the variables of the system, hence allowing us to define the transition relation. In order to achieve this a Boolean relation is written in terms of the current and next state.
- **Lower and Upper Fixpoints** — by means of the upper and lower fixpoint reserved keywords, we are allowed to write Boolean Relations which have upper and lower fixpoints. These in turn allow us to write μ -calculus formulas which we use to specify the properties of the system.
- **Classes** — these are definitions which group together related type declarations and assignments that, as explained before, can be reused more than once. Modules also have input and output parameter signals which allow it, when instantiated, to be plugged into the system as a complete stand alone component (much like a black-box).

5.8 Conclusion

This chapter has focused on the modelling stage of model checking. We have seen how verification in model checking may be automated by means of appropriate algorithms for the relevant temporal logics. Moreover we have discussed some model checkers which implement the algorithms for *CTL* (SMV, NuSMV), *LTL* (Cadence SMV, NuSMV), *ATL* (mocha) and μ -calculus (μ cke).

In the last three chapters we have taken a detailed look at model checking by exploring each of its stages. We have seen how modelling may be achieved by means of both an enumerative representation and symbolical one. We have also explored how specification of properties about such models may be achieved using temporal logics such as *CTL*, *LTL*, *CTL** and μ -calculus. Finally, we briefly considered how verification may be automated using appropriate algorithms implemented in a number of model checkers.

In the next chapter we will show how modelling and specification may be applied for a particular type of system: game systems. We show how a game may be easily translated into a model by the use of a Kripke structure, how properties may be written for such models using the temporal logics mentioned earlier and hence how they can be automatically verified by the use of a model checker. As an example we make use of a very simple game which we call “tictac”. Tictac is basically a

simpler version of the tictactoe game which we use as an illustration of how model checking may be applied to games.

GAMES

You have to learn the rules of the game.

And then you have to play better than anyone else.

Albert Einstein

6.1 Overview

This chapter introduces the main idea proposed in this body of work. Building on the knowledge acquired in the previous chapter we first show how a game can be viewed as a system with states and transitions, hence allowing it to be modelled. We then show how properties about a game may be specified so that the model can be verified by a model checker such as SMV or Mucke. As a proof of concept we introduce a very simple game and show how it can be modelled and specified in preparation for verification using the model checkers just mentioned.

6.2 Introduction

The type of games that we will consider in our discussion are standard board games which can be played by two players such as tictactoe, connect four, chess, checkers and bridget, amongst many others. In all such games, the main objective is for a player to achieve a goal before the opponent does. This can be accomplished by making strategic moves on the board which are in the favour of the player or which hinder the opponent. Each move tries to make the prospect of reaching the

goal more probable, or tries to hinder the opponent by reducing the probability of them reaching their goal.

Every game in our category has its own set of rules regarding how the initial setup of the board is, what moves can be done by the players to lead to a goal and what the goal of the game is in terms of the state of the board. More often than not the initial setup of the board is singular and predetermined, such as for example in tictactoe the board consists of an empty three by three grid, while in chess the chess-pieces are set up symmetrically on each side of the board, mimicking a battle about to start. The moves which can be done on the board vary from game to game. Games like tictactoe and connect four allow moves which introduce board pieces on the board as the game proceeds while others such as chess and checkers start with a fixed number of pieces which are moved around the board and removed when captured. The goals of the games differ depending on the games themselves and whether such goals have been reached or not can be deduced directly from the board's configuration during gameplay. For example in tictactoe the goal is to have three markers representing a player in a row which is immediately visible from the board. In chess the goal is to immobilise the King piece such that every possible next move causes the piece to be captured. This too can be deduced from the board. In the next section we will describe how the above games can be viewed as finite state systems which can be modelled and specified for verification using model checking.

6.3 Modelling Game Systems

In order to model check a game one must first translate it into a representation which can be accepted for model checking such as a Kripke structure. Translation of a game into a Kripke structure is effortless if one views the game as any normal system with its own states and transitions. The only requirement is to map components of the game to components of the Kripke structure (the state space S , initial states S_0 , transition relation R and labelling function L) and provide an adequate set of atomic propositions of the form *variable = value* related to games such as for example *turn = player1* which means that it is currently player 1's turn or *location_12 = red* which means that location 12 on the board is marked with a red marker.

Before moving on to the state space, we first require a representation of a single state ($s \in S$) of the

model. As explained earlier a state consists of a set of variables whose values are what is needed to define the system at any point in time. In the case of games the variables required for such a state would naturally hold the configuration of the game board (what each of the board locations contain or the position of the pieces on the board) and the next player to make a move (the player with the turn). These values are enough because had the game required to be stopped and then resumed at a later time, they would allow the game to proceed as if the interruption had never occurred. Hence the combination of game board configuration variables and player turn variable will form the state of the Kripke structure representing our game. We will from now on refer to such a state as a *game state*. Moreover, when we refer to game state variables we imply the variables of the game state, that is, the game board configuration variables (usually a variable for each board location where the value of the variable represents the marker or piece currently in or on the board location) and the player turn variable.

The state space S contains all the possible states of the system, both valid and invalid ones. In terms of game systems the state space translates to all the possible configurations the game's board and the player turn may be in, regardless of the game rules. Hence, some of these combinations of game board and player turn are valid ones which can be obtained during normal game-play while others are invalid since they result from invalid moves which go against the game rules. At a more mathematical level the game state space implies the cross product of the domains of the variables representing the board locations and the variable representing the player turn.

The initial states of a system, denoted by S_0 , consist of a set of valid states from which the system starts to operate. When players decide to play a game, before the first move is done the board game is set up and the first player is chosen. As mentioned earlier, initial board setup usually consists of an empty game board (e.g. tictactoe and connect four) or a fixed positioning of the game pieces on the board as according to the game rules (e.g. checkers and chess). The first player to make a move on the board is chosen at random or by a simple game rule (such as White always begins with some games such as chess). We can regard these combinations of the initial board configuration and the player chosen to start the game, as the set of initial states of our game system, S_0 . Moreover, this set of initial game states is a subset of the game state space.

If the model is written correctly the transition relation R of the Kripke structure shows us how we can move from one valid state to another valid state. Translating this to game systems, we

require something which takes us from one valid game state to another possible valid game state in accordance to the game rules. In games such a transition from one game state to another is known as a player move. If in the current game state a player has their turn, they will decide which move to make from those which are available to them as according to the game rules. In this way they alter the board configuration and change the turn to their opponent. In doing so they take the game to the next game state. We can thus simply say that for a Kripke structure modelling a game, a player's move which follows the rules of the game corresponds to a transition defined in the transition relation of the Kripke structure.

While playing a game the players can deduce the value of the board locations and the player turn at any one time by viewing the board itself and by knowing who made the last move respectively. This provides them with a complete overview of the game's status. Such an action can be viewed as the labelling function L for our game-oriented Kripke structure, since given a game state such a function can give a complete description of the game state by providing its respective atomic propositions.

The systems modelled by Kripke structures lead to infinite computation trees. Each computation path in such trees can be seen as a game session where starting from the initial board setup the players take turns to make moves on the board. It is however important to notice that while in Kripke structures such paths are infinite in length, in reality a game session is finite and eventually always reaches a point where there are no moves, either because a player has won or because the game is drawn, making any further moves illegal. We can model a finite game session using an infinite computation path by writing the transition relation in such a way that certain game states, when reached, repeat themselves indefinitely. Examples of such states are player winning game states or draw game states. When such game states occur, the successor game state is the state itself.

6.4 Specifying Game Systems

The specification of game systems is achieved in the same way as with any normal system, that is by using temporal logics. Temporal logics allow us to reason about games in the same way as with any other system. They allow us to verify properties such as whether a game is fair to allow both

players an equal chance of winning or if there are game sessions (paths) which allow a player to win in a certain amount of steps such that the opponent loses whatever move he or she makes. It is our interest to investigate how model checking fairs while trying to verify such properties and its ability to scale up to larger games.

As some simple examples consider the properties below:

- “All game sessions lead to Player 1 winning”, written in CTL as the formula

$$Player1AlwaysWin \stackrel{\text{def}}{=} \mathbf{AF} \text{ winner_player1}$$

First of all we must define what *winner_player1* is. This can be achieved by a formula in terms of the variables of the board’s locations, written according to the game rules. Using the **AF** operator pair we state that for all paths starting at the start state there is eventually a state on the path where *winner_player1* is true. For games to be interesting we would like this property to be false. If this is the case the model checker can provide us with a counterexample by proving the existence of a path where this property fails.

- “Player 1 can force a win in their favour in their third move”, written in CTL as the formula

$$Player1ForceWin3rdMove \stackrel{\text{def}}{=} \mathbf{EX AX EX AX EX} \text{ winner_player1}$$

The formula *winner_player1* is defined as previously. Using alternation of the operator pair **EX** and **AX** we can describe game play in such a way that the former signifies a possible play by Player 1 while the latter means all the possible moves by Player 2 (which are beyond the control of Player 1). The final result is of the type “does a move exist which whatever move my opponent does...” or more directly “is there a move which forces ...”. This alternation can be used a number of times to achieve properties which model check games to verify that a player can or cannot force a win by means of a number of possible moves. The last operator pair of the sequence **EX** is required as the finishing move, that is the move which actually wins the game session.

- “The game board configurations where Player 1 can force a win”, written in μ -calculus as the formula

$$Player1ForceWin \stackrel{\text{def}}{=} \mu Z. \mathbf{EX} \text{ winner_player1} \vee \mathbf{EX AX} Z$$

The subformula *winner_player1* is defined as previously and represents the set of states where it is true. The subformula $\mathbf{EX} \text{ winner_player1} \vee \mathbf{EX AX} Z$ enables us to recursively

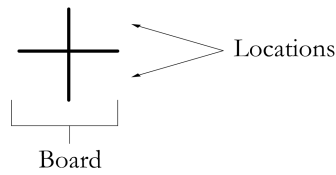


Figure 6.1: Board for Tictac

define Player 1 forcing a win in n moves, depending on how many times Z is nested. When Z is nested zero times *Player1ForceWin* refers to all the game states where Player 1 wins the game session in one move. When Z is nested once *Player1ForceWin* represents the set of states where either Player 1 can force a win or a move exists he/she can make where Player 2's moves are useless as they result in Player 1 winning (Player 1 forces the game session to end in his/her favour in two moves or less). If nested twice then Player 1 can force a win using three certain moves or less, and so on. The set obtained by the least fixpoint of such a formula is the set of states from which Player 1 can force a win.

6.5 A very simple game: Tictac

We will now describe a very simple game which we will use as a case in point to show how modelling can be achieved. Moreover we will specify some properties which can be written for such a game. The game is very trivial and provides no challenge for the players. However it allows us to show how a game may be modelled without going into details which are due to the game itself. We can thus focus on the translation of the game into a model (Kripke Structure).

Our game will be a slight variation of tictactoe which we will call *tictac*. The board consists of a grid of two by two which starts out as empty as shown in Figure 6.1. Similarly to tictactoe, two players alternately position their respective markers (such as for example, cross and circle) on the board until a line of two is achieved either horizontally, vertically or diagonally.

It can also be immediately deduced that the player who makes the first move wins the game as no matter what move his opponent does he/she still has two board locations to do the finishing move. Moreover, we realise that by the third move (first player's second move) the game ends with three pieces on the board. Finally it is impossible for the game to be drawn. Using temporal logic it is possible to verify that such properties are in fact true as we shall see in this section. An example

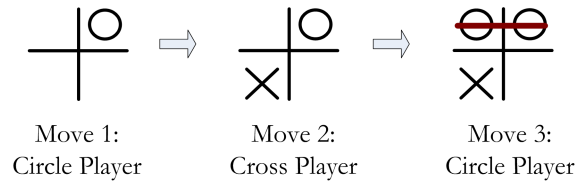


Figure 6.2: A complete game of Tictac. Circle Player wins horizontally.

complete game is shown in Figure 6.2. Here circle makes the first move and wins the game.

6.5.1 Modelling Tictac

Our game of tictac will be modelled as a Kripke structure $M_{tt} = S, S_0, R, L$ where S , S_0 , R and L have their usual meanings as described in Chapter 3. To describe the state of our system at any point in time we will use five variables: $board_1$, $board_2$, $board_3$, $board_4$ and $turn$. The variables $board_1$ to $board_4$ represent the board locations and range over the domain $D_b = \{empty, player1, player2\}$ since each board location can be either empty, marked by Player 1's marker or marked by Player 2's marker respectively. The variable $turn$ represents the turn and ranges over the domain $D_t = \{player1, player2\}$ for when it is Player 1's or Player 2's turn respectively. Hence the definition of M_{tt} is as follows:

- $S = D_b \times D_b \times D_b \times D_b \times D_t$.

As can be noted, even for such a simple system the state space is quite substantial. There are in fact $3 \times 3 \times 3 \times 3 \times 2 = 162$ possible states in the state space. Of these not all are valid. For example, some states involve four markers on the board while others have three markers of the same colour on the board. Such states can never be reached if the system is well modelled according to game rules.

- $S_0 = \{(empty, empty, empty, empty, player1)\}$.

This is the initial board state possible if we start with an empty board. As can be deduced we will assume that Player 1 always makes the first move.

- $R = \{((empty, empty, empty, empty, player1), (empty, empty, empty, player1, player2)),$
 $((empty, empty, empty, empty, player1), (empty, empty, player1, empty, player2)),$
 $((empty, empty, empty, empty, player1), (empty, player1, empty, empty, player2)),$
 $((empty, empty, empty, empty, player1), (player1, empty, empty, empty, player2)),$

$$\begin{aligned}
& ((empty, empty, empty, player1, player2), (empty, empty, player2, player1, player1)), \\
& ((empty, empty, empty, player1, player2), (empty, player2, empty, player1, player1)), \\
& \dots \\
& ((player1, player2, player1, empty, player2), (player1, player2, player1, empty, player2))\}
\end{aligned}$$

The transitions of R take the game state through valid transitions that lead to states which are correct according to the game rules. The first transition shown in the above set shows an example initial move by Player 1 where he/she picks the fourth board location. The last transition shown on the other hand is one where Player 1 has won the game. When this occurs the board location variables and turn variable retain their value indefinitely as the game session has ended and further moves are invalid.

- $L((empty, empty, empty, empty, player1)) = \{board_1 = empty, board_2 = empty, board_3 = empty, board_4 = empty, turn = player1\},$
 $L((empty, empty, empty, empty, player2)) = \{board_1 = empty, board_2 = empty, board_3 = empty, board_4 = empty, turn = player2\},$
 $L((empty, empty, empty, player1, player2)) = \{board_1 = empty, board_2 = empty, board_3 = empty, board_4 = player1, turn = player2\},$
 $L((empty, empty, player1, empty, player2)) = \{board_1 = empty, board_2 = empty, board_3 = player1, board_4 = empty, turn = player2\},$
 \dots

The labelling function L is defined in such a way that it labels each state with the atomic propositions true in that state. The first case for example labels the state with the atomic propositions which thus show that this state is an initial state as it fits the definition of S_0 .

The computation paths available for our small game are many. If we consider just the paths which start with Player 1 making the first move, there are four paths at first, which quickly branch out into twelve possible paths, each of which representing a possible game session given the two moves made by the players. The computation tree we are discussing can be viewed in Figure 6.3. Player 1 has the circle marker while Player 2 has the cross marker. If one examines the paths of Figure 6.3, especially the first four moves, on first glance one concludes that there are in fact four initial empty locations and hence four moves. However by symmetry we realise that in reality there is only one actual move and that the four possible ones are variations of this one move as shown in

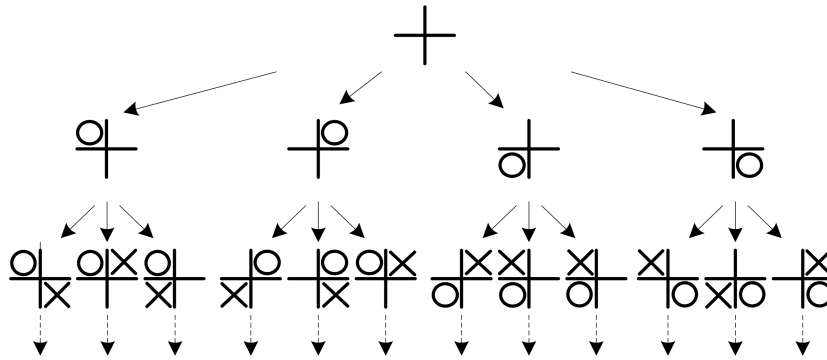


Figure 6.3: Initial Part of Computation Tree in a game of Tictac where Player 1 (Circle) makes the first move

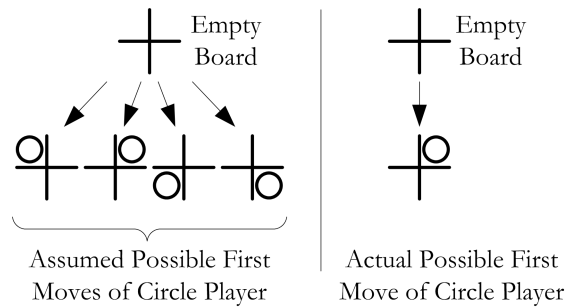


Figure 6.4: Assumed and Actual Possible First Moves of Player 1 (Circle) in Tictac

Figure 6.4. Hence, due to such symmetry it is possible to reduce the Kripke structure modelling the game system to a fraction of its initial size. Note that this does not reduce the state space, but merely decreases the valid states rendering the model more compact. Note however that it is possible with suitable encodings to somewhat reduce the state space, which is often important to reduce the problems of state explosion. The minimised Kripke structure is shown in Figure 6.5.

6.5.2 Specifying Tictac

We will now specify some properties which we can use as examples to model check our model. We will use circle to represent Player 1 and cross to represent Player 2. Again we assume that Player 1 always makes the first move.

Some properties:

- “Player 1 always wins” written in CTL as the formula

$$Player1AlwaysWins \stackrel{\text{def}}{=} \mathbf{AF} \text{ winner_circle}$$

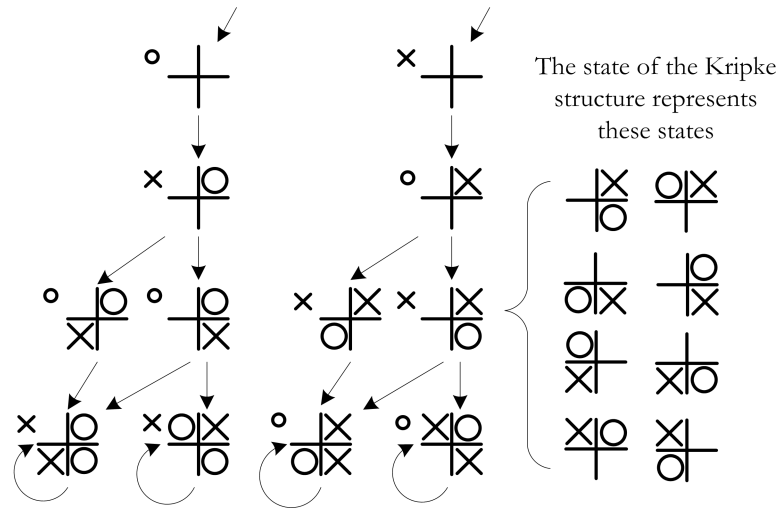


Figure 6.5: A Minimised Kripke Structure for Tictac. Some of the states shown represent more than one state which may be left out due to symmetry as shown with the example state on the right hand side of the figure. Player 1 uses the circle marker while Player 2 uses the cross marker.

where

$$winner_circle \stackrel{\text{def}}{=} ((board_1 = circle) \wedge (board_2 = circle)) \vee ((board_3 = circle) \wedge (board_4 = circle)) \vee$$

$$((board_1 = circle) \wedge (board_3 = circle)) \vee ((board_2 = circle) \wedge (board_4 = circle)) \vee$$

$$((board_1 = circle) \wedge (board_4 = circle)) \vee ((board_2 = circle) \wedge (board_3 = circle))$$

This definition represents the condition required for Player 1 to win a game session. Said definition, when proceeded by the operator pair **AF**, results in a formula which means that always eventually Player 1 wins. If this formula is verified to be true, Player 1 will therefore finally win the game at all times. If this is false it is possible for Player 2 to win.

- “Player 1 can force a win with the second move” written in CTL as the formula

$$Player1ForceWinSecondMove \stackrel{\text{def}}{=} \mathbf{EX AX EX winner_circle}$$

We define *winner_circle* as previously. The formula **EX AX EX winner_circle** states that there exists a move by the circle player which whatever moves the cross player does leads to a another (possible) move by the circle player which causes the latter to win the game (i.e. circle player can force a win in two moves). Proving that this is true ensures that we check that Player 1 can win in two moves. If this is false no move Player 1 can make forces Player 2 to lose in two moves.

- “There are game sessions which result in a full board (all locations are used)” written in CTL as the formula

$$GameEndsFullBoardExists \stackrel{\text{def}}{=} \mathbf{EF} \text{ board_full}$$

where

$$\begin{aligned} \text{board_full} \stackrel{\text{def}}{=} & (\neg(\text{board}_1 = \text{empty}) \wedge \neg(\text{board}_2 = \text{empty}) \wedge \\ & \neg(\text{board}_3 = \text{empty}) \wedge \neg(\text{board}_4 = \text{empty})) \end{aligned}$$

This definition represents the condition required for the board to be full. The formula $\mathbf{EF} \text{ board_full}$ means that there exists a path where eventually on this path all the board locations are marked by the players. If model checking this formula leads to a positive result it is possible for the model to reach such a game state, which according to our rules is not possible. If the result is negative then our model is correct and there are no game sessions which lead to a full board.

- “There are no games which end with a draw” written in CTL as the formula

$$GameEndsDrawGameNotPossible \stackrel{\text{def}}{=} \mathbf{AF} \neg \text{draw_game}$$

where

$$\text{draw_game} \stackrel{\text{def}}{=} (\neg \text{winner_cross}) \wedge (\neg \text{winner_circle}) \wedge \text{board_full}$$

This definition derives its value in terms of the previous definitions of *winner_circle*, *board_full* and a new definition *winner_cross* which is the equivalent form of *winner_circle* for the cross player. The meaning of this formula is that a draw game board is one where neither of the two players win and the board is full. We use the temporal operator pair \mathbf{AF} with the negation of *draw_game* which gives us a formula which means that in all paths starting from the start state it is eventually true that the game is not drawn. Checking this statement and finding it to be false means that it is possible to draw the game.

6.5.3 Verifying Tictac

To verify a game such as tictac we first translate its Kripke structure model into the equivalent code which a model checker can use. Model checkers, such as SMV or Mucke, use such code to internally build the required Kripke structure (or OBDD representation in the case of symbolic model checking). Specification is then done, also using the model checker’s constructs to write the

required formulas in the temporal logic language accepted by the model checker. Finally verification is done automatically by the model checker and if any specification about the game is false it provides a counterexample when this is possible. Using such a procedure it is possible to verify tictac and any similar board game.

6.6 Conclusion

As we have seen in this chapter, model checking can be used to reason and automatically prove various properties about various board games. Moreover using counterexamples and witnesses it is sometimes possible to automatically generate game sessions which show why the specified properties failed or why they are true. In the next chapters we will show how model checking fairs with games such as tictactoe and connect four. These games have a larger state space than tictac even when the board size used is the smallest possible. By scaling up the board sizes and attempting to proving various properties about such games we can see whether model checking is efficient when applied to games and proving properties about game systems. This is important because it is often the case that in such systems the state space grows quickly with every state variable introduced. We are interested in knowing whether enumerative model checking, or symbolic model checking for that matter, is able to cater for such systems in an adequate manner.

CASE STUDY 1: TICTACTOE

*Man is a game-playing animal,
and a computer is another way to play games.*

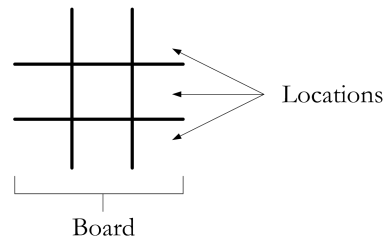
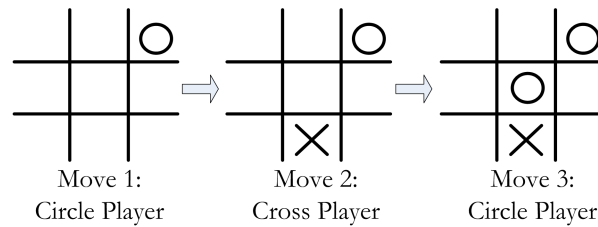
Scott Adams

7.1 Overview

This chapter aims at using the tictactoe game as a case study for the model checking of game systems. We will first introduce the game and its rules. Following this we show how the game can be modelled by means of a Kripke structure and specified by means of various properties using *CTL* and μ -calculus temporal logics. As a means of verification we employ SMV and μ cke as our model checkers for *CTL*-based and μ -calculus-based specifications respectively. Moreover we show how, by repeating the above exercise for tictactoe boards of different sizes, we can attempt to test model checking's ability to scale up to larger game systems.

7.2 Game Rules

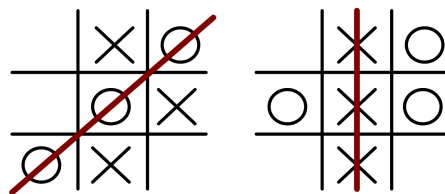
Tictactoe is a two-player game with players usually designated as cross and circle. Any other variation of the two may be used, such as cross and naught, and black and white. The game board typically consists of a matrix of 3×3 locations (as shown in Figure 7.1) but which can be generalised for $n \times n$ locations. A valid tictactoe game follows the rules below:

Figure 7.1: A standard 3×3 tic-tac-toe boardFigure 7.2: Example moves in a 3×3 tic-tac-toe board

1. The two players take turns to fill locations on the board.
2. In each turn a player takes up one of the locations and marks it with his own symbol (a cross or a circle respectively depending on the marker chosen).
3. Once a location is used up it cannot be used again in a subsequent turn.

Figure 7.2 shows some example moves of a typical tic-tac-toe game.

The main aim for the players in the game is to be the first player to produce a row of n of their own symbols either horizontally, vertically or diagonally. To do so, the players must make strategic moves that put themselves at an advantage over their opponent and increase their chance of winning. Two examples of finished games are shown for an 3×3 board in Figure 7.3. In the first example circle wins diagonally, while in the second scenario cross wins vertically.

Figure 7.3: Example winning boards in a 3×3 tic-tac-toe board

X	O	O
O	X	X
O	X	O

O	X	O
O	O	X
X	O	X

Figure 7.4: Example draw boards in a 3×3 tictactoe board

Apart from any of the two players winning, this game can also end in a draw game where the board locations are all used up but no one has achieved the winning condition of n markers in a row. Two examples of such cases are shown for a 3×3 board in Figure 7.4.

7.3 Modelling Tictactoe

In this section we will see how tictactoe may be modelled in terms of a Kripke structure and we will examine its respective computation tree.

7.3.1 Game State

Before moving on to the Kripke structure used to model tictactoe we first show how the state of the structure can be represented by the use of appropriate variables. In our discussion we will assume a tictactoe grid of size 3×3 . Note however that our model can be easily generalised for $n \times n$ board locations.

In order to represent a state s for our model of a 3×3 tictactoe game we require two items: a number of board variables ($3 \times 3 = 9$), each representing a board location and a variable to represent the current player turn. The values of such variables dictate the state of the model at any point in time. As stated earlier each of these states represent the game board and turn during a gaming session. More formally a state can be described as a duple

$$s = (locs, turn)$$

where,

- $locs$ is the set of nine board location variables. We denote each of these variables by $locs_i$,

$locs_1$	$locs_2$	$locs_3$
$locs_4$	$locs_5$	$locs_6$
$locs_7$	$locs_8$	$locs_9$

Figure 7.5: The positioning of $locs$ variables in a 3×3 tictactoe board

	X	O
X	O	
	X	

Figure 7.6: This is the board state represented by the formula $(locs_1 = empty) \wedge (locs_2 = player1) \wedge (locs_3 = player2) \wedge (locs_4 = player1) \wedge (locs_5 = player2) \wedge (locs_6 = empty) \wedge (locs_7 = empty) \wedge (locs_8 = player1) \wedge (locs_9 = empty)$. Player 1 is represented by the cross symbol while player 2 is represented by the circle symbol.

where i ranges from 1 to 9 (n^2 for a generic tictactoe). The type of each of these variables can take one of three values: one representing an empty board location (*empty*), one representing a board location marked by Player 1 (*player1*) and one representing a board location marked by Player 2 (*player2*). Hence each of these variables ranges over the domain defined by $D_b = \{empty, player1, player2\}$. It is important to notice for later on that i also represents where on the board the location may be found. Location 1 or $locs_1$ is at the upper left corner of the board while the last board location or $locs_9$ is at the lower right corner of the board. Figure 7.5 shows as a 3×3 tictactoe board and how $locs_1$ to $locs_9$ are positioned on the board. Thus to represent the board's state in the current state all that we require is a valuation of the set of variables $locs$ from which we extract the equivalent formula, e.g. a possible tictactoe board state is the valuation $\langle locs_1 \leftarrow empty, locs_2 \leftarrow player1, locs_3 \leftarrow player2, locs_4 \leftarrow player1, locs_5 \leftarrow player2, locs_6 \leftarrow empty, locs_7 \leftarrow empty, locs_8 \leftarrow player1, locs_9 \leftarrow empty \rangle$ from which we derive the formula $(locs_1 = empty) \wedge (locs_2 = player1) \wedge (locs_3 = player2) \wedge (locs_4 = player1) \wedge (locs_5 = player2) \wedge (locs_6 = empty) \wedge (locs_7 = empty) \wedge (locs_8 = player1) \wedge (locs_9 = empty)$. This is equivalent to the board state shown in Figure 7.6 where Player 1 is represented by the cross symbol and Player 2 is represented by the circle symbol.

- *turn* is the variable which represents the current game state's turn, i.e. whose turn it is to

make a move on the board. The type of *turn* is like that for *locs*, however we remove the value *empty*, resulting in the domain defined as $D_t = \{player1, player2\}$. For example if in the current game state $turn = player1$, it is player 1's turn. Note that we assume that the player selects where they will mark the board in the current board state, but the actual change in the board locations, i.e. *locs*, is reflected in the next board state after the current one.

Hence an example of a possible game state is

$$\begin{aligned} & (locs_1 = empty) \wedge (locs_2 = player1) \wedge (locs_3 = player2) \wedge \\ & (locs_4 = player1) \wedge (locs_5 = player2) \wedge (locs_6 = empty) \wedge \\ & (locs_7 = empty) \wedge (locs_8 = player1) \wedge (locs_9 = empty) \wedge \\ & (turn = player2). \end{aligned}$$

7.3.2 Game Model

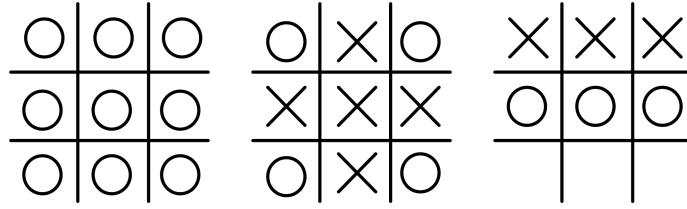
Having defined how a state of the Kripke structure can be represented we move on to the structure itself. As we have seen in Chapter 6 a board game such as a 3×3 tictactoe can be seen as a system and hence modelled as a Kripke structure. To achieve this we once more require a state space S , a set of initial (valid) states S_0 , a transition relation R and a labelling function L which together form the required Kripke structure

$$G_{ttt} = (S, S_0, R, L)$$

where,

$$- S = D_b^9 \times D_t.$$

Note that the power of 9 indicates the product of D_b nine times. This can of course be generalised to n^2 times for an $n \times n$ tictactoe. The state space of a 3×3 tictactoe contains $3^9 \times 2 = 39366$ states. Not all of these states are valid as they do not follow the game's rules. For example some states exist where the number of cross and circle markers differ by more than one. These states are illegal and can never be reached by the correct transition relation by starting from the initial board configurations. Other examples are the board states where all the board is filled with crosses or filled with circles. See Figure 7.7 for some invalid game states.

Figure 7.7: Some illegal board configurations in a 3×3 tictactoe board

- $S_0 = \{(empty, empty, empty, empty, empty, empty, empty, empty, empty, player1)\}$.

This is the initial board state which represents the game starting with an empty board. We assume that Player 1 always makes the first move. Whether Player 1 or Player 2 makes the first move is inconsequential for us as we shall see later on.

- $R = \{((empty, empty, empty, empty, empty, empty, empty, empty, empty, player1), (player1, empty, empty, empty, empty, empty, empty, empty, empty, player2))$
 $((empty, empty, empty, empty, empty, empty, empty, empty, empty, player1), (empty, player1, empty, empty, empty, empty, empty, empty, empty, player2))$
 $((empty, empty, empty, empty, empty, empty, empty, empty, empty, player1), (empty, empty, player1, empty, empty, empty, empty, empty, empty, player2))$
 $((empty, empty, empty, empty, empty, empty, empty, empty, empty, player1), (empty, empty, empty, player1, empty, empty, empty, empty, empty, player2))$
 \dots
 $((player1, player1, player1, player2, player2, empty, empty, empty, empty, player2), (player1, player1, player1, player2, player2, empty, empty, empty, empty, player2))\}$

The above shown transitions of R take the game state from a valid state to another valid state as according to the game rules. The first transition shown in the above set shows Player 1 picking the first board location, $locs_1$. The last transition shown depicts a state where its successor is the same state itself. This occurs upon reaching an end-game condition such as a win by either of the players or a draw game. In our case Player 1 has won. The state does not change because further changes in the board locations lead to illegal states outside of the game's rules.

- $L((empty, empty, empty, empty, empty, empty, empty, empty, empty, player1)) = \{locs_1 = empty, locs_2 = empty, locs_3 = empty, locs_4 = empty, locs_5 = empty,$

$$\begin{aligned}
& locs_6 = empty, locs_7 = empty, locs_8 = empty, locs_9 = empty, turn = player1\}, \\
& L((player1, empty, empty, empty, empty, empty, empty, empty, empty, player2)) = \\
& \{locs_1 = player1, locs_2 = empty, locs_3 = empty, locs_4 = empty, locs_5 = empty, \\
& locs_6 = empty, locs_7 = empty, locs_8 = empty, locs_9 = empty, turn = player2\}, \\
& \dots
\end{aligned}$$

The labelling function is defined as usually. It labels each state with the atomic propositions true in that state. This allows us to distinguish each of the different states of the state space through their label.

7.3.3 End-Game Definitions

Apart from the model we also require some definitions which we have already mentioned briefly before when we discussed repeating states. These definitions can be used to detect whether: the board is full, Player 1 has won, Player 2 has won, the game is a draw or the game has ended. All of these are defined in terms of the board location variables and are given respectively below:

$$board_full \stackrel{\text{def}}{=} \bigwedge_{i=1}^9 (locs_i \neq empty)$$

$$winner_player1 \stackrel{\text{def}}{=}$$

$$\begin{aligned}
& [(locs_1 = player1) \wedge (locs_5 = player1) \wedge (locs_9 = player1)] \vee \\
& [(locs_3 = player1) \wedge (locs_5 = player1) \wedge (locs_7 = player1)] \vee \\
& (\bigvee_{i=1,4,7} [(locs_i = player1) \wedge (locs_{i+1} = player1) \wedge (locs_{i+2} = player1)]) \vee \\
& (\bigvee_{i=1,2,3} [(locs_i = player1) \wedge (locs_{i+3} = player1) \wedge (locs_{i+6} = player1)])
\end{aligned}$$

$$winner_player2 \stackrel{\text{def}}{=}$$

$$\begin{aligned}
& [(locs_1 = player2) \wedge (locs_5 = player2) \wedge (locs_9 = player2)] \vee \\
& [(locs_3 = player2) \wedge (locs_5 = player2) \wedge (locs_7 = player2)] \vee \\
& (\bigvee_{i=1,4,7} [(locs_i = player2) \wedge (locs_{i+1} = player2) \wedge (locs_{i+2} = player2)]) \vee \\
& (\bigvee_{i=1,2,3} [(locs_i = player2) \wedge (locs_{i+3} = player2) \wedge (locs_{i+6} = player2)])
\end{aligned}$$

$$draw_game \stackrel{\text{def}}{=} (\neg winner_player1) \wedge (\neg winner_player2) \wedge draw_game$$

$$end_game \stackrel{\text{def}}{=} winner_player1 \vee winner_player2 \vee board_full$$

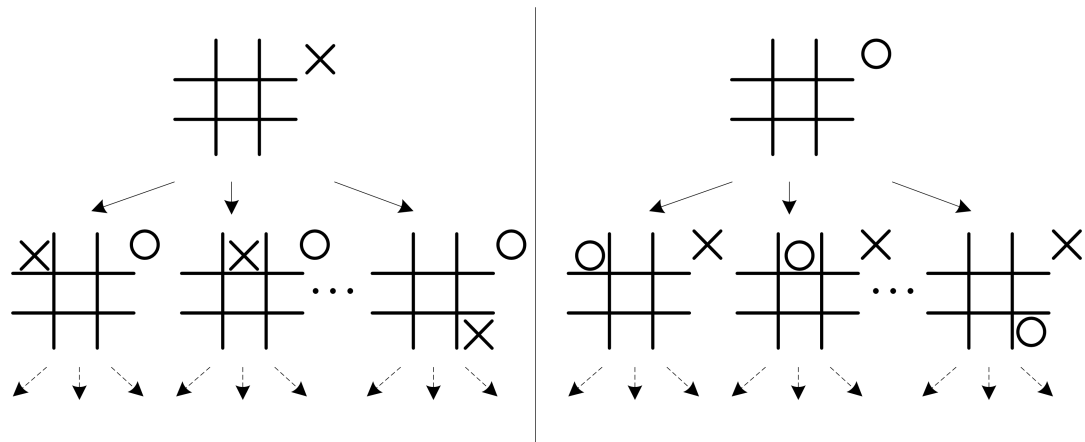


Figure 7.8: Initial part of computation tree of a 3×3 tictactoe board with two initial states, one for each of the players. The left computation tree shows the paths possible if game play starts with the cross marker, similarly on the right for the circle marker. As can be seen by inspection the trees are symmetrical.

7.3.4 Game Computation Tree

The finite Kripke structure model generates an infinite computation tree as with all systems modelled using such structures. Had we used two initial game states which differ in whom of the players makes the first turn the computation tree would be separated into two independent symmetrical sub-trees as shown in Figure 7.8.

Initially the two computation subtrees rooted at the initial game states start out with just one computation path each. From these states the computation paths quickly branch out in nine possible different computation paths each. There are nine computation paths because of the nine possible initial moves a player can make when presented with an empty board. Each of these nine branches further subdivide into eight possible paths and so on in a factorial fashion. This is due to the fact that after every move the choice of board locations available for marking diminishes. This of course occurs until an end-of-game state is reached in the Kripke structure modelling our game. Since the structure is defined in such a way that such states repeat themselves indefinitely and since all game sessions eventually must end due to the finite size of the board or due to a winning or draw condition, the paths of the computation tree eventually all should lead to infinite paths which no longer branch but where the states on the path are all repetition of the same state in the Kripke structure. The result of this is that both computation trees branch out quickly at first. This branching then slowly decreases until the tree fans out completely and branching no

longer occurs (after the board has filled out) or until an end-of-game state is reached which causes the branching to stop abruptly (before the board has filled out). Once this occurs the branches continue indefinitely, repeating the same game state.

Another important point to notice from tictactoe's computation tree if two start states were used is that due to the inherent symmetry and due to the fact that there are two separate sub-trees we can assume that Player 1 will make the first move thus reducing our model to half its size. Modelling the game in this way as opposed to choosing who makes the first move non-deterministically has no negative affect in our case, especially in terms of what specifications may be verified on the model. Both the model and specifications can easily be rewritten to swap the players thus allowing the other half of the computation tree to be verified separately.

7.3.5 Game Strategy

As with all board games, various game strategies exist for tictactoe which allow the players a better chance of winning. In our case we are not interested in any particular strategy. Our aim is to prove properties about the game itself and not on the game play restricted to a number of particular moves. Hence during game play each of the board locations which has not been previously used has an equal chance of being selected as if the player were playing at random. To achieve this we make use of non-determinism. Hence the model checker when verifying our model attempts all possible combinations of the moves.

7.3.6 SMV Model

If one assesses the game board one realises that there is a lot of symmetry such as along the y-axis, along the x-axis, along the diagonals and rotationally. For a 3×3 tictactoe board one can view the board as consisting of three types of locations situated at differing areas of the board. For example, locations 1, 3, 7 and 9 all lie in the corners of the board and can be grouped as *corners*. Locations 2, 4, 6 and 8 all lie on the *sides* of the board. Finally location 5 is unique as it lies at the *centre* of the board. Figure 7.9 shows the mapping of the nine board locations to this new view of the board. We attempted to use such a view so as to potentially create a more compact model. However it does not scale up easily for larger board sizes since a different model is required for every $n \times n$ tictactoe

1	2	3	corner	side	corner
4	5	6	side	centre	side
7	8	9	corner	side	corner

Figure 7.9: Symmetrical view of a 3×3 tictactoe board

board size. Moreover, the model created is harder to understand and does not provide advantages in terms of state space reduction. We therefore decided to create the SMV models which use the traditional indexed approach, i.e. each board location is seen as unique. Every state of the model includes as input an index variable which determines which location a player has chosen in the current state and which determines the location to be changed from empty to the player's marker in the next state. These kinds of models are easy to understand, are compact in both model code and state space size and can easily scale up for larger board sizes as required.

7.3.7 μ cke Model

The tictactoe model written in μ cke's syntax language is identical to the one for SMV, that is, although the models differ in how they are expressed, the initial states and transitions defined are identical. The only difference is minimal and related to the fact that for the tictactoe models written for μ cke we were able to abstract away the index variable used as input for tictactoe models written for SMV. This allows for a slightly smaller model but the end-result in the model achieved is the same.

7.4 Specifying Tictactoe

This section will introduce the properties which we utilised to specify tictactoe. These properties are not exhaustive and many more may be added as required. We have selected ten properties each for *CTL* and μ -calculus and used these as our specification.

7.4.1 CTL Properties

In the *CTL* properties described here we will assume a 3×3 tictactoe board size. For larger board sizes such as 4×4 , 5×5 and so on, these properties have been altered slightly as required to enable the properties to scale up with the larger models.

$$1. AFEndGame \stackrel{\text{def}}{=} \mathbf{AF} \text{ end_game}$$

This property is a very trivial one we would like to specify about our game. However it is important as it model checks the game to verify whether *all the paths of the game eventually lead to a state where the end game condition is satisfied*, that is whether all paths lead to either of the two players winning or a draw game. In doing so we prove that our game model does not enter any computation path that leads it to an infinite and hence incorrect game which never stops. It is true that all computation paths are infinite when modelled over Kripke structures. However we are assuring that for all computation paths, the initial prefix is finite and that such a prefix ends at a state where the end of game condition is satisfied. After such a state is reached it is inconsequential for us that the rest of the path is infinite, as long as the end of game state is reached in a finite number of steps. It would be interesting in *LTL* or μ -calculus to specify a properties similar to this one such as $\mathbf{AFG} \text{ end_game}$ and $\mathbf{AGF} \text{ end_game}$. In doing so we specify that once *end_game* is reached it repeats indefinitely, and that all states eventually lead to an end of game state, respectively. *CTL* can express two similar expressions defined as follows

$$AGAFEndGame \stackrel{\text{def}}{=} \mathbf{AG} \mathbf{AF} \text{ end_game} \text{ and}$$

$$AFAGEndGame \stackrel{\text{def}}{=} \mathbf{AF} \mathbf{AG} \text{ end_game}.$$

The first formula means that all game states of all the paths lead to game states where *end_game* is eventually satisfied along all paths originating from said game states. The second one states that in all paths eventually a game state is reached where all game states on all paths originating from such a game state satisfy *end_game*. Both these latter formulas should be true for our model of tictactoe.

$$2. AFPlayer1Wins \stackrel{\text{def}}{=} \mathbf{AF} \text{ winner_player1}$$

This property is also trivial. It model checks the game to see if *all the computation paths lead Player 1 to win the game* hence checking whether the game is totally biased towards such a player. Similar alterations to the property written for *end_game* may be done, which result in

$$AGAFPlayer1Wins \stackrel{\text{def}}{=} \mathbf{AG} \mathbf{AF} \text{ winner_player1} \text{ and}$$

$AFAGPlayer1Wins \stackrel{\text{def}}{=} \mathbf{AF} \mathbf{AG} \text{ winner_player1}.$

These are exactly the same but are aimed at checking whether *winner_player1* is true, instead of *end_game*. Note that these properties are false for tictactoe and that model checking should result in an error trace which shows a set of game states which lead to Player 2 winning or a draw game.

3. $Player1StartsPlayer1AFWins \stackrel{\text{def}}{=} (turn = player1) \Rightarrow \mathbf{AF} \text{ winner_player1}$

This property is similar to the previous one albeit more interesting. We ask whether *if Player 1 makes the first move, he/she will always finally win the game*, that is whether in fact Player 1 can win just because he is the first player to make a move. This property should be false for tictactoe and the model checker should provide a counterexample which shows how if Player 1 starts playing Player 2 can win or a the game can result in a draw. Notice also that since we modelled the game with Player 1 always making the first move, this specification is the same as the previous one, that is the $(turn = player1)$ subformula is extra. However in reality the meaning of this and the previous formulas differ by the fact that one asks if Player 1 can eventually win for all paths if he/she makes the first move while the other asks whether Player 1 can eventually finally win for all paths with no restriction, respectively.

4. $Player1StartsCentrePlayer1AFWins \stackrel{\text{def}}{=} ((turn = player1) \wedge (index = 5)) \Rightarrow \mathbf{AF} \text{ winner_player1}$

In this property we attempt to dispel a tictactoe myth which many people think is true, that is that *if Player 1 is the first player to make a move and that move is in the centre of the board (Location 5) he/she will always eventually win the game*. We use model checking to see if this myth is true or not. As we mentioned earlier, we have modelled the game to pick randomly between the different locations at each game state. One of the initial randomly selected game states is the middle location. By the use of the subformula $((turn = player1) \wedge (index = 5))$ we verify that there exists an initial move by Player 1 where said player picks the centre location to make their first move. If this is the case, the model checker uses it to try to see if eventually finally all the game sessions end up satisfying *winner_player1*. If it is false a counterexample trace will be provided for us which shows Player 2 winning or a draw game. A better way to model check this property is to modify slightly the model to make the first move by Player 1 pick the centre location always (hence making $((turn = player1) \wedge (index = 5))$ true for the only initial state possible) and then random locations are chosen thereof (as before the modification). Note that we can modify this property to check

various similar tictactoe attributes, such as:

- We can check this for all the board locations. That is whether there are other locations which guarantee a win if used in the first move. This is achieved by changing the *index* value to any of the nine board locations. If on the other hand we want to prove the above for different index locations together such as for example all the corners or the sides are chosen in the beginning we must modify the model to force pick non-deterministically between them and so try them all out. We then specify the indices by the disjunction of their respective atomic propositions, such as for example all the sides $((index = 2)|(index = 4)|(index = 6)|(index = 8))$.
- We can check whether the opposite is true. That is whether there are locations which guarantee a failure (player 2 wins) for the first player to make a move. This is achieved by changing the subformula *winner_player1* to *winner_player2*.
- We can check whether there are locations which if chosen by the first player to make a move always result in a draw game, achieved simply by changing *winner_player1* to the draw game end-game definition *draw_game*.

5. *Player1StartsCentrePlayer2EFWin* $\stackrel{\text{def}}{=}$

$$((turn = player1) \wedge (index = 5)) \Rightarrow \mathbf{EF} \text{ winner_player2}$$

This property is provided as a complement to the previous one. We check if there are game sessions which allow Player 2 to win if Player 1 starts in the centre location of the board. If the above property is false then this one should be obviously true. We can alter the model as before by making the first player pick the centre location if required. The alterations done to the previous property can be done to this property as well, that is:

- We can check this for all indices or combinations of such indices. Altering the model as highlighted above is required if we need to check the disjunction of more than one index atomic proposition.
- We can alter *winner_player2* to *winner_player1* to check whether in fact starting at a given index results in their being a possibility of that player winning the game.

- We can check whether if a player starts with a certain location, there exists a possibility of a draw game.

6. $Player1ForceWin3Moves \stackrel{\text{def}}{=} \mathbf{EX\ AX\ EX\ AX\ EX\ winner_player1}$

This property checks whether *Player 1 can force a win in three moves by making a certain combination of moves*. To do so we use a sequence of **EX AX** where **EX** stands for a possible move by Player 1 while **AX** stands for all the moves Player 2 can make, that is **EX AX** on its own means *can a move be done (by Player 1 in our case) which whatever move there is after (by Player 2) leads to ...*. This property can be of course written for any number of moves by repeating the **EX AX** sequence per every two moves by the players. We are obviously limited by the size of the tictactoe board which for a 3×3 board, enables at most four sequences of **EX AX**. As stated earlier the last operator pair **EX** is required as the finishing move done by Player 1 to actually win the game. As we shall see later on, μ -calculus is more flexible as it allows us to write this property while generalising for n sequences of **EX AX**. This property should be always false since Player 2 can always force a draw.

7. $Player1StartsCentreForceWin3Moves \stackrel{\text{def}}{=} (((turn = player1) \wedge (index = 5)) \Rightarrow \mathbf{EX\ AX\ EX\ AX\ EX\ winner_player1})$

In this property we combine Properties 4 and 6 to create the property which states *can Player 1 force a win in three moves if he/she starts play with a move at the centre location?* This property can be altered in many ways to create similar properties:

- We can alter the number of moves required to force a win.
- We can alter the locations where the first move is made such as for example all the corners (indices 1, 3, 7, 9) or all the sides (indices 2, 4, 6, 8). Again the model has to be altered accordingly to allow more than one index to be model checked at a time.

8. $Player1StartCentrePlayer2ReplyCornerForceWin3Moves \stackrel{\text{def}}{=} (((turn = player1) \wedge (index = 5)) \wedge \mathbf{AX} ((turn = player2) \wedge ((index = 1) \vee (index = 3) \vee (index = 7) \vee (index = 9)))) \Rightarrow \mathbf{EX\ AX\ EX\ AX\ EX\ winner_player1}$

This property asks whether *if Player 1 starts play in the centre location, and Player 2 replies in one of the corners, can Player 1 force win the game in three moves?* To verify this property we must alter the

model so that instead of randomly selecting the first and second moves, in the first move Player 1 selects the centre location while in the second move Player 2 selects one of the corner locations of the board. In this way we ensure that the subformulas $((turn = player1) \wedge (index = 5))$ and **AX** $((turn = player2) \wedge ((index = 1) \vee (index = 3) \vee (index = 7) \vee (index = 9)))$ are true for Player 1 and Player 2 respectively. The rest of the locations are then selected at random hence making the model checker check all the possible paths. Using such alterations we then ask the model checker whether it is possible for it to find game sessions which allow Player 1 to force win the game after 3 moves. Again like in the previous property we can change the property so as:

- We can change Player 1's initial move to one or more different indices, provided we also alter the model accordingly.
- We can change Player 2's reply move to one or more different indices, provided we also alter the model accordingly.
- We can also alter the number of moves required to force a game into a win for Player 1.

9. $Player1AFWinBoardConfiguration \stackrel{\text{def}}{=} \mathbf{AG} (board_configuration_1 \Rightarrow \mathbf{AF} \text{winner_player1})$

where $board_configuration_1 \stackrel{\text{def}}{=} (locs_1 = player1) \wedge (locs_2 = player1) \wedge (locs_3 = empty) \wedge$

$$(locs_4 = player1) \wedge (locs_5 = player2) \wedge (locs_6 = player2) \wedge \\ (locs_7 = empty) \wedge (locs_8 = player2) \wedge (locs_9 = empty)$$

defines a particular game board configuration which we believe is a forced win for Player 1. This temporal logic formula describes a property which verifies the system to see whether *it is true that if a certain board configuration occurs during gameplay (in our case the game board configuration (player1, player1, empty, player1, player2, player2, empty, player2, empty) it is the case that Player 1 always eventually wins*. If this property is true and while playing this board configuration is reached, then no matter what Player 2 does, he/she has already lost. We can alter this property in various ways:

- Check whether it is possible for Player 2 to win by substituting, **AF** *winner_player1* with **EF** *winner_player2*.
- Check whether it is possible for game play to result in a draw by substituting, **AF** *winner_player1* with **EF** *draw_game*.

10. $Player1ForceWin2MovesBoardConfiguration \stackrel{\text{def}}{=}$

$$\mathbf{AG} (board_configuration_1 \Rightarrow \mathbf{EX AX EX} \text{ winner_player1})$$

where $board_configuration_1$ is defined as in the previous property. This property states that *it is true that if a certain board configuration occurs during gameplay (in our case the game board configuration (player1, player1, empty, player1, player2, player2, empty, player2, empty)) it is the case that Player 1 can force win the game in two moves*. This implies that this board state is a winning one for Player 1 and that no matter what Player 2 does, he/she has already lost the game proving future moves useless. This property is thus quite similar to the previous one. However this property checks to see if the winning condition for Player 1 can occur after 2 moves while in the previous property we do not impose such a restriction. This property may be modified in a similar way to the previous property. We can also modify it by changing the moves required to force a win.

7.4.2 μ -calculus Properties

For the μ -calculus properties we will assume as above and we shall alter the properties as required for different board sizes. Some of the properties here are translations of CTL properties into μ -calculus while others cannot be expressed by the former language and are unique for μ -calculus.

1. $AFEndGame$

$$\stackrel{\text{def}}{=} \mathbf{AF} \text{ end_game} \quad (\text{CTL})$$

$$\stackrel{\text{def}}{=} \mu Z. (\text{end_game} \vee \mathbf{AX} Z) \quad (\mu\text{-calculus})$$

This property is synonymous to the first CTL property expressed earlier but has been translated to the fixpoint representation of μ -calculus. As before it checks whether *all games always finally lead to an end of game state, that is, whether Player 1 wins, Player 2 wins or the game results in a draw*. Since the above μ -calculus formula on its own represents only the set of states where such formula is true, we need to write another formula to make sure that our model satisfies the above property, that is, we need to specify whether all the start states imply the property. This can be expressed using the formula:

$$\forall s. (\text{start_game}(s) \Rightarrow AFEndGame(s))$$

where s is of type Game State and

$$\text{start_game} \stackrel{\text{def}}{=}$$

$$((\text{turn} = \text{player1}) \wedge$$

$$\begin{aligned}
& (locs_1 = \text{empty} \wedge locs_2 = \text{empty} \wedge locs_3 = \text{empty} \wedge \\
& locs_4 = \text{empty} \wedge locs_5 = \text{empty} \wedge locs_6 = \text{empty} \wedge \\
& locs_7 = \text{empty} \wedge locs_8 = \text{empty} \wedge locs_9 = \text{empty}))
\end{aligned}$$

The formula *start_game* represents the set of initial states (in our case just one state with an empty board and Player 1's turn). If this property is verified to be true then it is true that for our game, the game always finally ends.

2. *AFPlayer1Wins*

$$\stackrel{\text{def}}{=} \mathbf{AF} \text{ winner_player1} \quad (\text{CTL})$$

$$\stackrel{\text{def}}{=} \mu Z. (\text{winner_player1} \vee \mathbf{AX} Z) \quad (\mu\text{-calculus})$$

Again, this property is similar to the one written earlier in *CTL* and quite similar to the above μ -calculus property. This is a basic property one would like to prove about a game, that is, *whether Player 1 always eventually wins the game*. We are required to add another formula such that the model checker proves the above property:

$$\forall s. (\text{start_game}(s) \Rightarrow \text{AFPlayer1Wins}(s))$$

where *s* and *start_game* have the same type and definition as before, respectively. Using this property we can check whether a game is completely biased towards a player and whether the opposing player has a chance to win even if the game is unfair.

3. *AGIfPlayer1StartsMiddleImpliesAFPlayer1Wins*

$$\stackrel{\text{def}}{=} \mathbf{AG} (\text{Player1StartsMiddle} \Rightarrow \mathbf{AF} \text{ winner_player1}) \quad (\text{CTL})$$

$$\stackrel{\text{def}}{=} \nu Z. (\text{Player1StartsMiddleImpliesAFPlayer1Wins} \wedge \mathbf{AX} Z) \quad (\mu\text{-calculus}) \text{ where}$$

$$\text{Player1StartsMiddleImpliesAFPlayer1Wins} \stackrel{\text{def}}{=}$$

$$\text{Player1StartsMiddle} \Rightarrow \text{AFPlayer1Wins}$$

and where

$$\text{Player1StartsMiddle} \stackrel{\text{def}}{=}$$

$$((\text{turn} = \text{player2}) \wedge$$

$$(\text{locs}_1 = \text{empty} \wedge \text{locs}_2 = \text{empty} \wedge \text{locs}_3 = \text{empty} \wedge$$

$$\text{locs}_4 = \text{empty} \wedge \text{locs}_5 = \text{player1} \wedge \text{locs}_6 = \text{empty} \wedge$$

$$\text{locs}_7 = \text{empty} \wedge \text{locs}_8 = \text{empty} \wedge \text{locs}_9 = \text{empty}))$$

and *AFPlayer1Wins* is defined as previously.

This formula, split into many sub-formulas, attempts to verify the game to check if it is the case

that it is *always globally true* that if Player 1 starts play in the middle location of the tictactoe board they *always finally win the game*. If this formula is true, Player 1 can win simply by starting play in the middle location of the board. As can be deduced, *Player1StartsMiddle* is the set of states where Player 1 has selected to start gameplay in the middle location. Subsequently the formula *Player1StartsMiddleImpliesAFPlayer1Wins* represents the states where if Player 1 starts in the middle imply that they will always finally win. We then obtain the states for our system where this is true for all states in all paths originating from such states by means of the actual property. Finally we require the following formula:

$$\forall s. (start_game(s) \Rightarrow$$

$$AGIfPlayer1StartsMiddleImpliesAFPlayer1Wins(s))$$

where s and *start_game* have the same type and definition as before, respectively. In this way we check whether the property is true for our model by checking if it is true for the start state.

4. *AGIfPlayer1StartsMiddleImpliesEFPlayer2Wins*

$$\stackrel{\text{def}}{=} \mathbf{AG} (Player1StartsMiddle \Rightarrow \mathbf{EF} \text{ winner_player2}) \quad (\text{CTL})$$

$$\stackrel{\text{def}}{=} \nu Z. (Player1StartsMiddleImpliesEFPlayer2Wins \wedge \mathbf{AX} Z) \quad (\mu\text{-calculus}) \text{ where}$$

$$Player1StartsMiddleImpliesEFPlayer2Wins \stackrel{\text{def}}{=}$$

$$Player1StartsMiddle \Rightarrow EFPlayer2Wins$$

where

$$EFPlayer2Wins$$

$$\stackrel{\text{def}}{=} \mathbf{EF} \text{ winner_player2} \quad (\text{CTL})$$

$$\stackrel{\text{def}}{=} \mu Z. (\text{winner_player2} \vee \mathbf{EX} Z) \quad (\mu\text{-calculus})$$

and *Player1StartsMiddle* is defined as before.

This formula asks whether it is *always the case that it is possible for Player 2 to win if Player 1 starts play in the middle location*. The subformula *EFPlayer2Wins* refers to the set of states where starting from such states paths exist which eventually lead to states where Player 2 wins. Hence, the states which satisfy the formula *Player1StartsMiddleImpliesEFPlayer2Wins* are those where if Player 1 starts play in the middle, there exists the possibility that Player 2 wins. The complete formula checks whether this is true for all the states on all the paths of the model. To model check the property we write the formula:

$$\forall s. (start_game(s) \Rightarrow$$

$$AGIfPlayer1StartsMiddleImpliesEFPlayer2Wins(s))$$

where s and $start_game$ have the same type and definition as before, respectively.

5. $AGIfPlayer1StartsMiddlePlayer2RepliesCornerImpliesAFPlayer1Wins$

$$\begin{aligned} &\stackrel{\text{def}}{=} \mathbf{AG} (Player1StartsMiddleAndPlayer2RepliesCorner \Rightarrow \mathbf{AF} winner_player1) \quad (\text{CTL}) \\ &\stackrel{\text{def}}{=} \nu Z. (Player1StartsMiddleAndPlayer2RepliesCornerImpliesAFPlayer1Wins \\ &\quad \wedge \mathbf{AX} Z) \quad (\mu\text{-calculus}) \end{aligned}$$

where

$$\begin{aligned} Player1StartsMiddleAndPlayer2RepliesCornerImpliesAFPlayer1Wins &\stackrel{\text{def}}{=} \\ &Player1StartsMiddleAndPlayer2RepliesCorner \Rightarrow AFPlayer1Wins \end{aligned}$$

and where

$$\begin{aligned} Player1StartsMiddleAndPlayer2RepliesCorner &\stackrel{\text{def}}{=} \\ &((turn = player1) \wedge \\ &((locs_1 = player2 \wedge locs_2 = empty \wedge locs_3 = empty \wedge locs_4 = empty \wedge locs_5 = player1 \wedge \\ &locs_6 = empty \wedge locs_7 = empty \wedge locs_8 = empty \wedge locs_9 = empty) \vee \\ &(locs_1 = empty \wedge locs_2 = empty \wedge locs_3 = player2 \wedge locs_4 = empty \wedge locs_5 = player1 \wedge \\ &locs_6 = empty \wedge locs_7 = empty \wedge locs_8 = empty \wedge locs_9 = empty) \vee \\ &(locs_1 = empty \wedge locs_2 = empty \wedge locs_3 = empty \wedge locs_4 = empty \wedge locs_5 = player1 \wedge \\ &locs_6 = empty \wedge locs_7 = player2 \wedge locs_8 = empty \wedge locs_9 = empty) \vee \\ &(locs_1 = empty \wedge locs_2 = empty \wedge locs_3 = empty \wedge locs_4 = empty \wedge locs_5 = player1 \wedge \\ &locs_6 = empty \wedge locs_7 = empty \wedge locs_8 = empty \wedge locs_9 = player2))) \end{aligned}$$

and $AFPlayer1Wins$ is defined as previously.

In this property we take Property 3 a step forward by checking whether it is always the case that if Player 1 starts in the middle location and Player 2 replies in one of the corner locations, Player 1 always finally wins. The formula $Player1StartsMiddleAndPlayer2RepliesCorner$ represents the set of states where Player 1 has marked the centre location and Player 2 has marked one of the corner locations. Using this subformula we build the formula

$Player1StartsMiddleAndPlayer2RepliesCornerImpliesAFPlayer1Wins$ which refers to the set of states where if the former subformula is true this implies that Player 1 is always finally the winner. This allows us to obtain the required property which checks that in all the states on all the paths of the system if the players make their moves as described, Player 1 always eventually wins.

To complete the property and allow us to check it on the model we require the formula:

$$\forall s. (start_game(s) \Rightarrow$$

$$AGIf Player1StartsMiddlePlayer2RepliesCornerImpliesAF Player1Wins(s))$$

where s and $start_game$ have the same type and definition as before, respectively.

6. $AGIf Player1StartsMiddlePlayer2RepliesCornerImpliesEF Player2Wins$

$$\stackrel{\text{def}}{=} \mathbf{AG} (Player1StartsMiddleAndPlayer2RepliesCorner \Rightarrow \mathbf{EF} \text{ winner_player2}) \quad (\text{CTL})$$

$$\stackrel{\text{def}}{=} \nu Z. (Player1StartsMiddleAndPlayer2RepliesCornerImpliesEF Player2Wins \wedge \mathbf{AX} Z) \quad (\mu\text{-calculus})$$

where

$$Player1StartsMiddleAndPlayer2RepliesCornerImpliesEF Player2Wins \stackrel{\text{def}}{=} \\ Player1StartsMiddleAndPlayer2RepliesCorner \Rightarrow EF Player2Wins$$

This property checks whether it is always the case that if Player 1 starts in the middle location and Player 2 replies in one of the corner locations, there exists a path which eventually leads Player 2 to win.

This property complements the previous one. If the previous property is true this property should be false and viceversa. $Player1StartsMiddleAndPlayer2RepliesCorner$ represents the same set of states as before. $Player1StartsMiddleAndPlayer2RepliesCornerImpliesEF Player2Wins$ represents the set of states where if the former subformula is true this implies that Player 2 may be finally the winner. Our final property proves this for all the states of the game model. We also require the formula:

$$\forall s. (start_game(s) \Rightarrow$$

$$AGIf Player1StartsMiddlePlayer2RepliesCornerImpliesEF Player2Wins(s))$$

where s and $start_game$ have the same type and definition as before, respectively.

7. $Player1ForceWin \stackrel{\text{def}}{=}$

$$\mu Z. (winner_player1 \vee ((turn = player1 \wedge \mathbf{EX} Z) \vee (turn = player2 \wedge \mathbf{AX} Z)))$$

This property checks whether Player 1 may force a win in a number of moves starting from a particular board state. Such a property may only be written in μ -calculus since no equivalent CTL property exists. In CTL it is only possible to define a fixed number of moves required to force a win while μ -calculus allows us to leave the upper bound undetermined such that verification may check the property for any number of moves. The result of such a property is an interesting one for any form of game as it shows that there exist board states where a player may win no matter what

their opponent does hence proving future moves useless. If such board states exist, it would be a player's main target to achieve them so as to ensure a win in their favour. In order to model check this property we require the formula:

$$\exists s. \text{Player1ForceWin}(s)$$

where s is of type Game State.

Using this formula we check for the existence for such states. Moreover, a state count of this property gives us the number of board states where Player 1 can force a win.

$$8. \text{Player1CanForceAWinIfStartsMiddle} \stackrel{\text{def}}{=} \text{Player1StartsMiddle} \wedge \text{Player1ForceWin}$$

where $\text{Player1StartsMiddle}$ and Player1ForceWin are defined as before. This μ -calculus –only property verifies whether it is possible for Player 1 to force a win in a number of moves if they start play in the middle board location. Doing so we may affirm or dispel a myth which some people believe is true for tictactoe, that is, if Player 1 starts in the middle location they win at all the costs. We use the formula:

$$\exists s. \text{Player1CanForceAWinIfStartsMiddle}(s)$$

where s is of type Game State. In this way we check for the existence of the required states which satisfy both the two sub-properties.

$$9. \text{Player1CanForceAWinIfPlayer1StartsMiddleAndPlayer2RepliesCorner} \stackrel{\text{def}}{=}$$

$$\text{Player1StartsMiddleAndPlayer2RepliesCorner} \wedge \text{Player1ForceWin}$$

where $\text{Player1StartsMiddleAndPlayer2RepliesCorner}$ and Player1ForceWin are defined as previously. We use this property to show whether if Player 1 starts play in the middle and Player 2 replies in a corner location, Player 1 can force a win in a number of moves. We use this property to check if it is true that replying to a move with another allows a player to force a win, as it is often believed. We can of course try this property using other start/reply locations or to try to see if instead of Player 1, Player 2 can force a win. To check this property we require the formula:

$$\exists s. \text{Player1CanForceAWinIfPlayer1StartsMiddleAndPlayer2RepliesCorner}(s)$$

where s is of type Game State. Again, in this way, we check for the existence of the required states which satisfy the property.

$$10. \text{Player2MayDraw} \stackrel{\text{def}}{=}$$

$$\mu Z. (\text{draw_game} \vee ((\text{turn} = \text{player1} \wedge \mathbf{EX} Z) \vee (\text{turn} = \text{player2} \wedge \mathbf{EX} Z)))$$

This property defines the set of states where Player 2 may attempt to draw the game. We can model

check the game model using this property by means of the additional formula:

$$\forall s. (start_game(s) \Rightarrow Player2MayDraw(s))$$

where s is of type Game State. This formula states that for all states it is true that if the state is a starting one it implies that Player 2 may draw the game.

7.5 Verifying Tictactoe

In order to verify the above properties on tictactoe a model generator was created in C# which generates SMV and μ cke code scripts for tictactoe models. This generator is able, given a board size and two player name parameters, to generate the appropriate $x \times x$ tictactoe code script for two players. It does so by automatically creating the appropriate definitions of the initial game state, the transition relation and end-game state conditions defined earlier. Using this script, which can be saved and edited as necessary, we added the specifications above in SMV *CTL* syntax and μ cke μ -calculus syntax, taking care to modify the properties according to the board size since some of the above properties are restricted to a tictactoe board size of 3×3 . The system used for model checking consisted of an AMD Athlon 64 3700+ 2.19GHz processor with 2.00 GB of memory, running a 32 bit operating system.

7.5.1 Results

Verifying tictactoe of different sizes using the above specifications has yielded the following results as shown in the tables in the following pages.

SMV Results

Verification Result		Property									
Board Size		1	2	3	4	5	6	7	8	9	10
	3 × 3	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
	4 × 4	TRUE	FALSE	FALSE	FALSE	DNF	C	DNF	C	FALSE	C
	5 × 5	C	C	C	C	C	C	C	C	C	C

Table 7.1: Tictactoe Verification Results for the *CTL* properties verified using SMV

Verification Time		Property									
Board Size		1	2	3	4	5	6	7	8	9	10
	3 × 3	0.59375 s	1.23438 s	1.25 s	1.03125 s	0.671875 s	1.20313 s	0.859375 s	0.984375 s	0.75 s	0.5625 s
	4 × 4	106.316 s	259.078 s	250.835 s	409.538 s	DNF	C	DNF	C	221.875 s	C
	5 × 5	C	C	C	C	C	C	C	C	C	C

Table 7.2: Tictactoe Verification Times for the *CTL* properties verified using SMV

Counterexample Trace		Property									
Board Size		1	2	3	4	5	6	7	8	9	10
	3 × 3	N/A	Full	Full	Full	N/A	No	No	No	N/A	N/A
	4 × 4	N/A	Full	Full	Full	DNF	C	DNF	C	Full	C
	5 × 5	C	C	C	C	C	C	C	C	C	C

Table 7.3: Tictactoe Counterexample Trace Results for the *CTL* properties verified using SMV

State Variable Count		Property									
Board Size		1	2	3	4	5	6	7	8	9	10
	3 × 3	23	23	23	23	23	23	23	23	23	23
	4 × 4	37	37	37	37	DNF	C	DNF	C	37	C
	5 × 5	C	C	C	C	C	C	C	C	C	C

Table 7.4: Tictactoe State Variable Counts for the *CTL* properties verified using SMV

Notes:

“DNF” Did Not Finish: SMV was stopped when paging of memory started to take place and the processor was spending approximately 0% working on SMV verification.

“C” Crashed: SMV crashed with an exception during verification.

μ cke Results

Verification Result		Property									
Board Size		1	2	3	4	5	6	7	8	9	10
	3 × 3	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	FALSE	TRUE
	4 × 4	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	FALSE	TRUE
	5 × 5	TRUE	FALSE	FALSE	DNF	FALSE	DNF	DNF	DNF	DNF	DNF
	6 × 6	DNF	DNF	DNF	DNF	DNF	DNF	DNF	DNF	DNF	DNF

Table 7.5: Tictactoe Verification Results for the μ -calculus properties verified using μ cke

Verification Time		Property									
Board Size		1	2	3	4	5	6	7	8	9	10
	3 × 3	0.22 s	0.21 s	0.19 s	0.21 s	0.19 s	0.22 s	0.25 s	0.25 s	0.25 s	0.18 s
	4 × 4	7.79 s	2.11 s	2.86 s	23.2 s	2.13 s	21.1 s	45.5 s	44.0 s	44.1 s	24.9 s
	5 × 5	284 s	390 s	403 s	DNF	403 s	DNF	DNF	DNF	DNF	DNF
	6 × 6	DNF	DNF	DNF	DNF	DNF	DNF	DNF	DNF	DNF	DNF

Table 7.6: Tictactoe Verification Times for the μ -calculus properties verified using μ cke

Notes:

“DNF” Did Not Finish: μ cke was stopped when paging of memory started to take place and the processor was spending approximately 0% working on μ cke verification.

7.6 Conclusion

In this chapter we have shown how tictactoe can be modelled, specified using temporal logics such as *CTL* and μ -calculus, and verified using symbolic model checking. To do so we have utilised two model checkers and employed different board sizes. We have also presented the results we have obtained. In the next chapter we will consider how model checking may be applied to another game: connect four.

CASE STUDY 2: CONNECT FOUR

*Please don't ask me what the score is,
I'm not even sure what the game is.*

Ashleigh Brilliant

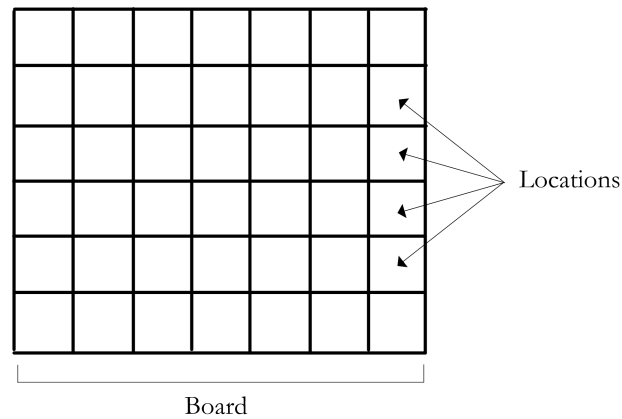
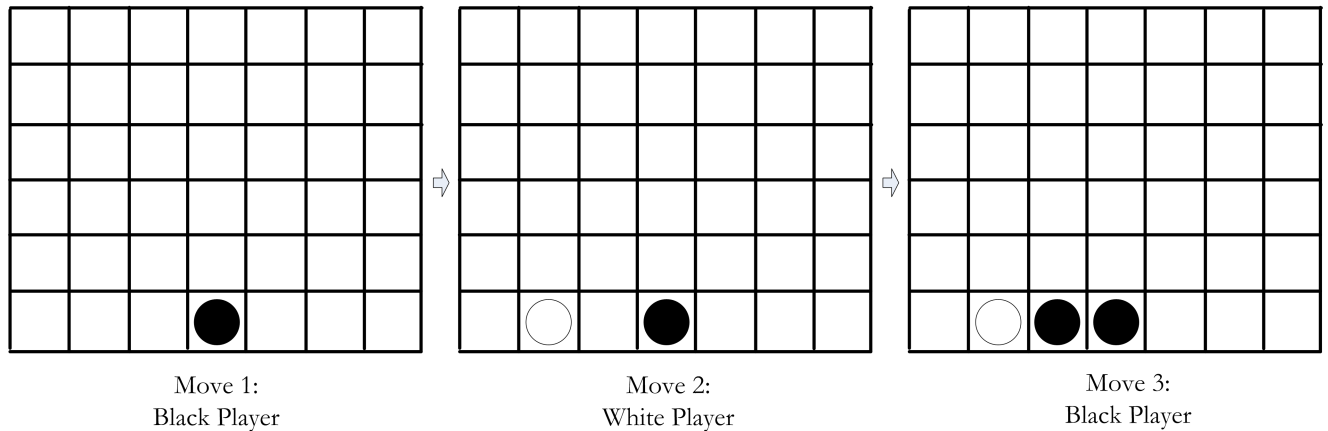
8.1 Overview

This chapter follows in the same pattern as the previous one by first introducing the game at hand, connect four. We then explain how the game may be modelled by means of a Kripke structure. Specification of the game is again done by a number of properties in both *CTL* and μ -calculus. We make use of SMV and μ cke to verify these properties on the game by taking into consideration different board sizes.

8.2 Game Rules

8.2.1 Introduction

Connect four is a two player game which is very similar to tictactoe. The players are once more designated using two different markers, traditionally using yellow and red or any other variation such as black and white. The typical game board consists of a matrix of forty-two locations arranged in seven vertical columns of six locations each, as shown in Figure 8.1. The board can

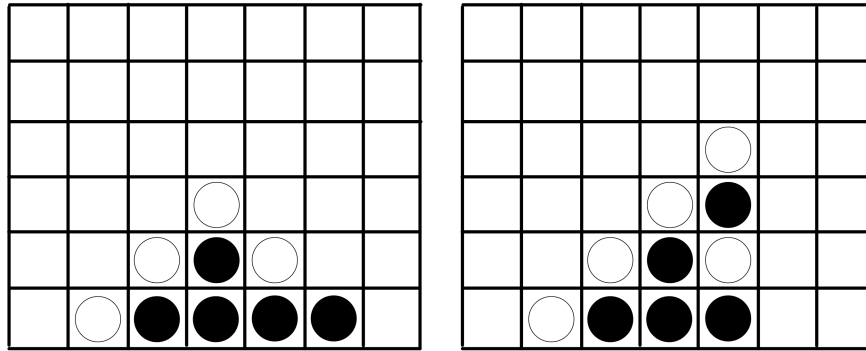
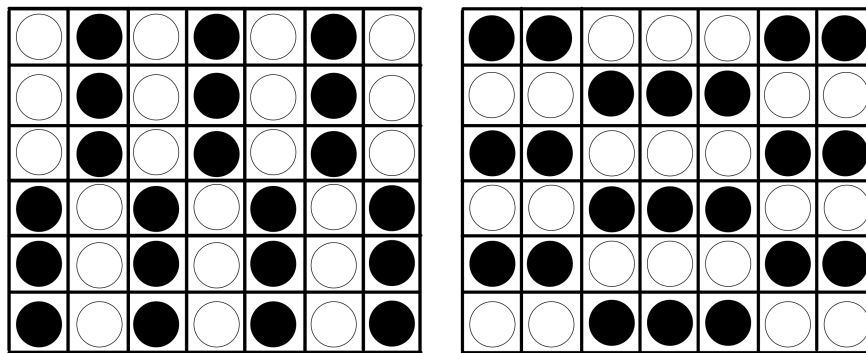
Figure 8.1: A standard 7×6 connect four boardFigure 8.2: Example moves in a 7×6 connect four board

however be of any size with the minimum board size being 4×4 .

The rules of a valid connect four game are as follows:

1. The two players take turns to fill the locations on the board.
2. In each turn a player chooses a column of locations and puts his marker in the lowest available location in that column such that as the column is filled up it resembles a stacked pillar of markers. No empty locations may be left between filled locations and the former exist only at the top of the column unless the column has been completely filled up.
3. Once a location has been marked its marker cannot be changed in a subsequent turn.

The first player to produce a row of four of their own symbols either horizontally, vertically or diagonally wins the game. In order to do so, like in tictactoe, they must make strategic moves in

Figure 8.3: Example winning boards in a 7×6 connect four boardFigure 8.4: Example draw boards in a 7×6 connect four board

order to increase their advantage over their opponent and hence increase their chance of winning. Two examples of finished games are shown for a 7×6 board in Figure 8.3. The first example on the left shows black winning horizontally while in the second case white wins by forming a diagonal of markers.

The game can also end in a draw game. This occurs when the board locations are all filled by markers before any of the two players completes a set of four markers. Two examples of such cases are shown for a 7×6 board in Figure 8.4.

8.3 Modelling Connect Four

As we shall see in this section, modelling connect four is quite similar to any other board game. In fact it differs from tictactoe's model mainly due to the different playing rules which translate into a different transition relation as we shall see later on.

8.3.1 Game State

Once more we first introduce how the states of our Kripke structure will be defined prior to describing the structure itself. It is important to notice that although we assume a connect four board size of 4×4 , it is possible to generalise our model for a $m \times n$ board.

Our game state for connect four will have the same structure as that used for tictactoe, that is it will hold a number of board location variables (4×4 in our case) and a variable to represent the current player turn. By providing a value to each of these variable we will know the state of the game at a particular point in time.

We will formally define the connect four game state as

$$s = (locs, turn)$$

where,

- *locs* is the set of sixteen board location variables. Again we denote each of these variables with $locs_i$ where i ranges from 1 to 16 ($m \times n$ for a generic connect four). The type of these variables consists of three values: *empty*, *player1* and *player2* which refer to an empty board state, a location marked by Player 1 and a location marked by Player 2, respectively. The domain of each $locs_i$ is D_b defined earlier as $\{empty, player1, player2\}$. The indexing of the location variables differs slightly from that of tictactoe. In fact for connect four we will assume that $locs_1$ is situated at the lower left corner location of the board while $locs_{16}$ is at the top right corner location of the board, that is the first row of locations is $locs_1, locs_2, locs_3$ and $locs_4$ and the second row on top of it starts at $locs_5$ and so on. Figure 8.5 shows as a 4×4 connect four board and how $locs_1$ to $locs_{16}$ are positioned on the board. By a valuation of these variables we can represent the required game board state as a formula. For example consider the valuation $\langle locs_1 \leftarrow player1, locs_2 \leftarrow player1, locs_3 \leftarrow player1, locs_4 \leftarrow player2, locs_5 \leftarrow player2, locs_6 \leftarrow player2, locs_7 \leftarrow empty, locs_8 \leftarrow empty, locs_9 \leftarrow empty, locs_{10} \leftarrow empty, locs_{11} \leftarrow empty, locs_{12} \leftarrow empty, locs_{13} \leftarrow empty, locs_{14} \leftarrow empty, locs_{15} \leftarrow empty, locs_{16} \leftarrow empty \rangle$ represented using the formula: $(locs_1 = player1) \wedge (locs_2 = player1) \wedge (locs_3 = player1) \wedge (locs_4 = player2) \wedge (locs_5 =$

$locs_{13}$	$locs_{14}$	$locs_{15}$	$locs_{16}$
$locs_9$	$locs_{10}$	$locs_{11}$	$locs_{12}$
$locs_5$	$locs_6$	$locs_7$	$locs_8$
$locs_1$	$locs_2$	$locs_3$	$locs_4$

Figure 8.5: The positioning of *locs* variables in a 4×4 connect four board

●	●		
○	○	○	●

Figure 8.6: This is the board state represented by the formula $(locs_1 = player1) \wedge (locs_2 = player1) \wedge (locs_3 = player1) \wedge (locs_4 = player2) \wedge (locs_5 = player2) \wedge (locs_6 = player2) \wedge (locs_7 = empty) \wedge (locs_8 = empty) \wedge (locs_9 = empty) \wedge (locs_{10} = empty) \wedge (locs_{11} = empty) \wedge (locs_{12} = empty) \wedge (locs_{13} = empty) \wedge (locs_{14} = empty) \wedge (locs_{15} = empty) \wedge (locs_{16} = empty)$. Player 1 is represented by the white marker while player 2 is represented by the black marker.

$player2) \wedge (locs_6 = player2) \wedge (locs_7 = empty) \wedge (locs_8 = empty) \wedge (locs_9 = empty) \wedge (locs_{10} = empty) \wedge (locs_{11} = empty) \wedge (locs_{12} = empty) \wedge (locs_{13} = empty) \wedge (locs_{14} = empty) \wedge (locs_{15} = empty) \wedge (locs_{16} = empty)$. This formula in terms of the location variables describes the game board state shown in Figure 8.6. In the figure, Player 1 is represented by white markers while Player 2 is represented by black markers.

- *turn* is the variable representing the current game state's turn, that is the player who will make the next move. Since a game session's turn alternates between the two players, the domain of *turn* is the set D_t defined as $\{player1, player2\}$. Again we assume that if in the current game state it is Player 1's turn to make a move, the location variables which reflect this are the ones of next game state and not of the current one.

An example of a possible game state is

$$\begin{aligned}
 &(locs_1 = player1) \wedge (locs_2 = player1) \wedge (locs_3 = player1) \wedge (locs_4 = player2) \wedge \\
 &(locs_5 = player2) \wedge (locs_6 = player2) \wedge (locs_7 = empty) \wedge (locs_8 = empty) \wedge \\
 &(locs_9 = empty) \wedge (locs_{10} = empty) \wedge (locs_{11} = empty) \wedge (locs_{12} = empty) \wedge
 \end{aligned}$$

$empty, empty, empty, empty, empty, player1),$
 $(player1, empty, empty, empty, empty, empty, empty, empty, empty, empty, empty, empty,$
 $empty, empty, empty, empty, empty, player2))$
 $((empty, empty, empty, empty, empty, empty, empty, empty, empty, empty, empty,$
 $empty, empty, empty, empty, empty, player1),$
 $(empty, player1, empty, empty, empty, empty, empty, empty, empty, empty, empty, empty,$
 $empty, empty, empty, empty, empty, player2))$
 $((empty, empty, empty, empty, empty, empty, empty, empty, empty, empty, empty,$
 $empty, empty, empty, empty, empty, player1),$
 $(empty, empty, player1, empty, empty, empty, empty, empty, empty, empty, empty, empty,$
 $empty, empty, empty, empty, empty, player2))$
 $((empty, empty, empty, empty, empty, empty, empty, empty, empty, empty, empty,$
 $empty, empty, empty, empty, empty, player1),$
 $(empty, empty, empty, player1, empty, empty, empty, empty, empty, empty, empty, empty,$
 $empty, empty, empty, empty, empty, player2))$
 \dots
 $((player1, player1, player1, player1, player2, player2, player2, empty, empty, empty, empty,$
 $empty, empty, empty, empty, empty, player2),$
 $(player1, player1, player1, player1, player2, player2, player2, empty, empty, empty, empty,$
 $empty, empty, empty, empty, empty, player2)))$

The transitions of R are based on the game rules and progress the game from one valid state to another valid one. As examples consider the first and the last transitions in the set shown above. The first transition shows Player 1 picking the first board location, $locs_1$. The last transition shows a case where Player 1 has won causing the transition to lead from an end-game state to itself.

- $L((empty, empty, empty, empty, empty, empty, empty, empty, empty, empty, empty,$
 $empty, empty, empty, empty, player1)) =$
 $\{locs_1 = empty, locs_2 = empty, locs_3 = empty, locs_4 = empty, locs_5 = empty,$
 $locs_6 = empty, locs_7 = empty, locs_8 = empty, locs_9 = empty, locs_{10} = empty,$
 $locs_{11} = empty, locs_{12} = empty, locs_{13} = empty, locs_{14} = empty, locs_{15} = empty,$

$$\begin{aligned}
& locs_{16} = empty, turn = player1\}, \\
& L((player1, empty, empty, empty, empty, empty, empty, empty, empty, empty, empty, \\
& empty, empty, empty, empty, player2)) = \\
& \{locs_1 = player1, locs_2 = empty, locs_3 = empty, locs_4 = empty, locs_5 = empty, \\
& locs_6 = empty, locs_7 = empty, locs_8 = empty, locs_9 = empty, locs_{10} = empty, \\
& locs_{11} = empty, locs_{12} = empty, locs_{13} = empty, locs_{14} = empty, locs_{15} = empty, \\
& locs_{16} = empty, turn = player2\}, \dots
\end{aligned}$$

The labelling function performs the same function as with any modelled system. It allows us to distinguish each of the different states of the state space by means of their label which consists of the atomic propositions true in that state.

8.3.3 End-Game Definitions

In this section we present the end-game definitions for connect four which are similar to the ones for tictactoe mentioned earlier and are also defined in terms of the board locations variables. Respectively these define when: the board is full, Player 1 has won, Player 2 has won, the game is a draw or the game has ended:

$$board_full \stackrel{\text{def}}{=} \bigwedge_{i=1}^{16} (locs_i \neq empty)$$

$$\begin{aligned}
winner_player1 & \stackrel{\text{def}}{=} \\
& [(locs_1 = player1) \wedge (locs_6 = player1) \wedge (locs_{11} = player1) \wedge (locs_{16} = player1)] \vee \\
& [(locs_{13} = player1) \wedge (locs_{10} = player1) \wedge (locs_7 = player1) \wedge (locs_4 = player1)] \vee \\
& (\bigvee_{i=1,5,9,13} [(locs_i = player1) \wedge (locs_{i+1} = player1) \wedge (locs_{i+2} = player1) \wedge (locs_{i+3} = player1)]) \vee \\
& (\bigvee_{i=1,2,3,4} [(locs_i = player1) \wedge (locs_{i+4} = player1) \wedge (locs_{i+8} = player1) \wedge (locs_{i+12} = player1)])
\end{aligned}$$

$$\begin{aligned}
winner_player2 & \stackrel{\text{def}}{=} \\
& [(locs_1 = player2) \wedge (locs_6 = player2) \wedge (locs_{11} = player2) \wedge (locs_{16} = player2)] \vee \\
& [(locs_{13} = player2) \wedge (locs_{10} = player2) \wedge (locs_7 = player2) \wedge (locs_4 = player2)] \vee \\
& (\bigvee_{i=1,5,9,13} [(locs_i = player2) \wedge (locs_{i+1} = player2) \wedge (locs_{i+2} = player2) \wedge (locs_{i+3} = player2)]) \vee \\
& (\bigvee_{i=1,2,3,4} [(locs_i = player2) \wedge (locs_{i+4} = player2) \wedge (locs_{i+8} = player2) \wedge (locs_{i+12} = player2)])
\end{aligned}$$

$$draw_game \stackrel{\text{def}}{=} (\neg winner_player1) \wedge (\neg winner_player2) \wedge draw_game$$

$$end_game \stackrel{\text{def}}{=} winner_player1 \vee winner_player2 \vee board_full$$

8.3.4 Game Computation Tree

The computation tree of the Kripke structure which models our game of connect four has a surprising much different structure than that of tictactoe, even though the two games are quite similar in nature and in goals. Structure-wise the only property they have in common is that like tictactoe's computation tree, connect four's one is also symmetrical by who makes the first move. This implies that the computation tree rooted at the initial game state in which Player 1 makes the first move and the corresponding computation tree rooted at the initial game state in which Player 2 makes the first move are symmetrical and also independent from one another, hence allowing us to reduce our model by half by writing our model in such a way that we force one of the player to make the first move (in our case Player 1).

At the first state there are four possible transitions (in case of our 4×4 connect four), that is one of the four possible columns which the player may use to make his move. Each of these possible initial transitions allow us four possible moves, again, due to the four columns. Hence at each game state node the computation tree fans out into four possible next state nodes. The growth is thus exponential (4^n) at most states. We use the term *most* and not *all* due to the simple fact that eventually either an end-game state node is reached and hence the computation tree grows singularly indefinitely (same node repeats itself) or one or more of the columns is full up of markers and hence in the computation tree's fanning out factor decreases at this point depending on how many available columns there are. In the second case, if a fanning out factor of zero is reached it means that a draw state was reached by the players since there are no more columns in which to play. This state repeats itself in the computation tree indefinitely as it is another form of end-game state node. Figure 8.8 shows the initial part of the computation tree for a 4×4 connect four in which Player 1 (black markers) makes the first move.

8.3.5 Game Strategy

For our game strategy we will utilise the same one that we used with tictactoe, that is all of the columns have the same possibility of being selected by the players to make their moves. To achieve this non-determinism is used so that all possible combinations of moves are tried out by the model checker.

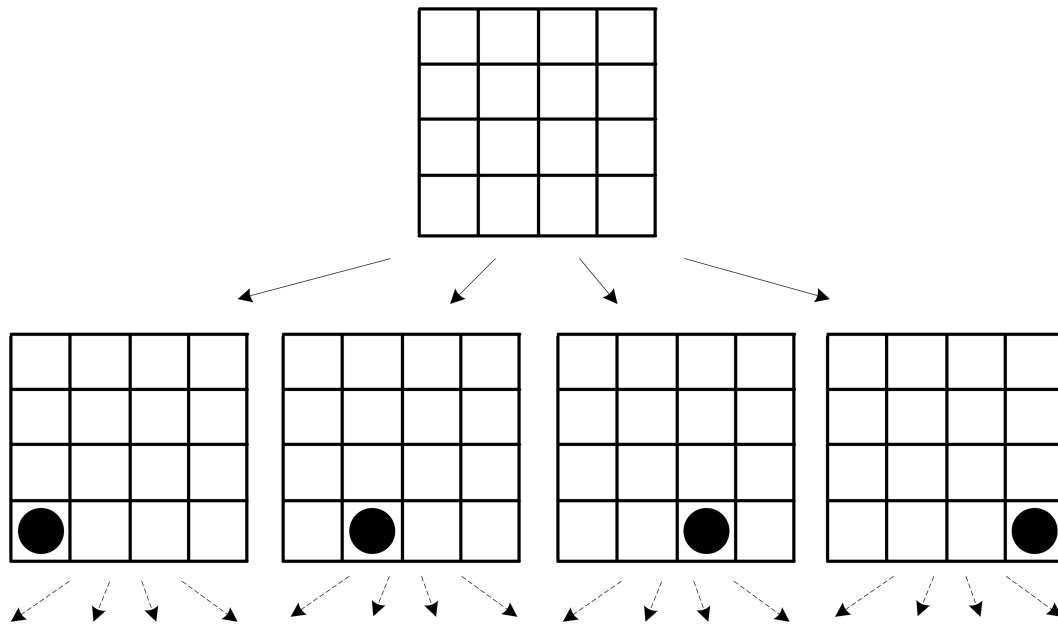


Figure 8.8: Initial part of the computation tree of a 4×4 connect four board where Player 1 (black) makes the first move.

SMV Model

As we stated earlier, each of the board location variables ranges over three values, one for when the location is empty, one for when it is occupied by Player 1's marker and one for when it is occupied by Player 2's marker. Since three values are required they are represented by two Boolean variables which allow us four values, one of which is thus redundant. In order to remove such redundancy and hence obtain a smaller state space we have written the SMV model in such a way as to use a more efficient encoding of the board locations.

The encoding we will use will still consist of columns of board locations. However the locations in each column are grouped together in twos, starting from the bottom row upwards. In this way for example, the first location of the first row is grouped with the first location of the second row, while the first location of the third row is grouped with the first location of fourth row and so on. In case of an odd number of rows the top row is not grouped but encoded as before. As an example consider a 4×4 connect four using the above grouping as shown in Figure 8.9. The eight dual locations obtained from the sixteen normal locations are numbered as seen in this figure, using the same numbering system as before.

13	14	15	16	13	14	15	16
9	10	11	12	5	6	7	8
5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8
				1	2	3	4
				1	2	3	4

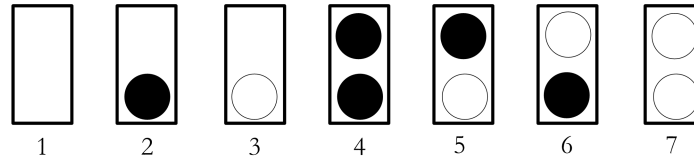
Figure 8.9: Grouping of the board locations of a 4×4 connect four board.

Figure 8.10: Possible values of dual locations and their codes.

Each of the newly formed dual locations has one of seven possible values which is made up of the combinations of the values of the two separate locations according to game rules. These combinations are depicted in Figure 8.9. Each of these values is represented by a code, also shown in the figure. Since now we require to represent a type with seven values (the codes 1 to 7) the number of Boolean variables required is three ($2^3 = 8$) with an extra unused code (8). If we had to use the original modelling, we would require two Boolean variables per board location due to the fact that each location has three possible values. This results in four Boolean variables and two extra unused codes. Hence using our encoding we reduce the number of Boolean variables by one and also reduce the redundant values by one. More importantly we reduce our state space considerably.

Consider once more a 4×4 connect four board. Before we needed 32 Boolean variables (16 locations of 2 Boolean variables each) to represent the board locations. Now using our encoding we require only two-thirds of that amount, that is 24 Boolean variables (8 locations of 3 Boolean variables each) for the board locations. This also translates to reducing the state space from 86093442 states ($= 3^{16} \times 2$) to 11529602 states ($= 7^8 \times 2$), that is by a factor of almost 7.5.

μ cke Model

The μ cke model for connect four models the same behaviour as that for SMV, hence the only difference between the two is the syntax used by the respective language. Again in the μ cke models we managed to abstract away the explicit index operator used in the SMV model hence allowing

us to produce a slightly smaller model.

8.4 Specifying Connect Four

We will once more make use of a number of *CTL* and μ -calculus properties to specify connect four. These properties are almost equivalent to the ones specified for tictactoe in Section 7.4. In fact some of the properties are identical for both games, while others are altered so that they can be used to specify properties about connect four similar to the ones for tictactoe. It should also be noted that the alterations available to the properties written for tictactoe can be applied for the following properties of connect four as well.

8.4.1 CTL Properties

The *CTL* properties selected are as follows:

1. $AFEndGame \stackrel{\text{def}}{=} \mathbf{AF} \text{ end_game}$

This property, like the one for tictactoe, checks the game to verify whether *all the paths of the game eventually lead to a state where the end game condition is satisfied*.

2. $AFPlayer1Wins \stackrel{\text{def}}{=} \mathbf{AF} \text{ winner_player1}$

This property model checks the game to see if *all the computation paths lead Player 1 to win the game hence checking whether the game is totally biased*. Again this is similar to the property for tictactoe explained previously.

3. $Player1StartsPlayer1AFWins \stackrel{\text{def}}{=} (\text{turn} = \text{player1}) \Rightarrow \mathbf{AF} \text{ winner_player1}$

Here we check whether if *Player 1 makes the first move, he/she will always finally win the game*. Please refer to Section 7.4 for more information.

4. $Player1StartsSecondColumnPlayer1AFWins \stackrel{\text{def}}{=} ((\text{turn} = \text{player1}) \wedge (\text{index} = 2)) \Rightarrow \mathbf{AF} \text{ winner_player1}$

This property checks whether if *Player 1 makes the first move and he/she picks the second column, the game will always finally end in his favour*. As with the tictactoe property similar to this one (see *Player1StartsCentrePlayer1AFWins*), one of the first possible moves is for Player 1 to put his marker in the second column. Using this game state this property tries to find out if all game

sessions lead Player 1 to win. Another possible way to do this is to alter the model slightly and make the first move always a mark in the second column. In this way there is only one initial move possible and the model checker verifies the above formula based on this restriction.

5. $Player1StartsSecondColumnPlayer2EFWin \stackrel{\text{def}}{=} ((turn = player1) \wedge (index = 2)) \Rightarrow \mathbf{EF} \text{ winner_player2}$

This property compliments the previous one and checks *if Player 1 starts play by choosing the second column, can Player 2 win?*

6. $Player1ForceWin4Moves \stackrel{\text{def}}{=} \mathbf{EX AX EX AX EX AX EX} \text{ winner_player1}$

Altered for connect four by using four moves instead of three (as connect four needs four-in-a-row), here we check whether *Player 1 can force a win in four moves starting from any of the initial states.*

7. $Player1StartsSecondColumnForceWin4Moves \stackrel{\text{def}}{=} (((turn = player1) \wedge (index = 2)) \Rightarrow \mathbf{EX AX EX AX EX AX EX} \text{ winner_player1})$

In this property we once again combine Properties 4 and 6. The property created in this way asks whether *Player 1 can force a win in four moves if he/she starts play with a move in the second column.*












8. $Player1StartSecondColumnPlayer2ReplyThirdColumnForceWin4Moves \stackrel{\text{def}}{=} (((turn = player1) \wedge (index = 2)) \wedge \mathbf{AX} ((turn = player2) \wedge (index = 3))) \Rightarrow \mathbf{EX AX EX AX EX AX EX} \text{ winner_player1}$

This property asks whether *if Player 1 starts play in the second column, and Player 2 replies in the third column, Player 1 can force win the game in four moves.* As with the similar tictactoe property, the model must be altered so that the first and second moves are no longer random. We force the model to make Player 1's first move a marker in second column and Player 2's first move a marker in the third column. After these moves the locations are selected once more at random. We hence ask whether it is possible using such two moves, for Player 1 to win with his fourth move.

9. $Player1AlwaysFinallyWinBoardConfiguration \stackrel{\text{def}}{=} \mathbf{AG} (\text{board_configuration}_1 \Rightarrow \mathbf{AF} \text{ winner_player1})$

where $\text{board_configuration}_1 \stackrel{\text{def}}{=} (locs_1 = player1) \wedge (locs_2 = empty) \wedge (locs_3 = player2) \wedge (locs_4 = player1) \wedge$

$(locs_1 = player1) \wedge (locs_2 = empty) \wedge (locs_3 = player2) \wedge (locs_4 = player1) \wedge$

7	1	5	2
5	1	4	7

Figure 8.11: Board configuration used to verify Property 9 on a 4×4 connect four. On the left we see the single location model where Player 1 is represented by the White marker while Player 2 is represented by the Black marker. On the right we see the dual location encoding model where the numbers in the locations refer to the ones in Figure 8.10

$$\begin{aligned}
& (locs_5 = player2) \wedge (locs_6 = empty) \wedge (locs_7 = player2) \wedge (locs_8 = player1) \wedge \\
& (locs_9 = player1) \wedge (locs_{10} = empty) \wedge (locs_{11} = player1) \wedge (locs_{12} = player2) \wedge \\
& (locs_{13} = player1) \wedge (locs_{14} = empty) \wedge (locs_{15} = player2) \wedge (locs_{16} = empty)
\end{aligned}$$

defines a particular game board configuration which we believe is a forced win for Player 1. This temporal logic formula describes a property which verifies the system to see whether *it is true that if a certain board configuration occurs during gameplay it is the case that Player 1 always eventually wins*. The board configuration is as follows: $(player1, empty, player2, player1, player2, empty, player2, player1, player1, empty, player1, player2, player1, empty, player2, empty)$, shown in Figure 8.11 (left). Here we have defined the values of the locations in terms of the actual base model with single locations. However given the encoding used in the model, the formula for $board_configuration_1$ using dual locations is $(locs_1 = 5) \wedge (locs_2 = 1) \wedge (locs_3 = 4) \wedge (locs_4 = 7) \wedge (locs_5 = 7) \wedge (locs_6 = 1) \wedge (locs_7 = 5) \wedge (locs_8 = 2)$ represented by game state encoding $(5, 1, 4, 7, 7, 1, 5, 2)$ as shown in Figure 8.11 (right).

10. $Player1ForceWin2MovesBoardConfiguration \stackrel{\text{def}}{=} \mathbf{AG} (board_configuration_1 \Rightarrow \mathbf{EX AX EX} \text{ winner_player1})$

where $board_configuration_1$ is defined as in the previous property. This property states that *it is true that if a certain board configuration occurs during gameplay (again the previously mentioned board configuration $(player1, empty, player2, player1, player2, empty, player2, player1, player1, empty, player1, player2, player1, empty, player2, empty)$) it is the case that Player 1 can force win the game in two moves*. Hence if this property is true the board state is a winning one for Player 1 and that no matter what Player 2 does, he/she has already lost the game proving future moves useless.

8.4.2 μ -calculus Properties

The μ -calculus properties selected are as follows:

1. *AFEndGame*

$$\stackrel{\text{def}}{=} \mathbf{AF} \text{ end_game} \quad (\text{CTL})$$

$$\stackrel{\text{def}}{=} \mu Z.(\text{end_game} \vee \mathbf{AX} Z) \quad (\mu\text{-calculus})$$

This property checks whether the game *always eventually ends*. We require another formula to verify the above property on our model. This is:

$$\forall s. (\text{start_game}(s) \Rightarrow \text{AFEndGame}(s))$$

where s is of type Game State and

$$\text{start_game} \stackrel{\text{def}}{=}$$

$$\begin{aligned} & ((\text{turn} = \text{player1}) \wedge \\ & (\text{locs}_1 = \text{empty} \wedge \text{locs}_2 = \text{empty} \wedge \text{locs}_3 = \text{empty} \wedge \text{locs}_4 = \text{empty} \wedge \\ & \text{locs}_5 = \text{empty} \wedge \text{locs}_6 = \text{empty} \wedge \text{locs}_7 = \text{empty} \wedge \text{locs}_8 = \text{empty} \wedge \\ & \text{locs}_9 = \text{empty} \wedge \text{locs}_{10} = \text{empty} \wedge \text{locs}_{11} = \text{empty} \wedge \text{locs}_{12} = \text{empty} \wedge \\ & \text{locs}_{13} = \text{empty} \wedge \text{locs}_{14} = \text{empty} \wedge \text{locs}_{15} = \text{empty} \wedge \text{locs}_{16} = \text{empty})) \end{aligned}$$

The formula *start_game* represents the set of initial states. If this property is verified to be true the game always finally ends with a win for either of the two players or a draw.

2. *AFPlayer1Wins*

$$\stackrel{\text{def}}{=} \mathbf{AF} \text{ winner_player1} \quad (\text{CTL})$$

$$\stackrel{\text{def}}{=} \mu Z.(\text{winner_player1} \vee \mathbf{AX} Z) \quad (\mu\text{-calculus})$$

This property checks *whether Player 1 always eventually wins the game*. The required complimentary formula is:

$$\forall s. (\text{start_game}(s) \Rightarrow \text{AFPlayer1Wins}(s))$$

where s and *start_game* have the same type and definition as before, respectively. This formula allows us to check for a basic form of fairness even if it is not unbiased. If this formula is true the game is useless in terms of interest as all gameplay sessions lead to Player 1 winning.

3. *AGIfPlayer1StartsSecondColumnAFPlayer1Wins*

$$\stackrel{\text{def}}{=} \mathbf{AG} (\text{Player1SecondColumn} \Rightarrow \mathbf{AF} \text{ winner_player1}) \quad (\text{CTL})$$

$$\stackrel{\text{def}}{=} \nu Z.(\text{Player1StartsSecondColumnImpliesAFPlayer1Wins} \wedge \mathbf{AX} Z) \quad (\mu\text{-calculus})$$

where

$$\begin{aligned} \text{Player1StartsSecondColumnImpliesAFPlayer1Wins} &\stackrel{\text{def}}{=} \\ &\text{Player1StartsSecondColumn} \Rightarrow \text{AFPlayer1Wins} \end{aligned}$$

and where

$$\begin{aligned} \text{Player1StartsSecondColumn} &\stackrel{\text{def}}{=} \\ &((\text{turn} = \text{player2}) \wedge \\ &(\text{locs}_1 = \text{empty} \wedge \text{locs}_2 = \text{player1} \wedge \text{locs}_3 = \text{empty} \wedge \text{locs}_4 = \text{empty} \wedge \\ &\text{locs}_5 = \text{empty} \wedge \text{locs}_6 = \text{empty} \wedge \text{locs}_7 = \text{empty} \wedge \text{locs}_8 = \text{empty} \wedge \\ &\text{locs}_9 = \text{empty} \wedge \text{locs}_{10} = \text{empty} \wedge \text{locs}_{11} = \text{empty} \wedge \text{locs}_{12} = \text{empty} \wedge \\ &\text{locs}_{13} = \text{empty} \wedge \text{locs}_{14} = \text{empty} \wedge \text{locs}_{15} = \text{empty} \wedge \text{locs}_{16} = \text{empty})) \end{aligned}$$

and AFPlayer1Wins is defined as previously.

This formula verifies the game to check if it is *always globally true* that if Player 1 starts play in the second column they *always finally* win the game. The formula $\text{Player1StartsSecondColumn}$ is the set of states where Player 1 has selected to start the game by dropping his marker in the second column. The derived formula $\text{Player1StartsSecondColumnImpliesAFPlayer1Wins}$ refers to the states where if Player 1 starts in the second column imply that they will always finally win. Our final property goes a step further by checking this for all the states of our game system. In order to verify we require the formula:

$$\begin{aligned} \forall s. (\text{start_game}(s) \Rightarrow \\ \text{AGIfPlayer1StartsSecondColumnImpliesAFPlayer1Wins}(s)) \end{aligned}$$

where s and start_game have the same type and definition as before, respectively. We hence verify the property on our system by checking if the latter is true for the set of start state (one state). If the property is true, Player 1 should always start with the second column so as to ensure they will eventually win in all cases.

4. $\text{AGIfPlayer1StartsSecondColumnImpliesEFPlayer2Wins}$

$$\stackrel{\text{def}}{=} \mathbf{AG} (\text{Player1StartsSecondColumn} \Rightarrow \mathbf{EF} \text{winner_player2}) \quad (\text{CTL})$$

$$\stackrel{\text{def}}{=} \nu Z. (\text{Player1StartsSecondColumnImpliesEFPlayer2Wins} \wedge \mathbf{AX} Z)$$

(μ -calculus)

where

$$\text{Player1StartsSecondColumnImpliesEFPlayer2Wins} \stackrel{\text{def}}{=}$$

$$Player1StartsSecondColumn \Rightarrow EFPlayer2Wins$$

where

$$EFPlayer2Wins$$

$$\stackrel{\text{def}}{=} \mathbf{EF} \text{ winner_player2} \quad (\text{CTL})$$

$$\stackrel{\text{def}}{=} \mu Z. (\text{winner_player2} \vee \mathbf{EX} Z) \quad (\mu\text{-calculus})$$

and $Player1StartsSecondColumn$ is defined as before.

This formula complements the previous one by checking whether it is *always the case that Player 2 has the possibility of winning if Player 1 starts play in the second column*. $EFPlayer2Wins$ refers to the states where starting from the latter paths exist which finally lead to Player 2's winning states. The formula $Player1StartsSecondColumnImpliesEFPlayer2Wins$ is true for those states where if Player 1 initially plays as mentioned earlier, there exists the possibility that Player 2 wins. The complete formula checks whether this is true for all the states on all the paths of the model. The required formula to model check is:

$$\forall s. (\text{start_game}(s) \Rightarrow$$

$$AGIfPlayer1StartsSecondColumnImpliesEFPlayer2Wins(s))$$

where s and start_game have the same type and definition as before, respectively.

$$5. AGIfPlayer1StartsSecondColumnPlayer2RepliesFirstColumnImpliesAFPlayer1Wins$$

$$\stackrel{\text{def}}{=} \mathbf{AG} (\text{Player1StartsSecondColumnAndPlayer2RepliesFirstColumn} \Rightarrow \mathbf{AF} \text{ winner_player1}) \quad (\text{CTL})$$

$$\stackrel{\text{def}}{=} \nu Z. (\text{Player1StartsSecondColumnAndPlayer2RepliesFirstColumnImpliesAFPlayer1Wins} \wedge \mathbf{AX} Z) \quad (\mu\text{-calculus})$$

where

$$Player1StartsSecondColumnAndPlayer2RepliesFirstColumnImpliesAFPlayer1Wins$$

$$\stackrel{\text{def}}{=} \text{Player1StartsSecondColumnAndPlayer2RepliesFirstColumn} \Rightarrow \text{AFPlayer1Wins}$$

and where

$$Player1StartsSecondColumnAndPlayer2RepliesFirstColumn \stackrel{\text{def}}{=}$$

$$((\text{turn} = \text{player1}) \wedge$$

$$(\text{locs}_1 = \text{player2} \wedge \text{locs}_2 = \text{player1} \wedge \text{locs}_3 = \text{empty} \wedge \text{locs}_4 = \text{empty} \wedge$$

$$\text{locs}_5 = \text{empty} \wedge \text{locs}_6 = \text{empty} \wedge \text{locs}_7 = \text{empty} \wedge \text{locs}_8 = \text{empty} \wedge$$

$$\text{locs}_9 = \text{empty} \wedge \text{locs}_{10} = \text{empty} \wedge \text{locs}_{11} = \text{empty} \wedge \text{locs}_{12} = \text{empty})$$

$$locs_{13} = \text{empty} \wedge locs_{14} = \text{empty} \wedge locs_{15} = \text{empty} \wedge locs_{16} = \text{empty}))$$

and $AFPlayer1Wins$ is defined as previously. Here we check whether if Player 1 starts in the second column and Player 2 replies in the first column, Player 1 always finally wins. The formula $Player1StartsSecondColumnAndPlayer2RepliesFirstColumn$ represents the set of states where Player 1 has started with the second column and Player 2 has replied with the first column. The formula $Player1StartsSecondColumnAndPlayer2RepliesFirstColumnImpliesAFPlayer1Wins$ refers to the set of states where if the former subformula is true this implies that Player 1 is always finally the winner. Using our property we can check this for all the states on all the paths of the system. Moreover, we require the formula:

$$\forall s. (start_game(s) \Rightarrow$$

$$AGIfPlayer1StartsSecondColumnPlayer2RepliesFirstColumnImpliesAFPlayer1Wins(s))$$

where s and $start_game$ have the same type and definition as before, respectively.

6. $AGIfPlayer1StartsSecondColumnPlayer2RepliesFirstColumnImpliesEFPlayer2Wins$

$$\stackrel{\text{def}}{=} \mathbf{AG} (Player1StartsSecondColumnAndPlayer2RepliesFirstColumn \Rightarrow \mathbf{EF} \text{ winner_player2}) \quad (\text{CTL})$$

$$\stackrel{\text{def}}{=} \nu Z. (Player1StartsSecondColumnAndPlayer2RepliesFirstColumnImpliesEFPlayer2Wins \wedge \mathbf{AX} Z) \quad (\mu\text{-calculus})$$

where

$$Player1StartsSecondColumnAndPlayer2RepliesFirstColumnImpliesEFPlayer2Wins$$

$$\stackrel{\text{def}}{=} Player1StartsSecondColumnAndPlayer2RepliesFirstColumn \Rightarrow EFPlayer2Wins$$

This property checks whether it is always the case that if Player 1 starts play in the second column and Player 2 replies in the first one, there exists a path which eventually leads Player 2 to win.

$Player1StartsSecondColumnAndPlayer2RepliesFirstColumnImpliesEFPlayer2Wins$ represents the set of states where if Player 1 and Player 2 make the previously mentioned moves it is true that Player 2 may be finally the winner. The full property proves this for all the states of the game model. We also require the formula:

$$\forall s. (start_game(s) \Rightarrow$$

$$AGIfPlayer1StartsSecondColumnPlayer2RepliesFirstColumnImpliesEFPlayer2Wins(s))$$

where s and $start_game$ have the same type and definition as before, respectively.

7. $Player1ForceWin \stackrel{\text{def}}{=}$

$$\mu Z. (winner_player1 \vee ((turn = player1 \wedge \mathbf{EX} Z) \vee (turn = player2 \wedge \mathbf{AX} Z)))$$

This property checks whether *Player 1* may force a win in a number of moves starting from a particular board state. The result of this property shows whether there exist board states where a player may win no matter what their opponent does. In order to model check this property we require the formula:

$$\exists s. Player1ForceWin(s)$$

where s is of type Game State. Using this formula we check for the existence for such states.

8. *Player1CanForceAWinIfStartsSecondColumn*

$$\stackrel{\text{def}}{=} Player1StartsSecondColumn \wedge Player1ForceWin$$

where *Player1StartsSecondColumn* and *Player1ForceWin* are defined as before. This property verifies whether it is possible for *Player 1* to force a win in a number of moves if they start play in the second column. We use the formula:

$$\exists s. Player1CanForceAWinIfStartsSecondColumn(s)$$

where s is of type Game State, to check the existence of such states.

9. *Player1CanForceAWinIfPlayer1StartsSecondColumnAndPlayer2RepliesFirstColumn* $\stackrel{\text{def}}{=}$

$$Player1StartsSecondColumnAndPlayer2RepliesFirstColumn \wedge Player1ForceWin$$

where *Player1StartsSecondColumnAndPlayer2RepliesFirstColumn* and *Player1ForceWin* are defined as previously. This shows whether if *Player 1* starts play in the second column and *Player 2* replies in the first column, *Player 1* can force a win in a number of moves. To check this property we require the formula:

$$\exists s. Player1CanForceAWinIfPlayer1StartsSecondColumnAndPlayer2RepliesFirstColumn(s)$$

where s is of type Game State.

10. *Player2MayDraw* $\stackrel{\text{def}}{=}$

$$\mu Z. (draw_game \vee ((turn = player1 \wedge \mathbf{EX} Z) \vee (turn = player2 \wedge \mathbf{EX} Z)))$$

This property defines the set of states where *Player 2* may attempt to draw the game. We can model check the game model using this property by means of the additional formula:

$$\forall s. (start_game(s) \Rightarrow Player2MayDraw(s))$$

where s is of type Game State.

8.5 Verifying Connect Four

Verifying connect four was achieved in the same way as was done for tictactoe. A model generator was written in C#. This generator is able to automatically create $x \times y$ connect four games in both SMV and μcke syntax allowing us to obtain incrementally larger models. It does so by creating the start game definition, the transition relation and the end-game definitions. As for tictactoe, the specifications have to be entered manually as they depend on what the user would like to verify about the game. In our case we entered the properties mentioned above, making the necessary modifications for different board sizes. Using the two model checkers the generated models were then checked to see if they adhered to the specifications.

8.6 Results

Verifying connect four of different sizes using the above specifications has yielded the following results as shown in the tables in the following pages.

SMV Results

Verification Result		Property									
Board Size		1	2	3	4	5	6	7	8	9	10
	4 × 4	C	C	FALSE	FALSE	C	C	FALSE	FALSE	FALSE	FALSE
	4 × 5	DNF	C	DNF	C	C	DNF	C	FALSE	C	C
	5 × 4	DNF	DNF	C	C	C	DNF	DNF	DNF	DNF	DNF
	5 × 5	C	C	C	DNF	DNF	C	DNF	C	C	C

Table 8.1: Connect Four Verification Results for the *CTL* properties verified using SMV

Verification Time		Property									
Board Size		1	2	3	4	5	6	7	8	9	10
	4 × 4	C	C	201.484 s	161.625 s	C	C	283.25 s	26.3281 s	176.141 s	90.3438 s
	4 × 5	DNF	C	DNF	C	C	DNF	C	119 s	C	C
	5 × 4	DNF	DNF	C	C	C	DNF	DNF	DNF	DNF	DNF
	5 × 5	C	C	C	DNF	DNF	C	DNF	C	C	C

Table 8.2: Connect Four Verification Times for the *CTL* properties verified using SMV

Counterexample Trace		Property									
Board Size		1	2	3	4	5	6	7	8	9	10
	4 × 4	C	C	Full	Full	C	C	No	No	Full	Partial
	4 × 5	DNF	C	DNF	C	C	DNF	C	No	C	C
	5 × 4	DNF	DNF	C	C	C	DNF	DNF	DNF	DNF	DNF
	5 × 5	C	C	C	DNF	DNF	C	DNF	C	C	C

Table 8.3: Connect Four Counterexample Trace Results for the *CTL* properties verified using SMV

State Variable Count		Property									
Board Size		1	2	3	4	5	6	7	8	9	10
	4 × 4	C	C	27	27	C	C	27	27	27	27
	4 × 5	DNF	C	DNF	C	C	DNF	C	35	C	C
	5 × 4	DNF	DNF	C	C	C	DNF	DNF	DNF	DNF	DNF
	5 × 5	C	C	C	DNF	DNF	C	DNF	C	C	C

Table 8.4: Connect Four State Variable Counts for the *CTL* properties verified using SMV

Notes:

“DNF” Did Not Finish: SMV was stopped when paging of memory started to take place and the processor was spending approximately 0% working on SMV verification.

“C” Crashed: SMV crashed with an exception during verification.

μ cke Results

Verification Result		Property									
		1	2	3	4	5	6	7	8	9	10
Board Size	4 × 4	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	FALSE	TRUE
	4 × 5	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	FALSE	TRUE
	5 × 4	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	FALSE	TRUE
	5 × 5	E	FALSE	FALSE	TRUE	FALSE	TRUE	E	E	E	TRUE
	5 × 6	DNF	DNF	DNF	DNF	DNF	DNF	DNF	DNF	DNF	DNF

Table 8.5: Connect Four Verification Results for the μ -calculus properties verified using μ cke

Verification Time		Property									
		1	2	3	4	5	6	7	8	9	10
Board Size	4 × 4	2.38 s	0.93 s	1.27 s	1.85 s	1.29 s	1.84 s	2.11 s	2.08 s	2.06 s	1.45 s
	4 × 5	41.2 s	31.2 s	35.4 s	37.6 s	35.7 s	37.8 s	106 s	104 s	105 s	29.7 s
	5 × 4	102 s	40.6 s	53.4 s	81.4 s	54.6 s	82.3 s	238 s	204 s	210 s	75.1 s
	5 × 5	E	1144 s	1512 s	1174 s	1714 s	1156 s	E	E	E	1018 s
	5 × 6	DNF	DNF	DNF	DNF	DNF	DNF	DNF	DNF	DNF	DNF

Table 8.6: Connect Four Verification Times for the μ -calculus properties verified using μ cke

Notes:

“DNF” Did Not Finish: μ cke was stopped when paging of memory started to take place and the processor was spending approximately 0% working on μ cke verification.

“E” Error: ABCD BDD package reported that the maximum number of nodes was exceeded.

8.7 Conclusion

In this chapter we have shown how a game such as Connect Four may be modelled using an appropriate modelling structure and how it may be specified using *CTL* and μ -calculus. Verification was achieved using the same two model checkers used for tictactoe in the previous chapter. The results obtained through verification reflect the ability of model checking to scale up to large connect four board sizes.

In the next chapter we will evaluate the results for both tictactoe and connect four and discuss the ability of model checking for the verification of games. Moreover we will compare and contrast model checking with other known forms of game verification methods.

EVALUATION & RELATED WORK

One of the greatest resources people cannot mobilize themselves is that they try to accomplish great things. Most worthwhile achievements are the result of many little things done in a single direction.

Nido Qubein

9.1 Overview

In the previous chapters have seen how model checking may be applied to games and how it enables us to obtain some interesting results regarding the games' properties. In this chapter we will evaluate whether model checking is in fact appropriate for game systems by discussing the results obtained. We will also compare model checking to other work which has been applied to games such as theorem proving, minimax analysis, retrograde searching and alpha-beta searching.

9.2 Evaluation of Results

The results we obtained show that model checking may be applied to proving properties about games. The modelling step was achieved effortlessly because modelling a game reduces to translating the latter into a Kripke structure. As we have seen this is a fairly straight forward task since the states of the Kripke structure may be represented by the board state, while the transitions of the

same structure may be immediately mapped to moves which the game rules allow the players to perform on the game board. Specification was done by writing the required properties in *CTL* and μ -calculus temporal logics. The verification process was accomplished as with any other system by the use of the SMV and μ cke model checkers.

Analysing the results for both games using the two model checkers gives us reason to believe that although model checking may be utilised for games, the highly combinatorial nature of such systems causes problems in verification. This can be immediately seen from the results for the two games verified using both *CTL* -based and μ -calculus -based model checking. For example, in our first experiment for tictactoe with SMV the properties were verified quickly (1.25 seconds or less) for a small board of 3×3 . However, a slight increase of the board size to 4×4 caused a drastic increase in verification time with some properties failing to verify altogether due to system limitations. Considering that the increase in state variable count is just fourteen (23 for 3×3 tictactoe to 37 for 4×4 tictactoe) we believe that with some minor modifications to the model written in SMV we may achieve better results. For example, a simple encoding which groups the columns of the tictactoe board as one unit, instead of just separate board locations allows for fewer state variables and may contribute to better overall results. The only problem with this encoding is the complexity in creating an $n \times n$ generator for different board sizes as we required. We have proposed and used a similar encoding for connect four to the one suggested for tictactoe which however groups every two board locations in a column. The results for this game have in fact been more favourable considering that its smallest board size is still larger than the smallest tictactoe board. Thanks to our encoding the initial state variable count for a 4×4 connect four was reduced from 37 to 27 variables allowing for larger board sizes to be model checked. Another available option which could possibly allow us to model check larger board sizes is to try to use state space reduction techniques such as abstraction, symmetry and induction (see [20] for more information on these subjects). Of these we believe that symmetry might be the most appropriate. As an example consider how in 3×3 tictactoe all the four corners may be mapped to one another allowing us to consider just one corner. If this is applied to the centre location and the side locations, we could in fact just model check the games for three types of locations instead of nine separate ones. The only problem with this reduction technique is related to its ability to scale up and the complexity in writing an $n \times n$ model generator. It is still however an interesting area of study to see how model checking and

symmetry behave for game systems.

Hence, on the whole our results reflected the fact that since board size is the major contributor to the size of the state space, the larger the board size the longer the time for verification and the lesser the chance of verification success. As another example consider the verification times for connect four verified using μcke . We notice how for a 4×4 board size the verification times ranged from a meagre 0.97–2.38 seconds up to a maximum of almost half an hour for a board size of just 5×5 . On first inspection one might think that our systems are suitable for the MBFS used in SMV and μcke allowing for short verification times even for larger board sizes than 5×5 . The computation trees of our game systems are more inclined towards breadth rather than length due to the fact that initially the board allows for a maximum number of moves which decreases gradually hence allowing the tree to spread out quickly at first and starting to taper on most paths. Moreover, the length of the deepest path is at most the size of the board itself. This contrasts with other games such as chess and checkers where loops exist in the model of such games because a some moves may be reversed and redone for example. In fact for small board sizes model checking was quite successful as we mentioned earlier. Problems however arose as the depth was increased. In all cases a small increase in depth has caused a significant toll on model checking times. For example: an increase of depth 7 from a tictactoe maximum depth of 9 for 3×3 to a maximum depth of 16 for 4×4 has almost doubled the depth of the computation tree. This fact is clearly reflected in the difference in verification times for these two board sizes. The impact is less noticeable for the board sizes of connect four due to the smaller increments in board size.

Overall, if we compare the results for tictactoe and connect four for both SMV and μcke we realise that μcke has provided us with better results than SMV. The reasons for this could be numerous. The OBDD libraries for SMV and μcke might have different implementations related to how variable orderings are chosen which could cause performance differences between the two model checkers. Since we are dealing with OBDDs different variable orderings might cause a great impact on verification time. The results seem to suggest that μcke 's variable ordering is more suited for games than SMV's. A slight extension to our work would be to provide pre-defined variable orderings for both model checkers which might be more suitable for our systems hence allowing a better comparison. It might also however be the case that the state space of our games themselves represent a difficult case for the OBDD representation in a similar fashion to the difficulty this rep-

representation has with the representation of integer multiplication as seen in [11]. It would be an interesting extension to our work to see how other representations besides OBDDs, such as SAT-based, fair with game systems. Another reason which could have lead to the discrepancy between the two model checker's results relates to the fact that *CTL* -based model checking is iterative in nature and a considerable number of intermediate steps are required to model check a system through *CTL* especially if we have a substantial number of nested temporal operator pairs. These operator pairs have to be worked out one at a time starting from the inner formula and expanding outwards until the entire formula is model checked. The μ -calculus model checking on the other hand does not behave in this manner. The fixpoint representation of formulas allows us to express most properties using one recursive definition even if in *CTL* they involve a lot of nesting. Hence for most properties written in μ -calculus there are no subformulas to be worked out before the final property may be verified allowing perhaps verification to be much quicker and available for larger board sizes.

An important thing to notice about our model checking results is that properties which involve existential quantification or alternating path quantifiers are problematic when model checking games. In fact for tictactoe using SMV we encountered our first problem verifying properties when we attempted to verify that there exist paths which allow Player 2 to win if Player 1 starts gameplay in the centre location. Moreover, the properties which checked for the existence of force win states using alternating path quantifiers for Player 1 both unrestricted or restricted to for example starting in the centre location, have failed to verify for quite a small tictactoe board size. This is also the case for μ cke results, this time for a board size of 5×5 . Connect four results follow in the same manner for SMV and μ cke. The difference in the model checkers and hence the type of model checking is again shown here. The μ -calculus model checker behaves as we observed for connect four as it did for tictactoe. It found some problems when model checking properties involving existential quantifiers and alternating path quantifiers. In fact the first problems were encountered when attempting to verify properties involving alternating path quantifiers for this game. SMV's results for connect four behave in a slightly different manner. While a property for unrestricted forced winning did not verify, attempting to verify properties related to forced wins restricted to a certain initial moves was successful, at least for the smallest board size of 4×4 . These results seem to suggest two things:

- It would be interesting to know how *ATL* compares to both *CTL* and μ -calculus as this temporal logic has been specifically created for alternating path quantifiers. Writing properties in *ATL* and model checking them with an appropriate model checker such as Mocha should be an interesting way to compare the three temporal logics and moreover to attempt to determine which temporal logic is more suited for games. As we mentioned earlier *CTL* is not able to express certain properties about games. We may find whether *ATL* is enough or whether μ -calculus is still required for game systems.
- SMV's results for connect four seem to suggest that perhaps larger board sizes may be model checked if we write a number of different models for every board size, where each model differs from the other by where Player 1 does their first move. It should be then possible to combine the results together for all the separate models for the result of the actual model.

Another minor observation related to connect four is that our results indicate that a 4×5 board size is clearly different from a 5×4 board size even if the number of board locations is 20 in both cases. This can be deduced from the verification times which indicate that μ cke took longer to model check the 5×4 board size than the 4×5 one. The factors leading to this could be numerous as the two board sizes' state space, transition relation and computation tree's overall structure differ completely from one another even if they share the same amount of locations and computation tree maximum depth. A more thorough comparison can be attained if we manage, in future work, to verify larger cases in both SMV and μ cke.

As we have stated earlier in Chapter 3, temporal logics have different expressiveness depending on their definition. This translates to what properties they can express about a particular system. A part of our study was concerned with comparing two such key temporal logics: *CTL* and μ -calculus. We wanted to know whether *CTL* is enough to express all the properties one would like to verify on game systems or whether such a temporal logic restricted us by what it can express. We have seen through some of the properties we have written that some properties in μ -calculus cannot be expressed by *CTL*. An example of this are the formulas which relate to forced wins. It was found impossible to write *CTL* formulas which allowed us to check whether a player may force win a game in an unpredetermined number of moves. *CTL* limited us to write properties with a preknown, fixed number of moves while μ -calculus allowed us to remove this upper limit. As an example of this consider the μ -calculus formula:

it is similar to model checking in the sense that it views a game as a system of equations. These equations follow strictly the game rules themselves (such as for example how a piece may move) and are applied on the current state of the game to produce a number of possible next states. Theorem proving has allowed researchers to create what are known as endgame databases. These databases store all the sequences of all the moves which may originate from a particular board configuration [32]. In [32] Hurd discusses this subject matter. His work is related to ours due to the use of BDDs which he employs with the HOL4 theorem prover to construct an endgame database for all the four piece pawnless positions possible of chess. As with our case, BDDs help him produce compact representations of the states of the game, in similar way to how we represent sets of states from our game model's state space using OBDDs. Also by means of his work he has found which states consisting of a small number of end pieces are force win ones for a player. We attempted to obtain these states as well by the use of the forced win μ -calculus formula mentioned earlier.

Min-max or minimax analysis [30] is a form of search whereby each node in the computation tree is given a value depending on how profitable it is for the player if he/she is in that particular state. The value of each node is determined by a heuristic evaluation function. The search algorithm attempts to always pick the most profitable state of those available for a player thereby maximising the chance of him/her winning and subsequently attempting to minimise the chance of a favourable outcome for the opponent. To do so a search is done and if in the available nodes it is the player's turn to move, the maximum value node is selected, while if it is the opponent's turn, the node with the minimum value is taken as a move. The maximum value node is taken in the player's turn because it reflects that the player is trying to maximise their chance of winning while the minimum value node is taken for the opponent's turn because it reflects that the opponent is also trying to win for themselves [30]. Minimax has been applied for various games, as it can be used to attempt to find the best gameplay possible for a player playing a particular game. A form of minimax has been used by Victor Allis to solve the game of connect four which was found out to be a win for Player 1 if perfect play is employed [1]. If we compare minimax analysis to model checking we see that they are both exhaustive in nature, that is, all paths are considered thoroughly. However, while minimax gives weights to paths according to perceived notions by the researcher, thus preferring some paths over others to reach its required states and potentially ignoring certain

paths which could turn out to be advantageous, model checking is less forgiving as it attempts all the paths equally unless it is restricted otherwise. Compared to minimax, model checking is more thorough and takes no chances. Minimax has however been used with considerable success due to its practical approach.

Two important search techniques (often employed with minimax analysis) which have been used for games are those of retrograde search and alpha-beta search [41]. In retrograde search we start from a goal state we want to reach, such as for example a forced win state for a player, and we work backwards by means of depth first search tree to find which states may lead us to such a desired state. This fact makes retrograde search a form of backwards search in the computation tree. Alpha-beta search on the other hand is the opposite of retrograde analysis. It starts from a start state and finds which moves lead to a winning state by means of a depth first search on the search tree. Combining these two techniques provides with the advantages of both together. Simplistically, alpha-beta is used to start searching from the start state while in parallel retrograde analysis works in reverse from the desired state until there is a meeting point at a particular level of depth for both. These two techniques have been applied separately and together to study and solve various games, often with success. An example of this is the work by Gasser who showed how using retrograde and alpha-beta he can solve a game called nine men's morris [29]. His results show that the game is in fact a draw. More recently (2007) checkers has been solved as a draw in a similar manner by a team consisting of Schaeffer et al. [41], again using the same two techniques. Their approach using these two techniques has some similarities and differences to our work. In contrast to these two techniques, as we mentioned earlier, model checking utilises a breadth first search instead of a depth first search. However both techniques involve a search of all the required paths which meet specification, such as "the states where Player 1 can force a win". Due to its forward search method, our model checking approach to games may be seen to resemble alpha-beta searching which Gasser and Schaeffer et al. use more than it does with their use of retrograde. Both model checking and alpha-beta, when used with games start from a initial board location and work onwards until they find the required board states which they want. Also it is obvious that in our approach to game verification we did not use backwards searching techniques such as retrograde as this is not used in model checking.

During the course of this body of work, we came across a paper by Zheng Zhang which discusses

how model checking may be used to verify properties about a 3×3 tictactoe [43]. Zhang attempts to model check a property for the states which are forced wins for a tictactoe player using bounded *CTL*, *ATL* and μ -calculus. He successfully manages to use SMV to verify this property for *CTL* whilst failing in his attempt with μ cke (as we did at first) and mocha which he wished to use for the other temporal logics. Our work can be seen as an extension of Zhang's work as our approaches are very similar to each other. We however extend more on Zhang's work by checking more properties and attempting larger board sizes for both tictactoe and connect four.

Another paper which describes work related to our own is by Madhusudan et al. [36]. In [36] Madhusudan et al. take a more generic approach than ours and although they consider symbolic model checking as a technique to solve certain simple games which they call Pursuit-Evasion and Swap and extract properties about them, they also discuss other symbolic methods, of note being the SAT-based symbolic technique and *ATL* model checking. Similarly to us they use μ cke and have also tried board sizes of varying sizes. Their results which concern the use of BDDs compare to our results by the fact that with increasing board sizes, symbolic model checking techniques suffered, sometimes not managing to finish after a considerable amount of verification time.

9.4 Conclusion

In this chapter we have evaluated our results and what they reflect on model checking when applied to games. All things considered, our results show that we were successful in what we set out to do. Model checking is a reasonable technique to deduce properties about games and with some further work and improvements it may even become comparable to the other methods we mentioned earlier. In the next chapter we will discuss any potential research areas which stem from our work and give our final conclusions regarding our research.

FUTURE WORK & CONCLUSION

I never think of the future.

It comes soon enough.

Albert Einstein

10.1 Overview

In this chapter we will examine some possible future work which may be derived from our study and present our final remarks and conclusions.

10.2 Future Work

Our results suggest that standalone symbolic model checking is a bit limited when trying to prove properties about games. To mitigate this problem we have made use of suitable encodings. This however is not always enough since there are scaling problems involved and the complexity of the model written for the model checker makes the former hard to write. A new study could be carried out to find out if an existent state space reduction technique may be used to help mitigate the state space explosion problem. For example, we believe that despite the fact that it might make model writing for a model checker harder, symmetry is a viable technique which could substantially reduce our models and enable us to check larger board sizes. Induction on the board size may also be perhaps attempted. If no existent technique is found to satisfy our needs, perhaps the

study carried out could attempt at discovering a technique which is domain specific for the model checking of games.

As suggested by Zhang's [43] and Madhusudan et al.s' [36] work and our study, an interesting extension of our work would be to utilise *ATL* model checking for games. A similar study to ours but which includes *ATL* would allow us to contrast and compare *ATL* with *CTL* and μ -calculus in relation to game systems. We can once more, for example, find out if *ATL* is enough or whether μ -calculus is still required to prove certain properties about games. *ATL* is seen as a good candidate since games can be seen as an instance of one of the classes of systems it was specifically designed for. Moreover, alternating path quantifiers which we often make use of whilst proving our properties about games seem to be quite heavy for *CTL* model checking. It would be interesting to see how *ATL* model checking is able to scale up for larger board sizes even though it makes use of alternating path quantifiers.

In our study we focused on symbolic model checking based on binary decision diagrams. It would be interesting to compare BDDs with other symbolic techniques such as SAT-based model checking in a similar fashion to Madhusudan et al.s' work. Model checkers like NuSMV which support both OBDD- and SAT-based algorithms may allow us to compare the two's different verification times when applied to verifying properties about games.

Using model checking it is possible to extend our work to carry out a study on game strategies and their impact upon a particular game. In our work this subject matter was broached briefly and we saw how strategies can be included either through properties or by means of a change in the model itself as part of the transition relation. Model checking may be used to attempt to find the success rates of possible gameplay strategies. An example of this is the strategy used for "perfect" gameplay, which when found about a particular game, allows us to find out whether an opponent may always draw the game, making thus the game fair, or whether it is biased towards a player.

10.3 Conclusion

Despite initial problems we had due to the lengthy verification times for even some of the smaller board sizes, the overall result was positive and we achieved what we set out to do. We managed to apply model checking to game systems and have used it to verify a number of properties about two

commonly-played games, tictactoe and connect four using SMV for *CTL* verification and μ cke for μ -calculus verification. Moreover since the properties were written in both *CTL* and μ -calculus we were able to see if *CTL* was capable enough alone to express any required game-related property. As we have seen this is not the case as it cannot be used to write some very important properties.

Another factor we studied was scalability. We believe that with a little more work and improvement model checking may be used to study games with larger board sizes such as for example the standard 7×6 connect four which managed to elude us and perhaps with time and research, other, more complex games such as checkers and chess. Eventually model checking might be added to the current collection of tools used to study games as another, viable approach.

GAME MODEL GENERATOR TOOL

*We shall neither fail nor falter; we shall not weaken
or tire...give us the tools and we will finish the job.*

Winston Churchill

A.1 Overview

This appendix is dedicated to the game model generator we wrote as a tool to help us with our study. The model generator was written in .Net 2.0 using the C# language. It is able to create models of tictactoe and connect four of size up to 9×9 which may be model checked with SMV and Mucke. The script generated is correct in terms of the two model checkers' syntax language and only requires the specification properties to be added after generation is complete. This is possible through a basic editor which forms part of the tool itself. After the model has been generated and the specifications added, the tool allows the user to save the script file in a format which may be immediately used by the respective model checker.

A.2 Aim

The aim behind our tool is to assist us in getting an indication of the scalability of model checking with game board size. Since using our encodings, creating a 3×3 game model for both games is the same as creating one for a 4×4 , 5×5 and so on, we decided to automate the task.

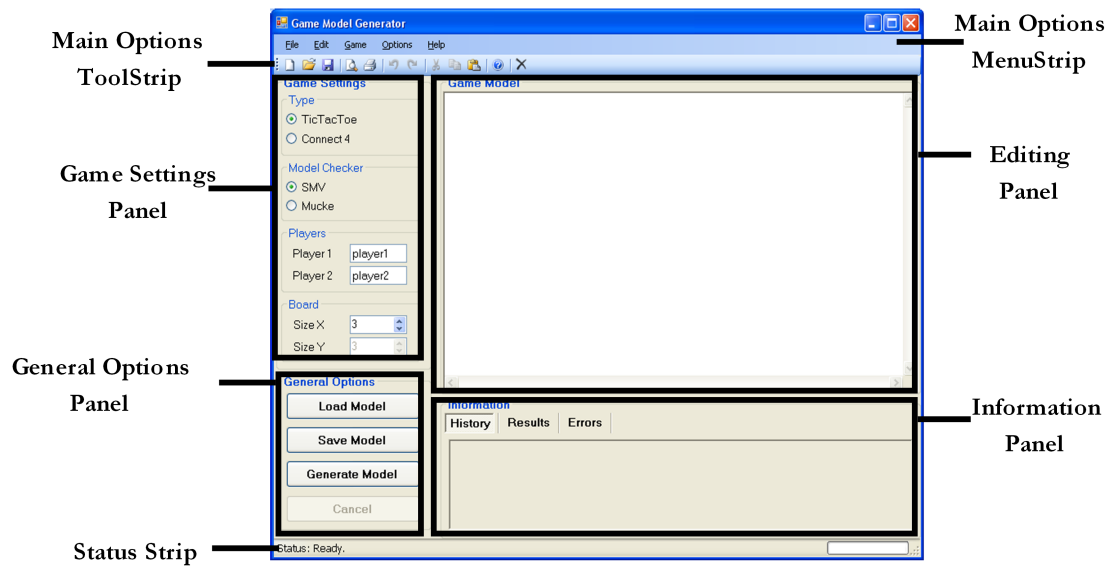


Figure A.1: Game Model Generator Tool Main Window

A.3 Options

The program consists of a main window and a help window. The main window is divided into seven sections as shown in Figure A.1. The sections are:

Main Options Menustrip

This section contains:

- Menu Options which give the user the option to create, open, save, print preview and print model files.
- Editing Options such as undoing, redoing, cutting, copying and pasting.
- Game Parameter Options such as game type, model checking language, player identifiers, board size and model generation.
- Editor Options related to what panels of the interface are hidden or visible and word warp.
- Help on the use of the model generator.

Main Options Toolstrip

This section replicates some of the common options from the Main Options Menustrip for easy, one-click use. Options available are:

- creating, opening, saving, print-previewing and printing game model files
- cutting, copying, pasting text from a model file
- help on the use of the model generator
- exiting the model generator

Game Settings Panel

This panel allows the user to edit the game model parameters prior model generation. Parameters available are:

- **Game Type** — Select between Tictactoe and Connect Four.
- **Model Checker** — Select between SMV and Mucke model checker.
- **Players** — Player 1 and Player 2 identifiers may be entered here.
- **Board Size** — Enter the size of the board game. The size of the sides of the board for Tictactoe ranges from 3 to 9 while that for Connect Four ranges from 4 to 9. Note also that Tictactoe's board size is restricted to grow only in a square fashion.

General Options Panel

This panel allows the user to:

- Load a Model in both Mucke or SMV using a Load File Dialog.
- Save a Model as a Mucke Model or SMV Model or a generic text format using a Save File Dialog.
- Generate a Model using the game parameters set up before in the Game Settings Panel. The newly generated model appears in the Editing Panel.

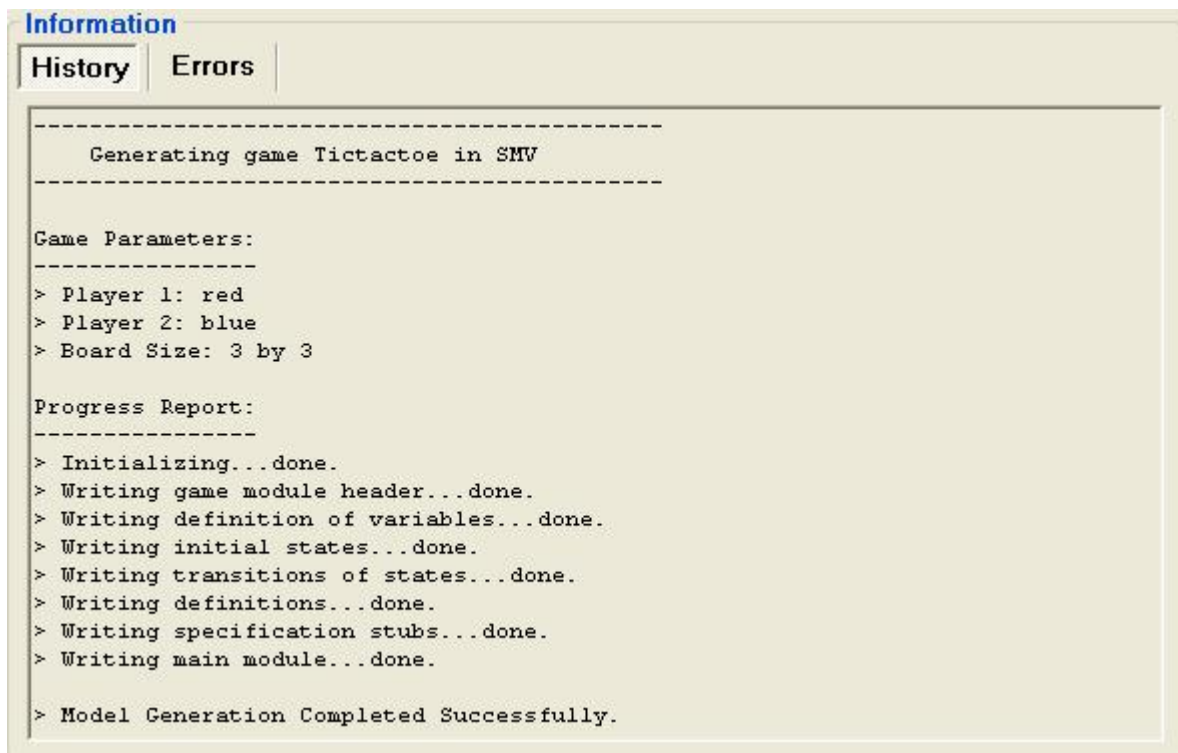


Figure A.2: Game Model Generator Information Panel History Tab Window

Editing Panel

The Editing Panel consists of a basic editor which has the following options:

- Displays a generated model.
- Allows editing of the generated model.
- Allows the addition of specification properties before the model is saved for use with the appropriate model checker.

Information Panel

The Information Panel is divided into two informative tabs:

History — displays model generation history. As an example see Figure A.2.

Errors — displays any errors which may occur prior generation. The errors handled are:

- One or both of the players' identifiers are missing.



Figure A.3: Game Model Generator Information Panel Errors Tab Window

- Identifier(s) entered is incorrect according to the model checker. Identifiers should start with an alphabetic character or underscore and proceed with an alphanumeric character or underscore.
- Identifier(s) entered is a model checker language reserved word.
- Identifier(s) cannot be the same.

As an example consider Figure A.3.

Status Strip

This is a status bar which displays information of immediate attention to the user. Examples of status messages are:

- Whether the tool is ready for model generation.
- Whether errors were found which do not allow model generation to occur.
- The current step being performed during model generation, such as generation of definition of the initial states or the transition relation.
- Whether model generation was completed successfully.

A.4 User Manual

A.4.1 Generating a new Game Model

To generate a model one must follow the following steps:

1. Select the Game Type (Tictactoe or Connect Four).
2. Select the Model Checker (SMV or μ cke).
3. Enter two variable names, one for each player.
4. Select the board size required.
5. Generate the Model by clicking on the Generate Model Button.
6. Enter the specification properties required where suggested in the generated script.
7. Save the script.
8. Use SMV or μ cke to run the script depending on the Model Checker selected in 2.

A.5 Contents of CD-ROM

The cd-rom contains:

- The Visual Studio .Net 2005 project and source code
- The tool executable
- The generated scripts we used to verify the games
- A soft copy of this document.

BIBLIOGRAPHY

- [1] Victor Allis. A knowledge-based approach of connect-four - the game is solved: White wins.
- [2] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, 2002.
- [3] Rajeev Alur, Thomas A. Henzinger, Freddy Y. C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. MOCHA: Modularity in model checking. In *Computer Aided Verification*, pages 521–525, 1998.
- [4] Henrik Reif Andersen. An introduction to binary decision diagrams, October 1997. notes for 49285 Advanced Algorithms E97.
- [5] André Arnold and Damian Niwiński. *Rudiments of μ -calculus*, volume 146 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 2001.
- [6] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 164–176, New York, NY, USA, 1981. ACM Press.
- [7] A. Biere. *Efficient Model Checking of the Mu-Calculus with Binary Decision Diagrams*. PhD thesis, Karlsruhe University, Karlsruhe, Germany, 1997.
- [8] A. Biere. mucke — efficient mu-calculus model checking. *International Conference on Computer-Aided Verification*, 1254:468–471, 1997.

- [9] J. Bradfield and C. Stirling. Modal logics and mu-calculi: an introduction, 2001.
- [10] J. Bradfield and C. Stirling. Modal mu-calculi, 2006.
- [11] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
- [12] Randal E. Bryant. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *IEEE/ACM International Conference on Computer Aided Design, ICCAD, San Jose/CA*, pages 236–243. IEEE CS Press, Los Alamitos, 1995.
- [13] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.
- [14] R. Cavada, A. Cimatti, G. Keighren, E. Olivetti, M. Pistore, and M. Roveri. *NuSMV 2.4 Manual*. ITC-irst and CMU, Via Sommarive 18, 38055 Povo (Trento) – Italy.
- [15] R. Cavada, A. Cimatti, G. Keighren, E. Olivetti, M. Pistore, and M. Roveri. *NuSMV 2.4 Tutorial*. ITC-irst and CMU, Via Sommarive 18, 38055 Povo (Trento) – Italy.
- [16] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of LNCS, Copenhagen, Denmark, July 2002. Springer.
- [17] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings Eleventh Conference on Computer-Aided Verification (CAV’99)*, number 1633 in Lecture Notes in Computer Science, pages 495–499, Trento, Italy, July 1999. Springer.
- [18] E. Clarke, O. Grumberg, and D. Long. Model checking. In *Proceedings of the NATO Advanced Study Institute on Deductive program design*, pages 305–349, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.
- [19] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

- [20] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
- [21] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.
- [22] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 343–354, New York, NY, USA, 1992. ACM Press.
- [23] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.
- [24] Edmund M. Clarke, Orna Grumberg, and David E. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium*, pages 124–175, London, UK, 1994. Springer-Verlag.
- [25] E. A. Emerson and J. Y. Halpern. “sometimes” and “not never” revisited: on branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, 1986.
- [26] E. Allen Emerson and Joseph Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *STOC '82: Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 169–180, New York, NY, USA, 1982. ACM Press.
- [27] E. Allen Emerson and Chin-Laung Lei. Modalities for model checking (extended abstract): branching time strikes back. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96, New York, NY, USA, 1985. ACM Press.
- [28] J.L. Lyons et al. Ariane 5: Flight 501 failure. Board of inquiry report, European Space Agency, July 1996.
- [29] R. Gasser. Harnessing computational resources for efficient exhaustive search, 1994.
- [30] David Harel. *Algorithmics: the spirit of computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

- [31] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.
- [32] Joe Hurd. Formal verification of chess endgame databases. In *Theorem proving in higher order logics: Emerging trends proceedings*, pages 85–100. Oxford University Computing Laboratory, August 2005. Technical Report PRG-RR-05-02.
- [33] L. Lamport. “sometime” is sometimes “not never”: On the temporal logic of programs. In *POPL ’80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–185, New York, NY, USA, 1980. ACM Press.
- [34] G. Le Lann. An analysis of the Ariane 5 flight 501 failure - A system engineering perspective. *Proceedings of the International Conference and Workshop on Engineering of Computer-Based Systems, 1997.*, pages 339–346, 1997.
- [35] Nancy Leveson. Medical devices: The therac-25. In *Safeware: System Safety and Computers*. Addison-Wesley, 1995.
- [36] P. Madhusudan, Wonhong Nam, and Rajeev Alur. Symbolic computational techniques for solving games. *Electr. Notes Theor. Comput. Sci.*, 89(4), 2003.
- [37] K. L. McMillan. *Symbolic Model Checking: An approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.
- [38] K. L. McMillan. *Getting started with SMV*. Cadence Berkeley Labs, 2001 Addison St. Berkeley, CA 94704, USA, March 1999.
- [39] K. L. McMillan. *The SMV Language*. Cadence Berkeley Labs, 2001 Addison St. Berkeley, CA 94704, USA, March 1999.
- [40] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.
- [41] Jonathan Schaeffer, Neil Burch, Yngvi Bjornsson, Akihiro Kishimoto, Martin Muller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, pages 1144079+, July 2007.

-
- [42] Patriot missile defense: Software problem led to system failure at Dhahran, Saudi Arabia. Report GAO/IMTEC-92-26, Information Management and Technology Division, United States General Accounting Office, Washington, D.C., February 1992.
- [43] Zheng Zhang. Playing tic-tac-toe game using model checking. Technical report, University of Illinois at Chicago, 2004.