EFFICIENT CTL* MODEL CHECKING USING GAMES AND AUTOMATA

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Science and Engineering

June 1998

By
Willem C. Visser
Department of Computer Science

(DWW3Q)

Th 21020 ✓

Contents

List of Tables

List of Figures

Abstract

Formal verification, where a system is verified with respect to a desired behaviour, has now become popular in industry, especially in mission and safety critical applications. Specifically model checking methods, which can be fully automated, are being used extensively to verify that a finite state system meets a desired behaviour. The desired behaviour is often specified by a temporal logic formula. In this thesis we are interested in efficient algorithms for CTL* model checking, where the system to be verified is specified as a Kripke structure and the formula to be checked is given in the branching time temporal logic CTL*.

Efficient linear time model checking algorithms were developed by adopting the approach of translating the temporal formulas to nondeterministic automata on infinite words. At first, the theoretical link between linear time model checking and automata theory was shown, and only later did this lead to the discovery of efficient model checking algorithms. Recently it has been shown that automata-theoretic model checking for branching time temporal logic is possible by translating the temporal formulas to alternating automata. The aim of the work presented here is to show that, in a similar way to the linear time case, this link with automata theory can lead to the development of efficient model checking algorithms for branching time temporal logic. In the automata-theoretic approach to CTL* model checking the model checking problem reduces to checking the nonemptiness of an alternating tree automaton

(more precisely that the language that the automaton recognises is nonempty). We show that this nonemptiness check can be reformulated as a 2-player game, which we refer to as the *nonemptiness game*. We develop a novel way, by playing so-called *new games*, of ensuring that results obtained during the nonemptiness game can be safely reused in later stages to make the algorithm both space and time efficient.

Model checkers for the sublogic CTL of CTL* are very popular in industry since for some types of Kripke structure very efficient model checking can be done for this logic. Although CTL is not as expressive as CTL* it is often the case that a syntactically more succinct CTL* formula can be expressed as CTL. An interesting open problem is therefore to determine whether a given CTL* formula is equivalent to a CTL formula. Here we show that the structure of the alternating automaton translated from a CTL* formula can be used, not only to determine whether an equivalent CTL formula exists, but can be used to find an equivalent formula (if it exists). We show further that the structure of alternating automata can also lead to interesting results for another sublogic of CTL* for which model checking is popular, namely LTL. Specifically it is shown that CTL* formulas that are both expressible in CTL and LTL can be model checked very efficiently in our nonemptiness game setting.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

The Author

The author received a *B.Sc.* degree (Computer Science and Applied Mathematics) in 1989, a *B.Sc. Hons* degree (Computer Science) in 1991, and an *M.Sc.* (Computer Science) in 1993 from the University of Stellenbosch (South Africa).

His interest in formal methods and specifically model checking started during 1990. His M.Sc. work was concerned with the validation, via model checking, of a locally developed operating system kernel. The thesis was entitled "A run-time environment for a validation language".

After completing his studies at the University of Stellenbosch he spent 9 months working for DataFusion Systems (SA). His primary task was to help develop a prototype directory enquiry system for the South African telephone company (Telkom).

He started his Ph.D. work in January 1995 under the supervision of Professor Howard Barringer in the Department of Computer Science, at the University of Manchester.

Acknowledgements

Money is always important no matter what people say! So I'll start off by thanking those institutions that made it possible for me to be able to write this section in the first place: ORS, the Harry Crossley Foundation and the South African Foundation for Research Development. A special thanks goes to Rose of the FRD for always answering my questions and reminding me when United lost a game, which, thankfully, wasn't very often during the past three years.

Howard, my supervisor, must be commended for always finding time for me in a schedule that can only be described as *extremely* full. Once we got going he would sit there for hours explaining to me the finer points of temporal logic and many other subjects of which I knew very little. His "library" of papers and the (near) perfect filing system astounded me. Most importantly he always found a way to fund my trips to conferences, for which I'll be forever grateful. Then, of course, there were the fly-fishing trips to the Lakes and anywhere else Howard could think of that might be home to a few trout.

I shared an office with Alan, to whom I am indebted so much that it is impossible to put down in a few sentences: from standing in line for hours to get us tickets to watch United, to listening to my stupid ideas and always patiently putting me on the right track. Not to mention all the pints he bought me over the past three years. Or whenever I had a more technical problem I could always turn to the third member of the office, Donal, who would always

know the best way to fix just about anything to do with Unix or C. And what a fountain of knowledge Graham is; I wish he could read every paper I ever write because if there is an error in it, he'll find it. When I needed a different view on things or just a quick pint there was always the "people down the road": Clare, Michael and Alexander.

I cannot possibly forget to acknowledge the massive influence of my great friend Pieter on my work and life in general. He introduced me to model checking, whisky and great food. Not to mention his encouragement for me to do a PhD outside of South Africa. The number of emails between Manchester and Stellenbosch often numbered 10 a day.

Away from work I could not have asked for a more interesting bunch of people to mingle with. From the Flat 20 brigade (Tim, Warren, Ben, Lee, Tony, Peter and Jamie) to my current flat mates in Greville street (Nicola and Joe) and the many others I've met along the way. Then, of course, there is Amanda without whom life would have been much less interesting.

The last word must go to my parents who have supported me through thick and thin albeit from 5000 miles away. How glad I am that we installed that email system before I left. It was a joy to hear from you nearly every day for three years.

Chapter 1

Introduction

How many people raise an eyebrow at the following names: A320/330/340 and B777? Maybe we all should, since these are the series numbers of a new generation of aircraft developed by Airbus and Boeing respectively, which are so-called "fly-by-wire" — their primary flight control is achieved through computers. Unfortunately, some of these aircraft have already been involved in accidents [Lad]. Even more disturbingly, some of these accidents are reported to be computer related:

A320, Warsaw, 14/9/93: Pilots were unable to activate any of the braking systems until 9 seconds after landing and consequently ran off the end of the runway.

A330, Toulouse, 30/6/94: The aircraft rolled sharply during a test-flight and the pilots could not regain control in time to avoid hitting the ground.

Probably the most publicised incident of software failure in the past five years was the explosion of the Ariane 5 rocket [Ari]. The explosion was caused by a complete loss of guidance information 30 seconds after take off, which caused the rocket to veer off its flight path. Here is a quote from the official report on the incident:

> This loss of information was due to specification and design errors in the software of the inertial reference system. The extensive reviews and tests carried out during the Ariane 5 development programme did not include adequate analysis and testing of the inertial reference system or of the complete flight control system, which could have detected the potential failure.

Testing

Testing has for a long time been the most common way of determining whether a system is robust enough for release in both the software and hardware industries. Although testing can find many errors in a system, it cannot guarantee to find *all* the errors [BF91, FW91]. This is rather graphically illustrated by the quote above. In today's rapid development of complex computerised systems a more reliable method than testing for detecting errors is required. *Formal Verification* is such a method.

Formal Verification

During formal verification a system is verified to meet a desired behaviour by checking whether a mathematical model of the system satisfies a formal specification that describes the behaviour. The desired behaviour is specified in a *formal specification language*, with a precise semantics to eliminate any ambiguity in what it means for a behaviour to be correct. Furthermore, formal verification is *exhaustive*: a system is correct with regards to its specification when *all* its behaviours satisfy the specification. Therefore, since there is a precise meaning of correctness and all behaviours of a system are checked, formal verification is clearly superior to testing. Another advantage of formal verification is that it can be used during the design phase, when errors are still comparatively harmless and easy to correct.

Unfortunately, formal verification is not the solution for achieving *complete* correctness of a system, since the system is only verified to be correct with regards to the behaviours specified in the formal specification[1]. The problem of determining which formal specifications to verify for a system is beyond the scope of this thesis.

Automated Formal Verification

It is worthwhile to observe why testing is so popular. It is largely an automatic procedure; most work goes into determining which tests should be run, but the actual tests themselves can be executed automatically and the results observed at a later stage. In order for a formal verification technique to be popular in industry the automatic nature of testing must be preserved. This is why *proof based* formal verification techniques have not been widely taken up in industry. In proof based techniques the model of the system is augmented with assertions in a formal specification language and proofs are constructed to relate these assertions. Although theorem provers and proof checkers have been developed to reduce the effort involved during the construction of these proofs, there is still considerable effort and expert knowledge required to use these tools [SOR93, BBC+96]. *Model checking* is a formal verification technique based on the exploration of the states of a model and can in many cases be fully automated.

Model Checking

Since a model checker explores the reachable state space of a model it can only handle *finite-state* systems[2]. Many interesting systems are essentially finite-state systems: hardware systems, communication protocols, operating system

[1] A similar situation occurs with testing: only when the *appropriate* tests are being run can one draw any conclusions about the system's correctness.

[2] Infinite state systems can be handled by a combination of proof based and state-exploration based formal verification.

kernels, flight controllers etc. Besides that it can be fully automated when checking finite-state systems, another advantage of model checking is that it can report how an error can be generated within a system. Unfortunately, model checking also has a drawback: its performance is dependent on the size of the model. For example, when checking a system consisting of concurrent components the number of states to explore can grow exponentially in the number of components. This is often referred to as the *state-explosion* problem. Alleviating the state explosion problem is a challenging area of research in model checking.

Temporal Logic

Temporal Logic is a popular formal specification language for use with a model checker. A temporal logic is a logic augmented with temporal modalities to allow the specification of event orders in time, without having to introduce time explicitly. For example, a temporal logic with the modalities *always* and *eventually* will be able to specify the following property: "for all future moments in which p holds there will be a future moment in which q holds". Whereas traditional logics can specify properties relating to the initial and final states of terminating systems a temporal logic is better suited to describe on-going behaviour of non-terminating and interacting (reactive) systems.

Linear and Branching Temporal Logic

There are two main kinds of temporal logics: *linear* and *branching* [Lam80]. In linear temporal logics, each moment in time has a unique possible future, while in branching temporal logics, each moment in time may have several possible futures. Linear temporal logic formulas are therefore interpreted over linear sequences and are regarded as specifying the behaviour of a single computation of a system. Branching temporal logics, on the other hand, are interpreted

over structures that can be viewed as trees, each describing the behaviour of the possible computations of a nondeterministic system. We will refer to *linear time model checking* when the property to be checked during model checking is specified as a linear temporal formula and *branching time model checking* when the property is specified as a branching temporal formula.

Model Checking in Industry

Since the first algorithms appeared in the early 1980's temporal logic model checking has become very popular not only in academia but also in industry. In the literature, numerous examples exist for the application of model checking in the formal verification of commercially used software and hardware systems [BCDM86, CGH+95, tEM95, HWT95, Low96, Kar96, RL97, SECH98]. In fact, companies are not only recruiting model checking experts (INTEL, NASA, Lucent, Chrysalis, etc.), but in some cases are even selling model checkers (FormalCheck from Lucent [For98] and Design INSIGHT from Chrysalis [Ins98]). Most of this commercial interest has been centred around two types of model checkers: linear time model checkers for the propositional linear temporal logic (LTL) and branching time model checkers for computation tree logic (CTL). It is interesting to observe the reasons for the popularity of these two types of model checkers.

Industrial Model Checking: CTL

CTL model checkers were the first model checkers to be introduced [CE81, QS82] and as such had a head start, even more so when the original algorithms were improved to show that model checking can be done in time linear in the size of the state graph (reachability graph of states) for the system and the length of the formula. Unfortunately it also had the drawback that the complete state graph had to be kept in memory throughout the model checking procedure

and therefore due to the state-explosion problem it had limited applicability. A major breakthrough for CTL model checking came with the realisation by McMillan [McM92a] (after initial work by Bryant [Bry86]) that it is unnecessary to store the state graph explicitly during model checking. He encoded the state graph as an ordered binary decision diagram (OBDD) that in many cases would require less memory than when the graph was stored explicitly. Model checking is then performed on these OBDD encodings by applying boolean functions to them according to the fixpoint characteristics of the CTL formulas to be checked. Unfortunately, for the OBDD encodings to be efficient, the state space must exhibit some form of regularity. The good news is that for the state graphs of synchronous hardware systems this is often the case, and as such OBDD based CTL model checkers found their niche market so to speak.

Industrial Model Checking: LTL

When Pnueli first showed that temporal logic is suitable for specifying properties of reactive system, the logic that he used was a linear time temporal logic [Pnu77]. Ever since then linear time has been the logic of choice for writing temporal specifications for systems [MP92]. Although many different types of linear time temporal logics were used for specifying reactive systems, most of the model checking work centred around the use of LTL. LTL model checkers were however less popular than CTL model checkers, since their model checking complexity was exponential in the length of the formula (whereas it is linear for CTL). The breakthrough for LTL model checkers came when it was realised that the complete state graph need not be kept in memory throughout model checking. In fact, these model checkers were often referred to as "on-the-fly" model checkers [VW86b, JJ89, CVWY92] since the reachable states were generated during the model checking procedure. Furthermore, only the parts of the state graph required to (in)validate the formula would be analysed (this is often referred to as *local model checking*). Interestingly, the reason for the

discovery of efficient model checking algorithms for LTL, was due to the close relationship between linear temporal logics and the theory of automata on infinite words [VW86b, VW94]. The basic idea is to associate with each formula an automaton on infinite words that accepts exactly all the computations that meet the behaviour specified by the formula. This enables the model checking problem for LTL to be reduced to the automata-theoretic problem of testing nonemptiness of an automaton.

Model Checking: CTL versus LTL

Although the OBDD based model checkers can in some cases handle larger systems than the explicit state exploration approach of the LTL model checkers, they do suffer from another disadvantage: model checking may require expert knowledge and is in some cases an interactive procedure. This is due to the fact that the efficient representation of an OBDD is sensitive to the ordering of its boolean variables [Bry86, Bry92]. In many cases to obtain an efficient variable ordering[3], i.e. one for which the model checker will not run out of memory, expert knowledge about the system to be checked and also about the way the OBDD encodings work is required [Dil96].

Since practical model checking algorithms for LTL are based on automata theory and those for CTL on fixpoint characterisations of the formulas combined with OBDD encodings it is hard to compare them adequately. Clearly a simple comparison on their respective (theoretical) model checking complexities does not give a satisfactory measure for comparing their applicability for industrial-scale model checking. Furthermore, of course, CTL and LTL cannot be compared in expressiveness either [BG94]. What is required is a model checking algorithm that would be able to check both CTL and LTL properties in the same setting and be practical (efficient) enough for undertaking model

[3]Note that in some cases no efficient variable ordering exists.

checking of industrial-strength systems.

1.1 Thesis Goal

Constructing such an efficient model checking algorithm is the goal of this thesis. In order to achieve this we define (and implement) a model checker for the branching time temporal logic CTL* that contains as sublogics both CTL and LTL. This will enable the model checking of, not only CTL and LTL properties, but also properties that cannot be expressed by either.

Although LTL model checking can be done with an automata-theoretic approach, it has only comparatively recently been shown that by using *alternating tree automata* this is also possible for CTL*[Ber95]. Essentially the CTL* formula to be checked is translated to a special type of alternating automaton, *hesitant alternating automaton* (HAA), and the model checking problem then reduces to checking the nonemptiness of this type of automaton. In [Ber95] two different algorithms are presented for doing the nonemptiness checking: one is time efficient and the other is space efficient. Although this was a major first step, what is required is an algorithm that is *both* time and space efficient.

We extend the work of [Ber95] by showing that recasting the nonemptiness problem of alternating tree automata as a special type of a two-player game, called the *nonemptiness game*, allows CTL* model checking that is both time and space efficient. We show that the rules for the nonemptiness games are different when checking CTL formulas (CTL nonemptiness games) than when LTL formulas (LTL nonemptiness games) are checked. In fact it turns out that CTL nonemptiness games are in general of higher complexity than LTL nonemptiness games. This is different from the traditional model checking complexities, where LTL model checking has higher complexity.

We show that the differences in the rules of the CTL and LTL nonemptiness

games manifest themselves in the structure of the HAA[4] translated from the formula to be checked and can be used as a means to determine whether a CTL* formula has an equivalent CTL and/or LTL formula. Determining whether a CTL* formula has a CTL equivalent has been an open problem since 1988 [CD88]. We not only show that with the aid of HAA this problem can be solved, we also show how to find the equivalent CTL formula if one exists. This could enable the use of OBDD based CTL model checkers whenever an equivalent CTL formula exists for the CTL* formula to be checked.

Since an aim is to construct a *practical* model checking algorithm close attention to the implementation details will be given throughout. In fact, this is a secondary goal of the thesis: to give the reader enough detail about implementation issues to allow the algorithm described here to be implemented by him/her.

1.2 Thesis Layout

The thesis consists of the following three parts:

Background: Chapter 2 introduces CTL* and Chapter 4 (except section 4.7) introduces the automata-theoretic approach to model checking.

Model Checking Review: Chapter 3.

Practical Model Checking: Section 4.7 introduces the logic LinearCTL*. Chapter 5 gives the basic algorithm, Chapter 6 shows the different versions of the algorithm for CTL and LTL as well as showing how CTL* formulas can be classified as either CTL and/or LTL or strictly CTL*. Chapter 7 shows how the algorithm can be used for practical model checking. Section 4.7 was first published in [VBF+97], section 7.2 first appeared in

[4]Throughout the thesis, in an abuse of notation, the abbreviation HAA stands for either hesitant alternating automata or hesitant alternating automaton, depending on the context.

[VB96] and section 7.3 in [VBF+97].

Here follows a brief synopsis of each chapter:

Chapter 2 contains the syntax and semantics of CTL*. The use of CTL* as a specification language for properties of reactive systems is also illustrated by some examples. The linear and branching sublogics of CTL* are briefly compared with regards to their expressive power and how they are used for model checking.

Chapter 3 presents an overview of model checking for CTL, LTL and CTL*. It starts by giving characteristics that can be used to classify different types of model checkers and continues by describing different model checking approaches according to this characterisation. This is by no means an exhaustive overview of model checkers, but the model checking approaches selected for the review have some form of significance in model checking history: either being the first of their kind, or illustrations of different approaches.

Chapter 4 focuses on automata-theoretic model checking. It begins with an overview of the use of first nondeterministic automata and then alternating automata as an automata-theoretic counterpart for CTL*. The remainder of the chapter is devoted to translating CTL, LTL and CTL* formulas to HAA in a similar fashion to that proposed in [Ber95]. It is also shown how the structure of the translation rules for translating LTL to alternating word automata can be exploited to define a sublogic of CTL* called *Linear*CTL* that allows a linear translation from the formulas of the logic to HAA. Note, in general, this translation from CTL* to HAA is exponential. Lastly, some implementation issues for translating formulas to automata are discussed.

Chapter 5 introduces the nonemptiness game for HAA. The rules of the game are first given and are followed by a first attempt at implementing the game. It is shown that in order to make it time efficient information must be stored

during the game to be reused in later stages. This requires the novel approach of playing new games to ensure the information stored is correct. Pseudo code for the algorithm is given and explained in detail. The chapter concludes by comparing the algorithm to related work.

Chapter 6 contains the main contribution of the thesis. Here it is shown how the structure of the HAA can be exploited to optimise the nonemptiness game when CTL or LTL formulas are being model checked. Combining the rules for CTL and those for LTL formulas and adding a trivial proviso turns out to be sufficient to also play efficient nonemptiness games for full CTL*. Lastly, it is shown how to classify CTL* formulas, as either CTL, LTL or neither, and how to utilise this information for efficient model checking.

Chapter 7 takes a look at the more practical issues involved in model checking. A structure for a model checking system is discussed in which the model checking algorithm introduced here can be fitted. This structure allows the model checking of different input formalisms with little change to the existing system. Two novel approaches to alleviate the effect of the state-explosion problem during model checking are introduced. Lastly, it is shown how the model checker described in this thesis is used to check properties of asynchronous hardware systems.

Chapter 8 contains a retrospective view of the goals and achievements of the work presented as well as some future extensions being planned.

Chapter 2

CTL* : Syntax, Semantics and Usage

Pnueli, in his seminal 1977 paper [Pnu77], introduced the use of temporal logic to the specification of program behaviour. Since then much research has gone into developing different temporal logics that are suitable for different applications (an excellent survey can be found in [Eme96]). Here we will focus on the branching time logic CTL* [EH86].

2.1 Syntax and Semantics of CTL*

CTL* can express both linear and branching time properties, and is therefore more expressive than the linear time logic LTL [LP85] and the branching time logic CTL [CE81, CES86]. In fact, both these logics are sublogics of CTL*. For technical convenience only *positive* CTL* formulas will be used here, i.e. formulas with negations only applied to atomic propositions. Any CTL* formula can be transformed into a positive form by pushing negations inward as far as possible by using De Morgan's laws and dualities. There are two types of formula in CTL*: formulas whose satisfaction is related to states, *state* formulas,

and those whose satisfaction is related to paths, *path* formulas. Let $q \in Props$, where $Props$ is a set of atomic propositions. The syntax of CTL* state (S) and path formulas (P) is then given by the following two BNF rules:

$$\begin{aligned} \mathbf{S} &::= true \mid false \mid q \mid \neg q \mid \mathbf{S} \wedge \mathbf{S} \mid \mathbf{S} \vee \mathbf{S} \mid A\mathbf{P} \mid E\mathbf{P} \\ \mathbf{P} &::= \mathbf{S} \mid \mathbf{P} \wedge \mathbf{P} \mid \mathbf{P} \vee \mathbf{P} \mid X\mathbf{P} \mid \mathbf{P}\ U\ \mathbf{P} \mid \mathbf{P}\ V\ \mathbf{P} \end{aligned}$$

A ("for all") and E ("there exists") are referred to as path quantifiers and X ("next"), U ("Until") and V ("release") as path operators. The sublogics CTL and LTL are now defined as:

CTL Every occurrence of a path operator is immediately preceded by a path quantifier.

LTL Formulas of the form $A\mathbf{P}$ where the only state subformulas of $\mathbf{P}$ are propositions.

The semantics of CTL* is defined with respect to a *Kripke* structure $K = (Props, S, R, s_0, L)$, where $Props$ is a set of atomic propositions, S is a set of states, $R \subseteq S \times S$ is a transition relation that must be total (for every $s_i \in S$ there exists a s_j such that $(s_i, s_j) \in R$), $s_0 \in S$ is an initial state and $L : S \to 2^{Props}$ maps each state to the set of atomic propositions true in that state. For $(s_i, s_j) \in R$, s_j is the *successor* of s_i and s_i the *predecessor* of s_j. The branching degree, i.e. the number of successors, of a state s is denoted by $d(s)$. A path in K is an infinite sequence of states, $\pi = s_0, s_1, s_2, \ldots$ such that $(s_i, s_{i+1}) \in R$ for $i \geq 0$. The suffix $s_i, s_{i+1}, \ldots$ of π is denoted by π^i. $K, s \models \varphi$ indicates that the state formula φ holds at state s and $K, \pi \models \psi$ indicates the path formula ψ holds at the path π of the Kripke structure K. $s \models \varphi$ and $\pi \models \psi$ are written when K is clear from the context. The relation $\models$ is inductively defined as:

- $\forall s \in S$, $s \models true$ and $s \not\models false$

- $s \models p$ for $p \in Props$ iff $p \in L(s)$
- $s \models \neg p$ for $p \in Props$ iff $p \notin L(s)$
- $s \models \varphi_1 \wedge \varphi_2$ iff $s \models \varphi_1$ and $s \models \varphi_2$
- $s \models \varphi_1 \vee \varphi_2$ iff $s \models \varphi_1$ or $s \models \varphi_2$
- $s \models A\psi$ iff for every path $\pi = s_0, s_1, \ldots$, with $s_0 = s$, then $\pi \models \psi$
- $s \models E\psi$ iff there exists a path $\pi = s_0, s_1, \ldots$, with $s_0 = s$, and $\pi \models \psi$
- $\pi \models \varphi$ for a state formula φ, iff $s_0 \models \varphi$ where $\pi = s_0, s_1, \ldots$
- $\pi \models \psi_1 \wedge \psi_2$ iff $\pi \models \psi_1$ and $\pi \models \psi_2$
- $\pi \models \psi_1 \vee \psi_2$ iff $\pi \models \psi_1$ or $\pi \models \psi_2$
- $\pi \models X\psi$ iff $\pi^1 \models \psi$
- $\pi \models \psi_1\ U\ \psi_2$ iff $\exists i \geq 0$ such that $\pi^i \models \psi_2$ and $\forall j, 0 \leq j < i$, $\pi^j \models \psi_1$
- $\pi \models \psi_1\ V\ \psi_2$ iff $\forall i \geq 0$ such that if $\pi^i \not\models \psi_2$ then $\exists j, 0 \leq j < i$, $\pi^j \models \psi_1$

A state s satisfies $A\psi$ ($E\psi$) if every path (some path) π from the state s satisfies ψ, while a path satisfies a state formula if the initial state of the path does. $X\psi$ holds of a path when ψ is satisfied in the next state on the path, whereas $\psi_1\ U\ \psi_2$ holds of a path if ψ_1 holds on the path until ψ_2 becomes true. V is the dual of U since $\neg(\psi_1\ U\ \psi_2) = \neg\psi_1\ V\ \neg\psi_2$ and is referred to as the "release" operator: $\psi_1\ V\ \psi_2$ holds for a path, if ψ_2 remains true until ψ_1 "releases" the path from its obligation.

The following well known abbreviations will also be used:

$F\varphi = true\ U\ \varphi$ — φ holds in a future (hence the "F") state on a path. Also referred to as the "eventually" operator.

$G\varphi = false\ V\ \varphi$ — φ holds globally (hence the "G") on a path. Also referred to as the "always" operator.

The $\psi_1\ U\ \psi_2$ operator is a *strong* until, since ψ_2 must eventually become true. A *weak* until operator, W, can be defined in terms of the strong until: $\psi_1\ W\ \psi_2 = G\psi_1 \vee \psi_1 U \psi_2$. Note for the weak until to be satisfied ψ_2 need not become true. In a similar fashion the V operator is a weak release operator, whereas the dual of W, call it R, defined as $\psi_1\ R\ \psi_2 = \neg(\neg\psi_1\ W\ \neg\psi_2)$, is a strong release operator. These operators, W and R, will only be used in the rest of the thesis to show translations between equivalent CTL and LTL formulas. Two formulas ψ_1 and ψ_2 are equivalent if for all Kripke structures K we have that $K \models \psi_1$ *iff* $K \models \psi_2$.

The closure of a CTL* formula ψ, $cl(\psi)$, is defined as all the state subformulas of ψ including ψ but excluding true and false. For example, $cl(A(EXp\ V\ AXq)) = \{A(EXp\ V\ AXq), EXp, p, AXq, q\}$.

2.2 CTL* for Property Specification

Temporal logic is mostly used in the specification and verification of reactive systems [HP85, Pnu86]. An important property of reactive systems is that they interact continuously with their environments and do not compute a final value on termination — in fact, they are usually designed not to terminate at all. A reactive system is also fundamentally concurrent, firstly, since it executes concurrently with its environment, but also the system itself generally consists of concurrent processes. The temporal specifications therefore tend to specify properties of the interaction between processes in the reactive system.

Since reactive systems are so diverse, ranging from operating systems, process control systems, communication protocols through to microprocessors, the nature of the temporal specifications have also been numerous. Originally two

types of specification were distinguished [Lam77]:

Safety Properties: state that "something bad never happens" (a program never enters an unacceptable state)

Liveness Properties: state that "something good will eventually happen" (a program eventually enters a desirable state)

Safety properties are expressed by $AG\neg p$, where p characterises a "bad" state. Mutual exclusion, $AG(\neg crit1 \vee \neg crit2)$, is a well known safety property. Liveness properties are more difficult to classify syntactically, but in general they are formulas with the F operator present. For example[1], $AG(sent \rightarrow Freceived)$, is a liveness property of a protocol that specifies whenever a message is sent by the sender it will always eventually be received by the receiver. Manna and Pnueli extended the liveness classification to give a more appropriate classification of LTL formulas [MP92]. This classification is given below with an example of a formula in each class:

Guarantee: Specifies that an event will eventually happen, but does not promise repetitions. [AFp: At least one state on a path satisfies p]

Obligation: This is the class of properties that cannot be expressed by safety and guarantee formulas alone and is therefore a disjunction of the two classes. [$A(Gp \vee Fq)$: Either p holds always on a path or q holds at some state.]

Response: Specifies that an event will happen infinitely many times. The fact that for every stimulus there is a response in a system can be specified by this class. [$AGFq$: Infinitely many states on a path satisfy p.]

Persistence: Specifies the eventual stabilisation of some system condition after

[1] $\psi_1 \rightarrow \psi_2$ is defined in the usual way as $\neg\psi_1 \vee \psi_2$. Note, if $\psi_1 \notin Props$ then the negation must be pushed inside.

an arbitrary delay. [$AFGp$: For all states on a path from a certain point on p holds.]

Reactivity: This is the maximal class and is formed from the disjunction of response and persistence properties. [$A(GFp \vee FGq)$: Either the path contains infinitely many states for which p holds or from a certain point on q holds continuously.]

Another classification that is important when specifying properties is that of fairness [Fra86]. Below follow the main fairness classes with example specifications:

Unconditional Fairness - $AGFq$**:** q holds infinitely often

Weak Fairness - $A(FGp \rightarrow GFq)$**:** if p is continuously true then q must be true infinitely often. Note, in positive form $A(FGp \rightarrow GFq)$ is written as $A(GF\neg p \vee GFq)$.

Strong Fairness - $A(GFp \rightarrow GFq)$**:** if p is true infinitely often then q must be true infinitely often. Note, in positive form $A(GFp \rightarrow GFq)$ is written as $A(FG\neg p \vee GFq)$.

The fairness classes are subsumed within Manna and Pnueli's classification, since unconditional and weak fairness are both response properties and strong fairness is a reactivity property.

2.2.1 Example

Consider the mutual exclusion problem for two processes where each process ($i = 1, 2$) can be in one of three code regions: noncritical (N_i), trying (T_i) or critical (C_i). A binary semaphore S is used to protect the critical region. The value of the semaphore is indicated by S_i in Figure 2.1, where i can be 0 or 1.

A process can only enter its *critical* region from its *trying* region if the value of the semaphore is 0. When a process enters its critical region the value of the semaphore becomes 1 and on leaving the critical region and entering the *noncritical* region the value of the semaphore becomes 0 again.

Each state is labelled with the values of all variables in that state. The set of states W is therefore given by $(P_1, P_2, Sema)$, where P_i can be T_i, N_i or C_i, with $i = 1, 2$ and $Sema$ is either S_0 or S_1. The Kripke structure for this system is therefore given by $K = (2^{Props}, W, R, w_0, L)$, where $Props = \{N_1, N_2, T_1, T_2, C_1, C_2, S_0, S_1\}$, the transition relation and the labelling is given in Figure 2.1 and the initial state, w_0, is $(N_1 N_2 S_0)$.

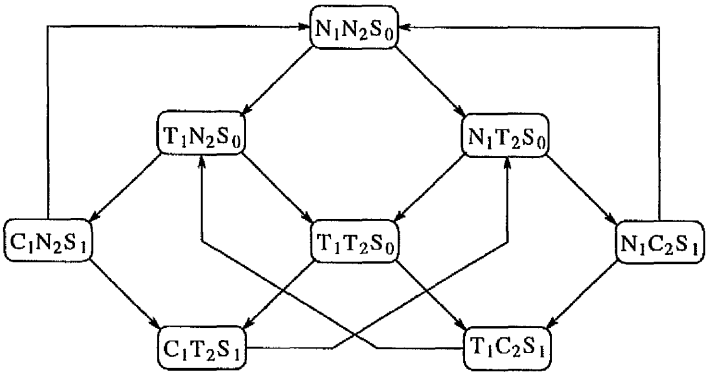

Figure 2.1: Transition relation for the mutual exclusion system

Here are three examples of specifications with their truth-values for the mutual exclusion system modelled by K.

Safety Property (Mutual Exclusion): The two processes will never be in their critical regions at the same time. $\psi = AG(\neg C_1 \vee \neg C_2)$, with $K, w_0 \models \psi$.

Guarantee Property (Safe Liveness): Neither of the two processes can enter their critical regions until one of them receives the semaphore (semaphore state become S_1). $\psi = A((\neg C_1 \wedge \neg C_2)\ U\ S_1)$, with $K, w_0 \models \psi$.

Response Property (Absence of Starvation): Once a process has entered its

trying region it will eventually enter its critical region. $\psi = AG(T_1 \rightarrow F(C_1))$. Due to the possible unfair behaviour in which process 2 can be continuously given the semaphore, seen in Figure 2.1 by the cycle (T_1, N_2, S_0), (T_1, T_2, S_0), (T_1, C_2, S_1), (T_1, T_2, S_0), $K, w_0 \not\models \psi$. A fairness constraint can be added to exclude this behaviour resulting in $K, w_0 \models AG(T_1 \rightarrow (F(C_1) \vee GF(C_2)))$. Essentially the constraint $GF(C_2)$ specifies that the cycle given above should be considered acceptable behaviour.

2.3 Expressiveness: Linear vs. Branching

In the previous section classifications of temporal formulas, used to express properties of reactive system, were given for the linear fragment of CTL*. Do they still hold for the strictly branching time logic CTL? No. When considering the syntactic definitions of the classes as given in [MP92] it is clear that safety properties have equivalent CTL formulas. Guarantee properties however don't; for example, the formula $AF(p \wedge Xp)$ is not expressible in CTL [EH86]. Furthermore, since in the syntactic definition all the other classes (obligation, response, persistence and reactivity) build on guarantee properties, none of the classes can be fully expressed in CTL. This, however, does not mean that a meaningful subset of each class doesn't have a set of equivalent CTL formulas. For example AFp (guarantee), $A(Gp \vee Fq)$ (obligation), $AGFp$ (response) all have equivalent CTL formulas respectively AFp, $A((p \wedge \neg q)\ W\ AFq)$ and $AGAFp$. If we consider the fairness classification, then both unconditional $[AGAFq]$ and weak fairness $[AG(A(AF(\neg p \vee q)\ W\ \neg p) \wedge A(AF(\neg p \vee q)\ W\ q))]$ can be expressed in CTL, but strong fairness cannot.

It is of course easy to see which CTL formulas cannot be expressed by an equivalent LTL formula: all formulas that include an existential path quantifier (E). One such formula is AGEFp that states from all states in the system it is possible to reach a state where p holds. This type of formula can therefore

be used to show absence of deadlock or livelock in a system. For example, the mutual exclusion system from Figure 2.1 contains no deadlocks or livelocks since the initial state can be reached from all states of the system, i.e. $K, w_0 \models AGEF(N_1 \wedge N_2 \wedge S_0)$. There are also CTL formulas that only contain universal (A) quantifiers, but are still not expressible in LTL, for example AFAGp [CD88].

Lastly, when considering conjunctions or disjunctions of some of the above formulas we can construct a formula that is a CTL* formula, but neither CTL nor LTL, for example $AFGp \wedge AFAGq$. CTL* is therefore strictly more expressive than both CTL and LTL, but CTL and LTL cannot be compared in expressiveness.

2.4 CTL* for Model Checking

When writing temporal specifications that are to be used by a model checker, many factors must be considered, first and foremost the capabilities of the temporal logic at your disposal. For example, if you can express a property to be checked in both LTL and CTL, LTL is to be preferred, since LTL formulas are more succinct and readable than CTL, one reason for this is that it does not require unnecessary path quantifiers before all the temporal operators. However, an argument against the use of LTL is that its model checking algorithm is exponential in the size of the formula [LP85]. Unfortunately, writing an equivalent formula in CTL (for the LTL formula to be checked) might not be the solution. For example, the LTL formula $A(Gp_0 \vee Gp_1 \vee \ldots \vee Gp_n)$ has an equivalent CTL formula that is exponential in n. Furthermore, if the original LTL formula cannot be expressed in CTL, then writing it in LTL is the only option. It must be noted here that CTL model checkers, specifically those based on OBDDs [BCM+90, McM92a], can in certain cases outperform LTL model checkers by orders of magnitude (on the same Kripke structure and equivalent formula).

Most model checkers available today are either for CTL or LTL, and hence the option of which logic to use is out of the practitioners' hands, so to speak. It is our experience of using a CTL* model checker for the validation of asynchronous hardware designs [VBF+97] that linear time formulas are written more often, in as much as branching time formulas are used only when no equivalent linear time formula exists (e.g. AGEFp). Classifying the properties that we write according to the classification given in section 3.1 it follows that safety $[AGp]$ and response $[AG(p \rightarrow Fq)]$ properties are written frequently. Fairness properties, in all three categories, are also used regularly for excluding unfair behaviour of a circuit whilst model checking response properties (in case of it being strong fairness the property then becomes a reactivity property).

Lichtenstein and Pnueli observed that in most cases the size of the formula is much smaller than the size of the Kripke structure [LP85]. We believe it is counter productive to express too complex a property, since if the property does not hold for a design it is difficult to see what is wrong in the design, or even worse, the design may be correct but the property does not capture the property that the user was intending.

2.5 Concluding Remarks

Although LTL is more popular for expressing properties of reactive systems [Pnu77, MP92], the model checking community tend to favour CTL model checkers [CES86, McM92a]. An important issue is therefore finding equivalent LTL and CTL formulas. One of the aims of this thesis is thus to answer the open problem [CD88]: *Given a CTL* formula, is there an equivalent CTL formula?* In section 2.3 we gave some CTL equivalences for LTL formulas that are non-trivial, for example the formula for weak fairness. In Chapter 6 it will be shown how these were obtained.

Chapter 3

A Tour of Model Checking Approaches

A model checker determines whether a model of a system, given as a Kripke structure, satisfies a temporal logic specification. Model checking has enjoyed strong support in the formal methods (verification) community, since it can be fully automated in most cases, for example in the particular case we are concerned with here: models given as finite-state Kripke structures and properties specified in CTL*. Full automation means that the user of the model checker need not know the inner-workings of the system, thus making it more widely applicable to the computer science community as a whole. This is in marked contrast with theorem proving, where user input and knowledge of the system is often crucial to the successful completion of a verification task.

The first model checking algorithms were independently developed in the early 1980's by Clarke and Emerson [CE81] and Queille and Sifakis [QS82]. Although both these model checkers used a branching time temporal logic as a specification language, the former is better known since it introduced the logic CTL. In [CES86] the complexity of the CTL model checking algorithm was

reduced to being linear in both the number of states in the Kripke structure, $|S|$, and the size of the formula, $|f|$ (the original algorithm was polynomial in $|S|$ and $|f|$). Clarke and Sistla were the first to analyse the complexity of LTL model checking, showing that it is PSPACE-complete [SC85]. In [LP85] LTL model checking is shown to be linear in the size of the Kripke structure, but exponential in the size of the formula. Emerson and Lei [EL87] extended this result by showing that CTL* has the same complexity as LTL model checking.

Here we will look in more detail at the different techniques employed in the development of model checkers for CTL and LTL. Rather than grouping the algorithms by whether they are CTL or LTL model checkers, we classify existing work according to the model checking approach that was adopted.

3.1 Model Checking Classification

CTL and LTL model checkers can be classified by two characteristics: whether they are structural or automata based and whether they are local or global algorithms.

3.1.1 Structural versus Automata Based

Model checking algorithms based directly on the structure of the formula to be checked fall into the structural class. Both the first two model checking algorithms [CE81, QS82], mentioned above, fall into this category. Automata based algorithms can be thought of as indirectly based on the structure of the formula, since the formula is first translated to an automaton and the model checking then proceeds in an automata-theoretic fashion. Structural algorithms therefore have the advantage that they do not require an extra translation phase, whereas the principal advantage of the automata approach is that existing methods and

results from automata theory can be used. Note, however, that the automaton translated from a formula can be more succinct than the original formula and hence, despite the time spent during the translation phase, can allow more efficient model checking.

3.1.2 Local versus Global

A global model checker labels all the states in a Kripke structure in which a state formula holds. A local model checker, on the other hand, answers the question whether a state formula holds in the initial state of a Kripke structure. In the global case, all the states in the Kripke structure need to be visited, whereas in the local case it might be possible to label the initial state without visiting all the states. In the worst-case all the states must be visited by a local model checking algorithm.

3.1.3 Road Map

	Structural	Automata
Global	CTL [CE81, QS82, McM92a]	CTL [BVW94]
Local	LTL [LP85, Fis92, BCG95] CTL [VL93]	LTL [VW86b, BFG89, CVWY92] CTL [BVW94]

Table 3.1: Classifying CTL and LTL Model Checkers

The algorithms which will be discussed in the remainder of the chapter are classified in Table 3.1. Note however that this is by no means an exhaustive survey of CTL and LTL model checkers; the intention is rather to show examples of the major approaches for developing CTL and LTL model checkers. An interesting observation from Table 3.1 is that all the LTL model checkers we survey are local model checkers. The reason is that the satisfaction of LTL formulas are determined with respect to paths in the Kripke structure, and since all reachable states are on a path from the initial state of the Kripke

structure it follows that a local model checking algorithm is the more natural approach. In contrast, a CTL formula is true or false in a state of the Kripke structure and hence either a local or global algorithm can be developed for CTL model checking.

To the best of our knowledge all the algorithms discussed have been implemented with the exception of [LP85], [VW86b] and [BVW94][1]. Implementations of improved versions of the algorithms of [LP85] and [VW86b] are discussed in respectively [BFG89] and [CVWY92]. Our algorithm for obtaining efficient CTL* model checking, discussed in the remaining chapters of this thesis, indicates how the two algorithms of [BVW94] can be improved to allow an efficient implementation.

3.2 Global and Structural

3.2.1 Clarke, Emerson and Sistla

[CE81, CES86]

This algorithm requires the complete Kripke structure to be checked to be stored in memory throughout the model checking process. The *length* of a formula is calculated in terms of the maximum depth of the state subformulas within it. For example, for $\psi = AG((AFp \vee EGp)\ U\ AFq)$, p and q are of length 1, AFp, EGp and AFq are of length 2, $AFp \vee EGp$ is of length 3 and $length(\psi) = 4$. The algorithm operates in stages when checking a formula ψ in a Kripke structure K: first, all the states in which state subformulas of length 1 hold are labelled; then in the next stage those of length 2 are labelled in all states. After stage i all subformulas of length smaller or equal to i will be labelled at the states in which they are true. The algorithm terminates when all the states for which the original formula holds are labelled, i.e. when

[1] In this paper two algorithms for CTL are given, one local and one global (see section 3.4.1).

$i = length(\psi)$. If for the initial state s_0 of K we have that ψ is labelled at that state then $K, s_0 \models \psi$. The algorithm is structural, since the labelling at each stage is dependent on the structure of the formula and it is global since at every stage, all the states in which a subformula is true are labelled.

Labelling states with the negations of propositions, $\wedge$, $\vee$, AX and EX is straightforward. The more interesting cases are those for U and V. Here only a brief description of the algorithms for labelling states with universal and existentially quantified U formulas (V, being the dual of U, is similar) is given; the interested reader is referred to [CES86] for a more detailed description. First, note that because states are labelled with the shortest subformulas first, when a formula of the form $\psi_1\ U\ \psi_2$ is to be labelled, both ψ_1 and ψ_2 are already labelled in all the states they are true. A state is labelled by a formula of the form $E(\psi_1\ U\ \psi_2)$ by first finding all the states in which ψ_2 is labelled and working backwards, by using the converse of the transition relation, to find all the states that can reach these ψ_2-labelled states on a path on which every state is labelled by ψ_1. All such states are labelled with $E(\psi_1\ U\ \psi_2)$. States are labelled with formulas of the form $\psi = A(\psi_1\ U\ \psi_2)$ with a depth-first algorithm: in the current state if ψ_2 is labelled then ψ is also labelled, if ψ_2 is not labelled, but ψ_1 is, the successor states are traversed in a depth-first manner; as soon as a cycle on the current path is found or a state where neither ψ_1 nor ψ_2 are labelled then $\neg\psi$ is labelled. An important aspect of this algorithm is that a *state store* can be maintained that contains all the states already visited by the algorithm; when a state is revisited its successors are not traversed again and the algorithm proceeds depending on the labelling of the revisited state. Due to the use of the store, each state in K will only be visited once during the labelling of a $A(\psi_1\ U\ \psi_2)$ formula.

The complexity of the algorithm is $O(length(\psi) \times |S|)$, where ψ is the formula being checked and $|S|$ is the number of states in the Kripke structure. In [CES86] the algorithm is extended to handle fairness, by adding a component

to the Kripke structure, call it F, that contains a collection of predicates on the states of K. A path p is now a *fair* path *iff* for each $g \in F$, there are infinitely many states on p which satisfy predicate g. The fair version of CTL has the same semantics as CTL, except that the path quantifiers range over fair paths.

3.2.2 Queille and Sifakis

[QS82, QS83]

They use a branching time logic with similar operators to those of CTL: $POT(\psi)$, potentially ψ, is the same as $EF\psi$, $INEV(\psi)$, inevitably ψ, is the same as $AF\psi$, $ALL(\psi)$, always ψ, is the same as $AG\psi$ and $SOME(\psi)$, sometimes ψ, is the same as $EG\psi$. In [QS83], they extend the operators to be similar to existential and universal quantification of U and V operators, e.g. $POT(\psi)$ becomes $POT(\psi_1, \psi_2)$ $(= E(\psi_1\ U\ \psi_2))$. The algorithm is based on the fixpoint characterisations of the operators and is dependent on the pre-image of a state being available (the states from which the current state can be reached). The actual algorithm will not be elaborated on in this section since it is very similar to the one on which OBDD based model checkers are based, which is described in Appendix A. The original algorithm of [QS82] is extended, in a similar fashion as described in the previous section, to handle fairness [QS83] (called *fair reachability of predicates*). They also show how formulas expressed over fair paths can be transformed to equivalent formulas over unfair paths. Since their notion of fairness is similar to unconditional and weak fairness this result is similar to our own translations given in section 2.3.

3.2.3 McMillan

[McM92a]

McMillan first considered model checking without representing the transition relation explicitly in 1987. Instead he used an ordered binary decision diagram (OBDD) encoding of the transition relation. Since then OBDD based model checking, or *symbolic model checking* as its often referred to, has become very popular [McM92a, McM92b, EFT91, HDDY92, HD93, HBK93, GL93, DB95, BMS95]. In Appendix A an introduction to OBDDs is given as well as a detailed description of how OBDDs are used for CTL model checking.

The global algorithms have a serious drawback in practice due to the fact that the complete transition relation (state graph) must be kept in memory throughout the model checking process. OBDD based model checkers help to combat this, since they are global and can in some cases handle much larger systems than those where the state space is encoded explicitly.

The early (non-symbolic) model checkers could only handle models with at most 10^7 states, whereas symbolic model checkers quickly pushed this limit to 10^{20} states [BCM+90, McM92a] and even 10^{120} [BCL91], albeit only for well chosen examples. Although this might seem overwhelming evidence for using symbolic model checkers, it is however a fact that many models still suffer from a state explosion when the variable ordering within the OBDD is not well chosen. At AT&T, for instance, models are first model checked using non-symbolic methods, and only when those fail (i.e. run out of memory) do they use OBDD based model checking[Kur95]. Unfortunately, computing a variable ordering for an OBDD such that the OBDD is of minimal size is NP-complete [THY93]. Developing heuristics for finding an efficient ordering and even doing on-the-fly reordering of variables in an OBDD is an active area of research [Bry86, BMS95]. For some functions, most notably multiplication [Bry91], the size of the OBDD grows exponentially in the word size regardless of the variable ordering.

3.3 Local and Structural

3.3.1 Vergauwen and Lewi

[VL93]

This interesting and above all practical algorithm was the first local CTL model checker to be developed. Or more correctly, *specifically* for CTL, since much of the early work on local model checking was done for the μ-calculus [SW89, SW91, BS90] and CTL has a straightforward translation to the μ-calculus [Dam92].

The algorithm proceeds depth-first through the Kripke structure and only labels states with formulas when required. When trying to label a state with a formula ψ some of the subformulas of ψ might be required to be labelled first. For example when a state is to be labelled with $\psi = A(\psi_1\ U\ \psi_2)$ then the algorithm will proceed to first check whether ψ_2 is labelled at the state by making a recursive call if ψ_2 is not already labelled. This is the opposite approach to the global algorithm of Clarke, Emerson and Sistla where the subformulas are labelled on all the states in which they hold regardless of whether they will be required for labelling any other formulas.

Negation, boolean connectives and the next formulas (AX and EX) are again straightforward to label in a depth first manner. In the $A(\psi_1\ U\ \psi_2)$ case the same algorithm as in [CES86] is used, but in the need-driven fashion described above. The interesting case is labelling $E(\psi_1\ U\ \psi_2)$ in a depth first manner. Note in [CES86] this was achieved by first computing all the states in which ψ_2 holds and then working backwards. In the depth-first approach the problem is that a straightforward solution for labelling $E(\psi_1\ U\ \psi_2)$ exists, but this algorithm runs in quadratic time when nested $E(\ldots\ U\ \ldots)$ formulas are encountered. The problem arises, because during the labelling many states are visited, but none of the states are labelled, only the initial state is labelled. A

novel approach of keeping track of the visited states that can also be labelled with $E(\psi_1 \; U \; \psi_2)$ is developed. In essence their algorithm traded space for time.

3.3.2 Lichtenstein and Pnueli

[LP85]

This was one of the first algorithms for doing LTL model checking. They define a closure for an LTL formula, $CL(\psi)$, and show that the size of this closure $|CL(\psi)| \leq 5|\psi|$. This closure is essentially a maximal tableau that characterises all the models for the formula. When checking whether a formula is true in the start state of a Kripke structure K, the product of the state transition graph of K is taken with the closure of the formula (referred to as the product graph P). Each node in P therefore consists of two components (s, f) where $s \in K$ and $f \in CL(\psi)$. The number of nodes in P is bounded by $|S| \times 2^{5|\psi|}$, where $|S|$ is the number of states in the Kripke structure and $|\psi|$ refers to the size of the formula. The rest of the algorithm is based on the fact that any infinite path in P will get *trapped* within a strongly connected component (SCC) of P. Since, during the construction of P all the formulas true at a node are part of the second component (from the closure of the formula), the algorithm constructs all the maximal SCCs and checks whether there is a node that is labelled with the original formula, ψ.

This algorithm is local since, although it requires the product graph to be constructed and kept in memory, when a node is found in an SCC that is labelled with the formula being checked the algorithm reports that the formula is satisfiable without labelling anymore nodes in the graph. Since every node in the product graph needs to be visited during the construction of the SCCs the complexity of the algorithm is $O(|S| \times 2^{5|\psi|})$. Since the algorithm constructs essentially a maximal tableau for the formula to be checked it is not considered to be a practical algorithm, and, to the best of our knowledge, no implementation

exists.

3.3.3 Fisher

[Fis92]

This was one of the first LTL algorithms to construct the product graph on-the-fly during the model checking procedure. It uses the common approach to LTL model checking of negating the formula and looking for the existence of an accepting path (if such a path exists then the original formula does not hold). Since only the parts of the graph required to find an accepting path is constructed, it follows that memory and time will be saved in the case where the path are found before the complete graph is constructed. In other words, the algorithm is especially efficient when $s \not\models A\psi$. An interesting point to note about this algorithm is that instead of constructing SCCs, it keeps track of all subformulas of ψ that are *unsatisfiable* at a node in the graph, therefore making sure that formulas will not be checked more than once from a specific node in the graph.

3.3.4 Bhat, Cleaveland and Grumberg

[BCG95]

Here the following approach is taken for doing LTL model checking: rather than looking for the existence of a path that satisfies the negation of the formula, all the paths must satisfy the formula. A form of closure construction, called *subgoals*, is used in a similar fashion to [LP85]. Furthermore, an efficient algorithm for the on-the-fly construction of SCCs (due to Tarjan [Tar72]) is used to find infinite paths that cannot be accepted. An infinite path is accepted only if there exists a state on the path that is labelled with a V operator. The SCCs are built in memory, but are discarded when all the paths through them

are accepting (i.e. have a state labelled with a V operator). For efficiency, all states visited are recorded as well as all the subformulas that do not hold in those states.

3.4 Global and Automata

3.4.1 Bernholtz, Vardi and Wolper

[BVW94]

The first approach to doing automata-theoretic CTL model checking was to translate the CTL formulas to nondeterministic Büchi automata [VW86b]. This translation however caused an exponential size increase, and, since the complexity of CTL model checking is known to be linear in both the size of the Kripke structure and the formula, this approach found no favour in the CTL model checking community. In [BVW94] it was shown that CTL formulas can be translated linearly to alternating tree automata and also that both local and global CTL model checking are possible with these automata. The use of alternating tree automata for efficient CTL* model checking is discussed in detail in the following chapters.

3.5 Local and Automata

3.5.1 Vardi and Wolper

[VW86b, VW94]

For linear time temporal logics, notably LTL, a close relationship with nondeterministic automata has been established [VW86b, VW94]. Essentially, with each linear time formula, an automaton over infinite words is associated that accepts exactly all the computations that satisfy the formula. Therefore if we

consider the Kripke structure to be an automaton as well, call it A_K, with the automaton describing the formula, A_ψ, then model checking can be described as a language containment problem:

$$\mathcal{L}(A_K) \subseteq \mathcal{L}(A_\psi)$$

This can be rewritten as a nonemptiness problem of intersecting automata:

$$\mathcal{L}(A_K) \cap \mathcal{L}(\overline{A_\psi}) = \emptyset$$

LTL formulas can express properties on infinite behaviours, therefore automata that accept infinite sequences (words) are required. Nondeterministic Büchi automata (NBA) [Büc62] can accept infinite sequences and are often used for automata-theoretic LTL model checking [VW86b, Tho90, VW94, Var96].

A Büchi automaton A is a 5-tuple $(\Sigma, S, \rho, s_0, F)$, where Σ is a finite alphabet, S is a finite set of states, $s_0 \in S$ is the initial state[2], $\rho : S \times \Sigma \to 2^S$ is a transition function and $F \subseteq S$ is the set of accepting states. Intuitively, $\rho(s, a)$ is the set of states A can move into when it reads symbol a when in state s. Since it can move to a set of states, the Büchi automaton is nondeterministic. If an infinite word, $w = a_0, a_1, \ldots$ over Σ is given as input to A then a *run* of A is the sequence $s_0, s_1, \ldots$ where $s_{i+1} \in \rho(s_i, a_i)$, for all $i \geq 0$ (we also refer to a run as a path of states). If we define $inf(\pi)$ as the set of states that occur infinitely often on the infinite path π, then π is an accepting path *iff* $inf(\pi) \cap F \neq \emptyset$.

A Kripke structure $K = (Props, S, R, s_0, L)$ can be viewed as a Büchi automaton $A_K = (\Sigma, S, \rho, s_0, S)$, where $\Sigma = 2^{Props}$ and $s' \in \rho(s, a)$ *iff* $(s, s') \in R$ and $a = L(s)$. The automaton A_K has as its accepting set all the states in the automaton and therefore any run of the automaton is accepting. Thus, $\mathcal{L}(A_K)$ is the set of computations (possible behaviours) of K.

In [VW94, Var96] it is proven that for an LTL formula ψ a Büchi automaton A_ψ can be constructed such that $\mathcal{L}(A_\psi)$ is the set of computations that satisfies

[2]In the general case there is a set of initial states, but for model checking we only require a single initial state.

formula ψ with the number of states of $\mathcal{L}(A_\psi)$ in $O(2^{O(|\psi|)})$. Furthermore, in [Var96] it is shown that for Büchi automata the following holds: $\mathcal{L}(\overline{A_\psi}) = \mathcal{L}(A_{\neg\psi})$.

In general the following does not hold for Büchi automata: $\mathcal{L}(A_1 \times A_2) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$, since this implies that the two automata must go infinitely often and *simultaneously* through accepting states. Here, however, since all the states of A_K are accepting we have:

$$\mathcal{L}(A_K \times A_{\neg\psi}) = \mathcal{L}(A_K) \cap \mathcal{L}(A_{\neg\psi})$$

Automata based LTL model checking can therefore be described by the following three steps:

1. Negate formula ψ and create the NBA $A_{\neg\psi}$.
2. Construct the product automaton $A_{K,\neg\psi} = K \times A_{\neg\psi}$.
3. **If** $\mathcal{L}(A_{K,\neg\psi}) \neq \emptyset$ report invalid **else** report valid.

As an example, consider checking whether $AFG\neg p$ is true in the initial state of K given in Figure 3.1, i.e. is the language accepted by the Büchi automaton for K included in the language of the Büchi automaton for $FG\neg p$. First we negate the formula $FG\neg p$ and generate the Büchi automata for GFp given in Figure 3.2. The product automata, Figure 3.3, is nonempty since it has a run that infinitely cycles through an accepting state $(h, 2)$, therefore we have $K, x \not\models AFG\neg p$.

A Büchi automaton accepts some word *iff* there exists an accepting state reachable from the initial state and from itself [TB73, VW94]. It is easy to see that a linear time algorithm exists to find such an accepting state, thus matching the result in [EL87]. Decompose the state graph of the automaton into SCCs, which can be done in time linear in the size of the automaton [Tar72]; the

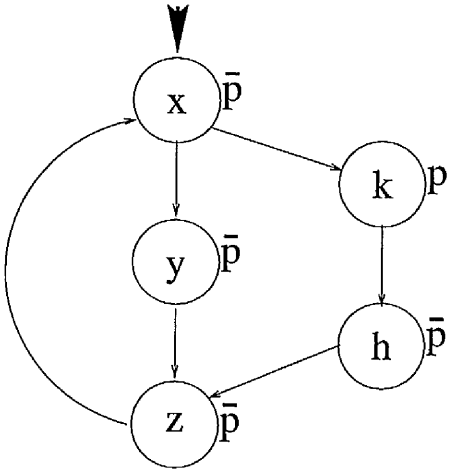

Figure 3.1: Kripke structure $K = (\{\{p\}, \{\neg p\}\}, \{x, y, z, k, h\}, R, x, L)$

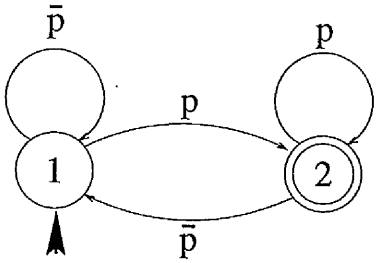

Figure 3.2: Büchi automaton for GFp $A_{GFp} = (\{\{p\}, \{\neg p\}\}, \{1, 2\}, \rho, 1, \{2\})$

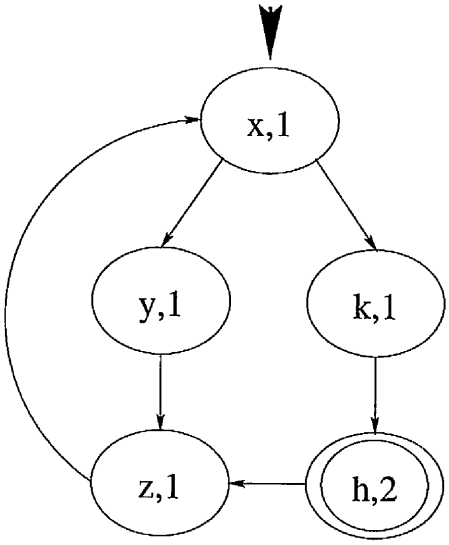

Figure 3.3: Büchi automaton for product $A_K \times A_{GFp}$.

automaton is nonempty *iff* an accepting state exists in any of the SCCs. Since checking whether a Büchi automaton accepts some word can be done in time linear in the size of the automaton and an LTL formula ψ can be translated to a Büchi automaton with $O(2^{O(|\psi|)})$ states this gives model checking complexity $O(|S| \times (2^{O(|\psi|)}))$ where $|S|$ is the number of states in the Kripke structure to be checked. The complexity matches the results of [LP85, SC85] that was obtained without the use of automata theory.

3.5.2 Barringer, Fisher and Gough

[BFG89]

This algorithm was one of the first automata-theoretic LTL model checking algorithms to be implemented. This algorithm takes a similar approach than the one described above except that rather than constructing the Büchi automaton for a formula, a tableau containing just enough information to characterise the possible models for a formula is constructed[Gou84]. In a similar fashion to the algorithm above a product graph is constructed from the tableau for the formula and the Kripke structure and checked for accepting paths by examining the terminal SCCs in the graph. In the implementation of the model checker an efficient algorithm for constructing the terminal SCCs is used based on an algorithm by Tarjan [Tar72].

3.5.3 Courcoubetis et al.

[CVWY92]

They show that during the nonemptiness check of a Büchi automaton the computation of SCCs can be avoided. Note that constructing SCCs is not very memory efficient since the states in the SCCs must be stored during the procedure. The idea is to use a nested depth-first search to find accepting states

that are reachable from themselves.

```
1     dfs(state s)
2        Add (s,0) to VisitedStates;
3        FOR each successor t of s DO
4          IF (t,0) not in VisitedStates THEN dfs(t) END
5        END
6        IF s is an accepting state THEN seed = s; 2dfs(s) END
7     END;
8
9     2dfs(state s)
10       Add (s,1) to VisitedStates;
11       FOR each successor t of s DO
12         IF (t,1) not in VisitedStates THEN 2dfs(t) END
13         ELSIF t==seed THEN report nonempty END
14       END
15    END
```

Figure 3.4: Nested DFS

Such an algorithm is given in Figure 3.4. *VisitedStates* is a data-structure, usually a hash table, that keeps track of all states already seen during the search. The algorithm works as follows: when the first search backtracks to an accepting state a second search is started to look for a cycle through this state. In [CVWY92] it was stated that the memory requirements of the nested depth-first search would be double that of a single depth-first search, but in [GH93] it is shown that only two bits need to be added to each state to separate the states stored in *VisitedStates*. Unfortunately, the time might double when all the states of the automaton are reachable in both searches and there are no cycles through accepting states.

Of all model checkers, SPIN [Hol91] is probably the most widely used: in [HP96] it is mentioned that there are more than 2000 installations of SPIN[3], with an even spread among commercial and academic usage. Correctness properties in SPIN are specified by the so-called *never claim*, which is essentially a Büchi automaton expressing unacceptable behaviour (hence the name *never*

[3]More recently it was estimated nearer 4000 [Hol97a]

claim). Checking nonemptiness of the automaton comprising the product of the *never claim* with the model of the system is done with the nested depth-first algorithm shown in Figure 3.4 [Hol91, HPY96]. The product automaton is also built on-the-fly during the depth-first search for memory efficiency. An interface for doing LTL model checking exists by translating a LTL formula to a *never claim* (Büchi automaton).

The nested depth-first search to determine whether an accepting state can be reached from itself, is the most memory efficient of the LTL model checking algorithms described here. Building SCCs (as in [BCG95]) can be very memory inefficient when model checking reactive systems. The reason is that reactive systems do not terminate and hence it is not uncommon for all reachable states of the system to be reachable from each other, i.e. all the states to be in one SCC. An example, albeit a trivial one, of such a system is the mutual exclusion system shown in Figure 2.1.

Lastly it is worth noting that there have been some attempts at doing automata-theoretic LTL model checking with OBDDs [BCM+90, Kur94]. The COSPAN model checker [HHK96], similar to SPIN, checks whether a property specified as a Büchi automaton is satisfied in a Kripke structure. COSPAN can either operate in an explicit state mode or in an OBDD mode. In the OBDD mode interesting heuristics are used to determine when cycles in the product automaton exist with no accepting states (so-called *bad cycles*) [HKSV97]. Unfortunately it is observed that these heuristics only work in limited cases.

3.6 Discussion

3.6.1 Local over Global

An advantage of local model checking over global model checking is that the transition relation (state space) of the Kripke structure can be generated *on-the-fly* during the model checking process. The description of a reactive system to be checked can therefore be given as transition rules describing how a state must be transformed to enable the transition to a successor state. On-the-fly model checking is more space efficient than global model checking since in some cases the complete state space of a model need not be generated to determine whether a formula is true or false in the initial state.

We argue that in practice, space efficiency is more important than time efficiency: one is quite prepared to wait ten minutes for a model checker to finish executing, but one learns nothing if after ten seconds the program aborts due to a lack of memory. This is an argument for using on-the-fly (local) as opposed to global model checking algorithms. Although OBDD based algorithms can be more space and time efficient than local model checkers, their behaviours are highly dependent on the problem being tackled and therefore in many cases only experts can achieve good results.

3.6.2 Automata over Structural

Although the time complexity of the automata based algorithms and those based on the structure of the formula are the same for LTL model checking, namely linear in the size of the Kripke structure and exponential in the size of the formula, the actual time taken in the structural approach is dependent on the exact formula. Structural algorithms may take different amounts of time (and space) to check equivalent formulas on the same Kripke structure. For example, in [BC96] it is observed that the model checker of section 3.3.4

finds it *easier* to check the formula $A(FG(p \wedge Fq))$ than the equivalent formula $A(FGp \wedge GFq)$. The automata approach does not tend to suffer this problem: the two formulas above give the same Büchi automaton when using the LTL to Büchi translation of [GPVW95] (see Figure 3.5, the T on the one transition stands for true and indicates any input that is read can cause the transition). Since formulas used for model checking are "small" the cost of the translation from LTL to NBA is negligible.

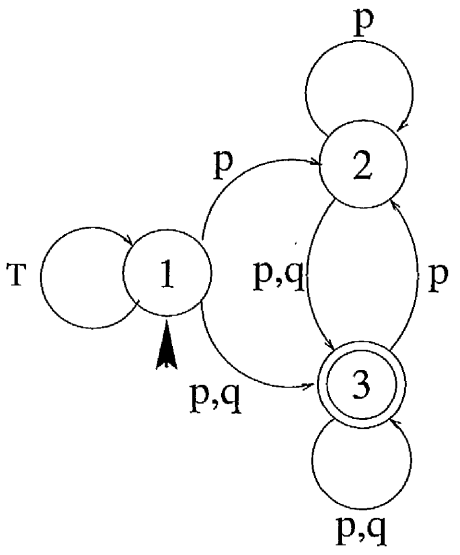

Figure 3.5: Büchi Automaton for $FG(p \wedge Fq)$ and $FGp \wedge GFq$

3.7 Concluding Remarks

An interesting observation is that the LTL model checking algorithms described in this chapter can easily check formulas of the form $E\psi$ where ψ contains no state subformulas. In the approaches of [LP85, BFG89, Fis92, HPY96], if a formula of the form $E\psi$ is to be checked the formula is *not* negated and if a path satisfying ψ is found the formula holds in the Kripke structure. In the algorithm of [BCG95] the formula $E\psi$ is negated and if the model checker

finds that all paths satisfy $\neg\psi$, the original formula does not hold, otherwise it holds. Although LTL formulas are traditionally defined over *all* paths, in the rest of this thesis we will consider both universally quantified formulas $A\psi$ and existentially quantified formulas $E\psi$ to be LTL formulas (where the only state subformulas in ψ are propositions). Clearly this will not alter the complexity of the model checking algorithms, but will add expressive power. Both SPIN [HPY96] and the model checker of Bhat *et al* [BCG95] allow formulas to be given in both the $E\psi$ and $A\psi$ forms.

It is surprising to find that although there are numerous model checkers for the sublogics CTL and LTL, CTL* model checking has not received the same amount of attention. A reason for this might be that theoretically the CTL* model checking problem is the same as LTL model checking. LTL model checkers check formulas of the form $A\psi$ and $E\psi$, where ψ contains no state subformulas. But similarly, they can also be extended to handle the cases where ψ does contain state subformulas, by calling themselves recursively to determine whether the subformula is true or false. For example, given the formula $\psi = A(FGp \wedge EFq)$, when the truth-value of EFq is required in a state s_i, the model checker is called recursively to first solve the problem $s_i \models EFq$ before continuing.

The LTL model checker described in section 3.3.4 is extended in the above fashion to do local CTL* model checking [BCG95]. As was mentioned in the previous section, this model checker is not as memory efficient as the SPIN system and furthermore its behaviour is dependent on the exact form of the formula to be checked.

In [Dam92] it is shown that CTL* formulas can be encoded in the powerful branching time logic the μ-calculus. Model checking algorithms for the μ-calculus are well studied [SW89, SW91, BS90, Cle90, CKS92], but unfortunately no polynomial time algorithm has yet been discovered. Using model

checking algorithms that can handle the full μ-calculus therefore seem unlikely to yield efficient CTL* algorithms. Interestingly, the algorithm of section 3.3.4 has been extended to model check two powerful fragments of the μ-calculus [BC96]. One of these fragments has more expressive power than CTL*.

In this thesis we are interested in finding efficient model checking algorithms for CTL*. Specifically, we are interested in the use of automata-theoretic techniques for finding model checking algorithms that are efficient in both time and space usage. In the next three chapters we show how *alternating tree automata* and the theory of 2-player games can be combined to yield an efficient model checking algorithm for CTL*.

Chapter 4

Automata for Temporal Logic

Automata theory and temporal logic are closely related, since we can associate with each temporal formula a finite automaton on infinite objects that accepts exactly all the computations that satisfy the formula. For linear time formulas the automata take as input infinite words [WVS83, SVW87, VW94] and for branching time the input is infinite trees [ES83, SE84, Eme85, EJ88, VW86a]. The temporal logic satisfiability and model checking problems can thus be reduced to the nonemptiness checking of automata.

The automata-theoretic approach to temporal reasoning has many advantages. On the one hand, automata-theoretic techniques provide the only known efficient satisfiability testing procedures for CTL* [ES83] and the μ-calculus [EJ88]. On the other hand, new types of automata have been defined to facilitate reasoning about different temporal logics [MP87, EJS93, BG93, BVW94]. The quest for optimal decision procedures for temporal logics has also led to improvements in automata theory. For example in [Saf88] the determinization of automata, i.e. translation of a nondeterministic automaton to a deterministic automaton, on infinite words is reduced to a single exponent (in the size of the automaton) and in [SVW87] a similar result is achieved for the complementation of automata on infinite words.

In this chapter we first consider nondeterministic automata and then, the more general, alternating automata as the automata-theoretical counterpart for linear and branching time temporal logic. In the linear time case we look at word automata (i.e. nondeterministic and alternating word automata) and in the branching time case it is tree automata (i.e. nondeterministic and alternating tree automata). From work by Bernholtz [Ber95] it is known that a special type of alternating tree automata, called *hesitant alternating tree automata* (HAA), is required for efficient CTL* model checking. In the second part of the chapter efficient translations from formulas in a number of sublogics of CTL* to HAA are given.

4.1 Nondeterministic Automata

4.1.1 Word Automata - Linear Time

Nondeterministic automata over infinite words are the automata-theoretic counterpart of linear time temporal logic [WVS83, SVW87, Tho90, Kur94, VW94, GPVW95]. Specifically, nondeterministic Büchi automata (NBA) have been popular for reasoning about LTL decision problems. In section 3.5.1 the use of NBA for doing LTL model checking was described. Efficient LTL model checking with automata, however, requires efficient translations from the temporal formula to the Büchi automata. In the worst case this translation is exponential in the size of the formula [WVS83, VW94, Var96, GPVW95].

The first translation from an LTL formula to a Büchi automaton was presented by Wolper, Vardi and Sistla [WVS83, VW94]. It is based on constructing the intersection of two automata: the *local* automaton (that takes care of the state to state consistency of a run) and the *eventuality* automaton (that checks that accepting states are visited infinitely often on a run). This construction

was developed to show the theoretical connection between LTL and Büchi automata and its correctness. However, if it is applied blindly it leads to an NBA of exponential size. It is also a global construction, in the sense that only after the whole automaton is constructed can it be reduced by removing unreachable nodes.

In [Gou84] and [KMMP93], where the goal was satisfiability checking (does a model exist for a formula?) rather than model checking (is a specific model a model for the formula?), tableau constructions for linear temporal logic are given. These algorithms improve on the global algorithm given above, since they create the tableau incrementally and therefore does not realise the exponential increase in size immediately (although they will in the worst-case). The resulting tableau corresponds to a Büchi automaton and can be used for model checking [BFG89].

The algorithm of [GPVW95] allows the Büchi automaton to be built on-the-fly during the model checking process. This is different from the previous algorithm, where the automaton must be built before model checking can commence. The LTL formula is translated to a *generalised* Büchi automaton[1] [CVWY92] using a simple depth-first algorithm. Unlike some of the earlier algorithms, notably the algorithms based on the intersection of the local and eventuality automata, this algorithm is easy to implement and is used for the LTL to NBA translation within the SPIN model checker (section 3.5.1). Note that it is straightforward to translate a generalised Büchi automaton to a classical Büchi automaton as defined in section 3.5.1.

[1]The accepting condition is a set of sets of states and for a path to be accepting at least one state from each set must be visited infinitely often.

Nonemptiness Checking

Checking the nonemptiness of an NBA has been discussed in detail in section 3.5.1. An interesting observation about checking nonemptiness of an NBA is that the problem can be reduced to a 1-letter nonemptiness problem, i.e. the nonemptiness check for an NBA with a single letter alphabet. When checking whether there is an accepting path in the automaton, one only needs to find an accepting state that is reachable from the initial state and from itself; the input letters that are read when the automaton moves from one state to the next are not important and hence can be considered to be the same letter.

4.1.2 Tree Automata - Branching Time

The automata-theoretic counterpart for branching time temporal logic is automata over infinite trees [Rab69, MSS86, MSS88, VW86a]. The study of such tree automata has been instrumental in finding optimal decision procedures for various branching time temporal logics [ES83, SE84, Eme85, EJ88, VW86a]. For branching time, unlike for linear time, satisfiability and model checking complexity do not coincide (see Table 4.1); model checking is typically much easier than checking satisfiability. Nondeterministic tree automata cannot compete with this gap, essentially since the translation from formulas to automata can incur an exponential blow-up in size. Therefore, when using nondeterministic tree automata as a basis for model checking the resulting algorithm's time complexity will be exponential in the size of the temporal formula.

	Satisfiability	Model Checking
LTL	PSPACE-complete	PSPACE-complete
CTL	EXPTIME-complete	Linear Time
CTL*	2EXPTIME-complete	PSPACE-complete

Table 4.1: Satisfiability and Model Checking Complexity of LTL, CTL and CTL*

Since nondeterministic automata have traditionally been used for automata-theoretic model checking and since for certain branching time temporal logics model checking is linear (e.g. CTL), automata-theoretic techniques have been considered inapplicable to branching time model checking. In [BVW94, Ber95] it is shown that the use of *alternating* automata over infinite trees is the automata-theoretic counterpart of branching time temporal logics that allows efficient model checking.

4.2 Alternating Automata

Alternating automata generalise nondeterministic automata, since they can express both existential and universal choice, whereas nondeterministic automata can only express existential choice. In fact, the name refers to the automaton's ability to *alternate* between existential and universal choice. Alternation is studied in the context of automata over finite objects in [BL80, CKS81, Lei81, Var96] and infinite objects in [MH84, MSS86, MSS88].

In order to define the alternating automata of interest here, the following definition is required: for a given set X, let $B^+(X)$ be the set of positive Boolean formulas over X (i.e. boolean formulas built from elements in X using $\wedge$ and $\vee$), where the formulas *true* and $false$ are also allowed. $Y \subset X$ satisfies $\beta \in B^+(X)$ if β is satisfied when assigning true to the members of Y and false to $X - Y$. For example, the set $\{s_0, s_1\}$ satisfies the formula $(s_0 \vee s_2) \wedge (s_1 \vee s_3)$, but the set $\{s_0, s_2\}$ does not.

4.2.1 Word Automata - Linear Time

First we show the difference between nondeterministic and alternating Büchi word automata with the aid of the function $B^+(X)$ (defined above).

For a nondeterministic Büchi word automaton $A = (\Sigma, S, \delta, s_0, F)$ the transition function δ maps a state s and an input symbol $a \in \Sigma$ to a set of states indicating the possible nondeterministic choice for the automaton's next state. The function δ can be represented by using $B^+(S)$; for example, $\delta(s, a) = \{s_0, s_1, s_2\}$ can be written as $\delta(s, a) = s_0 \vee s_1 \vee s_2$. When using alternating automata, however, δ can be an arbitrary formula from $B^+(S)$; for example

$$\delta(s, a) = (s_0 \wedge s_1) \vee (s_2 \wedge s_3)$$

meaning that the automaton accepts the word aw (where a is a symbol and w is a word) when it accepts a in state s and accepts w from both states s_0 and s_1 or from both s_2 and s_3. The transition combines therefore both existential choice (disjunction) and universal choice (conjunction).

Formally, an alternating Büchi word automaton is a tuple $A = (\Sigma, S, \delta, s_0, F)$, where Σ is a finite alphabet, S is a finite set of states, $s_0 \in S$ is the initial state, F is the set of accepting states and $\delta : S \times \Sigma \to B^+(S)$ is a partial transition function. Since an alternating Büchi automaton can express universal choice, a run of the automaton is a tree. Each node in the tree is mapped to a state in S. The branches in the tree can be either finite or infinite: they are finite when the boolean expression in the transition function is either **true** or **false**. A run is accepting if all its branches are either finite and terminated by a **true** or infinite and nodes labelled by a state in F are seen infinitely often on the branch.

Alternating Büchi word automata generalise nondeterministic Büchi word automata: nondeterministic automata correspond to alternating automata where the class of transition functions used are restricted to those that relate states s_i with disjunctions of the form $\bigvee s_i$. In [MH84] it is shown that they have the same expressive power, but alternating automata can be more succinct, exponentially more succinct in fact. When considering the translation of LTL

formulas to Büchi automata the succinctness of alternating automata over nondeterministic automata is very apparent: LTL formulas can be translated linearly to alternating Büchi automata (see next paragraph and [MSS88]), but can cause an exponential increase when translated to nondeterministic Büchi automata.

Given an LTL formula $A\varphi$, an alternating Büchi word automaton $A_\varphi = (2^{Props}, S, \delta, s_0, F)$ can be constructed such that the language it accepts is exactly the set of infinite words satisfying φ. The set S is all the subformulas of φ and the acceptance condition F includes all subformulas of the form $\psi_1 \; V \; \psi_2$. The transition relation, δ, is defined in Figure 4.1.

- $\delta(q, a) =$ true if $q \in a$
- $\delta(q, a) =$ false if $q \notin a$
- $\delta(\neg q, a) =$ true if $q \notin a$
- $\delta(\neg q, a) =$ false if $q \in a$
- $\delta(\psi_1 \wedge \psi_2, a) = \delta(\psi_1, a) \wedge \delta(\psi_2, a)$
- $\delta(\psi_1 \vee \psi_2, a) = \delta(\psi_1, a) \vee \delta(\psi_2, a)$
- $\delta(X\psi, a) = \psi$
- $\delta(\psi_1 \; U \; \psi_2, a) = \delta(\psi_2, a) \vee (\delta(\psi_1, a) \wedge \psi_1 \; U \; \psi_2)$
- $\delta(\psi_1 \; V \; \psi_2, a) = \delta(\psi_2, a) \wedge (\delta(\psi_1, a) \vee \psi_1 \; V \; \psi_2)$

Figure 4.1: Translation of LTL to Alternating Büchi Word Automata

Example: Consider the LTL formula AGFp. The subformulas of the linear fragment GFp are: false V (true U p), true U p and p. Let us first look at the transition relation for $q_1 =$ true U p:

$$\delta(q_1, a) = \delta(p, a) \vee (\delta(true, a) \wedge q_1)$$

which simplifies to:

$$\delta(q_1, a) = \delta(p, a) \vee q_1$$

Now let us consider $q_0 = \text{false V } q_1$:

$$\delta(q_0, a) = \delta(q_1, a) \wedge (\delta(false, a) \vee q_0)$$

which simplifies to:

$$\delta(q_0, a) = (\delta(p, a) \vee q_1) \wedge q_0$$

State q_0 is of the form $\psi_1 \; V \; \psi_2$ and is therefore in the acceptance set F. In Figure 4.2 the alternating Büchi word automaton for GFp is given. Note that the transition function for the proposition p is combined with those for states q_0 and q_1 and is therefore not explicitly shown.

q	$\delta(q, \{\neg p\})$	$\delta(q, \{p\})$
q_0	$q_0 \wedge q_1$	q_0
q_1	q_1	true

Figure 4.2: $A_{GFp} = (\{\{p\}, \{\neg p\}\}, \{q_0, q_1\}, \delta, q_0, \{q_0\})$

Nonemptiness Checking

Unlike the nonemptiness problem for nondeterministic Büchi word automata, the nonemptiness problem for alternating Büchi word automata does not reduce to a 1-letter nonemptiness problem. The reason for this is that when an $\wedge$-choice occurs in a run, both branches must read the same input word. Take for example the alternating Büchi word automaton for GFp (Figure 4.2), when reading the two input words $\neg p, \neg p, \ldots$ and $\neg p, p, \neg p, \neg p, \ldots$ that coincide on the initial letter, but then diverge with one reading $\neg p$ infinitely and the other first reading a p then $\neg p$ infinitely. In the initial state q_0, if after reading the initial $\neg p$ the left-hand side of the $\wedge$-choice (i.e. the cycle through q_0) continues to read the first input, then that branch is accepting (since q_0 is an accepting state) and if the right-hand side reads the second input then it is also accepting since reading a p in state q_1 results in the **true** transition being taken. Clearly though, in neither of the two input words do we see a p infinitely often, which is

what GFp is stating. However if we let the two branches from the $\wedge$-choice read the same input word we get the desired result, i.e. the alternating automaton does not accept either word.

Since an alternating Büchi word automata can be translated to a nondeterministic Büchi word automata of exponential size [MH84] and checking nonemptiness of nondeterministic Büchi automata can be done in time linear in the size of the automaton [EL87], the nonemptiness problem for alternating Büchi word automata is decidable in exponential time. Interestingly, the 1-letter nonemptiness problem for alternating Büchi word automata is not linearly solvable either; in [VW86b] it is shown to be decidable in quadratic time. Furthermore, since an LTL formula can be translated to an alternating Büchi automaton of size linear in the length of the formula (see Figure 4.1) it again follows that LTL model checking can be done in time exponential in the size of the formula.

4.2.2 Tree Automata - Branching Time

In order to define automata over infinite trees we need the following basic definitions. A *tree* is a connected directed graph, with one *root* node, ε, and every other node has a unique parent. A tree τ over $\mathbb{N}$ is a subset of $\mathbb{N}^*$, such that if $x \cdot i \in \tau$, where $x \in \mathbb{N}^*$ and $i \in \mathbb{N}$, then also $x \in \tau$, and for all $0 \leq i' < i$, $x \cdot i' \in \tau$. For every $x \in \tau$ the nodes $x \cdot i$ where $i \in \mathbb{N}$ are called the *successors* of x. The *degree* of a node x, $d(x)$, in a tree τ is the number of successors of x in τ. A *leaf* is a node with no successors. A *path* π of a tree τ is a sequence of nodes starting with ε and for every $x \in \pi$, either x is a leaf or there exists a unique $i \in \mathbb{N}$ such that $x \cdot i \in \pi$. Let $inf(\pi)$ be the set of nodes that occur infinitely often on the infinite path π. Let $D \subseteq \mathbb{N}$, then a tree τ is a D-tree if τ is a tree over $\mathbb{N}$ and $d(x) \in D$ for all $x \in \tau$. A tree is called *leafless* if every node has at least one child. A Σ-labelled tree, for a finite alphabet Σ, is a pair

(τ, V), where τ is a tree and V is a mapping $V : \tau \to \Sigma$ that assigns to every node of τ a label in Σ. For model checking the type of Σ-labelled tree we are interested in has $\Sigma = 2^{Props}$ for a set of atomic propositions $Props$.

Automata over infinite trees (tree automata) run over leafless Σ-labelled trees. An *alternating tree automaton* is a tuple $(\Sigma, D, S, \delta, s_0, F)$. Here Σ is a finite alphabet, $D \subset \mathbb{N}$ is a finite set of branching-degrees, S is a finite set of states, $s_0 \in S$ is the initial state, F is the acceptance condition (the type of condition depends on the type of alternating automata; two types are discussed below) and $\delta : S \times \Sigma \times D \to B^+(\mathbb{N} \times S)$ is a partial transition function, where $\delta(s, a, k) \in B^+(\{0, \ldots, k-1\} \times S)$ for each $s \in S$, $a \in \Sigma$ and $k \in D$ such that $\delta(s, a, k)$ is defined.

A run r of an alternating automaton A on a leafless Σ-labelled tree (τ, V) is a tree where the root is labelled by s_0 and every other node is labelled by an element of $\mathbb{N}^* \times S$. Each node of r corresponds to a node of τ. A node in r, labelled by (x, s), describes a copy of the automaton that reads the node x of τ in the state s. Note that many nodes of r can correspond to the same node of τ. The labels of a node and its successors have to satisfy the transition function. Formally, a run r is a Σ_r-labelled tree $(\tau_r, \mathcal{T}_r)$ where $\Sigma_r = \mathbb{N}^* \times S$ and $(\tau_r, \mathcal{T}_r)$ satisfies the following:

1. $\varepsilon \in \tau_r$ and $\mathcal{T}_r(\varepsilon) = (\varepsilon, s_0)$
2. Let $y \in \tau_r$ with $\mathcal{T}_r(y) = (x, s)$ and $\delta(s, V(x), d(x)) = \theta$. Then there is a possibly empty set $Q = \{(c_0, s_0), (c_1, s_1), \ldots, (c_n, s_n)\} \subseteq \{0, \ldots, d(x) - 1\} \times S$, such that the following hold:
   - Q satisfies θ, and
   - $\forall i : 0 \leq i \leq n$, we have $y \cdot i \in \tau_r$ and $\mathcal{T}_r(y \cdot i) = (x \cdot c_i, s_i)$

A run is accepting if all its infinite paths satisfy the acceptance condition F. For example, the Büchi acceptance condition $F \subseteq S$ will be satisfied on an

infinite path π *iff* $inf(\pi) \cap F \neq \emptyset$. Note that we can get finite branches in the tree representing the run when either **true** or **false** is read in the transition function. However in an accepting run, only **true** can be found as leaves, since a path containing **false** is trivially not accepting.

Example: Let us consider the following transition function: $\delta(s_0, a, 2) = ((0, s_1) \vee (1, s_2)) \wedge ((0, s_3) \vee (1, s_1))$. When the automaton is in state s_0, reads input a and the branching degree of the input tree is 2 when a is read (i.e. there are 2 successor states from this state in the input tree) then the automaton will make two copies of itself (due to the $\wedge$ in δ) which could then proceed in different ways. One possibility is that one copy of the automaton proceeds to state s_1 and the next input this copy reads is in direction 0 of the input tree (which could be considered to be the first successor of the state in the tree where a was read), the other copy of the automaton could also go to state s_1 but read its input in direction 1 of the input tree (say in the direction of the second successor of the state in which a was read). In fact there are four possibilities for the automaton to proceed, summarised below:

- one copy proceeds in direction 0 in state s_1 and one copy proceeds in direction 0 in s_3.
- one copy proceeds in direction 0 in state s_1 and one copy proceeds in direction 1 in s_1.
- one copy proceeds in direction 1 in state s_2 and one copy proceeds in direction 0 in s_3.
- one copy proceeds in direction 1 in state s_2 and one copy proceeds in direction 1 in s_1.

Weak Alternating Automata

Weak alternating automata (WAA), introduced by Muller et al. [MSS86], was one of the first types of alternating automata to be used for reasoning about temporal logic. In [MSS86] they show that WAA define the set of weakly definable languages [2] defined by Rabin in [Rab70], hence the name weak alternating

[2] A language is weakly definable when it is definable by a formula in the weak monadic logic of the tree where one allows quantification only over finite sets.

automata. In [MSS88] WAA are used to explain the complexity of decision procedures for certain temporal logics. More recently, WAA were used to define linear time algorithms for model checking CTL [Ber95]. In [BVW94], Bernholtz et al. defined *bounded alternation* WAA, that also allow space efficient CTL model checking. In fact, it was shown that CTL model checking is in NLOGSPACE in the size of the Kripke structure.

A weak alternating automaton is defined as follows. Firstly, it uses a Büchi acceptance condition, $F \subseteq S$. The set of states of a WAA can be partitioned into disjoint sets S_i, such that each S_i is either an *accepting set*, i.e. $S_i \subseteq F$, or is a *rejecting set*, i.e. $S_i \cap F = \emptyset$. Furthermore, a partial order $\leq$ exists on the collection of S_i sets such that for every $s \in S_i$ and $s' \in S_j$ for which s' occurs in $\delta(s, a, k)$ for some $a \in \Sigma$ and $k \in D$, we have $S_j \leq S_i$. Thus, transitions from an S_i either lead to states in the same S_i or a lower one. An infinite path in the run of a WAA will therefore get trapped within some S_i; if this S_i is accepting then the path satisfies the acceptance condition.

Unfortunately WAA cannot be used for model checking CTL*, since CTL* can define languages that are not weakly definable. As an example we show that the alternating Büchi automaton translated from the CTL* formula EGFp (given in Figure 4.3) is not a WAA. If we consider the definition of the partial order between the S_i sets of states it is clear that both states q_0 and q_1 must be in the same S_i set: q_1 is referred to in $\delta(q_0, a, k)$ and q_0 is referred to in $\delta(q_1, a, k)$ hence q_1 and q_0 cannot be in different S_i sets because then there will not be a partial order between the two sets. Furthermore, only one of the states in the single S_i set is in the accepting set F, namely state q_1. In a WAA all S_i sets must either be accepting or rejecting, and this is clearly impossible here, since the S_i set is neither accepting nor rejecting with regards to the Büchi acceptance condition $F = \{q_1\}$.

q	$\delta(q, \{\neg p\}, k)$	$\delta(q, \{p\}, k)$
q_0	$\bigvee_{c=0}^{k-1}(c, q_0)$	$\bigvee_{c=0}^{k-1}(c, q_1)$
q_1	$\bigvee_{c=0}^{k-1}(c, q_0)$	$\bigvee_{c=0}^{k-1}(c, q_1)$

Figure 4.3: $A_{EGFp} = (\{\{p\}, \{\neg p\}\}, D, \{q_0, q_1\}, \delta, q_0, \{q_1\})$

Hesitant Alternating Automata

A stronger acceptance condition is therefore required for automata corresponding to CTL* formulas. In [Ber95] hesitant alternating tree automata (HAA) are defined that have a more restricted transition structure than WAA, but a more powerful acceptance condition. As with WAA, there exists a partial order between disjoint sets S_i of S. Furthermore, each set S_i is classified either as *transient*, *existential* or *universal*, such that for each S_i, and for all $s \in S_i$, $a \in \Sigma$ and $k \in D$ the following holds:

- if S_i is transient, then $\delta(s, a, k)$ contains no elements from S_i. Examples (assume $S_i = \{s_0\}$):
  - $\delta(s_0, a, 2) = (0, s_1) \wedge (1, s_2)$
  - $\delta(s_0, a, 2) = ((0, s_1) \vee (1, s_2)) \wedge (0, s_3)$
- if S_i is existential, then $\delta(s, a, k)$ only contains disjunctively related elements of S_i. Examples (assume $S_i = \{s_0\}$):
  - $\delta(s_0, a, 2) = (0, s_0) \vee (1, s_0)$
  - $\delta(s_0, a, 2) = ((0, s_0) \vee (1, s_0)) \wedge (0, s_1)$

  but here S_i is not an existential set
  - $\delta(s_0, a, 2) = ((0, s_0) \wedge (1, s_0)) \vee (0, s_1)$
- if S_i is universal, then $\delta(s, a, k)$ only contains conjunctively related elements of S_i. Examples (assume $S_i = \{s_0\}$):

- $\delta(s_0, a, 2) = (0, s_0) \wedge (1, s_0)$
- $\delta(s_0, a, 2) = ((0, s_0) \wedge (1, s_0)) \vee (0, s_1)$

but here S_i is not a universal set

- $\delta(s_0, a, 2) = ((0, s_0) \vee (1, s_0)) \wedge (0, s_1)$

The acceptance condition is a pair of sets of states, (G, B). From the above restricted structure of HAA it follows that an infinite path, ϕ, will either get trapped in an existential or universal set, S_i. The path then satisfies (G, B) iff either S_i is existential and $inf(\phi) \cap G \neq \emptyset$ or is universal and $inf(\phi) \cap B = \emptyset$.

Here we also define a subclass of HAA, called *1-HAA*, for which every S_i set contains only one state in the partial order. It will be shown that 1-HAA is the automata-theoretic counterpart of CTL, whereas HAA in general correspond to CTL* formulas. As an example consider again the WAA for EGFp (given in Figure 4.3) that can be considered an HAA if we change the acceptance condition to $(\{q_1\}, \{\})$; since the single S_i set contains two states q_0 and q_1 this is not a 1-HAA. Note also that the S_i set is an existential set.

As shown in [Ber95], complementing HAA is straightforward: complementing an HAA $A = (\Sigma, D, Q, \delta, q_0, (G, B))$ is $\overline{A} = (\Sigma, D, Q, \overline{\delta}, q_0, (B, G))$, where $\overline{\delta}$ is defined as switching all the true and false values and the $\wedge$ and $\vee$ symbols. For example, if $\delta(q, a, k) = p \vee (true \wedge g)$, then $\overline{\delta} = p \wedge (false \vee g)$.

4.3 Model Checking with HAA

Let us first consider the general approach to automata-theoretic branching time model checking. Recall that for linear time temporal logic each Kripke structure may correspond to infinitely many computations. Model checking is therefore reduced to checking inclusion between the set of computations allowed by the

Kripke structure and the language of an automata describing the formula (section 3.5.1). For branching temporal logic, each Kripke structure corresponds to a single nondeterministic computation. Therefore, model checking is reduced to checking the membership of this computation in the language of the automaton describing the formula [Wol89].

A Kripke structure $K = (Props, S, R, s^0, L)$ can be viewed as a tree (τ_K, V_K) that corresponds to the unwinding of K from s^0. Let $succ_R(s) = (s_0, \ldots, s_{(d(s)-1)})$ be an ordered list of s's R-successors. We define τ_K and V_K as follows:

1. $\varepsilon \in \tau_K$ and $V_K(\varepsilon) = s^0$.
2. For $y \in \tau_K$ with $succ_R(V_K(y)) = (s_0, \ldots, s_n)$ and for $0 \leq i \leq n$, we have $y \cdot i \in \tau_K$ and $V_K(y \cdot i) = s_i$.

Let φ be a branching time temporal formula and let $D \in \mathbb{N}$ be a set of degrees. Suppose that $A_{D,\varphi}$ is an alternating automaton that accepts exactly all the D-trees that satisfy φ. Consider a product of K and $A_{D,\varphi}$, i.e. an automaton that accepts the language $\mathcal{L}(A_{D,\varphi}) \cap \{(\tau_K, V_K)\}$. The language of this product automaton either contains a single tree, (τ_K, V_K), in which case $K \models \varphi$, or is empty in which case $K \not\models \varphi$. This discussion suggests the following automata-based model checking algorithm. Given a branching temporal formula φ and a Kripke structure K with degrees in D:

1. Construct the alternating automaton for the formula, $A_{D,\varphi}$.
2. Construct the product alternating automaton $A^K_{D,\varphi} = K \times A_{D,\varphi}$. This automaton simulates a run of $A_{D,\varphi}$ on the tree induced by the Kripke structure K.
3. If the language accepted by $A^K_{D,\varphi}$ is nonempty then φ holds for K, otherwise not.

Thus, a nonemptiness check for HAA is required to check CTL* properties in K. The general nonemptiness check for HAA cannot be done efficiently [Ber95]. Fortunately, taking the product with the Kripke structure K, results in a 1-letter HAA over words (i.e. an HAA with $|\Sigma| = 1$ and $D = \{1\}$), for which a nonemptiness check can be done in linear time [Ber95]. This is because the 2^{Props} labelling (resulting in the 1-letter reduction) and the branching structure of K (resulting in the automaton over words, rather than trees) are embodied in the states of $A^K_{D,\varphi}$, since every state of $A^K_{D,\varphi}$ is associated with a state in K. Therefore, all the copies of the product automaton that start in a certain state, say one associated with s, follow the same labelling: the one that corresponds to computations of K starting in s. Let us now define this product automaton. Let $A_{D,\varphi} = (2^{Props}, D, Q_\varphi, \delta_\varphi, q_0, (G_\varphi, B_\varphi))$ be an HAA which accepts exactly all the D-trees that satisfy φ and let $K = (Props, S, R, s_0, L)$ be a Kripke structure with degrees in D. The product automaton is then an HAA word automaton $A^K_{D,\varphi} = (\{a\}, S \times Q_\varphi, \delta, (s_0, q_0), (S \times G_\varphi, S \times B_\varphi))$ where δ is defined as:

- Let $Q \in Q_\varphi$, $s \in S$, $succ_R(s) = (s_0, \ldots, s_{(d(s)-1)})$ and $\delta_\varphi(q, L(s), d(s)) = \alpha$. Then $\delta((s,q),a) = \alpha'$, where α' is obtained from α by replacing each atom (c, q') in α by (s_c, q').

A run of an alternating automata is a tree; in the sequel we will display this tree as an And-Or tree with each infinite branch truncated when a node is revisited on a branch. Therefore the product automaton will be displayed in this fashion (note we do not show the $\wedge$ and $\vee$ choices when only one successor state exists in the product automaton). For example consider the product automaton of the Kripke structure in Figure 4.4 and the HAA for the CTL formula AGAFp (Figure 4.5) given as an And-Or tree in Figure 4.6. To illustrate how this product is obtained we show how the run proceeds from the initial state. In the initial state the automaton is in state q_0 and takes as input the label from

state x (namely $\neg p$) in the input tree induced by the Kripke structure K:

$$\delta(q_0, \{\neg p\}, 2) = ((0, q_1) \wedge (1, q_1)) \wedge ((0, q_0) \wedge (1, q_0))$$

If we consider y to be the first successor of x and k the second successor of x then we get:

$$\delta(q_0, \{\neg p\}, 2) = ((y, q_1) \wedge (k, q_1)) \wedge ((y, q_0) \wedge (k, q_0))$$

which is displayed graphically in Figure 4.6. Note that all the branches that reach the state (k, q_1) are trivially accepting since a **true** is read in the transition function. All other branches are infinite and their acceptance is determined by the acceptance condition $(\{\}, \{q_1\})$: the infinite branches with a q_0 component are accepting (since q_0 is in a universal set and $q_0 \notin B$, i.e. $inf(\pi) \cap \{q_1\} = \emptyset$, where π is any of the infinite branches with a q_0 component) whereas all the infinite branches with a q_1 component are not accepting (since q_1 is in a universal set and $inf(\pi) \cap \{q_1\} \neq \emptyset$, where π is any of the infinite branches with a q_1 component). The run of the alternating automata is therefore not accepting and hence $K, x \not\models AGAFp$.

In the following sections we consider the efficient translation of CTL, LTL and CTL* formulas to HAA.

4.4 CTL to 1-HAA Translation

Given a CTL formula φ and a set $D \subset \mathbb{N}$, a 1-HAA $A_{D,\varphi} = (2^{Props}, D, cl(\varphi), \delta, \varphi, (G, B))$ can be constructed such that the language which $A_{D,\varphi}$ recognises is the set of D-trees satisfying φ. The acceptance condition is (G, B), where G is the set of all EV formulas and B is the set of all AU formulas in $cl(\varphi)$. The transition function for all $a \in 2^{Props}$ and $k \in D$ is the following:

- $\delta(q, a, k) =$ true if $q \in a$

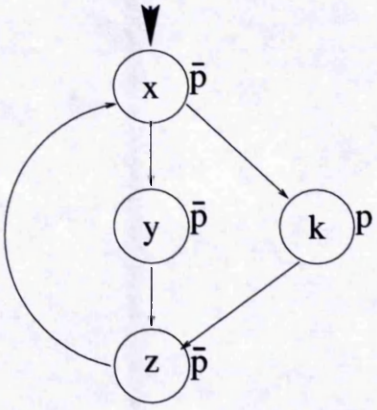

Figure 4.4: Kripke structure $K = (\{\{p\}, \{\neg p\}\}, \{x, y, z, k, h\}, R, x, L)$

q	$\delta(q, \{\neg p\}, l)$	$\delta(q, \{p\}, l)$
q_0	$\bigwedge_{c=0}^{l-1}(c, q_1) \wedge \bigwedge_{c=0}^{l-1}(c, q_0)$	$\bigwedge_{c=0}^{l-1}(c, q_0)$
q_1	$\bigwedge_{c=0}^{l-1}(c, q_1)$	true

Figure 4.5: $A_{D,AGAFp} = (\{\{\neg p\}, \{p\}\}, D, \{q_0, q_1\}, \delta, q_0, (\{\}, \{q_1\}))$

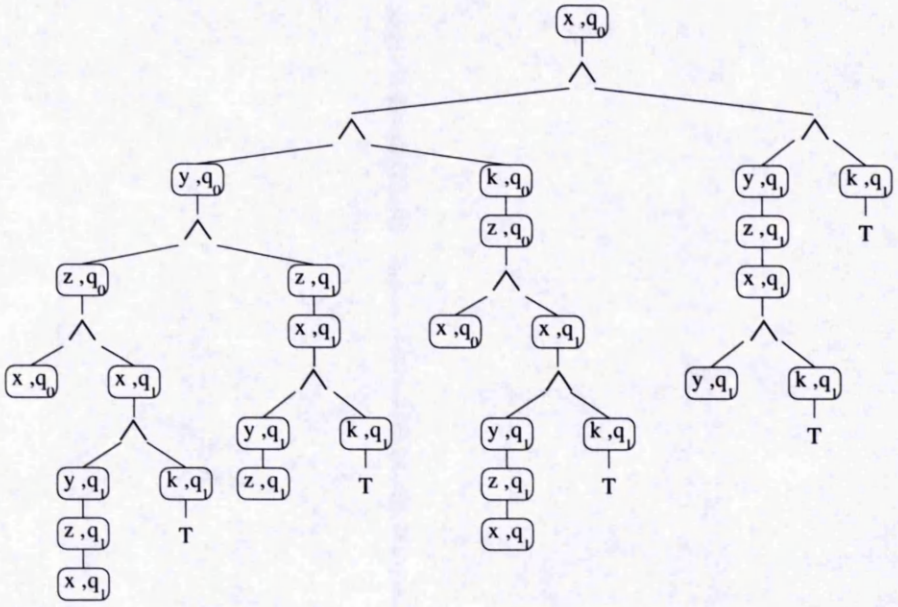

Figure 4.6: And-Or tree for the product automaton $K \times A_{D,AGAFp}$.

- $\delta(q, a, k) =$ false if $q \notin a$
- $\delta(\neg q, a, k) =$ true if $q \notin a$
- $\delta(\neg q, a, k) =$ false if $q \in a$
- $\delta(\psi_1 \wedge \psi_2, a, k) = \delta(\psi_1, a, k) \wedge \delta(\psi_2, a, k)$
- $\delta(\psi_1 \vee \psi_2, a, k) = \delta(\psi_1, a, k) \vee \delta(\psi_2, a, k)$
- $\delta(AX\psi, a, k) = \bigwedge_{c=0}^{k-1}(c, \psi)$
- $\delta(EX\psi, a, k) = \bigvee_{c=0}^{k-1}(c, \psi)$
- $\delta(A\psi_1 \; U \; \psi_2, a, k) = \delta(\psi_2, a, k) \vee (\delta(\psi_1, a, k) \wedge \bigwedge_{c=0}^{k-1}(c, A\psi_1 \; U \; \psi_2))$
- $\delta(E\psi_1 \; U \; \psi_2, a, k) = \delta(\psi_2, a, k) \vee (\delta(\psi_1, a, k) \wedge \bigvee_{c=0}^{k-1}(c, E\psi_1 \; U \; \psi_2))$
- $\delta(A\psi_1 \; V \; \psi_2, a, k) = \delta(\psi_2, a, k) \wedge (\delta(\psi_1, a, k) \vee \bigwedge_{c=0}^{k-1}(c, A\psi_1 \; V \; \psi_2))$
- $\delta(E\psi_1 \; V \; \psi_2, a, k) = \delta(\psi_2, a, k) \wedge (\delta(\psi_1, a, k) \vee \bigvee_{c=0}^{k-1}(c, E\psi_1 \; V \; \psi_2))$

and for the derived operators:

- $\delta(AF\psi, a, k) = \delta(\psi, a, k) \vee \bigwedge_{c=0}^{k-1}(c, AF\psi)$
- $\delta(EF\psi, a, k) = \delta(\psi, a, k) \vee \bigvee_{c=0}^{k-1}(c, EF\psi)$
- $\delta(AG\psi, a, k) = \delta(\psi, a, k) \wedge \bigwedge_{c=0}^{k-1}(c, AG\psi)$
- $\delta(EG\psi, a, k) = \delta(\psi, a, k) \wedge \bigvee_{c=0}^{k-1}(c, EG\psi)$

AW and EW have the same transition function as AU and EU respectively, with EW in G. Whereas, AR and ER have the same transition function as AV and EV respectively with AR in B. The special structure of the HAA follows from the fact that each formula ψ in $cl(\varphi)$ constitutes a singleton set $\{\psi\}$ in the partition and the partial order is defined by $\{\psi_1\} \leq \{\psi_2\}$ iff $\psi_1 \in cl(\psi_2)$. Note how the existential set is formed by the rules for EU and EV and the universal

set is formed by AU and AV. Furthermore, since each transition from a state ψ leads to states in $cl(\psi)$ it must follow that an infinite run will get trapped either in an existential or universal set.

Example : Consider the CTL formula $\varphi = AFAGp$. For every $D \subset \mathbb{N}$, the 1-HAA for φ is $A_{D,\varphi} = (\{\{p\}, \{\neg p\}\}, D, \{q_0, q_1\}, \delta, q_0, (\{\}, \{q_0\}))$, where δ is given by the following table:

q	$\delta(q, \{\neg p\}, k)$	$\delta(q, \{p\}, k)$
q_0	$\bigwedge_{c=0}^{k-1}(c, q_0)$	$\bigwedge_{c=0}^{k-1}(c, q_1) \vee \bigwedge_{c=0}^{k-1}(c, q_0)$
q_1	false	$\bigwedge_{c=0}^{k-1}(c, q_1)$

In state q_0 (AFAGp), if p holds then the automaton can choose either to check whether q_1 (AGp) holds or to postpone this check to the future. It would, however, not be able to postpone it forever, since this will violate the acceptance condition: q_0 must be seen only finitely often in the universal S_i set associated with q_0. In state q_1 the automaton expects a tree in which on all paths p holds.

4.5 LTL to HAA Translation

Here we consider formulas of the form $A\phi$ for which ϕ holds for all branches of the input tree and $E\phi$ where there exists a branch on which ϕ holds, where ϕ is a linear time formula.

In order to comply with the restricted transition structure of HAA the LTL formula must first be translated to a nondeterministic Büchi automaton. Note, for example, that the initial state in the alternating Büchi tree automaton for the formula $EGFp$, given in Figure 4.7, is neither part of an existential nor of a universal set.

The following approach is taken for the translation of the LTL formula to an HAA:

q	$\delta(q, \{\neg p\}, k)$	$\delta(q, \{p\}, k)$
q_0	$\bigvee_{c=0}^{k-1}(c, q_0 \wedge q_1)$	$\bigvee_{c=0}^{k-1}(c, q_0)$
q_1	$\bigvee_{c=0}^{k-1}(c, q_1)$	true

Figure 4.7: $A_{EGFp} = (\{\{p\}, \{\neg p\}\}, D, \{q_0, q_1\}, \delta, q_0, \{q_0\})$

- 1. If the formula is of the form $E\phi$, construct the Büchi word automaton that accepts ϕ, $B_\phi = (\Sigma, S, R, q_0, F)$.
  2. Translate B_ϕ to the corresponding HAA, $A_\phi = (\Sigma, D, S, \delta, q_0, (F, \emptyset))$.

- 1. If the formula is of the form $A\phi$, construct the Büchi word automaton that accepts $\neg\phi$, $B_{\neg\phi} = (\Sigma, S, R, q_0, F)$.
  2. Translate $B_{\neg\phi}$ to the corresponding HAA, $A_{\neg\phi}$. $A_{\neg\phi} = (\Sigma, D, S, \delta, q_0, (F, \emptyset))$.
  3. Negate $A_{\neg\phi}$, $\overline{A}_{\neg\phi} = (\Sigma, D, S, \overline{\delta}, q_0, (\emptyset, F))$.

The translation of a nondeterministic Büchi word automaton to an HAA is straightforward and essentially extends the word automaton by tracing it in a single branch of the tree automaton. Given a nondeterministic Büchi automaton $B = (\Sigma, S, R, q_0, F)$ the transition function of the corresponding HAA $A = (\Sigma, D, S, \delta, q_0, (F, \emptyset))$ is defined for all $s \in S$, $a \in \Sigma$ and $k \in D$:

$$\delta(s, a, k) = \bigvee_{c=0}^{k-1} \bigvee_{s_i \in R(s,a)} (c, s_i)$$

If $R(s, a) = \emptyset$ then $\delta(s, a, k) =$ **false**.

Due to the translation via nondeterministic automata it is clear that a formula of the form $E\phi$ will be translated to an HAA with only $\vee$-choices in its transition function whereas those of the form $A\phi$ will only have $\wedge$-choices. However, the translation via nondeterministic automata could also cause an exponential blow-up in size of the resulting HAA.

Example : Consider the LTL formula $\varphi = E(FGp \wedge GFq)$. For every $D \subset \mathbb{N}$, the HAA for φ is $A_{D,\varphi} = (2^{\{p,q\}}, D, \{s_1, s_2, s_3\}, \delta, s_1, (\{s_3\}, \{\}))$, where δ is given by the following table:

s	$\delta(s, \{\neg p, \neg q\}, k)$	$\delta(s, \{p\}, k)$	$\delta(s, \{q\}, k)$	$\delta(s, \{p, q\}, k)$
s_1	$\bigvee_{c=0}^{k-1}(c, s_1)$	$\bigvee_{c=0}^{k-1}(c, s_1 \vee s_2)$	$\bigvee_{c=0}^{k-1}(c, s_1)$	$\bigvee_{c=0}^{k-1}(c, s_1 \vee s_3)$
s_2	false	$\bigvee_{c=0}^{k-1}(c, s_2)$	false	$\bigvee_{c=0}^{k-1}(c, s_3)$
s_3	false	$\bigvee_{c=0}^{k-1}(c, s_2)$	false	$\bigvee_{c=0}^{k-1}(c, s_3)$

The nondeterministic Büchi word automaton for φ is given in Figure 3.5. There are two existential S_i sets in the partial order: one containing only state s_1 and the other consisting of states s_2 and s_3. State s_3 is in the set G, hence from the fact that s_3 is in an existential set a run will be accepting if s_3 is seen infinitely often on a branch.

4.6 CTL* to HAA Translation

This section is based on the description given in [Ber95]. First, the set of *maximal* state subformulas of a formula φ, $max(\varphi)$, needs to be defined: ψ is a maximal state subformula of φ if it is a state subformula and there are no other state subformulas of φ for which ψ is also a state subformula. For example, let φ = AF(Xq V AFAGp), then $max(\varphi) = \{q, AFAGp\}$.

Given a CTL* formula ψ and a set $D \subset \mathbb{N}$, we can construct an HAA $A_{D,\psi}$ such that $\mathcal{L}(\mathcal{A}_{D,\psi})$ is exactly the set of D-trees satisfying ψ, in the following fashion.

$A_{D,\psi}$ is constructed according to the structure of ψ. With each formula $\varphi \in cl(\psi)$ an HAA A_φ is associated that is composed from HAAs for the formulas in $max(\varphi)$. We assume that the state sets of the composed HAAs are disjoint and that for each one $\Sigma = 2^{Props}$ (i.e. if an HAA does not involve all the atomic propositions then it is extended). Now for each formula in

$max(\varphi) = \{\varphi_1, \ldots, \varphi_n\}$ we associate an HAA $A_{\varphi_i} = (\Sigma, S^i, \delta^i, s_0^i, (G^i, B^i))$ and its complement $\overline{A}_{\varphi_i}$, with $1 \leq i \leq n$. A_φ is now constructed as follows:

- If $\varphi = p$ or $\varphi = \neg p$, for some $p \in Props$, then A_φ is a one-state HAA.
- If $\varphi = \varphi_1 \wedge \varphi_2$ then $A_\varphi = (\Sigma, S^1 \cup S^2 \cup \{s_0\}, \delta, s_0, (G^1 \cup G^2, B^1 \cup B^2))$, where s_0 is a new state. For the states in S^1 and S^2, δ agrees with δ^1 and δ^2 respectively, but for s_0 and for all $a \in \Sigma$ we have that $\delta(s_0, a) = \delta(s_0^1, a) \wedge \delta(s_0^2, a)$. The set $\{s_0\}$ therefore constitutes a singleton transient set, with the ordering $\{s_0\} > S_i$ for all the sets S_i in S^1 and S^2.
- $\varphi = \varphi_1 \vee \varphi_2$ is similar to the one above except that $\delta(s_0, a) = \delta(s_0^1, a) \vee \delta(s_0^2, a)$.
- If $\varphi = E\phi$, first build an HAA A'_φ over the alphabet $\Sigma' = 2^{max(\varphi)}$, i.e. A'_φ regards the formulas in $max(\varphi)$ as atomic propositions. First build a Büchi word automaton, (Σ', S, R, s_0, F), for ϕ over the alphabet Σ', and translate this automaton to an HAA, $A'_\varphi = (\Sigma, D, S, \delta', s_0, (F, \emptyset)))$, as described in section 4.5. The HAA A'_φ accepts exactly all the Σ'-labelled tree models of φ. A_φ is obtained by adjusting A'_φ to accept the alphabet Σ. Intuitively, A_φ starts additional copies of the HAAs associated with formulas in $max(\varphi)$, ensuring that whenever A'_φ assumes that a formula in $max(\varphi)$ holds, it indeed holds, and whenever A'_φ assumes a formula does not hold, the negation of the formula holds. Formally, $A_\varphi = (\Sigma, D, S \cup \bigcup_i (S^i \cup \overline{S}^i), \delta, s_0, (F \cup \bigcup_i (G^i \cup \overline{G}^i), \bigcup_i (B^i \cup \overline{B}^i)))$, where δ is defined as follows: for states in $\bigcup_i (S^i \cup \overline{S}^i)$, it agrees with δ^i and $\overline{\delta}^i$ and for $s \in S$ and for all $a \in \Sigma$ we have

$$\delta(s, a) = \bigvee_{a' \in \Sigma'} (\delta'(s, a') \wedge \bigwedge_{\varphi_i \in a'} \delta^i(q_0^i, a) \wedge \bigwedge_{\varphi_i \notin a'} \overline{\delta}^i(\overline{q}_0^i, a))$$

The S_j sets in S are above all the sets S_i in $\bigcup_i (S^i \cup \overline{S}^i)$ in the partial order.

- If $\varphi = A\phi$ then construct and negate the HAA for $E\neg\phi$.

The correctness of this construction is shown in [Ber95].

Example : Consider the CTL* formula $\psi = AFG(EFp)$. The formula is of the form $A\phi$ hence we construct the HAA for $EGF(AG\neg p)$ and negate it to get the HAA, $A_{D,\psi}$ for all $D \subset \mathbb{N}$. The formula, $\varphi = AG\neg p$ is the only formula in $max(EGF(AG\neg p))$, and hence we need to construct the HAA for $A_{D,\varphi}$ and $\overline{A}_{D,\varphi}$ as follows:

$A_{D,\varphi} = (\{\{\neg p\},\{p\}\}, D, \{q_2\}, \delta, q_2, (\emptyset,\emptyset))$ with

q	$\delta(q,\emptyset,k)$	$\delta(q,\{p\},k)$
q_2	$\bigwedge_{c=0}^{k-1}(c,q_2)$	false

$\overline{A}_{D,\varphi} = (\{\{\neg p\},\{p\}\}, D, \{\overline{q}_2\}, \overline{\delta}, \overline{q}_2, (\emptyset,\emptyset))$ with

q	$\overline{\delta}(q,\emptyset,k)$	$\overline{\delta}(q,\{p\},k)$
$\overline{q}_2$	$\bigvee_{c=0}^{k-1}(c,\overline{q}_2)$	true

The next step is to create a Büchi automaton, $B_{EGF\varphi}$, for the linear time formula $EGF\varphi$ from which we can construct an HAA for $A'_{D,EGF\varphi}$.

$B_{EGF\varphi} = (\{\{\neg\varphi\},\{\varphi\}\}, \{q_0,q_1\}, R, q_0, \{q_1\})$ with

$$R(q_0,\{\neg\varphi\}) = R(q_1,\{\neg\varphi\}) = \{q_0\}$$

$$R(q_0,\{\varphi\}) = R(q_1,\{\varphi\}) = \{q_1\}$$

Thus, $A'_{D,EGF\varphi} = (\{\{\neg\varphi\},\{\varphi\}\}, D, \{q_0,q_1\}, \delta', q_0.(\{q_1\},\emptyset))$ with

q	$\delta'(q,\{\neg\varphi\},k)$	$\delta'(q,\{\varphi\},k)$
q_0	$\bigvee_{c=0}^{k-1}(c,q_0)$	$\bigvee_{c=0}^{k-1}(c,q_1)$
q_1	$\bigvee_{c=0}^{k-1}(c,q_0)$	$\bigvee_{c=0}^{k-1}(c,q_1)$

We can now compose the automata $A_{D,\varphi}$, $\overline{A}_{D,\varphi}$ and $A'_{D,EGF\varphi}$ into the one automaton over the alphabet $\{\{\neg p\},\{p\}\}$: $A_{D,EGF\varphi} = (\{\{\neg p\},\{p\}\}, D, \{q_0,q_1, q_2,\overline{q}_2\}, \delta, q_0, (\{q_1\},\emptyset))$ with

q	$\delta(q,\{\neg p\},k)$	$\delta(q,\{p\},k)$
q_0	$(\bigvee_{c=0}^{k-1}(c,q_0) \wedge \bigvee_{c=0}^{k-1}(c,\overline{q}_2)) \vee (\bigvee_{c=0}^{k-1}(c,q_1) \wedge \bigwedge_{c=0}^{k-1}(c,q_2))$	$\bigvee_{c=0}^{k-1}(c,q_0)$
q_1	$(\bigvee_{c=0}^{k-1}(c,q_0) \wedge \bigvee_{c=0}^{k-1}(c,\overline{q}_2)) \vee (\bigvee_{c=0}^{k-1}(c,q_1) \wedge \bigwedge_{c=0}^{k-1}(c,q_2))$	$\bigvee_{c=0}^{k-1}(c,q_0)$
q_2	$\bigwedge_{c=0}^{k-1}(c,q_2)$	false
$\overline{q}_2$	$\bigvee_{c=0}^{k-1}(c,\overline{q}_2)$	true

Note that for $\delta(q_0,\{p\},k)$ and $\delta(q_1,\{p\},k)$ the expressions are reduced from $(\bigvee_{c=0}^{k-1}(c,q_0) \wedge true) \vee (\bigvee_{c=0}^{k-1}(c,q_1) \wedge false))$. All that is required now is to negate this automaton to get $A_{D,\psi} = (\{\{\neg p\},\{p\}\}, D, \{\overline{q}_0,\overline{q_1},\overline{q}_2,q_2\}, \overline{\delta}, \overline{q}_0, (\emptyset,\{\overline{q}_1\}))$ with

q	$\delta(q,\{\neg p\},k)$	$\delta(q,\{p\},k)$
$\overline{q}_0$	$(\bigwedge_{c=0}^{k-1}(c,\overline{q}_0) \vee \bigwedge_{c=0}^{k-1}(c,q_2)) \wedge (\bigwedge_{c=0}^{k-1}(c,\overline{q}_1) \vee \bigvee_{c=0}^{k-1}(c,\overline{q}_2))$	$\bigwedge_{c=0}^{k-1}(c,\overline{q}_0)$
$\overline{q}_1$	$(\bigwedge_{c=0}^{k-1}(c,\overline{q}_0) \vee \bigwedge_{c=0}^{k-1}(c,q_2)) \wedge (\bigwedge_{c=0}^{k-1}(c,\overline{q}_1) \vee \bigvee_{c=0}^{k-1}(c,\overline{q}_2))$	$\bigwedge_{c=0}^{k-1}(c,\overline{q}_0)$
$\overline{q}_2$	$\bigvee_{c=0}^{k-1}(c,\overline{q}_2)$	true
q_2	$\bigwedge_{c=0}^{k-1}(c,q_2)$	false

Since $\overline{q}_1$ is in B and is part of a universal set (with $\overline{q}_0$), a run will only be accepting if the state $\overline{q}_1$ is seen only finitely often on every branch. This will happen if on every branch, from a certain state onwards, $\overline{q}_0$ is visited infinitely. From the transition function it can be seen that whenever a p is read in the input or when $\bigvee_{c=0}^{k-1}(c,\overline{q}_2)$ holds (which is equivalent to EFp being valid) a copy of the automaton visits $\overline{q}_0$. Therefore, a run is accepting if on every branch EFp holds from a certain position onwards, which is precisely what AFG(EFp) states.

4.7 *Linear*CTL* to HAA Translation

In the translation of both LTL (section 4.5) and CTL* (section 4.6) to HAA there is a possibly exponential increase in the size of the resulting automaton. This increase is due to the translation of the linear time fragments of the formulas to nondeterministic Büchi word automata. The translation of the linear time fragments to alternating Büchi word automata is however linear, and since

nondeterministic Büchi word automata is a subset of the alternating automata, it is possible to define a sublogic of CTL* for which a linear translation exists to HAA. We call this sublogic *Linear*CTL*.

Nondeterministic automata can only express existential choice and therefore if we consider the translation of LTL formulas to alternating Büchi word automata as given in Figure 4.1 it is clear that the $\wedge$-connectives, that allow universal choice, must be avoided in the transition function. From the translation rules in Figure 4.1 it can be seen that the following three rules can produce $\wedge$-connectives: $\psi_1 \wedge \psi_2$, $\psi_1\ U\ \psi_2$ and $\psi_1\ V\ \psi_2$. From the translation rules for these we can see the restricted form should be: $prop \wedge \psi$, $prop\ U\ \psi$ and $\psi\ V\ prop$, where *prop* is either an atomic proposition or its negation. The sublogic LinearCTL* can now be defined as follows:

$$
\begin{aligned}
\mathbf{S} &::= true \mid false \mid q \mid \neg q \mid \mathbf{S} \wedge \mathbf{S} \mid \mathbf{S} \vee \mathbf{S} \mid A\mathbf{P_A} \mid E\mathbf{P_E} \\
\mathbf{P_E} &::= \mathbf{S} \mid \mathbf{S} \wedge \mathbf{P_E} \mid \mathbf{P_E} \vee \mathbf{P_E} \mid X\mathbf{P_E} \mid \mathbf{S}\ U\ \mathbf{P_E} \mid \mathbf{P_E}\ V\ \mathbf{S} \\
\mathbf{P_A} &::= \mathbf{S} \mid \mathbf{P_A} \wedge \mathbf{P_A} \mid \mathbf{S} \vee \mathbf{P_A} \mid X\mathbf{P_A} \mid \mathbf{P_A}\ U\ \mathbf{S} \mid \mathbf{S}\ V\ \mathbf{P_A}
\end{aligned}
$$

It is interesting to note that LinearCTL* is a sublogic of LeftCTL* defined in [Sch97]. There the idea was to find sublogics of CTL* for which a translation exists to CTL. LeftCTL* is such a logic, but with a potential exponential blowup in the size of the resulting CTL formula. LinearCTL* is the sublogic of LeftCTL* for which this translation is linear. The difference between LinearCTL* and LeftCTL* is that the latter includes $\mathbf{P_E} \wedge \mathbf{P_E}$ and $\mathbf{P_A} \vee \mathbf{P_A}$ formulas, which indicates why the translation from LeftCTL* to CTL can be exponential.

4.8 Implementation Issues

We implemented a translator for translating CTL* formulas to HAA. The formula is translated into two formats of HAA that are used for different purposes:

- An internal data structure that is used for model checking.
- LaTeX source is generated for displaying the transition functions of the HAA in tabular form, to allow better analysis of the relationships between formulas and automata. This format is especially useful for debugging and optimisation of the translation algorithm as well as for generating examples that can be used in documentation.

The efficiency of the translation algorithm is dependent on the cost of the translation of linear time formulas to nondeterministic Büchi word automata, as required during the LTL and CTL* translations. The reason for this is that during this translation an exponential blow-up might occur. Note this will not happen when translating CTL formulas, since a linear translation to 1-HAA exists (see section 4.4).

We use an adaptation of the LTL to nondeterministic Büchi word automata algorithm of [GPVW95]. The original algorithm was designed to build the automaton on-the-fly during the model checking procedure, whereas our version builds the automaton beforehand. We argue, in accordance with the view of Lichtenstein and Pnueli [LP85], that temporal formulas for specifying properties to be model checked are generally not very long. Hence, even if there is a exponential size increase in the size of the resulting automaton, this automaton will seldom have more than a few hundred states. Building the automaton beforehand is therefore acceptable, but more importantly this allows us to reduce the size of the automaton by removing duplicate states. A state is considered to be duplicated by another state if both states have similarly labelled outgoing arcs and are either both accepting (in F) or both rejecting (not in F). A duplicate state is removed by replacing all its incoming arcs by arcs to the state it duplicates and then deleting it. Since, new duplicate states can be created by deleting others, this process is repeated until no more duplicates exist.

LTL Formula	Original	Optimised
$AFG(p \wedge Fq)$	6	3
$A(FGp \wedge GFq)$	11	3
$A((FFp \wedge G\neg p) \vee (Fp \wedge GG\neg p))$	22	2

Table 4.2: Comparing the number of states for two LTL to NBA translation algorithms.

In Table 4.2 the results of the original algorithm of [GPVW95] and the one optimised by removing duplicates are compared for three formulas. The first two formulas $AFG(p \wedge Fq)$ and $A(FGp \wedge GFq)$ are equivalent, but the unoptimised algorithm reports that the first one is more succinct, whereas after removing duplicate states the automata are identical (given in Figure 3.5). The third formula is taken from [GPVW95] and is the positive form of the unsatisfiable formula $A\neg(FFp \leftrightarrow Fp)$ (the form it is given in [GPVW95]).

4.9 Concluding Remarks

The translations from CTL, LTL and CTL* to HAA given in this chapter are adapted from [Ber95]. The observation that a sublogic of CTL* exists for which a linear translation exists to HAA is new (section 4.7).

In [Ber95] it is shown that the nonemptiness problem of the product automaton of a Kripke structure and the HAA for a CTL* formula reduce to the 1-letter nonemptiness problem and can be solved in time linear in the size of the product HAA. Furthermore, since CTL formulas translate linearly to HAA and the translation of CTL* formulas is exponential in the size of the formula, it follows that automata-theoretic model checking with HAA matches the known model checking complexity for CTL and CTL* model checking. In [Ber95] it is also shown that the 1-letter nonemptiness problem can be done in a space efficient fashion. Unfortunately, the algorithm proposed for time efficient

nonemptiness checking is incompatible with the one that allows space efficiency. This is because the time efficient algorithm is a global algorithm (product HAA must be kept in memory throughout the procedure) and the space efficient algorithm is a local algorithm (only the part of the HAA being explored is kept in memory).

In order to tackle large model checking examples an algorithm is required that is both time and space efficient. In the next chapter we consider the theory of 2-player games as a way of defining such an algorithm for the nonemptiness checking of HAA.

Chapter 5

Nonemptiness Games for HAA

Two-player games form a natural framework for the study of interactions. One player represents a *System* and the other represents the *Environment* of the System. A game can then be viewed as specifying the possible interactions between a System and its Environment. Two-player games are also often viewed as being played by a Player (System) and an Opponent (Environment). The Player is trying to win the game, whereas the Opponent, or spoiler, is trying to stop this from happening. Either Player or Opponent then has a *winning strategy* for a game if he/she can win any play of the game regardless of the other's moves.

Recently, much work has been done in the development of game semantics for programming languages [Abr97, AJ94, McC96]. Here the programs are modelled by the rules of how the System should play the game (strategies). Bisimulation equivalence can also be formulated as a two-player game, where the Player is trying to show that two systems are bisimilar and the Opponent is trying to show that they are not [Sti96, Sti97].

Emerson and Jutla were the first to use game-theory in combination with temporal logic [EJ88]. They used infinite Borel games to show that satisfiability checking for CTL* is in deterministic double exponential time. Stirling showed how *Ehrenfeucht-Fraïssé* games [Tho93] can be used to capture the expressive power of the extremal fixed point operators of the μ-calculus [Sti96]. To the best of our knowledge, Stirling was also the first to use two-player games for model checking [Sti95] when he reformulated the model checking problem for the μ-calculus as a two-player game.

Another application of game-theory is in the study of reactive systems, where the effective construction of a winning strategy for a game is an approach to the synthesis of reactive programs. The game represents a desired property of the system and by constructing a winning strategy for this game a reactive system that exhibits this property is developed. An automata-theoretic approach is taken in [PR89b, PR89a, Tho95, BLV95, Tho97] by considering the property to be checked to be an automaton on infinite words and the game to be the nonemptiness check of the language accepted by the automaton.

For synthesis the problem is to construct a reactive system that has a desired property. However for model checking we only need to check whether a given system exhibits a property. Here we will follow the games approach to synthesis for doing model checking, namely, to play a game to solve the nonemptiness problem for HAA. We refer to this game as the *nonemptiness game*.

We will show that formulating the nonemptiness problem for HAA as a game has two main advantages:

- The game is simple and can be played without prior knowledge of the automata-theoretic details.
- Although the game does not improve the worst-case complexity of the

nonemptiness check for HAA, it leads to a simple and efficient implementation for checking nonemptiness of an HAA.

In [SS98] it is also argued that the games approach to model checking allows better diagnostics when a property is invalid for a given system. Essentially the user plays a game against the winning strategy for showing the property is invalid. The information gained in this play can give the user a better understanding of why the property is invalid.

We first define the nonemptiness game and then show that an efficient implementation will require that winning positions (i.e. positions in the game from which a player wins a play) must be stored and later reused when these positions are visited in a different play. The playing of a *new game* from a position to establish the winning player from that position is introduced as a way to ensure that only "correct" winning positions are stored. We show pseudocode for the nonemptiness game algorithm and informally argue its correctness. In the last part of the chapter we analyse the complexity of the nonemptiness game and show how it relates to other work on CTL* model checking.

5.1 Nonemptiness Game

The nonemptiness game is defined as a two-player game, in which player 1 will try to show that the HAA is empty whilst player 2 will try to establish that it is nonempty. A *play* of the game is a possibly infinite sequence of *positions*[1] of the form $(q_0, s_0), (q_1, s_1), \ldots$ where each position (q_i, s_i) is a node in an And-Or tree (in our case, the product of the HAA for the formula and the Kripke structure). Which player makes the next move is determined by the structure of the And-Or tree: player 1 (Brandy) moves whenever there is an $\wedge$-choice and player 2 (Port) when there is an $\vee$-choice[2]. The players therefore do not

[1] Positions in the game setting are equivalent to states in the automata setting.
[2] The player names reflect when the player moves: Brandy and Port.

Player 1 (Brandy) Wins	Player 2 (Port) Wins
Play reaches a $false$	Play reaches a $true$
After a move by Port, that revisits a position in the current play, $infpos(play_\pi) \cap G = \emptyset$	After a move by Port, that revisits a position in the current play, $infpos(play_\pi) \cap G \neq \emptyset$
After a move by Brandy, that revisits a position in the current play, $infpos(play_\pi) \cap B \neq \emptyset$	After a move by Brandy, that revisits a position in the current play, $infpos(play_\pi) \cap B = \emptyset$

Figure 5.1: Winning Conditions for a Play in the Nonemptiness Game

take turns as is the case in many standard games. The winner of a play can be determined when either a node that is labelled $true$ (Port wins) or $false$ (Brandy wins) is found in the play or when a position in the current play is revisited.

When a position in a play is revisited it represents the scenario where there is an infinite path in the product HAA and therefore we need to consider the acceptance condition (G, B) to determine which player wins the play. Recall the acceptance condition for an infinite path, π, in an HAA:

1. if π gets trapped in an existential S_i and $inf(\pi) \cap G \neq \emptyset$
2. if π gets trapped in an universal S_i and $inf(\pi) \cap B = \emptyset$

Let us define a set of positions, $infpos(play_\pi)$, to be the positions in the current play, $play_\pi$, that are visited infinitely often on $play_\pi$. Existential S_i only contain disjunctively related elements, thus Port will be making the choices of which move to make in these S_i. Universal S_i, on the other hand, only contain conjunctively related elements, from which Brandy makes the next move. The definitions of existential and universal sets combined with 1 and 2 above are sufficient to define the winning conditions of a play, summarised in Figure 5.1.

Since the intentions of the players when making moves are backwards sound,

the following rule can be used to combine the results of plays: if Brandy (Port) moves from position s to s' in a play and s' is the start of a winning play for Brandy (Port), then Brandy (Port) also wins from s. A player has a *winning strategy* for a game if the player can win any play of the game from a position regardless of the opponent's moves.

Theorem 1 *Player 2 (Port) has a winning strategy in the nonemptiness game from the initial state of an HAA iff the language accepted by the HAA is nonempty.*

Proof: The correctness of the Theorem follows directly from the construction of the winning conditions of the game: each play in the game is essentially checking the acceptance of a (possibly) infinite path in the HAA. □

Example: If we play the nonemptiness game on the HAA, $K \times A_{D,AGEFp}$ from Figure 5.4, then it is clear that whenever Port has a move it can reach the position (k, q_1) which is a winning position for Port (the winning players are shown as labels on the positions in Figure 5.4). The only interesting plays are those that revisit (x, q_0) and (x, q_1), for example for the left-most play $(play_\pi)$, $infpos(play_\pi) = \{(x, q_0), (y, q_0), (z, q_0)\}$, but since B is empty Port wins this play. Port in fact wins every play, regardless of Brandy's moves, and from Theorem 1 it then follows that $K \times A_{D,AGEFp}$ is nonempty, and hence also that $K \models AGEFp$.

Theorem 2 *Given a Kripke structure K and a CTL* formula φ then $K \models \varphi$ iff Player 2 (Port) has a winning strategy for the nonemptiness game on $K \times A_{D,\varphi}$, where $A_{D,\varphi}$ is an HAA such that the language accepted by $A_{D,\varphi}$ is exactly the set of D-trees satisfying φ.*

Proof: Theorem 2 follows directly from Theorem 1 and captures the relationship between the model checking problem and the nonemptiness game. □

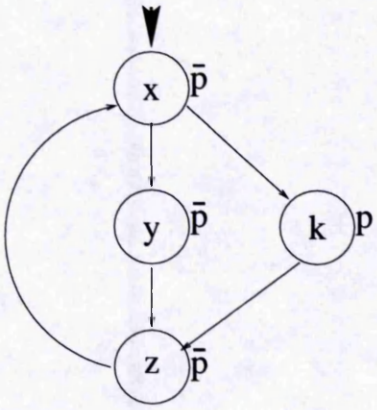

Figure 5.2: Kripke Structure $K = (\{\{\neg p\}, \{p\}\}, \{x, y, z, k\}, R, x, L)$

q	$\delta(q, \{\neg p\}, l)$	$\delta(q, \{p\}, l)$
q_0	$\bigvee_{c=0}^{l-1}(c, q_1) \wedge \bigwedge_{c=0}^{l-1}(c, q_0)$	$\bigwedge_{c=0}^{l-1}(c, q_0)$
q_1	$\bigvee_{c=0}^{l-1}(c, q_1)$	true

Figure 5.3: HAA $A_{D,AGEFp} = (\{\{\neg p\}, \{p\}\}, D, \{q_o, q_1\}, \delta, q_0, (\{\}, \{\}))$

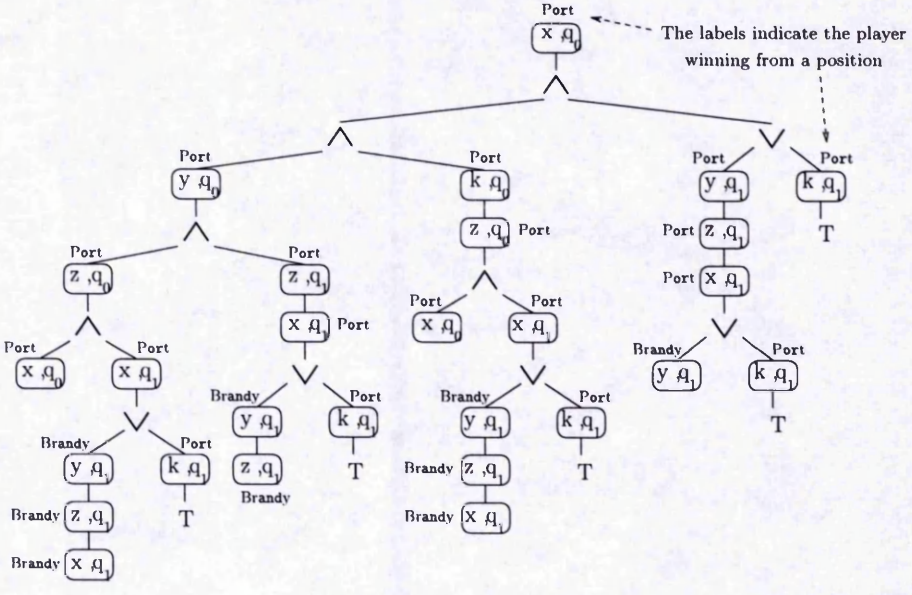

Figure 5.4: And-Or tree for the product automaton $K \times A_{D,AGEFp}$

From Theorem 2 we can now construct an efficient algorithm for doing CTL* model checking. The first part is to construct the HAA from the CTL* formula and then to play the nonemptiness game on the product of this HAA and the Kripke structure. The construction of the HAA from a CTL* formula is given in section 4.6. Next we will show how to implement the nonemptiness game in an efficient fashion.

5.2 Implementing the Nonemptiness Game

In the previous section it was shown that the moves of the nonemptiness game are determined by the structure of the And-Or tree. We have implemented a depth-first algorithm for finding winning plays in an And-Or tree. An efficient implementation of $infpos$ is obtained by keeping track of the positions in the current play on a stack data-structure, where a new position is pushed every time a move is made and popped whenever a winning play is found from the position. The elements in $infpos$ are therefore all the elements in the stack between the depth where a position is revisited and the current depth (value of the top of stack pointer). The stack is also used to keep track of the other possible moves from a position, but the moves themselves will only be made if the depth-first algorithm requires it later (i.e. after backtracking) in the search for a winning play. For example when looking for a winning play in the initial position of Figure 5.4 the left-hand choice at the $\wedge$-node is taken and the fact that there is a right-hand choice is recorded in the stack, but it is only explored when it is clear that the left-hand choice returns a winning play for Port. This approach is in general more memory efficient than a breadth-first algorithm where all the choices are explored simultaneously.

5.2.1 Storing Results

Unfortunately, although the algorithm outlined above is memory efficient, it is not time efficient. The reason for this is that "winning" games from a position can be replayed. Considering again the example of Figure 5.4, it is clear that in the play $(x, q_0), (y, q_0), (z, q_0), (x, q_1)$ there is a winning play for Port in position (x, q_1), but this position arises three more times in other plays and the fact that Port has a win from this position will be re-established each time. A *results* store is required to keep track of winning positions so that when they are revisited in different plays then the results can be reused. The problem is however to determine when to store the results, or to put it another way, when a potential winning position is stored to be certain that the winning position is indeed correct. One possibility is to store the winning position when all moves from a position have been played (i.e. when the depth-first algorithm backtracks). However, since a play is truncated whenever a position is revisited, it may happen that an incorrect result can be stored when all moves from a position have been made. The problem is that a winning play for a player may now be missed since that play may have been truncated at some point.

For example consider a nonemptiness game played in a left-most depth-first fashion on the And-Or tree of Figure 5.5 whilst storing winning positions in a results store. When considering the position (z, q_1) on the play $(x, q_0), (y, q_0), (z, q_0), (x, q_1), (y, q_1), (z, q_1), (x, q_1)$ this position, i.e. (z, q_1), will be recorded as a win for Brandy (since (x, q_1) is revisited). Clearly, however, from the position (x, q_1) the position (k, q_1) can be reached and this is a win for Port and hence (z, q_1) should be recorded as a win for Port. In Figure 5.5 it is indicated how recording (z, q_1) as a win for Brandy and then reusing this result causes the nonemptiness game on the And-Or tree of Figure 5.4 to be won by Brandy, which is incorrect.

A mechanism is required to ensure that when all moves from a position have

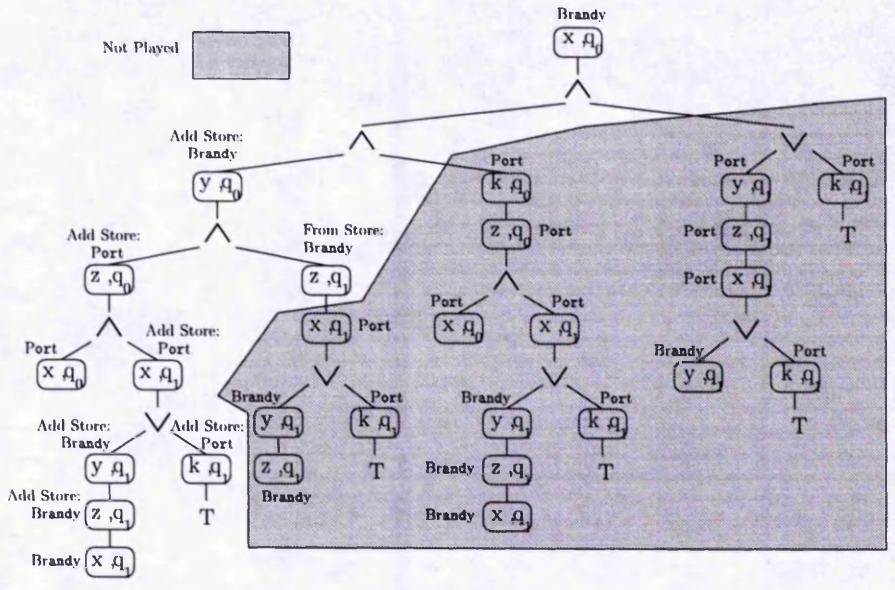

Figure 5.5: Incorrect Game with Results Store

been made that the player winning from this position is indeed correct.

5.2.2 New Games

In order to ensure that the results stored as winning positions are indeed correct, it is proposed that a *new* game is played whenever all the moves from a position, say s, have been played. This new game takes as its initial position s and a new stack and new results store are used. Since a new game uses a new stack and a new store, the intuition is that plays that were previously truncated will be played to completion in the new game, hence ensuring that the correct result is obtained for the initial position of the new game. When a new game is completed (i.e. the winning player from its initial position is found) the stack and results store for the new game are deleted and the result of the new game is stored in the original[3] results store. Whenever a position is visited in the new game it is first checked whether this position is not in the original store, since

[3]The store used for the initial game will be referred to as the original store and is never deleted.

if it is that result can be used. Note that when we refer to the *nonemptiness game* we refer to the initial (first) game together with all the new games that are played in order to determine the winning player for the initial position (of the first game).

New games may have to be played recursively, i.e. whilst playing a new game another new game can be started etc. Therefore, to ensure that new games will not be played infinitely, a new game is not allowed from a position from which a new game is already being played. In fact, when a new game is being played one can restrict the play of more new games from positions that are on the current play of any of the previous games (initial game and all new games currently being played). The reason for this restriction is that new games for positions that are on the current play of a previous game will be played later on (precisely, when the positions are backtracked in the previous game). This therefore has the effect of just postponing the new game. Note that the initial position of a new game is by definition also part of the current play of the previous game and therefore this restriction also ensures that new games cannot be played infinitely.

There is one important exception to this restriction of when to play new games: when a position is in the acceptance condition (G, B) and if no new game is currently being played for this position then a new game must be played from it. The reason for this exception is that positions in the acceptance condition influence the results of games on infinite plays and without playing new games from these positions infinite plays might be missed. In section 5.3 where the correctness of the nonemptiness games is discussed the need for this exception will be highlighted again.

Example: In Figure 5.6 it is shown how a new game is played when the only move from (z, q_1) has been played and the result of this new game is that (z, q_1) is a winning position for Port. Reusing this result when (z, q_1) is visited

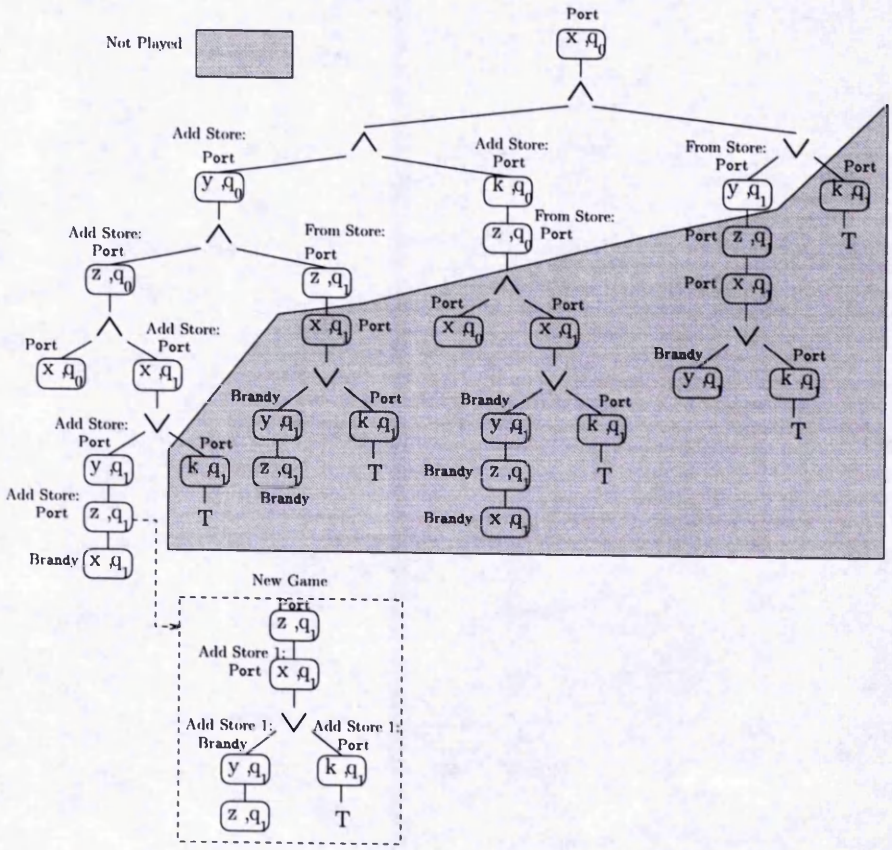

Figure 5.6: New Games combined with Results Store

again enables Port to win the game from the initial position which is the correct result. No further new games are played for the positions visited in the new game from position (z, q_1), since all these positions are on the current play of the initial game. New games are however played for the positions in the initial game, but in those cases the results are immediately found in the original results store (due to the depth-first nature of the plays).

5.2.3 Algorithm

An algorithm for playing the nonemptiness game is given in Figure 5.7. In order to illustrate how the algorithm works we first need to show that some moves made by the players relate only to the structure of the HAA for the formula, whereas other moves depend on the number of successor states from some state in the Kripke structure. For example, if we are in state (x, q_0) in Figure 5.4 then the next move depends on the following expression (from the transition table of Figure 5.3):

$$\bigvee_{c=0}^{k-1}(c, q_1) \wedge \bigwedge_{c=0}^{k-1}(c, q_0)$$

The $\wedge$ in the middle of this expression is a move for Brandy, and does not depend on the fact that state x has two successors ($k = 2$). Whereas the moves for $\bigvee_{c=0}^{k-1}(c, q_1)$ (Port moves) and $\bigwedge_{c=0}^{k-1}(c, q_0)$ (Brandy moves) are dependent on the number of successor states of the Kripke structure. We make a distinction between these two types of moves and they are handled by different parts of the algorithm: we call the former `AND` and `OR` moves and the ones dependent on the successor states `AND_SUC` and `OR_SUC` moves. Essentially this distinction allows the state of the Kripke structure to be hidden in the algorithm of Figure 5.7: we will only refer to `Alt_State` which is the state of the HAA for the formula being checked. `Alt_State` can take on 6 different values: `AND`, `OR`, `AND_SUC`, `OR_SUC`, `TRUE` and `FALSE`.

The algorithm consists of two functions `Game` and `Play` that both return a

```
1  Player Play(Alt_State s)
2  { while (!Finished) {
3    action = MoveInK(s);
4    if (action == no_more_moves) Finished = TRUE;
5    else {
6      switch (s) {
7      case AND_SUC:
8         switch (action) {
9           case new_position:
10            if ((result=Game(s,Expand))==Brandy) Finished = TRUE;
11            if NotPlayedBefore(s) {
12              PlayNextGame();
13              if ((result=Game(s,Expand))==Brandy) Finished = TRUE;
14              ReturnToPreviousGame(result);
15            }
16            s = BackTrack(result);
17            break;
18          case B_in_Infpos     : result = Brandy; Finished = TRUE; break;
19          case no_B_in_Infpos  : result = Port; break;
20          case Brandy_win      : result = Brandy; Finished = TRUE; break;
21          case Port_win        : result = Port; break;
22        }
23        break;
24     case OR_SUC:
25   // same as case AND_SUC, but with Port substituted for Brandy and G for B.
26        break;
27 } } }
28 return result;
29 }
30
31 Player Game(Alt_State s, int mode)
32 { if (mode == Play)
33     if (s == AND)
34       if (Game(s.left,Play) == Brandy)
35         return Brandy;
36       else
37         return Game(s.right,Play);
38     else if (s == OR)
39       if (Game(s.left,Play) == Port)
40         return Port;
41       else
42         return Game(s.right,Play);
43     else if (s == TRUE)  return Port;
44     else if (s == FALSE) return Brandy;
45     else return Play(s);
46   if (mode == Expand) {
47     s = CreateExpr(s);
48     return Game(s,Play);
49 } }
50
51 Initialise:
52   1. Generate HAA from formula with initial node of HAA being init.
53   2. if (Game(init,Expand) == Port) printf("Formula Satisfied");
55      else printf("Formula Invalid");
```

Figure 5.7: Algorithm for Nonemptiness Game.

winning player from a specific position. `Game`, when in the `Expand` mode, takes care of the lookups in the transition table (using the state of the HAA and the state of the Kripke structure as indexes) of the HAA for the formula to be checked. After an expression is expanded by `CreateExpr`, `Game` is called in the `Play` mode and plays the moves related to the structure of the HAA, i.e. for `AND` and `OR` (as defined above). It also checks whether Brandy or `Port` has a trivial win in the case when respectively a `FALSE` or `TRUE` was expanded in the transition table. However, when the next move requires the successor states of the current Kripke state to be evaluated then the `Play` function is called. Note that when `Play` is called only one of the two players will be making the moves. The function `MoveInK` makes a move by generating a successor state in the Kripke structure. If the position reached after this move has not been seen before in the game then `Game` is called in the `Expand` mode from this position.

Here follows a brief description of each function being called in `Game` and `Play`:

CreateExpr This function does the lookup in the transition table and expands the entry it finds to an And-Or expression or to `TRUE` or `FALSE`. Expressions are created in such a fashion that positions from lower S_i sets are to the left of the $\wedge$ or $\vee$. This, combined with the fact that the moves on the left sides of a $\wedge$ or $\vee$ choice are played first (see function `Game`), means that winning positions are first established for positions in lower S_i sets in the partial order. In terms of the CTL* formulas this has the effect of first establishing the truth-value of all the state-subformulas before the truth-value of the original formula.

MoveInK This function has several purposes, firstly it chooses a successor of the current position, it then checks whether this new position is on the current play, i.e. in the stack, if so it returns a value stating whether a position in B (or G) was found on this cycle, depending on whether it is a move

by Brandy or Port that causes the cycle to be closed (`B_in_Infpos,....`). If a cycle isn't closed by the new position a check is made in the results store to see if this position is winning for a player, if so either `Brandy_win` or `Port_win` is returned. Lastly, if the new position is neither on the stack nor in the results store, it is pushed onto the stack and the value `new_position` is returned. When all the successors have been visited from a position then `no_more_moves` is returned.

BackTrack Backtrack is called when a winning position for one of the players is found. Backtrack stores this result in the results store and also pops the entry from the stack and returns the state of the HAA so that successors of this (old) state can now be evaluated. The results store is implemented as a hash table for fast access. When Backtrack is called after a new game has finished the result is stored in the original results store (i.e. the one associated with the very first game). This ensures new games will only be played once for every position.

NotPlayedBefore Checks whether a game is already being played from the current position or whether the position is on the current play of any previous game and not in the sets G or B.

PlayNextGame Creates a new results store and new stack for the new game. Note, creating a new store and stack is simply done by incrementing a *Game* counter that would be used as part of the stack and store entries thus avoiding conflict with the entries from other games.

ReturnToPreviousGame Do the reverse of the previous function: clears the new results store and decrements the *Game* counter. Currently the value of the *Game* counter partitions the hash table (results store), thus making it easy to clear the table when a new game is finished. Note the new stack will be empty by definition, due to the `BackTrack` function.

5.3 Correctness

Since we only gave an informal description of our algorithm for playing the nonemptiness game, we will not formally proof its correctness. Instead we will give some informal arguments to illustrate why the use of the new games can guarantee that the correct player wins the nonemptiness game.

From Theorem 1 we know that the nonemptiness game without a results store will correctly determine the winning player from the initial position of an HAA. Clearly, one can generalise Theorem 1 so that it holds for a game started in any position, i.e. the nonemptiness game from position s will correctly determine the winning player from position s when no results are reused.

Let us call such a game that can determine a winning player from a position without reusing any results a *perfect game*. We can now define a *safe game* from position s as either being a perfect game or a game that can determine the winning player from s by only reusing the results from perfect games. The games we have looked at so far in this chapter are classified as follows. The game played on the tree of Figure 5.4 is a perfect game (hence also safe), since it didn't reuse any results (this result is direct from Theorem 1). The game played on the tree of Figure 5.5 is not safe, since the result that is reused in position (z, q_1) is not the result of a perfect game. The game played on the tree of Figure 5.6 is safe, since the new game from (z, q_1) is a perfect game.

In order to show that the nonemptiness game is correct, we need to show that the nonemptiness game is a safe game. Since we play a new game for every position in the initial game of the nonemptiness game, we only need to show that every new game played is a safe game.

Let us first consider the case without postponing new games if they are on the current play, in other words, during a new game further new games are played when a position is backtracked unless the position is one for which

we are already playing a new game (to avoid an infinite cycle of new games). The number of these new games is bounded since there is a finite number of positions in the And-Or tree representing the product of the finite state Kripke structure and finite state HAA for the formula. Since the number of new games is bounded it follows that one of them must be a perfect game and since a new game is played for every position it then follows that these games will either be perfect games or will, due to the depth-first nature in which the plays of the game are examined, be safe games since they reuse the results of the perfect games.

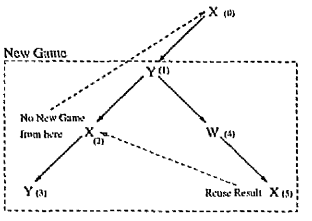

Figure 5.8: Unsafe new game.

Unfortunately, when we postpone new games, it can happen that during some new game a result from a position for which we are postponing a new game is reused. An example of this is shown in Figure 5.8 (with the depth-first order in which the positions are reached during the game shown in parentheses[4]), where a new game is played from position Y_1, but not from position X_2 since this position is on the current play of the previous game (shown as X_0). However on another (later) play from Y_1 the position X_5 is revisited and the result from the store (of the new game from Y_1) is reused. The new game from Y_1 is therefore not a safe game. (Note, the new game from Y_1 would have been safe if we had previously played a new game from X_2.) Let us refer to new games where results from positions from which we are postponing a new game are

[4]In the text we will show the depth-first order for the positions as subscripts. Note therefore that positions X_0, X_2 and X_5 all refer to the same position X with the subscript indicating when it is reached during the games (similarly Y_1 and Y_3).

reused as *semi-safe games*.

In order to show correctness of the nonemptiness game we therefore need to show that the result of a semi-safe game is correct. We will use Figure 5.8 to illustrate our arguments. Firstly, note that if only correct results are reused in the semi-safe game then of course the semi-safe game's result must be correct as well. Hence, the interesting case is when we assume an incorrect result is reused. Let us therefore assume the stored result for position X_2 is currently incorrect. This would mean there is some play from Y_1 that would give the correct result, but this play was not examined from X_2 since it got truncated when we found Y_3 on the current play. Let us assume this elusive play has a position W on it from which the correct result can be obtained. In Figure 5.8 the position W is shown on the same play as position X_5 for which the incorrect result is reused; this need not be the case since it could be on any play from Y_1 (if it is not on the same play, however, reusing the result from X_2 cannot influence the result from W and is therefore an uninteresting case). Note however it cannot be on a play "after" position X_5, since this would have meant, from the depth-first nature of the games, that we would have already examined this position on the left-hand play after X_2 and hence got the correct result for position X_2. Now let us assume reusing the result for position X_2 on a play from W causes the result for W to be incorrect. Note, only one play from W is sufficient since it is this play that causes the incorrect result to be obtained. If position W is not on the current play of any previous game, then a new game will be played from W. Since all new games cannot be semi-safe (again due to the And-Or Tree having a finite number of positions), we can assume the new game from W is a safe game. This then implies that the reuse of the incorrect result for position X_5 cannot influence the result of the new game from Y_1.

The last scenario to consider is the case where W is on a previous play and therefore the new game from W is postponed. It is not difficult to see that if a correct result can be obtained from W, but the only play from W

reaches position X_5, then the play we are missing is one that infinitely often visits position W. Furthermore, since we assumed from W we can obtain the correct result, but from X_2 (that is part of an infinite play) we cannot obtain the correct result, it means W must be in the acceptance condition (i.e. in set G or B). This is precisely why there is an exception to the rule of when to postpone a new game (see section 5.2.2). Since W is in G or B we therefore do not postpone a new game from W and this new game will obtain the correct result and hence reusing the incorrect result from X_2 cannot influence the result of the semi-safe game from Y_1.

Although we have only given an informal argument to support the correctness of our algorithm, we believe by using the simple notions of perfect, safe and semi-safe games the main points of interest are highlighted succinctly. Next we consider the complexity of the algorithm.

5.4 Complexity

Let the number of positions in the product HAA $K \times A_{D,\varphi}$ be n. Since a new game is required for each position and in the worst case each of the n positions needs to be visited for each new game the time complexity of the nonemptiness game is $O(n^2)$. However, due to the depth-first nature of the nonemptiness games and the fact that the original results store is checked for winning positions during a new game, new games tend only to traverse all n positions the first time the game is played and subsequently the results store provides the results. The time complexity of the nonemptiness game is therefore $O(cn)$, where $c \leq n$ and c is the number of new games that will not immediately find a winning position from the results store. In fact there is a relationship between c and the number of strongly connected components in the Kripke structure (and the product automaton therefore): for each SCC in $K \times A_{D,\varphi}$ a new game might be required that will visit at most all the positions in the SCC.

An example of this is the new game in Figure 5.6: the new game from (z, q_1) visits the positions (x, q_1), (y, q_1), (k, q_1) and (k, q_1) (which are all part of the same SCC), but the other new games only visit at most all their immediate successor positions.

The space complexity of the nonemptiness game is the amount of space required for the stack plus the space required for each of the games' results stores. The space requirement for one results store is linear in the number of positions n, and since we can reuse the space when a game is finished the space complexity is $O(k + n)$, where k is the space required for the stack (in general $k \ll n$)

5.5 Related Work

The version of the algorithm presented in Figure 5.7 that does not store winning positions (removing lines 11-15 and 20-21) is the game theoretic counterpart of the algorithm for nonemptiness checking of HAA described in [Ber95]. There it is shown that this algorithm is space efficient, $O(m \times log^2 n)$ (where m is the number of S_i sets in the HAA), but it is not time efficient. In fact, empirical results from our system and others (notably Holzmann's experience with the SPIN system) indicate the exponential time complexity of this algorithm is realised on even relatively small examples. In our case we checked whether it is possible to reach the initial state from any state (AGEF init) of a hand-coded model of the Address Interface of the AMULET asynchronous micro processor (described in [VBF+97] and section 7.3.1). This model contains only 32 unique states[5], but without storing winning positions 2402 positions were visited during the nonemptiness game (as opposed to 59 when a results store was used). Also in [Ber95] it is shown that the nonemptiness check for HAA

[5]The version of the address interface in section 7.3.1 is more complex (274 states) than the hand-coded version in [VBF+97] (32 states).

can be done in linear time ($O(n)$), but it requires the complete HAA to be kept in memory. The nonemptiness game, on the other hand, can label positions as winning without traversing the complete HAA.

The CTL* model checker of Bhat et al. [BCG95] is different from the one described here in two fundamental aspects: they traverse the state space based on the syntactic structure of the CTL* formula to be checked, and secondly, they construct SCCs (strongly connected components) in the state graph (product of the formula and Kripke structure) to determine the validity of formulas. As was pointed out in section 3.6.2 the automata approach does not suffer from the disadvantage that equivalent formulas might have model checking runs of different size. Furthermore, the automaton translated from a formula can in some cases be more succinct, for example, the formula $E(Gp \vee pUq)$ can be translated to an HAA with a single state[6]. Building SCCs in memory during model checking has the disadvantage that for certain systems the whole state space might be in an SCC and in these cases the algorithm will suffer from the state explosion problem. The nonemptiness game does not suffer from this, since instead of building SCCs, a new game is played to determine a winning position for a player. For example in Figure 5.4 a new game is played to determine whether (z, q_1) is a winning position for Port instead of building the SCC $(z, q_1), (x, q_1), (y, q_1), (k, q_1)$ to determine that (z, q_1) can reach **true** and is therefore a winning position for Port.

The nested depth-first search, described in section 3.5.3, used for doing nonemptiness checking of Büchi automata can be seen as a specialisation of the nonemptiness game described here. Whereas in the nonemptiness game a new game is played when backtracking from any position, in the nested depth-first search a second search is only started from an accepting state (state in the set F). Changing line 11 of the algorithm in Figure 5.7 so as to restrict the play

[6]The HAA contains the states **true** and **false**, but these are not counted since in the nonemptiness game they are trivial winning positions for Port and Brandy respectively and hence will not be expanded in the game.

of a new game to only states either in G or B will therefore be precisely the nonemptiness game equivalent of the nested depth-first search.

The work of Stevens and Stirling [SS98] is the most closely related to the algorithm described here. They use two-player games as the basis for a model checking algorithm for the μ-calculus. Since the μ-calculus is more expressive than CTL* their algorithm is more complex than ours. Specifically, they require *indexes* to keep track of the unwinding of fixed point variables, which is not required here. Another key difference is that instead of only recording winning positions that are definitely valid, they use *decisions* that represent a winning move for a player from a position under some *assumptions*. If there are no assumptions for a specific decision then the move will definitely lead to a win for the specific player and can always be taken. This is therefore equivalent to our results store. If however a decision is based on some assumptions that might not be valid, the game needs to be replayed from that position to ensure a win for the right player. In fact, a stack of possible decisions is kept at every position in a play, with the decision at the top of the stack being the "best" decision. The reason for the stack of decisions is that during a play it might occur that the assumption on which the best decision is based might be found to be invalid in which case the decision need to be discarded and the second best decision will be promoted. This approach is less memory efficient than ours, but can be more time efficient since not all the results stored in our game will be reused.

It is worth noting that the work presented in this thesis was developed completely independently and concurrently with that in [SS98].

5.6 Concluding Remarks

In the current implementation of our model checker, the system produces an error-trail to indicate to the user why the property failed for the reactive system under investigation. An error-trail is nothing more than a winning play for Brandy and is as such easy to produce. In [SS98], however, it is conjectured that an error-trail is not as useful to the user, since the user might rather want to have some indication why a specific play (execution sequence) is not a winning play for player. Clearly both these options could be useful to a user trying to understand the behaviour of a system. In [SS98] the user can play a game (and lose!) against the winning strategy for the model checking game, in order to get better insight into why the property holds, or otherwise. Since, in the algorithm presented here, a winning strategy for a player is available from the results in the results store, it would be a simple extension to allow a similar approach.

We have already shown that the nonemptiness game for nondeterministic Büchi automata is a specialisation of the nonemptiness game for HAA, and as such indicates how LTL model checking can be done more efficiently. In the next chapter we consider restrictions to allow efficient CTL model checking as well. Moreover, it is shown that the structure of the HAA for the CTL* formula can be used to determine the type of restriction that is required.

Chapter 6

Optimised Nonemptiness Games

Here we investigate different forms of the nonemptiness game based on different forms of HAA. Specifically, the structure of the HAA when dealing with CTL and LTL formulas will be analysed to see how more efficient nonemptiness games can be played for these automata.

It will be argued that the number of new games required during the nonemptiness game is a measure for comparing the complexity of CTL and LTL model checking in our games based automata-theoretic setting. Of course it is well-known that the model checking complexity of LTL is higher than that for CTL in general, since both are linear in the size of the Kripke structure but LTL is exponential in the size of the formula, whereas CTL is linear. These complexity results were however achieved in different settings: LTL in a *local* and *automata* setting and CTL in a *global* and *structural* setting. Here we compare them using the nonemptiness game setting, which is *local* and *automata* based.

An efficient nonemptiness game for HAA will need to exploit any special

structure of the HAA. In order to achieve this we show that it possible to determine from the structure of an HAA whether the formula that it was translated from is a CTL, LTL, both CTL and LTL, or strictly a CTL* formula.

6.1 LTL Nonemptiness Games

Here we are interested in CTL* formulas of the form Aφ and Eφ where φ contains no state-subformulas (call them linear time formulas).

From section 4.5 we know that the HAA obtained from formulas of the form Eφ only contain $\vee$-choices and those for Aφ only contains $\wedge$-choices. The nonemptiness games for linear time formulas can therefore be considered to be *boring* games for one of the two players, since either all the moves will be made by Port (for Eφ formulas) or by Brandy (for Aφ formulas). Furthermore, when Port makes all the moves the acceptance condition is $(G, \emptyset)$ and when Brandy makes all the moves it is $(\emptyset, B)$ (section 4.5). Therefore from the winning conditions in the nonemptiness game given in Figure 5.1, if Port (Brandy) moves and G (B) is empty then any position on the current play that is revisited means a win for Brandy (Port).

q	$\delta(q, \{\neg p\}, k)$	$\delta(q, \{p\}, k)$
q_0	$\bigwedge_{c=0}^{k-1}(c, q_1)$	$\bigwedge_{c=0}^{k-1}(c, q_0)$
q_1	$\bigwedge_{c=0}^{k-1}(c, q_1)$	$\bigwedge_{c=0}^{k-1}(c, q_0)$

Figure 6.1: $A_{D,AFGp} = (\{\{p\}, \{\neg p\}\}, D, \{q_0, q_1\}, \delta, q_0, (\{\}, \{q_1\}))$

However, if G (or B) is not empty then the fact that the S_i set is not necessarily singleton means some of the positions in the set can now be in G (B) and other positions not in G (B). For example the HAA translated from the formula AFGp (Figure 6.1) has an S_i set containing two states (q_0 and q_1) of which only one (q_1) is in the B set. When a nonemptiness game, without

any new games, are played for an automaton of the above form, the use of a results store might cause a cycle through a position in G (B) to be missed and consequently a winning position can be labelled incorrectly. A new game is thus required to find a cycle through positions that are either in B or G. This, combined with the fact that only one player moves in a game and that cycles in a play, where no position is in G (B) are trivially labelled, allows us to make the following optimised rule for linear time nonemptiness games:

- During the nonemptiness game for $K \times A_{D,\varphi}$ where φ is a linear time formula, new games need only be played from positions that are either in G or B.

This is the same rule that is used in the nested depth-first search used in the SPIN model checker [HPY96] (see also section 3.5.3). Here we have presented a justification for this rule in the setting of the nonemptiness game. Note that an LTL nonemptiness game is a semi-safe game (section 5.3), since incorrect results can be reused from positions that are not in G or B (since no new games are played for these positions).

6.2 CTL Nonemptiness Games

In section 4.4 it was shown that CTL formulas can be translated to 1-HAA (HAA with singleton S_i sets). Hence, unlike in the linear time case, the introduction of a results store cannot cause a position in G (B) to be labelled incorrectly as a win for the wrong player. Intuitively, when a cycle in a play is found during a CTL nonemptiness game if all the positions in $infpos$ are in G then Port wins the play (vice versa for B and Brandy); if one of these positions in $infpos$ is revisited in a later play then the result in the store can be reused since the new play will also have a cycle through the positions in $infpos$ (this is not necessarily true in the linear time case). This would seem to indicate

must be played for all positions visited during the nonemptiness game.

In the CTL nonemptiness game it is unnecessary to play new games for positions in the initial S_i set. The reason is that if we consider the positions in lower S_i sets already to be labelled then the boolean transition function for the states in the initial set reduces to only referring to positions from itself. Therefore, it can be considered to be a one-player set and no new games are required for these positions regardless whether the positions are in G or B. For example in the 1-HAA for $AGEFp$, positions with a q_0 component are in the initial S_{q_0} set and therefore no new games will be played for these positions.

CTL nonemptiness games are safe games (section 5.3), since the results from positions for which no new games are played cannot be incorrect (as is possible in the LTL case) and can therefore be safely reused.

6.3 LTL vs. CTL Nonemptiness Games

q	$\delta(q,\{\neg p,\neg q\},k)$	$\delta(q,\{p\},k)$	$\delta(q,\{q\},k)$	$\delta(q,\{p,q\},k)$
q_0	$\bigwedge_{c=0}^{k-1}(c,q_1)$	$\bigwedge_{c=0}^{k-1}(c,q_0)$	true	true
q_1	$\bigwedge_{c=0}^{k-1}(c,q_1)$	$\bigwedge_{c=0}^{k-1}(c,q_1)$	true	true

Figure 6.3: $A_{D,A(Gp\vee Fq)} = (2^{\{p,q\}}, D, \{q_0,q_1\}, \delta, q_0, (\{\}, \{q_1\}))$

Although in the LTL case all states in the HAA can be considered part of the same S_i set, splitting this set into singular sets where possible can allow more efficient nonemptiness games. In the CTL case we saw that for 1-HAA, when positions from one of the S_i sets are in G or B then no new games are required for these positions. Therefore, in the LTL case if the S_i set of the HAA is divided into smaller S_i sets and it is found that all the positions in G (B) are in singleton S_i sets then no new games are required. For example, consider the HAA for the formula $A(Gp \vee Fq)$ given in Figure 6.3. If we consider both

q	$\delta(q, \{\neg p\}, k)$	$\delta(q, \{p\}, k)$
q_0	$\bigvee_{c=0}^{k-1}(c, q_1) \wedge \bigwedge_{c=0}^{k-1}(c, q_0)$	$\bigwedge_{c=0}^{k-1}(c, q_0)$
q_1	$\bigvee_{c=0}^{k-1}(c, q_1)$	true

Figure 6.2: HAA $A_{D,AGEFp} = (\{\{\neg p\}, \{p\}\}, D, \{q_o, q_1\}, \delta, q_0, (\{\}, \{\}))$

that new games need only be played from positions that are neither in G nor B. This rule is however too weak.

In the linear time case we have seen that all the moves in a game are made by the same player. Let us now define S_i sets with all the transitions either consisting of only $\vee$-choices or only $\wedge$-choices to be *one-player* sets. Similarly, a set with transitions consisting of both $\vee$-choices and $\wedge$-choices is called a *two-player* set. In the linear time nonemptiness game all the S_i sets are therefore one-player sets. In the 1-HAA for the CTL formula $AGEFp$, given in Figure 6.2, the set S_{q_0} (i.e. S_i set containing q_0) is a two-player set, whereas the set S_{q_1} (i.e. S_i set containing q_1) is a one-player set. From the linear time case we know that for a one-player set no new games are required if the positions in the set are not in G nor B. This leads to the following stronger rule:

- During the nonemptiness game for $K \times A_{D,\varphi}$ where φ is a CTL formula, new games need only be played from positions that are neither in G nor in B but are referred to in the transition function of a two-player set.

In fact this rule can further be strengthened by observing that positions in transient S_i sets have trivial new games since all these positions lead to positions in lower S_i (in the partial order on the S_i sets) that would, because of the depth-first nature of the games, already be labelled as winning for one of the players. Therefore, positions in transient sets don't require new games (note that positions in transient sets cannot be in G or B). Again considering the 1-HAA for $AGEFp$ (Figure 6.2), both S_i sets of the automaton are referred to in the transition function of the two-player set S_{q_0} and therefore new games

must be played for all positions visited during the nonemptiness game.

In the CTL nonemptiness game it is unnecessary to play new games for positions in the initial S_i set. The reason is that if we consider the positions in lower S_i sets already to be labelled then the boolean transition function for the states in the initial set reduces to only referring to positions from itself. Therefore, it can be considered to be a one-player set and no new games are required for these positions regardless whether the positions are in G or B. For example in the 1-HAA for $AGEFp$, positions with a q_0 component are in the initial S_{q_0} set and therefore no new games will be played for these positions.

CTL nonemptiness games are safe games (section 5.3), since the results from positions for which no new games are played cannot be incorrect (as is possible in the LTL case) and can therefore be safely reused.

6.3 LTL vs. CTL Nonemptiness Games

q	$\delta(q,\{\neg p,\neg q\},k)$	$\delta(q,\{p\},k)$	$\delta(q,\{q\},k)$	$\delta(q,\{p,q\},k)$
q_0	$\bigwedge_{c=0}^{k-1}(c,q_1)$	$\bigwedge_{c=0}^{k-1}(c,q_0)$	true	true
q_1	$\bigwedge_{c=0}^{k-1}(c,q_1)$	$\bigwedge_{c=0}^{k-1}(c,q_1)$	true	true

Figure 6.3: $A_{D,A(Gp\vee Fq)} = (2^{\{p,q\}}, D, \{q_0,q_1\}, \delta, q_0, (\{\}, \{q_1\}))$

Although in the LTL case all states in the HAA can be considered part of the same S_i set, splitting this set into singular sets where possible can allow more efficient nonemptiness games. In the CTL case we saw that for 1-HAA, when positions from one of the S_i sets are in G or B then no new games are required for these positions. Therefore, in the LTL case if the S_i set of the HAA is divided into smaller S_i sets and it is found that all the positions in G (B) are in singleton S_i sets then no new games are required. For example, consider the HAA for the formula $A(Gp \vee Fq)$ given in Figure 6.3. If we consider both

states of the HAA to be in the same S_i set then new games must be played for positions with a q_1 component since these positions are in B. Whereas if we consider the HAA to have two S_i sets then it is clear that no new games are required.

LTL	CTL
1 For all states in HAA, set *NewGame* flag to false.	1 For all states in HAA, set *NewGame* flag to false.
2 Construct *minimal* S_i sets.	2 S_i sets are singleton (i.e. *minimal*).
3 For all states in G (B) and not in a singleton set, set *NewGame* true.	3 For all states not in G or B and referred to in the transition function of a two-player set, set *NewGame* true.
	4 Set *NewGame* for initial state to false.

Table 6.1: New game rules in nonemptiness Game for CTL and LTL

The different rules for determining when to play new games in the CTL and LTL environments are summarised in Table 6.1. First, the HAA for the CTL or LTL formula is built and then analysed according to the rules in Table 6.1. After completing the analysis each state in the HAA will have its *NewGame* flag either set to true or false. When the product with the Kripke structure is taken the product state's *NewGame* flag will be copied from the states of the HAA for the formula. New games are only played when a position has its *NewGame* flag set to true, hence line 11 of the algorithm in Figure 5.7

```
if NotPlayedBefore(s) {
```

now becomes

```
if (NotPlayedBefore(s) & (s->NewGame)) {
```

In the next section we compare the complexity of the CTL and LTL nonemptiness games.

6.3.1 Practical Considerations: LTL vs. CTL Nonemptiness Games

In section 5.4 we showed the complexity of the general nonemptiness game is dependent on the number of SCCs in the Kripke structure to be checked. This result is however based on the assumption that we play new games for all positions in the game. In the previous section we have shown that when considering HAA translated from LTL and CTL formulas then we can restrict the number of positions from which new games are required. If we therefore consider the Kripke structure to be fixed, we can compare the complexity of LTL and CTL nonemptiness games by considering the number of new games required in each case (i.e. the number of states in the HAA translated from the formula with its *NewGame* flag set to true).

In the translation of CTL formulas to 1-HAA a new state (and S_i set) is created for every state-subformula in the CTL formula, hence the number of AV, EU, AW and ER subformulas in a formula is an upper bound on the number of positions with their *NewGame* flag set to true. These subformulas translate to states in the 1-HAA that are neither in G nor B. Whereas in the LTL case new games are only required when an S_i set exists with more than one state and at least one in either G or B. We have already shown the relationship between LTL games and the nested depth-first search used for nonemptiness checking of Büchi automata; in [GH93] it is shown that for the nested depth-first search it is only necessary to distinguish between the states visited in the first search and the second (nested) search and the states need not be deleted after completing the nested search. If we recast this result in the LTL nonemptiness game it shows that only one new game will be required and all positions in the product HAA will at most be visited twice (once in the initial game and once in the new game). In the CTL nonemptiness game the number of new games depends on the formula being checked and can therefore not be bounded in this fashion.

Therefore, if we assume the HAA for CTL and LTL are of similar size, the LTL nonemptiness games are in general more efficient than the CTL games, since fewer new games are required.

In the next section we consider how this result influences the model checking complexity of CTL and LTL in our nonemptiness game setting.

6.3.2 Practical Considerations: LTL vs. CTL Model Checking

In [LP85] it is argued that when analysing the complexity of model checking the size of the Kripke structure is more important than the size of the input formula, since the size of the input formula is much smaller than the Kripke structure in practice. This argument supports our claim that the complexity of the nonemptiness game, which is based on the size and structure of the product automaton $K \times A_{D,\varphi}$, can also be used to determine model checking complexity. Previously the exponential blow-up in the size of $A_{D,\varphi}$ for an LTL formula φ, has led to the belief that LTL model checking is more complex than CTL model checking. However, in our experience the exponential blow-up seldom occurs in practice. Furthermore, even when the blow-up occurs, it tends to be small, since the LTL formulas used to specify properties tend to be small (seldom more than 5 path operators).

Therefore, if we argue that the size of the HAA for the formula, $A_{D,\varphi}$, will not greatly influence the size of the product automaton, $K \times A_{D,\varphi}$, then the complexity of the nonemptiness game can be considered also as the model checking complexity. From the previous section we know that if we fix the Kripke structure then LTL nonemptiness checking is in general easier than CTL nonemptiness checking, and hence, LTL model checking is in general easier than CTL model checking in our nonemptiness game setting.

One might argue that this result is not significant, since the nonemptiness

game is a very specific setting and cannot reflect on the general result that LTL model checking is more complex than CTL model checking. However, it is similar to the algorithm (nested depth-first search) used in the best-known LTL model checker, SPIN. Furthermore, the model checking complexity for CTL is achieved in a setting where the complete Kripke structure must be kept in memory throughout the procedure, which is often impossible due to the state-explosion problem. In the CTL nonemptiness game the Kripke structure can be built on-the-fly and in some cases only part of it needs to be explored to determine a winning position.

6.4 CTL* Nonemptiness Games

The rules for CTL and LTL nonemptiness games can be applied by first analysing the syntax of the formula to be checked to see whether it is a CTL or LTL formula. If however, after analysing the syntax, the formula is found to be neither CTL nor LTL, it must be a CTL* formula and the (unoptimised) nonemptiness game of section 5.2.3 must be played. However, a more efficient approach would be to apply the CTL and LTL rules where applicable. For example the CTL* formula $AFGp \wedge AGEFq$ can be checked by applying CTL rules to the states of the HAA for $AGEFq$ and LTL rules for the states of the HAA for $AFGp$. In order to achieve this, rules for the nonemptiness game must be based on the structure of the HAA and not on the syntax of the formula to be checked.

From the rules for CTL and LTL games it is clear that the characteristics of the S_i sets of the HAA can be exploited during the games. The following three characteristics of each S_i set have been used so far:

one-player: Is it a one-player or two-player set?

singleton : Is it a singleton set?

accepting : Is there at least one state from the set in G or B?

It is easy to see that these three features can be used to uniquely classify all possible S_i sets of an HAA that is translated from a CTL* formula. In the CTL case only *singleton* was used, whereas for LTL the definitive feature was *one-player*.

	one-player	accepting	singleton	Action
1	yes	no	yes	CTL and LTL: no new games
2	yes	no	no	LTL: no new games
3	yes	yes	yes	CTL and LTL: no new games
4	yes	yes	no	LTL (rule 3): new games
5	no	no	yes	CTL (rule 3): new games
6	no	no	no	Not covered
7	no	yes	yes	CTL: no new games
8	no	yes	no	Not covered

Table 6.2: Classification of S_i sets for CTL* games.

In Table 6.2 all 8 possibilities for the S_i sets of an HAA for a CTL* formula are listed together with the rule (CTL and/or LTL) that applies and whether new games are required. Interestingly, only two possibilities are not already covered by the rules for CTL and LTL. Both of these are when a two-player set is encountered, but the set is not singleton. Let us assume that we can use the following translation procedure to obtain an HAA for a CTL* formula:

1. If the CTL* formula φ that is translated to an HAA is syntactically a CTL formula then the linear translation of section 4.4 is used.

2. If φ is syntactically an LTL formula then the translation via a nondeterministic Büchi automaton is used (section 4.5).

3. If φ is a CTL* formula (i.e. neither LTL or CTL) then the translation of section 4.6 is used, and for any subformulas that are either CTL or LTL the above rules (1,2) are used.

In fact, this translation procedure is the one used within the implementation of our CTL* model checker. Furthermore, it allows the classification of the two outstanding cases in Table 6.2. From the translation procedure it follows that neither of the (outstanding) cases can be obtained from an LTL formula, since this would mean the S_i set must be a one-player set. Neither can the S_i set be from a CTL formula, since this would mean it must be a singleton set. Hence it can only be translated from a strictly CTL* formula. Of course, neither can it be a transient set, since transient sets are always singleton. This leaves only two possibilities: either it is translated from a (non-CTL) formula nested within a CTL formula, or, from a (non-LTL) formula nested within an LTL formula. But it cannot be from a formula nested within a CTL formula since this would mean the set would be a singleton set (rules 3 and 1). Hence the only remaining option is for the two cases to be related to a (non-LTL) formula nested within an LTL formula. The HAA for the formula AFG(EFp) given in section 4.6 is an example where the S_i set containing the states $\overline{q}_0$ and $\overline{q}_1$ is classified by case 8.

If the nested formula was a proposition then the S_i set would have been a one-player set (since it would have been translated from an LTL formula). Therefore, if during the game for a position in a set of case 6 or 8 the positions of the lower S_i set are labelled first then the set would become a one-player set. If we therefore make the rule that whenever a choice exists for a player's next move he/she always chooses a position from a lower S_i set when possible then case 6 would reduce to case 2 and case 8 to case 4. A similar argument is used for playing no new games for the positions in the initial set of an HAA for CTL formulas in section 6.2. Note that this rule is implemented in the algorithm for playing the nonemptiness game given in section 5.2.3 (see description of the CreateExpr function).

In Table 6.3 the rules for setting the value of the *NewGame* flag for the states of the HAA for a CTL* formula are given. Implementing the rules is

	CTL*
1	For all states in HAA set *NewGame* flag to false.
2	Construct minimal S_i sets.
3	For all states in G (B) and not in a singleton set, set *NewGame* true.
4	For all states not in G or B and referred to in the transition function of a two-player set, set *NewGame* true.
5	If the initial state is in a singleton set, set *NewGame* false.

Table 6.3: New game rules for CTL* nonemptiness games.

straightforward, since it is easy to determine whether an S_i set is singleton and whether it is a one-player set. At most two passes through all the states in the HAA are required: one to label the S_i sets as (not) singleton and (not) one-player, and another pass to do the analysis for rules 3 and 4. Rule 5 is an additional rule from the CTL case, where the initial set does not require new games, and is a simple test after completing the analysis for rule 3 and 4.

6.5 Classifying CTL* Formulas

An interesting question to be asked of a CTL* formula φ is whether there exists an equivalent CTL and/or LTL formula for φ. This question was addressed in [CD88] where a characterisation of those CTL* formulas that can be expressed in LTL was given. In the CTL case, however, they only showed when a CTL* formula cannot be expressed as a CTL formula. In both the LTL and CTL case, in order to characterise a formula, a Kripke structure must be provided by the user that will show certain properties of the formula, which therefore implies that their characterisation cannot easily be automated.

To the best of our knowledge, the complete characterisation of CTL formulas is still an open problem. Here we show that CTL formulas can be characterised by using 1-HAA.

6.5.1 CTL* Formulas Expressible as CTL

q	$\delta(q,\{\neg p\},k)$	$\delta(q,\{p\},k)$
q_0	$\bigwedge_{c=0}^{k-1}(c,q_0) \wedge \bigwedge_{c=0}^{k-1}(c,q_1)$	$\bigwedge_{c=0}^{k-1}(c,q_0)$
q_1	$\bigwedge_{c=0}^{k-1}(c,q_1)$	true

Figure 6.4: $A_{AGFp} = (\{\{p\},\{\neg p\}\}, D, \{q_0,q_1\}, \delta, q_0, (\{\}, \{q_1\}))$

In order to show how we will characterise CTL formulas we begin with a simple example of an LTL formula that has an equivalent CTL formula. In Figure 6.4 the HAA for the LTL formula AGFp is given. Note that this is a 1-HAA with two S_i sets: S_{q_0} (containing only state q_0) and S_{q_1} (containing only state q_1) and in the partial order we have $S_{q_1} \leq S_{q_0}$. If we consider the state q_1 first and expand the boolean expressions in the transition table we get:

$$q_1 \equiv (\neg p \wedge \bigwedge_{c=0}^{k-1}(c,q_1)) \vee (p \wedge true)$$

that can be reduced to:

$$q_1 \equiv p \vee \bigwedge_{c=0}^{k-1}(c,q_1)$$

If we consider the translation rules for translating CTL formulas to 1-HAA (section 4.4) then this is in the form (q_1 is in the B set):

$$\delta(AF\psi,a,k) \equiv \delta(\psi,a,k) \vee \bigwedge_{c=0}^{k-1}(c,AF\psi)$$

where $\psi = p$. Therefore we have $q_1 \equiv AFp$. Now consider the expression for q_0:

$$q_0 \equiv (\neg p \wedge \bigwedge_{c=0}^{k-1}(c,q_0) \wedge \bigwedge_{c=0}^{k-1}(c,q_1)) \vee (p \wedge \bigwedge_{c=0}^{k-1}(c,q_0))$$

that can be reduced to:

$$q_0 \equiv (p \vee \bigwedge_{c=0}^{k-1}(c,q_1)) \wedge \bigwedge_{c=0}^{k-1}(c,q_0)$$

However we know that $AFp \equiv (p \vee \bigwedge_{c=0}^{k-1}(c, q_1))$ and therefore we have:

$$q_0 \equiv AFp \wedge \bigwedge_{c=0}^{k-1}(c, q_0)$$

If we again consider the translation rules for CTL to 1-HAA (q_0 is not in B or G) we get $q_0 \equiv AGAFp$ Therefore, if we translated the CTL formula AGAFp to a 1-HAA we get the same 1-HAA as when we translate the LTL formula AGFp.

Theorem 3 *A CTL* formula φ has an equivalent CTL formula iff there exists a 1-HAA, $A_{D,\varphi}$, such that $\mathcal{L}(\mathcal{A}_{D,\varphi})$ is the set of D-trees satisfying φ.*

Proof: Firstly we need to show, if a CTL* formula φ has an equivalent CTL formula ψ then there exists a 1-HAA $A_{D,\varphi}$. However, since φ and ψ are known to be equivalent and from section 4.4 it is known that for every CTL formula a 1-HAA can be constructed, the 1-HAA for ψ, $A_{D,\psi}$, proves the existence of such a 1-HAA automaton. This proves the first direction.

The second direction, namely, if a 1-HAA exists for a CTL* formula φ then it has an equivalent CTL formula, will now be shown to hold by giving a translation from 1-HAA to CTL formulas. Since, we know that the 1-HAA must have been translated from a CTL* formula, we know two characteristics of the automata (from section 4.6):

- From a state in the automaton *all* successor states in the input tree will be considered. In other words, at no point in the input tree will the automata distinguish between subtrees. For example if an input tree has k successors at a state then the transition function cannot be of the form: $\bigvee_{c=0}^{j}(c, q_0) \wedge \bigvee_{c=j+1}^{k-1}(c, q_0)$, for some $0 \leq j < k-1$.
- The single state of an existential (universal) set cannot be in B (G).

The translation proceeds in a bottom-up fashion according to the partial order between the S_i sets of the 1-HAA: the first states to be translated to

formulas are those that do not refer to states in lower S_i sets. As soon as a state is translated to a CTL formula, all occurrences of that state in the transition functions of states of higher S_i sets in the partial order are replaced with the formula. Note the formula (that replaced a state) is therefore considered to be just a proposition in the transition function where it replaced a translated state. This process of translating states that do not refer to states in lower S_i sets and replacing their occurrences with formulas (propositions) is repeated until the initial state of the 1-HAA is translated to a formula. A state is translated to a formula when the boolean transition function for the state matches the right-hand side of the translation rules for the CTL formulas given in section 4.4.

All that is required now is to classify the positive Boolean formulas for the transition functions of the states of a 1-HAA.

First consider a transient set, i.e. a set that does not contain any reference to the state it contains. In these cases the boolean expression can only consist of atomic propositions or refer to states that are already labelled by formulas contained in lower S_i sets.

In the case of existential or universal sets the boolean formulas first need to be rewritten in either conjunctive or disjunctive normal form. First we form an abstract formula, by abstracting the successor relation to only contain one successor for the single state in the S_i set. For example the formula $q \vee (p \wedge \bigvee_{c=0}^{k-1}(c, q_0))$ becomes just $q \vee (p \wedge q_0)$. This is required in order to rewrite the formulas into CNF or DNF. Note this is a safe reduction since we know the automaton cannot distinguish subtrees (see above). Furthermore, we also *remember* whether the reduction removed $\wedge$ or $\vee$-choices (in the example above it removed $\vee$-choices). The single state of the S_i set that is also referred to in the boolean expression will be called the *repeat* state.

Let us consider the abstract formula to be rewritten in CNF. Some of the disjunctive terms will now contain the repeat state and some will not. Group

all the terms that do not contain the repeat state together and replace it with a single new variable, call it Z. Note that the variables in Z have already been labelled by formulas or contain propositions. The boolean expression now has the following form:

$$Z \wedge X$$

where X is in CNF with every disjunctive term containing a reference to the repeat state. If we call the repeat state q_0, then X can be rewritten as $Y \vee q_0$, where Y contains no reference to q_0. Hence the boolean expression now becomes:

$$Z \wedge (Y \vee q_0) \tag{6.1}$$

Similarly if we started by rewriting the formula in DNF the boolean expression will become:

$$Z \vee (Y \wedge q_0) \tag{6.2}$$

Now, depending whether a $\wedge$ or $\vee$-choice was removed during the abstraction and whether the repeat state is in G, B or neither, the formula the boolean expression translates to can uniquely be identified. The different cases are summarised in Table 6.4.

	Eq 6.1/ 6.2	$\wedge/\vee$	B/G/(N)either	Formula
1	6.1	$\wedge$	B	$A(Y\ R\ Z)$
2	6.1	$\wedge$	N	$A(Y\ V\ Z)$
3	6.1	$\vee$	G	$E(Y\ V\ Z)$
4	6.1	$\vee$	N	$E(Y\ R\ Z)$
5	6.2	$\wedge$	B	$A(Y\ U\ Z)$
6	6.2	$\wedge$	N	$A(Y\ W\ Z)$
7	6.2	$\vee$	G	$E(Y\ W\ Z)$
8	6.2	$\vee$	N	$E(Y\ U\ Z)$

Table 6.4: Translation Rules for 1-HAA to CTL.

This concludes the proof. Note that if we did not use the R and W operators then rules 1, 4, 6 and 7 would no longer be available, and the translation would

require to first translate to equation 6.1; see if it matches rules 2 or 3 and if not translate to equation 6.2 which will then allow a match with either rule 5 or 8. □

Example: Consider the HAA for the CTL* (LTL) formula $A(GF(\neg p) \vee GFq)$ (formula for weak fairness) given in Figure 6.5.

s	$\delta(s, \emptyset, k)$	$\delta(s, \{p\}, k)$	$\delta(s, \{q\}, k)$	$\delta(s, \{p, q\}, k)$
s_0	$\bigwedge_{c=0}^{k-1}((c, s_0) \wedge (c, s_1))$	$\bigwedge_{c=0}^{k-1}((c, s_0) \wedge (c, s_1) \wedge (c, s_2) \wedge (c, s_3))$	$\bigwedge_{c=0}^{k-1}(c, s_0)$	$\bigwedge_{c=0}^{k-1}((c, s_0) \wedge (c, s_3))$
s_1	$\bigwedge_{c=0}^{k-1}((c, s_1))$	$\bigwedge_{c=0}^{k-1}((c, s_1) \wedge (c, s_2)$	true	true
s_2	true	$\bigwedge_{c=0}^{k-1}(c, s_2)$	true	true
s_3	true	$\bigwedge_{c=0}^{k-1}((c, s_2) \wedge (c, s_3))$	true	$\bigwedge_{c=0}^{k-1}(c, s_3)$

Figure 6.5: $A_{D,WeakFair} = (2^{\{p,q\}}, D, \{s_0, s_1, s_2, s_3\}, \delta, s_0, (\{\}, \{s_2\}))$

Since the translation is done bottom-up, s_2 is translated first.

$$s_2 \equiv (\neg p \wedge \neg q) \vee (p \wedge \neg q \wedge \bigwedge_{c=0}^{k-1}(c, s_2)) \vee (\neg p \wedge q) \vee (p \wedge q)$$

After abstracting the successors to one, $k = 1$, and simplifying the last two terms to q we get the following:

$$\begin{aligned} s_2 &\equiv (\neg p \wedge \neg q) \vee (p \wedge \neg q \wedge s_2) \vee q \\ &\equiv ((\neg p \wedge \neg q) \vee q) \vee (p \wedge \neg q \wedge s_2) \end{aligned}$$

This is now in the form $Z \vee (Y \wedge s_2)$, with $Z = (\neg p \wedge \neg q) \vee q$ and $Y = p \wedge \neg q$. Since a $\wedge$-choice was abstracted away and s_2 is in B, this matches case 5 in Table 6.4 and therefore we have:

$$s_2 \equiv A((p \wedge \neg q)\ U\ ((\neg p \wedge \neg q) \vee q))$$

This can be further simplified to get $s_2 \equiv AF(\neg p \vee q)$. The translations for s_1 and s_3 are similar, here we show a reduced version for s_3:

$$s_3 \equiv (\neg p \wedge \neg q) \vee (p \wedge \neg q \wedge AF(\neg p \vee q) \wedge s_3) \vee (\neg p \wedge q) \vee (p \wedge q \wedge s_3)$$

$$\equiv ((\neg p \wedge \neg q) \vee (\neg p \wedge q)) \vee (p \wedge \neg q \wedge AF(\neg p \vee q) \wedge s_3) \vee (p \wedge q \wedge s_3)$$
$$\equiv \neg p \vee (((p \wedge \neg q \wedge AF(\neg p \vee q)) \vee (p \wedge q)) \wedge s_3)$$

This is now in the form $Z \vee (Y \wedge s_3)$, with $Z = \neg p$ and $Y = (p \wedge \neg q \wedge AF(\neg p \vee q)) \vee (p \wedge q)$. This matches case 6 (since s_3 is in neither B nor G), and we have:

$$s_3 \equiv A(((p \wedge \neg q \wedge AF(\neg p \vee q)) \vee (p \wedge q))\ W\ \neg p)$$

This can be reduced to $s_3 \equiv A(AF(\neg p \vee q)\ W\ \neg p)$. Similarly we have $s_1 \equiv A(AF(\neg p \vee q)\ W\ q)$. Lastly s_0 needs to be translated:

$$\begin{aligned} s_0 \equiv\ & (\neg p \wedge \neg q \wedge s_0 \wedge A(AF(\neg p \vee q)\ W\ q)) \vee \\ & (p \wedge \neg q \wedge s_0 \wedge A(AF(\neg p \vee q)\ W\ q) \wedge AF(\neg p \vee q) \wedge A(AF(\neg p \vee q)\ W\ \neg p)) \vee \\ & (\neg p \wedge q \wedge s_0) \vee \\ & (p \wedge q \wedge s_0 \wedge A(AF(\neg p \vee q)\ W\ \neg p)) \\ \equiv\ & ((\neg p \wedge \neg q \wedge A(AF(\neg p \vee q)\ W\ q)) \vee \\ & (p \wedge \neg q \wedge A(AF(\neg p \vee q)\ W\ q) \wedge AF(\neg p \vee q) \wedge A(AF(\neg p \vee q)\ W\ \neg p)) \vee \\ & (\neg p \wedge q) \vee (p \wedge q \wedge A(AF(\neg p \vee q)\ W\ \neg p))) \wedge s_0 \end{aligned}$$

This is of the form $Z \wedge (false \vee s_0)$ and therefore matches case 2, and after reductions we get:

$$s_0 \equiv AG(A(AF(\neg p \vee q)\ W\ q) \wedge A(AF(\neg p \vee q)\ W\ \neg p))$$

The LTL formula for weak fairness, $A(GF(\neg p) \vee GF(q))$, therefore has an equivalent CTL formula, given by s_0 above. Note that it is by no means an obvious translation from the LTL formula to the CTL formula. The translations from a CTL* formula to an HAA can however be automated as well as the translation from a 1-HAA to a CTL formula as described in Theorem 3. This provides a means to determine automatically whether a CTL* formula has a CTL equivalent and what it is:

1. Translate the CTL* to an HAA.
2. Check whether the HAA is a 1-HAA.
3. If yes, translate the 1-HAA back to CTL.

In our system the translation back to CTL is not automated yet, since a translation by hand gives more readable results. This is due to the user being able to reduce the size of the boolean expressions by using, sometimes non-obvious, equivalences, as was seen in the example above. Note that it is possible to translate a CTL formula into an HAA that is not a 1-HAA, but in those cases the HAA can always be transformed into one that is a 1-HAA. In our translation from CTL* formula to HAA, an efficient translation is made, that will always achieve the most succinct automaton, and therefore translates any HAA to a 1-HAA if one exists.

6.5.2 CTL* Formulas Expressible as LTL

Unfortunately, the LTL case cannot be classified by taking the same approach as in the CTL case above. We might argue the following:

A CTL* formula φ has an equivalent LTL formula *iff* there exists only one-player S_i sets in the HAA, $A_{D,\varphi}$, such that $\mathcal{L}(\mathcal{A}_{D,\varphi})$ is the set of D-trees satisfying φ.

Clearly the first direction holds, since we can translate LTL formulas via nondeterministic Büchi automata, as described in section 4.5. The other direction however does not hold. Essentially, an HAA with a single one-player set can easily be seen to be equivalent to a nondeterministic Büchi automaton, and nondeterministic Büchi automata are more expressive than LTL. For example the HAA in Figure 6.6 has a single one-player set, but it is well known that no LTL formula can capture the property: "there exists a path on which p holds at all even moments".

q	$\delta(q,\{\neg p\},k)$	$\delta(q,\{p\},k)$
q_0	false	$\bigvee_{c=0}^{k-1}(c,q_1)$
q_1	$\bigvee_{c=0}^{k-1}(c,q_0)$	$\bigvee_{c=0}^{k-1}(c,q_0)$

Figure 6.6: $A_{D,EvenMoments} = (\{\{p\},\{\neg p\}\}, D, \{q_0,q_1\}, \delta, q_0, (\{q_1\},\{\}))$

However if we restrict the type of HAA we consider to be those that can be constructed from CTL* formulas then the above type of HAA cannot be generated. With this restriction it can be argued that when a CTL* formula is translated to an HAA with only one-player sets, then there must exist an LTL formula that is equivalent to the CTL* formula. Showing this to be true would require a translation from the HAA to the LTL formula in a similar fashion as the one used in the CTL case. Unfortunately, we are not aware of any such translation, and conjecture that it will be difficult to establish since there is an exponential increase in size when translating the other way (LTL to HAA with only one-player sets).

With the restriction that we only consider HAA translated from CTL* formulas we can recast the result from [CD88] into the HAA setting. There it was shown that a CTL* formula can be expressed in LTL *iff* the formula is equivalent to the formula where all path quantifiers are removed and a single A is put at the front[1]. For example if AGAFp is equivalent to AGFp (remove both A's to get GF and then add one A to the front to get AGFp), then AGAFp has an LTL equivalent. The formula for which all the path quantifiers are removed is called ψ^d. Therefore if the CTL* formula is of the form Aψ then it must be equivalent to Aψ^d, and, similarly for Eψ and Eψ^d. Clearly, the formulas Aψ^d and Eψ^d will be translated to HAA with one-player sets. Therefore if the original formula can be translated to an HAA containing a two-player set the two formulas cannot be equivalent. If both the HAA for Aψ (Eψ) and Aψ^d (Eψ^d)

[1]Only formulas of the form Aψ are considered to be LTL. Here the same can be done with placing an E in front in the case where the original formula was of the form Eψ.

only contain one-player sets we need to show that they accept the same input tree (or input word in this case, since they consist only of one-player sets). For the Aψ case this can be shown by

$$\mathcal{L}(A_{A\psi} \times \overline{A}_{A\psi^d}) = \mathcal{L}(\overline{A}_{A\psi} \times A_{A\psi^d}) = \emptyset$$

where $A_{A\psi}$ is the HAA for the formula Aψ and $\overline{A}_{A\psi}$ is its negation, and, similarly $A_{A\psi^d}$ and $\overline{A}_{A\psi^d}$ are the HAA for $A\psi^d$ and its negation. The same can be done in the Eψ case.

q	$\delta(q, \{\neg p\}, k)$	$\delta(q, \{p\}, k)$
q_0	$\bigwedge_{c=0}^{k-1}(c, q_0)$	$\bigwedge_{c=0}^{k-1}(c, q_0) \vee \bigwedge_{c=0}^{k-1}(c, q_1)$
q_1	$\bigvee_{c=0}^{k-1}(c, q_1)$	false

Figure 6.7: $A_{D,AFAGp} = (\{\{p\}, \{\neg p\}\}, D, \{q_0, q_1\}, \delta, q_0, (\{\}, \{q_0\}))$

Example: Consider the HAA for the CTL formula AFAGp given in Figure 6.7. This HAA contains a two-player set and hence cannot be expressed in LTL. The same result is also shown in [CD88], where a Kripke structure is found for which AFAGp is false in the initial state, but AFGp is true.

6.5.3 Applications

The results of the previous two sections can be applied in many different ways. For example they can be used to identify the difference in expressive power of CTL and LTL. In Chapter 2 different classes of properties to be checked were given in LTL format. How many of these can also be expressed in CTL? By checking whether the formulas can be translated to a 1-HAA we can answer this as follows:

Safety: AGp is a CTL formula.

Guarantee: AFp is a CTL formula

Obligation: $A(Gp \vee Fq)$ can be translated to a 1-HAA (Figure 6.3) and is therefore equivalent to a CTL formula.

Response: $AGFp$ can be translated to a 1-HAA (Figure 6.4) and is therefore equivalent to a CTL formula.

Persistence: $AFGp$ has no equivalent CTL formula, since it cannot be translated to a 1-HAA (Figure 6.1).

Reactivity: $A(GFp \vee FGq)$ has no equivalent CTL formula since Persistence properties are a subclass.

Unconditional Fairness: $AGFp$ is the same as Response.

Weak Fairness: $A(FGp \rightarrow GFq)$ has a CTL equivalent, see Example in section 6.5.1.

Strong Fairness: $A(GFp \rightarrow GFq)$ is the same as Reactivity.

In the model checking community CTL model checkers have always been more popular than LTL model checkers, since the model checking complexity for CTL is lower than that for LTL. LTL is however more concise than CTL and is thus the favoured language to express properties of a system. The work presented here now allows this seeming anomaly to be overcome: a user can now express properties in LTL and when translations exist to CTL the translation can be done automatically and the formula used with a CTL model checker. Previous attempts to allow a more expressive logic than CTL, but retaining the CTL model checking complexity have focused on logics that can be translated into CTL: e.g. CTL^2 [BG94] and LeftCTL* [Sch97]. For LeftCTL* it is the case that it is equivalent to CTL, but in the case of CTL^2 it was shown that every formula can be translated to CTL, except formulas of the form $EG(\psi_1 \; U \; \psi_2)$. For this formula a special extension to the traditional bottom-up model checker of Clarke et al. [CES86] is made that preserves the CTL model checking complexity.

In section 6.1 and section 6.2 the rules for respectively playing the nonemptiness games for LTL and CTL formulas are given. But what about formulas that are both CTL and LTL? The following theorem establishes an interesting result for this class of formulas.

Theorem 4 *For all LTL formulas that have equivalent CTL formulas, no new games are required during the nonemptiness game.*

Proof: From Theorem 3 if an LTL formula ψ has a CTL equivalent then the HAA for the formula ψ must be a 1-HAA. But from the rules for new games, the *NewGame* flag is only set for two-player sets of 1-HAA, and since the formula is LTL it can only have one-player sets. □

This is a significant result, since the nonemptiness game without new games can reuse previous results and can therefore be played by visiting every position in the game only once. It is also interesting to note that most of the properties we currently check of our asynchronous hardware designs fall into the class of formulas covered by Theorem 4 (see section 7.3.2). The model checking complexity for the sublogic of LTL for which Theorem 4 holds (call this sublogic *CTLequiv*) is however still exponential in the size of the formula, the same as full LTL. The reason is that formulas from CTLequiv can in some cases have CTL equivalent formulas of exponential size. An example is the CTLequiv formula $E(Fp_1 \wedge Fp_2 \wedge \ldots \wedge Fp_n)$, that has a CTL equivalent of size exponential in n.

In section 4.7 the logic LinearCTL* was introduced that can be translated linearly to an HAA. From the way LinearCTL* is constructed it follows that the HAA for these formulas will always be 1-HAA. Therefore, if we consider only the LTL formulas that are in LinearCTL* (call this logic *LinearLTL*) then we get a sublogic of LTL that allows model checking of linear complexity in both the size of the Kripke structure and the size of the formula. LinearLTL is however not the complete sublogic of LTL for which model checking is linear.

For example, the obligation property $A(Gp \vee Fq)$ is not in LinearLTL, but has a linear sized CTL equivalent, $A(p\ W\ AFq)$, and can therefore allow linear model checking according to Theorem 4. Finding more expressive sublogics of LTL for which model checking is linear is an interesting avenue for future work.

So far we have not considered the generation of the Kripke structure representing the reactive system to be checked. This is however an active area of research and many interesting techniques are being employed to generate as little as possible of the state space of a system, whilst still preserving the validity of model checking results [CGL92, DGG93, Lon93, CGS95, ES93, CFJ93, ID93]. *Partial order rules* [GKPP95, God90, GW91, GW93, HP94, Pel94, PL90, Val90, WW96] is one the most widely used of these techniques. Essentially, it avoids the generation of states due to the execution of interleavings of independent transitions in different concurrent processes of a reactive system. From [GKPP95] it is known that partial orders for branching time (CTL*) model checking are less efficient than those for LTL model checking, in the sense that the partial order rules would allow more of the state space of a system to be generated during branching time model checking than during LTL model checking. Therefore, if a CTL* formula has an LTL equivalent then partial order rules for LTL can be employed rather than the less efficient rules for CTL*.

6.6 Concluding Remarks

In this chapter we have illustrated that CTL* model checking can be done in an efficient fashion by exploiting the theory of automata and games. However, in order for a model checker to be useful in practice, it is not only the model checking algorithm that must be efficient. In the next chapter we describe different approaches for implementing a results store as well as an efficient structure for a model checker based on the nonemptiness games and finally, as an example, show how to model check asynchronous hardware systems.

Chapter 7

Implementation Issues

In order to illustrate the model checking algorithm based on the nonemptiness game all the systems (to be checked) used so far have been small enough to be displayed as a state transition graph. In real-world examples systems are rarely simple enough to encode in this fashion. It is much more natural to describe the reactive system to be model checked in some appropriate formalism from which the state transition graph (i.e. Kripke structure) can be obtained. For example using the protocol description language PROMELA [Hol91] to describe a communications protocol for model checking with the SPIN model checker [Hol92, Hol97c]. In section 7.3 it will be illustrated how the model checker described here is used for checking properties of asynchronous hardware systems. In the sequel our model checker will be referred to as *AltMC* (**Alt**ernating Automata based **M**odel **C**hecker).

A model checker takes two inputs: a description of a system and a property to be checked. We propose a modular design for a model checker such that minimal change is required when the description formalism is changed. In section 7.1 the design is described in the context of AltMC.

The size of the Kripke structure and hence the product automaton to be

checked for nonemptiness is a major issue in determining the applicability of a model checker. Not only is the number of states in the automaton a limiting factor, but also the amount of storage required for a single state. In section 6.5.3 it was mentioned that reducing the number of states is an active area of research. Here, however, we show two novel techniques for reducing the amount of storage required for the results store (section 7.2).

7.1 Structure of Model Checker

We mentioned above that the systems to be checked are usually represented as a high-level description formalism depending on the type of system. Besides PROMELA, other examples of high-level formalism used for model checking include:

ESML: Imperative style language with complex data-structures for model checking operating system kernels [Vis93].

Petri Nets: Used by the PEP model checker [GB96].

SMV: Used for describing synchronous hardware for the SMV OBDD based model checker [CMCHG96].

Rainbow: Language framework for describing asynchronous hardware to be model checked by AltMC [BFGW97, BFG+97].

At a low level, systems described in these high-level formalisms are represented by a set of transition rules to evolve the current state to a set of next states. For example systems described in the languages above will be compiled into appropriate transition rules and when a new state is required in the Kripke structure the applicable rule(s) will be executed to generate the new state(s).

Example: Consider the two process mutual exclusion system of section 2.2.1 for which a reachability graph is given in Figure 7.1. Below we give a set

of transition rules for this system in a guarded action notation: *guard* $\rightarrow$ *action*, where the guard needs to be satisfied before the action can be executed. The state of the system consists of the values of the control variable for $process_1$, $process_2$ and the *semaphore* variable and is described by $(process_1, process_2, semaphore)$. An asterisk ("*") in the guard of a transition indicates that the field can hold any value for the guard to be satisfied in that state. The action part of the transition will indicate how each field is changed, with "-" indicating unchanged fields. The start state is (N_1, N_2, S_0) and the transitions are the following:

$$(N_1, *, *) \rightarrow (T_1, -, -) \tag{7.1}$$

$$(T_1, *, S_0) \rightarrow (C_1, -, S_1) \tag{7.2}$$

$$(C_1, *, *) \rightarrow (N_1, -, S_0) \tag{7.3}$$

$$(*, N_2, *) \rightarrow (-, T_2, -) \tag{7.4}$$

$$(*, T_2, S_0) \rightarrow (-, C_2, S_1) \tag{7.5}$$

$$(*, C_2, *) \rightarrow (-, N_2, S_0) \tag{7.6}$$

By executing these rules from the initial state the state transition graph of Figure 7.1 is obtained.

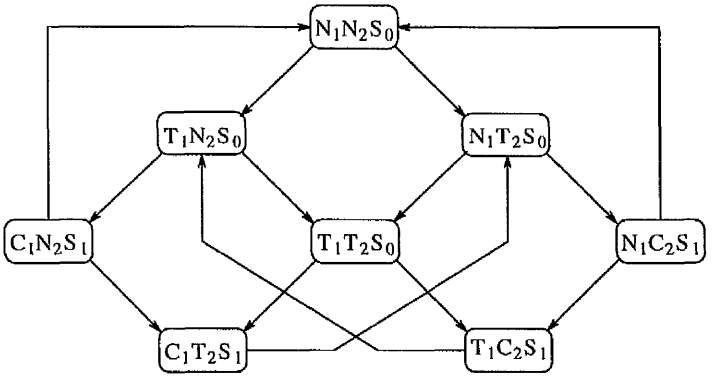

Figure 7.1: Reachability graph for the mutual exclusion system

Our goal is to design a model checking system that can check systems described in different formalisms. In [BFG89] such a model checking system is described: different input programs are compiled and executed to generate their reachability graphs in a format that can be used as input by existing model checkers. Since the reachability graph is generated before model checking starts this is not an on-the-fly model checking system. A novel feature of the system is that a program written in *any* language can be checked, since it not only takes as input the program, but also a parser and operational semantics for the language the program is written in.

Here we are not interested in how the states of the reachability graph for a program (in our case, reactive system) are generated, but whether the states can be generated on-the-fly during the execution of AltMC. We therefore assume the existence of an *Execution Unit* that can generate a successor state from the current state by executing an appropriate transition rule. Part of the execution unit is therefore the set of transition rules for the formalism currently being used as input to the model checker.

In the automata-theoretic approach to model checking a product automaton is created from the automaton for the formula and the one representing the Kripke structure. This is a synchronous product: each state in the product automaton is created by making one move each in the formula automaton and the Kripke automaton. In the on-the-fly approach the states of the Kripke automaton are generated when required, by executing state transition rules, during the creation of the product automaton. Therefore, if the goal is to design a model checker that can use different input formalisms, it is better to separate the execution of transition rules from the moves in the automaton for the formula.

In Figure 7.2 the structure of AltMC is given. It consists of four distinct parts:

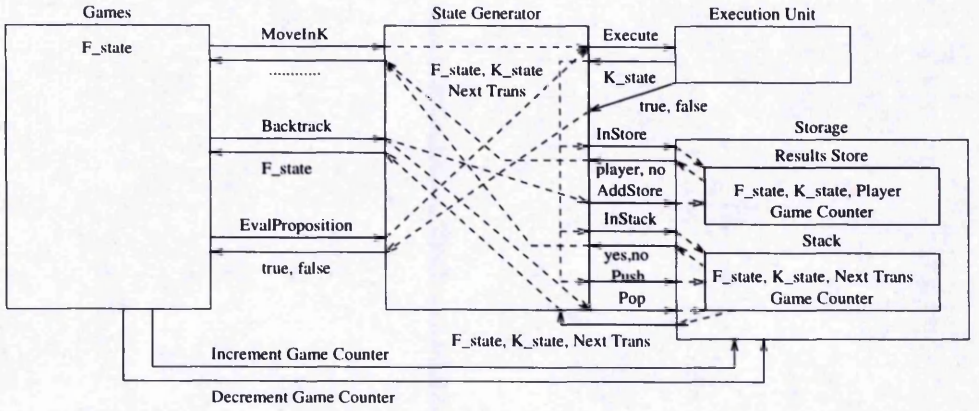

Figure 7.2: AltMC structure.

Games: This part contains the algorithm in Figure 5.7 and is the component driving the execution of the model checker. Its local variable is the state of the formula automaton (`F_state`). In order to know which boolean expression to expand from the transition table of the formula automaton, `EvalProposition` (that returns the truth-value of a proposition) must be called since the state of the Kripke structure is not known in this part of the model checker.

State Generator: This part can be seen as the interface between the formula automaton and the states of the Kripke structure. As its local variables it has the complete state of the product automaton (both the formula state and the Kripke structure's state, `K_state`) as well as the set of transitions that can still be executed in the current state (`Next Trans`). Another task of this component is to interface with the storage structures, stack and results store. The dotted lines indicate relationships between inputs being received and outputs being sent within the State Generator.

Execution Unit: This part is where the code is executed to generate a new state from the current state in the Kripke structure. It contains no local variables, since the current state and the transition to be next executed

are passed to it from the State Generator. Part of the transition rules is code to determine the truth-value of a proposition. When the Games component requires the value of a proposition it refers to it by an index, this index value is then used by the State Generator to determine which transition to execute to obtain the truth-value which is then passed back.

Storage: The results store and stack reside here. A stack entry consists of the state of the formula automaton, the state of the Kripke structure and the set of transitions still to be executed in the state of the Kripke structure. The entry in the results store consists of the state of the formula automaton, the state of the Kripke structure and the player for whom this is a winning position. Both the structures also contain the value of the Game counter that indicates which game is being played. When the Game counter is incremented, from the Games component, then a new stack and store are created. When it is decremented then these two structures are removed again. In the case of the stack this is achieved simply by using the same stack, and just changing the value of the game counter entries. The results store is implemented as a hash table and is partitioned according to the value of the counter to simplify the deletion of entries when the counter is decremented.

When the type of the system description language changes then at most the following components will be affected[1].:

State Generator: `K_state` and `Next Trans`.

Execution Unit: This component will be completely replaced by a different set of rules to execute.

Storage: The `K_state` components of the stack and results store will change, as well as the `Next Trans` set in the stack entries.

[1]Note that in some cases the type of language can change, but the transition rules and state descriptions can be the same as before.

With the exception of the execution unit, which might require major changes, the other components can easily be adapted to handle a new type of description language. As an illustration of the effectiveness of this structure for allowing different types of systems to be model checked, we changed the model checker to take descriptions of Rainbow designs (see section 7.3), rather than a state transition graph in less than one day. It must be noted that the transition rules for Rainbow designs already existed, since it was previously used as a basis for simulation.

It is interesting to note the different ways the two storage structures are used: the entries on the stack are both put onto the stack and taken off to be used again, but for the results store only the part relating to the winning player is retrieved. This indicates that the entries in the results store can be compacted. Since there are in general orders of magnitude more entries in the results store than on the stack, this will reduce the amount of memory required during model checking. Two compaction techniques are described in the next section.

7.2 Results Storage

The (product) state of the automaton can be represented by a vector of bits, called the *state vector*. In the nonemptiness game setting one bit is added to the state vector to indicate which player is winning from this position.

The representation of the results store has been the focus of much research [Hol88, GHP92, WL93, Gré96]. The most commonly used method is to represent the set as a large hash table of states. When a new state is generated it is hashed to obtain the index into the table. Since the states must be stored in their entirety, to allow for comparisons during the resolving of possible hash conflicts, this method is not very efficient when a large number of states must

be stored. The most novel approach to date is the so-called *bitstate hashing* [Hol88] used in the SPIN model checker. This technique requires a large vector of bits (of fixed size) to be maintained in memory to keep track of previously generated states. A hashing technique is used to compute an index into this bit vector from the value of each state. However, since the validation results can be invalid when a hash conflict occurs this technique is not always desirable when validating safety critical systems. The coverage obtained with bitstate hashing can be improved by either increasing the number of hashing functions used or using more than one bit to hash into (*hashcompact* [WL93]). A summary of bitstate hashing as well as comparisons with the hashcompact method can be found in [Hol95].

In the next two sections we propose two alternative techniques for representing the results store. Firstly, we show how a graph encoding, using ordered binary decision diagrams, can be used to represent the information in the results store and, secondly, we show how the state (vector) can be *compressed* without loss of information.

7.2.1 Graph Encoding with an OBDD

Appendix A gives an introduction to ordered binary decision diagrams (OBDDs) and also forms the background to the work presented here. If the reader is unfamiliar with OBDDs it is suggested that Appendix A.1 is read before proceeding.

If we consider a state, s, to be a vector of bits, $s = x_0, x_1, \cdots, x_n$ then when a state is visited then it can be represented by the boolean function $f(s) = 1$, or, in other words, $f(x_0, x_1, \ldots, x_n) = 1$. For example, consider a state vector with only three bits, x_1, x_2 and x_3 and a state is written as $< x_1 x_2 x_3 >$. Assume the following states are visited $< 000 >$, $< 010 >$ and $< 100 >$. Then the OBDD for the boolean function $f(x_1, x_2, x_3) = \overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1 x_2 \overline{x}_3 + x_1\overline{x}_2\overline{x}_3$ will represent

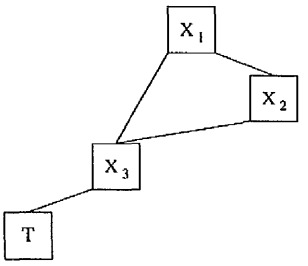

Figure 7.3: OBDD for function f, with variable ordering $x_1 < x_2 < x_3$.

the states visited. Thus if the variable ordering $x_1 < x_2 < x_3$ is assumed then the OBDD in Figure 7.3 represents the visited states for this example. To add a state, s, to the set of visited states it is therefore necessary to first convert s to its OBDD representation and then to perform an OR-operation on this OBDD and the one representing the set of states already visited. The resulting OBDD will represent the set of states with s added. To establish if a new state, s, is in the set of already visited states, the OBDD representing the visited states is traversed according to the values of the bits in s. For example, to check if the state $s =< 010 >$ is in the OBDD of Figure 7.3 we check if node x_1 has a left arc (since $x_1 = 0$ in s) this arc is traversed to reach node x_3; now since $x_3 = 0$ in s the left arc of x_3 is traversed and the resulting terminal node T is found indicating s is in the OBDD. If the existence of state $s =< 011 >$ was checked then at node x_3 the right branch would have been taken leading to the terminal F indicating s has not been visited yet. Only the following three operations are required to implement an OBDD state storage mechanism during on-the-fly model checking:

EncodeOBDD(s) This function takes as input a state and returns the OBDD encoding of the state.

OR(OBDD1,OBDD2) Returns the OBDD that is obtained after applying the transformation rules (given in Appendix A) to the result of adding OBDD1 to OBDD2.

Exists(OBDD1,s) Returns true if state s is in OBDD1 else false.

The advantage of using an OBDD to record visited states becomes apparent when large parts, or even, the complete reachable state space of a model are visited. Since this will in most cases cause the OBDD to become smaller, due to the applying of the transformation rules. For example consider the previous example where the states $< 000 >$, $< 010 >$ and $< 100 >$ were visited. If we now assume the rest of the reachable states of this example (the five states $< 001 >$,$< 011 >$, $< 101 >$,$< 110 >$ and $< 111 >$) are visited as well then the OBDD representing the visited states will only contain the terminal node T.

Example

Consider again the two process mutual exclusion example, from sections 2.2.1 with the transition rules given in section 7.1. The state of the system consists of the values of the control variable for $process_1$, $process_2$ and the $semaphore$ variable and is described by $(process_1, process_2, semaphore)$.The two control variables each have three possible values: N_i (0), T_i (1) or C_i (2). The semaphore has two values: S_0 (0) or S_1 (1). The two control variables will thus have two bits and the semaphore one bit allocated for it in the state vector (Figure 7.4).

State vector: 5 bits		
$Process_1$	$Process_2$	Semaphore
4..3	2..1	0

Figure 7.4: A State Vector for the Mutual Exclusion System.

During initialisation the initial state (N_1, N_2, S_0), which corresponds to the state vector $< 00000 >$, is inserted into the set of visited states. The next state to be visited is (T_1, N_2, S_0) $(< 01000 >)$ which is generated after executing transition 7.1. The OBDD resulting from adding this state to the initial state is shown on the left-hand of Figure 7.5. In the rest of Figure 7.5 the resulting

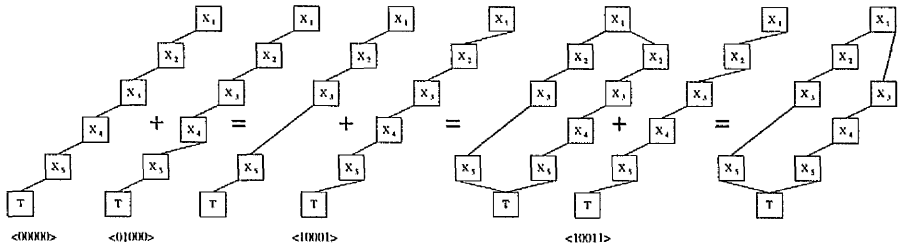

Figure 7.5: OBDD representations for adding the first four states of the mutual exclusion system to the state store.

OBDDs are shown after adding the next three states visited in the depth-first search. The OBDD on the right therefore contains the states $< 00000 >$, $< 01000 >$, $< 10001 >$ and $< 00010 >$. In Figure 7.6 the OBDD is shown after each of the remaining four states of this model is added to the set of visited states.

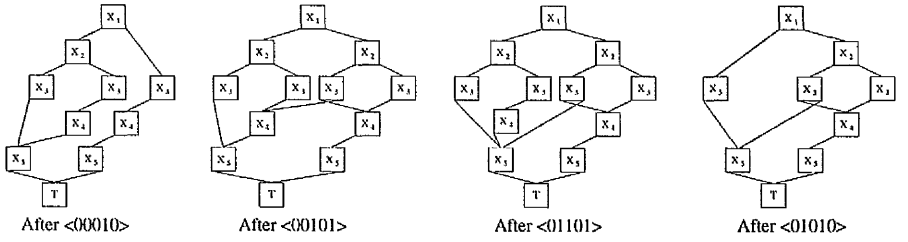

Figure 7.6: OBDD representations after the last four states of the example are added. The rightmost one represents all eight reachable states of the system.

The memory requirement for this method is that of the largest OBDD generated during the search. For this example it is the memory required to represent the second and third last OBDDs (see Figure 7.6) which both have 12 nodes (excluding the implicit F-terminal). Considering that the largest OBDD for a boolean function with five variables is 14, this example performs close to the worst-case[2]. In general the OBDD representing the state store will have a

[2]If n is the number of variables in a boolean function then there are $3 * 2^{\frac{n}{2}} - 2$ nodes when n is even and $2 * (2^{\lceil \frac{n}{2} \rceil} - 1)$ nodes if n is odd in the largest possible OBDD for the function (excluding the implicit F-terminal).

best-case performance that is *linear* and a worst-case that is *exponential* in the number of bits in the state vector.

7.2.2 State Compression

A system that has n bits in its state vector, seldom visits all 2^n states. Finding ways of reducing the state vector size, without losing information, would therefore seem a worthwhile pursuit. Strangely, however, *probabilistic* methods [Hol95, WL93, SD95] are attracting more attention than *safe* reduction methods [Vis93] in the research community. Probabilistic methods allow the state vector to be compressed in such a fashion that different states might have the same compressed state. During on-the-fly model checking this might cause a search to be truncated prematurely in which case certain parts of the state space might be ignored. Here a safe reduction method will be introduced that will reduce the memory requirements for a large class of models at the cost of an acceptable run-time overhead. The basic idea is to divide the state vector into parts and each part is then individually compressed with the aid of an indexed table recording all of the previous values that were assigned to the part (see Figure 7.7).

In general, consider the state vector, s, of length n to be constructed of $p+1$ parts each of length n_i with $i = 0 \ldots p$. The parts s_i might be the bits allocated to a process, a variable or even a byte boundary. For every part s_i allocate a table t_i with k_i entries. Let the compressed state, c, be of length m also consisting of $p+1$ parts each of length $m_i = log_2 k_i$. When a new state $s = s_0 s_1 \ldots s_p$ is generated take part s_i and check if these bits are in the table t_i; if so, take the table index and assign it to c_i; if s_i is not in the table find the next open slot (starting from slot 0), insert s_i, and assign the index to c_i. The compression technique is illustrated in Figure 7.7. This compression will fail to save memory when there are too many unique s_i parts, because then the table

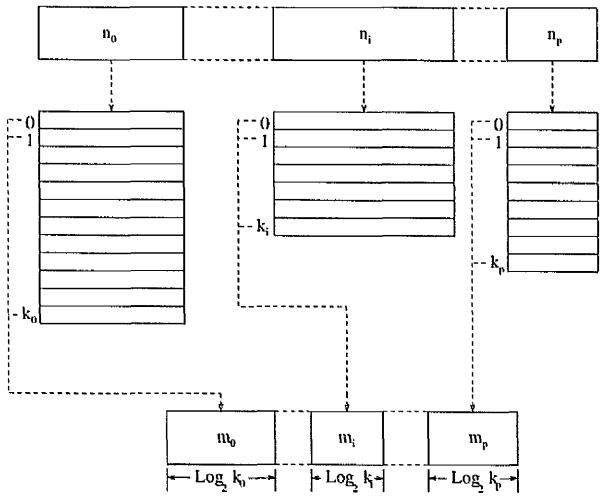

Figure 7.7: State vector compression via intermediate tables.

t_i becomes too large, or if a fixed table size is assumed, the table overflows. Thus this compression method is only suitable for models where each s_i part only has $k \ll 2^{n_i}$ unique states. There is also a break even point in the number of states that need to be generated before the memory used for the tables is offset. This value can be calculated as follows:

Let $|S|$ be the number of states generated. Memory will be saved when the memory used without compression is greater than the memory required if compression is used. Therefore,

$$|S| \sum_{i=0}^{p} n_i > |S| \sum_{i=0}^{p} log_2 k_i + \sum_{i=0}^{p} m_i n_i$$

and thus if,

$$|S| > \frac{\sum_{i=0}^{p} m_i n_i}{\sum_{i=0}^{p} n_i - \sum_{i=0}^{p} log_2 k_i}$$

This is read as: the number of states visited must be larger than the number of bits in the tables divided by the number of bits saved in each state. This fits in nicely with the intuition that the method will only work well if the tables

are relatively empty, i.e. each s_i part only has a few unique states.

7.2.3 Implementation

The graph encoding using OBDDs and the state compression method were developed before the AltMC model checker and has not yet been added to the system. However implementations of both techniques exist for the SPIN model checker [VB96]. The following conclusions can be drawn from the results of using the two techniques when model checking four example systems within SPIN:

OBDD Store: This was a failure since the number of bits in the state vector when using SPIN is too high (often > 500 bits). Note, the efficiency of the operations on the OBDD is dependent on the number of boolean variables in the OBDD and this number is in turn dependent on the number of bits in the state vector.

Compression + Hash Table: This gave the best results. The compression of the state vector reduced the memory required (four fold), but with an acceptable time overhead (1.46 times in the worst-case) over the case where a hash table was used by itself. Furthermore, in some cases the compression reduces the state vector length enough that time is actually saved since the hashing function executes faster.

Compression + OBDD: This combination achieved the best memory reduction, but on average took an order of magnitude longer to complete than when just a hash table is used.

7.2.4 Related Work

The work presented here on state compression and the OBDD results store, is an adaptation of the version that was originally published in [VB96]. At the

same time, Grégoire introduced another form of graph encoding, called *GE-sets* (Graph Encoded Sets) [Gré96] that encodes states at a higher level than OBDDs and is custom-made for use within SPIN. More recently, Puri and Holzmann, also investigated the use of graph encodings in SPIN [HP98].

In [Hol97b] Holzmann gives extensions of the basic state compression idea given here, called *recursive indexing*, and also implements it in the SPIN system. A two-phase compression is also given, where a second phase is used with larger tables, if during the first phase a table overflows.

7.3 Model Checking Rainbow Designs

Here we show the use of AltMC for checking properties of asynchronous hardware designs described in the Rainbow framework. We first outline the Rainbow framework for giving compact design representations, and then illustrate how this can lead to a reduction in the size of model generated.

The Rainbow asynchronous design framework offers a suite of unified design and modelling languages for giving mixed-view hierarchical descriptions of asynchronous micropipeline systems. Rainbow includes a control-flow sequential language, called Yellow, that is similar to OCCAM[Bur88], but uses an Ada-like rendezvous. A (static) dataflow-style language called Green uses micropipeline communication as primitive, thus hiding lower-level handshaking control components — we will concentrate on the latter here.

Micropipeline communication between sender and receiver components involves a 3-part handshake: a *request* control signal from the sender indicates that it is ready to offer some data, the receiver reads the data, and then gives an *acknowledge* signal to the sender. In Rainbow, the micropipeline communication is atomic, both at the user-language level — only a single (data) channel and the data transfer activity on the channel is seen by the designer —

and at the semantic level. In other modelling approaches, using for example CCS[Mil80, Mil89], OCCAM or PROMELA, the full handshake must be explicitly encoded, by the user and in the semantics, because the communication primitive is simply a synchronisation.

Green components include a (state-full) buffer node which can either read a value on its input channel when empty, or output a value when full. Other nodes are stateless; once the required values are present on their inputs, then the outputs can be generated, but the inputs are not released until all the outputs have been consumed. There is no implicit buffering between nodes.

A common underlying formal semantics for the component languages is given via a translation to a process term language called APA whose semantics is in turn defined using SOS-style transition rules. The semantics ensures that full interworking between components is supported at the micropipeline level, and provides the basis for formal analysis of designs. The APA transition rules are used for generating the states of the Kripke structure during model checking.

7.3.1 Example: Address Interface

Figure 7.8 shows the top-level hierarchical Green dataflow description of a simplified version of the AMULET1 microprocessor address interface[Pav94]. We first outline its behaviour: it performs several functions, so that only one memory address port MemOut is required on the processor. It generates sequences of increasing PC addresses, by cycling values through PC–MemAddr–Inc; these are used to fetch instructions in sequence from memory. It may also receive external requests (from the execute unit in the processor) modelled by the EXE_Input node. This introduces data read/write address values, which need to be passed to MemOut but do not enter the loop — in these cases, Arb suspends the normal PC value generation. Also, branch addresses can be input, so that the old PC value is replaced by the new branch value — it is therefore necessary to identify

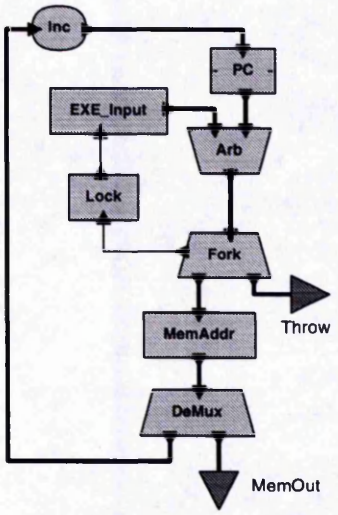

Figure 7.8: Green Description of Address Interface.

'old' and 'new' PC values, and this is achieved by tagging them with a 'colour', toggling this for every new value. The Fork node only passes on to MemAddr those PC values which match the current colour, and discards those that do not (Throw). The Lock node ensures that a new branch PC value can only be introduced when the old value has been discarded. In this model, we are interested mainly in the control and flow of address values in the interface, and so we can ignore the actual values given for each address, simply representing each one by its type (PC address or Data address) and its colour (red or green).

The design has the following state-full components: it has three data-buffers — 2 in the PC node (PC1, PC2), and MemAddr. Nodes Fork and EXE_Input store copies of the current colour and the Lock token is used to control branch address generation. In the initial state of the system both the buffers in the PC node are empty, the Lock is not set (i.e. Lock buffer is empty), and a green PC address is in the MemAddr buffer.

This version, with 2 buffers in PC will be shown below to be deadlock-free. However the 1 buffer version can be shown to deadlock.

7.3.2 Analysis

A state of the Kripke structure for a Rainbow design contains only the states of the buffers indicated above.

AltMC can now be used to analyse the properties of the design. The results are shown in Table 7.1, giving the number of states that need to be stored and the time required to check a selection of formulas for both the 2-buffer and 1-buffer versions of the interface.

	Design	Formula	AltMC		SPIN		
			States	Time	States	Time	
1	2-buf	Deadlock	274	31s	7615	7s	valid
2	2-buf	AG(PC →XF(Throw ∨ PC))	97	4s	405	0.9s	invalid
3	2-buf	AG(PC →(GF(DataAddr) ∨ XF(Throw ∨ PC)))	920	80s	15710	21s	valid
4	1-buf	Deadlock	66	4s	51	0.6s	invalid

Table 7.1: Analysis Results

We added to the model checker a *Deadlock* detection function that checks whether every reachable state of a system has at least one successor. A deadlock check for the Green version of the 2-buffer interface reports no deadlocks after examining all 274 reachable states of the system (row 1 in Table 7.1). A second useful property to be checked is that if there is a value in PC, then either it has the wrong colour and will be discarded by Fork (i.e. it will reach Throw) or it will again reach PC (row 2). However, this is invalid, since Arb may (unfairly) choose forever to accept data address values (DataAddr) from EXE_Input. When this possibility is allowed for in the formula (`'GF(DataAddr)'` in row 3) then the property is valid. This is an example of weak fairness being enforced on the behaviour of the address interface. Note also that, in accordance to the classifications given in section 6.5.1 this formula can be expressed as CTL. Finally, the deadlock in the 1-buffer version of the address interface is found (row 4).

7.3.3 Using PROMELA/SPIN to Check Properties

The address interface was also modelled in PROMELA, the input language of the LTL model checker SPIN. PROMELA simply uses synchronisation as the atomic communication between channels. This exposes the handshaking protocol between the components that was hidden in the Green model; consequently the address interface model is much larger (7615 states). The SPIN model checker also contains a built-in deadlock detection function which we used to establish that the PROMELA version of the interface is deadlock free. The results for checking the other LTL formulas in SPIN are shown in Table 7.1. It would also be interesting to compare results with those obtained from a CCS model of the address interface, using the approach advocated in [Liu95].

7.3.4 Observations

From the timings given in Table 7.1 it can be seen that SPIN is faster than AltMC (it examines more states per second). This is to be expected, since, unlike SPIN, AltMC is a new system that has not yet been redesigned for high performance[3].

A more interesting observation is that for both AltMC and SPIN when a property is invalid (rows 1 and 4 in Table 7.1) a counter example is found very quickly. In fact, it is our experience with using both SPIN and AltMC to check numerous systems that a counter example is often found, if one exists, by only examining a small portion of the reachable state space.

In the Rainbow framework a counter example is illustrated on the Green level by showing how the data flows in the system from the initial state to the one in which the counter example was found. Obviously, a short counter example will be desired over a long one, since it will be easier to follow. In

[3]Currently, nearly 80% of the time used to check a property of a Rainbow design is spent in the Execution Unit to generate a new state. This is an obvious area for improvement.

section 5.2.3 it was mentioned that when a choice exists for a player's next move we always make a move to a lower S_i set first; this has the effect of first examining state subformulas of a formula before the formula itself. This has the interesting side-effect of producing a "short" counter example (if one exists). Note however, since the successor states in the Kripke structure are examined in a depth-first fashion, we cannot guarantee to always get the shortest possible counter example.

7.4 Concluding Remarks

The designs we have model checked in the Rainbow framework have been fairly small until now, but we envisage this to change with the increase of new users. Therefore, although it is not currently required, state compression to reduce the memory required and partial order methods to reduce the search space will be added to AltMC in the near future.

The SPIN model checker is an example of a system that does not adhere to the structure described in section 7.1. In fact most of the Games and State Generator components reside in one function in SPIN (called `new_state`). This function is very long (928 lines[4]) and difficult to follow for all but the most experienced SPIN users. An interesting exercise would be to rewrite this code to follow the structure in Figure 7.2. SPIN is coded in the way it is for speed efficiency and therefore it would be of interest to see the trade-off between readability (understandability) and speed, when rewriting it in a structured way.

Currently, AltMC can check designs described in a reachability graph formalism or a design in the Rainbow framework. The next formalism to be added will be PROMELA, since this will allow the vast number of PROMELA designs

[4]SPIN version 3.0.0

existing already to be checked for CTL* correctness properties rather than the current LTL properties.

Chapter 8

Conclusions

The main goal of this thesis was to develop an efficient model checking algorithm for CTL*. In order to achieve this an automata-theoretic approach was taken, whereby the model checking problem was reduced to checking the nonemptiness of HAA. The nonemptiness check was recast as a two-player game which relies on the novel approach of playing new games to enable the safe reuse of previous results. It was shown how these games can be optimised depending on the structure of the HAA and therefore allowed not only efficient model checking for CTL* but also for the sublogics CTL and LTL.

Furthermore, in the setting of the nonemptiness game for HAA, the playing of new games can be used as a measure for the complexity of the algorithm. This leads to the, somewhat unexpected, result that the nonemptiness games when model checking CTL are in general more complex than when LTL formulas are being checked. When this is combined with the fact that formulas being checked are small in practice, and hence do not exhibit the exponential size increase that may occur in general, shows that LTL model checking is easier than CTL model checking in the nonemptiness game setting.

This is to the best of our knowledge the first time LTL and CTL model

checking have been compared in a unified setting. Previously, either one or the other type of model checker was taken as a basis:

- CTL model checking with an LTL model checker [BCG95]
- LTL model checking with a CTL model checker [CGH94].

Is the nonemptiness game an appropriate setting to compare the model checking complexity of CTL and LTL? We argue that it is when considering practical model checking, i.e. model checking capable of handling industrial-scale examples. Although both time and space efficiency are required of a practical algorithm, it is often the case that space is traded for time, since waiting longer for a result is more acceptable than getting no result at all (i.e. the "out of memory" message). The new games being played during the nonemptiness game can be seen as a way of avoiding building strongly connected components in the product automaton. The nonemptiness game therefore trades space for time. Another characteristic that can save both space and time is the fact that the nonemptiness game is a local algorithm, i.e. only the part of the product automaton required to label the initial state is traversed.

Just as alternating automata generalise nondeterministic automata, the nonemptiness game for HAA can be seen as a generalisation of the nested depth-first search for efficient nonemptiness checking of nondeterministic Büchi automata. In fact, the nonemptiness game algorithm is equivalent to the nested depth-first algorithm when LTL model checking is considered. We therefore claim that in a similar fashion as the link between LTL and nondeterministic automata led to the development of efficient model checking algorithms for LTL, the nonemptiness game shows that the link between CTL* and alternating automata also leads to efficient model checking algorithms for CTL*.

The thesis also addresses a long standing open problem [CD88]: for a given

CTL* formula does there exist an equivalent CTL formula? We show that the relationship between 1-HAA and CTL formulas can be exploited in order to answer this question. We believe that we even have an algorithm to construct the equivalent CTL formula for a CTL* formula if one exists. This algorithm is however based on the assumption that our translation from the CTL* formula to an HAA will always yield a 1-HAA if one exists. This assumption is not yet proven to hold, but we strongly believe it is the case. Furthermore, with the aid of this result we show that for the sublogic of CTL* containing all the formulas that can be expressed in both LTL and CTL, very efficient model checking is possible.

Next we consider some areas for future work: improving the system (partial order rules), new implementations (parallel model checking), model checking more expressive logics (ECTL*) and using alternating automata for model checking the μ-calculus.

In section 6.5.3 it was mentioned that partial order rules for LTL are more efficient than when a non-LTL formula is checked and therefore if we can determine from the HAA for the formula whether it has an LTL equivalent the more efficient rules can be used. However, we believe a finer distinction can be made on when to use linear time partial order rules and when to use branching time rules in a similar fashion to the classification for the S_i sets of the HAA for the formula given in Table 6.2. One possibility would be to use the linear time rules whenever a game is played from a position in a one-player set.

The nonemptiness game for HAA we describe here trades space for time, hence ways of making the algorithm faster need further investigation. We believe one way of improving the speed is to exploit the advances in computer hardware. Specifically we are interested in parallel architectures. Parallel algorithms for model checking have not been an active area of research, and only

a few examples exist and most of these are centred around developing parallel implementations of operations on OBDDs [LR93, SB96, SD96]. The main reason for parallel model checking not being very popular is the belief that communication overheads between different processors outweigh any efficiency increase gained from doing work in parallel. This is often true when the processors are distributed over a network, hence we propose to use a shared-memory multi-processor architecture for efficient parallel model checking. Furthermore, the new games being played within the nonemptiness game present a natural avenue for parallelisation: a new game can be played on a different processor, while the game that *spawned* it continues, and needs only read access to the results store of the *original* game. Developing a parallel implementation of the nonemptiness game together with the definition of sublogics of CTL* for which parallel model checking can be done efficiently (which ties in with the partial order work mentioned above) will form the basis of future work within our group.

Currently the model checker can only take a CTL* formula to be checked as input, however, it is a simple extension to enable the system to take any HAA as input. This would allow a larger class of properties to be checked than CTL*. For example the property that proposition p will only hold at all even moments, given in Figure 6.6, can be checked for a reactive system. In fact, it is easy to see that the nonemptiness game can check ECTL* [VW83] formulas. ECTL* is the extended version of CTL* where each path formula can be as expressive as ω-regular expressions, and therefore Büchi automata.

Can the nonemptiness game be of any use in μ-calculus model checking? Since ECTL* can be shown to be as expressive as the L_2 fragment of the μ-calculus introduced in [EJS93], the nonemptiness game can check formulas in this fragment of the μ-calculus. Unfortunately, but not unexpectedly since it would have given the elusive polynomial model checking algorithm, the full μ-calculus cannot be translated to HAA and hence cannot be checked by the

nonemptiness game. Interestingly, not even the alternation free fragment can be translated to HAA, and it is well known that this fragment can be model checked in linear time [CS91]. Alternation free μ-calculus formulas can however be translated to WAA [Ber95]. This, we believe, indicates that there are other classes of WAA (except for HAA) for which efficient model checking is possible. Furthermore, full μ-calculus formulas can be translated to alternating Rabin automata [Ber95], hence finding practical algorithms for such automata will be most interesting.

The work presented here is an attempt to make formal verification more appealing for use in industry by showing that practical model checking can be done for CTL*. Although it has been shown that the algorithm is at least as efficient as existing LTL and more efficient than existing practical CTL model checking algorithms, to show its true value it is most important that it is now applied in the model checking of industrial-scale examples. So far we have obtained very encouraging results from analysing asynchronous hardware designs expressed in the Rainbow framework.

Appendix A

OBDD Based Model Checking

A brief introduction to OBDDs is given in section A.1 followed by three sections describing how they are used for model checking.

A.1 Ordered Binary Decision Diagrams

Ordered binary decision diagrams (OBDDs) are directed acyclic graphs that represent boolean functions in a canonical form. As an example, let us consider the OBDD that represents the function $f(x_1, x_2, x_3) = \overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1 x_2 \overline{x}_3 + x_1\overline{x}_2\overline{x}_3$, where $\cdot$ denotes the AND operation and $+$ the OR operation. The decision tree for this function is given in Figure A.1. Left branches from a node indicates the variable is 0 (false) and similarly right branches indicate value 1 (true). Terminal nodes are labelled with T for true and F for false. To construct the OBDD representation of f a total ordering on the variables of f must be imposed. In Figure A.1 this ordering is $x_1 < x_2 < x_3$. There are three transformation rules on these graphs, that do not alter the function represented, but may reduce their size. An OBDD is the name given to the graph that cannot be reduced any further. The rules are:

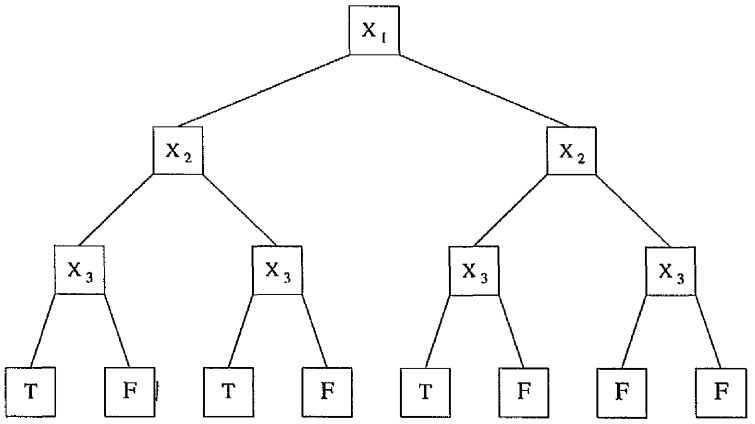

Figure A.1: Decision tree for the boolean function $f(x_1, x_2, x_3) = \overline{x}_1 \cdot \overline{x}_2 \cdot \overline{x}_3 + \overline{x}_1 \cdot x_2 \cdot \overline{x}_3 + x_1 \cdot \overline{x}_2 \cdot \overline{x}_3$

Remove Duplicate Terminals There must remain only one terminal with a given label. All arcs to duplicate labelled terminals must be redirected to the remaining one. A further optimisation that is not strictly required, is not to show any arcs leading to F, although they are implicitly there. In Figure A.2 this optimisation is performed on the decision tree for f.

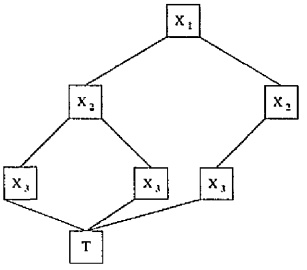

Figure A.2: Removing duplicate T terminals and explicit F terminals from the decision tree of f

Remove Duplicate Nonterminals If two nonterminal nodes have the same label and their left and right branches are the same then one must be removed. All arcs must be redirected to the remaining one. This is shown in Figure A.3 where the three nodes labelled x_3 (with left arc to T and

right to F) in Figure A.2 have been replaced by one x_3 node.

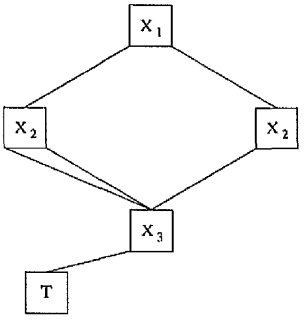

Figure A.3: Removing duplicate nonterminals.

Remove Redundant Tests If both arcs from a nonterminal, say n, point to the same node, say n', then n must be removed and all its incoming arcs must be redirected to n'. In Figure A.3 the left-hand x_2 is such a redundant node since both its left and right arc point to x_3. The OBDD for f is shown in Figure A.4 after this last transformation is performed.

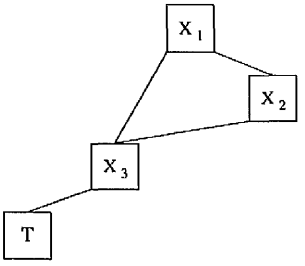

Figure A.4: OBDD for function f after removing redundant tests.

These rules must be applied repeatedly, since the application of one rule can cause more transformations to be possible on the resulting graph.

A.2 Representing Relations with OBDDs

OBDDs can provide extremely concise representations of relations over finite domains. If R is a n-ary relation over $\{0,1\}$ then R can be represented by an OBDD for its characteristic function f_R, where

$$f_R(x_1,\ldots,x_n) = 1 \textit{ iff } R(x_1,\ldots,x_n)$$

In order to construct complex relations it is convenient to extend propositional logic to permit quantification over boolean variables. The resulting logic is called QBF (Quantified Boolean Formulas). Given a set $V = \{v_1,\ldots,v_n\}$ of propositional variables, $QBF(V)$ is the smallest set of formulas such that

- $true$ and $false$ are formulas
- every variable in V is a formula
- if p and q are formulas, then $\neg p$ and $p \vee q$ are formulas
- if p is a formula and $v \in V$, then $\exists v p$ is a formula.

A *truth assignment* for $QBF(V)$ is a function $\sigma : V \to \{false, true\}$. Each QBF formula is equated with the set of truth assignments that satisfy the formula. Thus, $true$ represents the set of all truth assignments, and false the empty set. A propositional variable v represents the set of all truth assignments a such that $a(v) = true$. Furthermore,

1. $a \in (p \vee q)$ *iff* $a \in p$ or $a \in q$,
2. $a \in (\neg p)$ *iff* $a \notin p$, and
3. $a \in (\exists v p)$ *iff* $\sigma[true\backslash v] \in p$ or $\sigma[false\backslash v] \in p$.

In order to represent a QBF formula with an OBDD, we only need to show the representation of $\exists v p$. This is given by the following OBDD, if p is given by an

OBDD:

$$\exists v p = p|_{v \leftarrow false} + p|_{v \leftarrow true}$$

where $p|_{x_i \leftarrow b}(x_1, \ldots, x_n) = p(x_1, \cdots, x_{i-1}, b, x_{i+1}, \cdots, x_n)$ and $+$ is the *or* operation for OBDDs.

As will be seen in the next section, the quantification required during model checking of CTL specifications will be of the form:

$$\exists v[p(v) \wedge q(v)]$$

Thus, an efficient implementation of this *relational product* is required. The algorithm in Figure A.5, taken from [CGL93], computes the above relational product in one pass over the OBDDs $p(v)$ and $q(v)$. An important observation is that this algorithm computes the relational product without ever constructing the OBDD for $p(v) \wedge q(v)$, which can be very large. It also makes use of a result cache to avoid doing unnecessary work. Cache entries are of the form (p, q, E, h) where E is a set of variables that are quantified out and p, q and h are OBDDs. If an entry is in the cache, it means that a previous call to the function returned h as its result.

A.3 Fixpoint Characterizations of CTL

Clarke and Emerson showed that CTL operators can be characterized by extremal fixed points of appropriate functionals[CE81]. Let $M = (S, R, L)$ be an arbitrary finite Kripke structure. Let $Pred(S)$ denote the lattice of predicates over S where each predicate is identified with the set of states in S that make it true (the ordering is set inclusion). The least element in the lattice is therefore the empty set (false) and the greatest element is the set of all states (true). A functional $\tau : Pred(s) \rightarrow Pred(s)$ is called a *predicate transformer.* By definition,

```
Function RelProd(p,q: OBDD, E: set of variables): OBDD
BEGIN
  IF(p=false) OR (q=false) THEN
    RETURN false;
  ELSIF (p=true) AND (q=true) THEN
    RETURN true;
  ELSIF (p,q,E,h) in cache THEN
    RETURN h;
  ELSE
    let x be the top variable of p;
    let y be the top variable of q;
    let z be the top variable of x and y;
    h0 := RelProd(p|z=0, q|z=0,E);
    h1 := RelProd(p|z=1, q|z=1,E);
    IFz ∈ E THEN
      h := OR(h0, h1);
      /* OBDD for h0 ∨ h1 */
    ELSE
      h := IfThenElse(z,h1, h0);
      /* OBDD for (z ∧ h1) ∨ (¬z ∧ h0) */
    END;
    insert (p,q,E,h) in cache;
    RETURN h;
  END
END
```

Figure A.5: Relational Product Algorithm.

1. τ is *monotonic* when $P \subseteq Q$ implies $\tau[P] \subseteq \tau[Q]$

2. τ is $\cup$-*continuous* provided that $P_1 \subseteq P_2 \subseteq \ldots$ implies $\tau[\cup_i P_i] = \cup_i \tau[P_i]$

3. τ is $\cap$-*continuous* provided that $P_1 \supseteq P_2 \supseteq \ldots$ implies $\tau[\cap_i P_i] = \cap_i \tau[P_i]$

When S is finite, every increasing(decreasing) chain of subsets has a maximum (minimum) element. Therefore, in the finite case, monotonicity implies $\cup$-continuity and $\cap$-continuity. Tarski [Tar55] showed that a monotonic functional always has a least and a greatest fixed point with respect to inclusion ordering:

Theorem 5 (Tarski-Knaster) *If $\tau[Y]$ is monotonic, it has a least fixed point, $\mu Y.\tau[Y]$, and a greatest fixed point, $\nu Y.\tau[Y]$. If $\tau[Y]$ is also $\cup$-continuous, $\mu Y.\tau[Y] = \cup_{i \geq 0} \tau^i(false)$. If $\tau[Y]$ is also $\cap$-continuous, $\nu Y.\tau[Y] = \cap_{i \geq 0} \tau^i(true)$.*

The CTL operators can now be characterized as a least or greatest fixpoint of an appropriate predicate transformer:

Theorem 6 (Clarke-Emerson) *If we identify each CTL formula f with the predicate $\{s|M, s \models f\}$ in $Pred(S)$ and provided S is finite then*

- $A[p\ U\ q] = \mu Z.[q \vee (p \wedge AX\ Z)]$
- $E[p\ U\ q] = \mu Z.[q \vee (p \wedge EX\ Z)]$
- $AFp = \mu Z.[p \vee AX\ Z]$
- $EFp = \mu Z.[p \vee EX\ Z]$
- $AGp = \nu Z.[p \wedge AX\ Z]$
- $EGp = \nu Z.[p \wedge EX\ Z]$

```
Function (μY.τ[Y]{νY.τ[Y]})
BEGIN
  Y := false; {Y := true}
  DO Y' := Y; Y := τ[Y]; UNTIL Y = Y';
  RETURN Y;
END
```

Figure A.6: Calculate Least (Greatest) fixpoint.

The algorithm to calculate the least (greatest) fixpoint of a monotonic functional is shown in Figure A.6. It works by starting of with false (true) and iterating the functional until a fixed point is reached. Assuming S is finite, the algorithm will terminate in at most $|S| + 1$ iterations.

Let us consider the mutual exclusion example from section 2.2.1. Consider the validation of the following precedence property: $A(\neg(C_1 \vee C_2)\ U\ S_1)$. The predicate transformer τ will be given by

$$\tau(Z) = S_1 \vee (\neg(C_1 \vee C_2) \wedge AX\ Z)$$

Thus, $\tau^1(false) = S_1 \vee (\neg(C_1 \vee C_2) \wedge AX\ false) = S_1$. Therefore $\tau^1(false)$ is all the states with S_1 as part of the state. From Figure 2.1, it therefore follows that after the first iteration the set of states $\tau^1(false)$ contains $\{(C_1N_2S_1)$ $,(C_1T_2S_1)$, $(T_1C_2S_1)$, $(N_1C_2S_1)\}$. Repeating the process, after the next iteration, $\tau^2(false) = \tau^1(false) \cup\{(T_1N_2S_0)$, $(T_1T_2S_0)$, $(N_1T_2S_0)\}$. After the third iteration $\tau^3(false)$ contains all the reachable states in the model. Therefore after four iterations we get a fixpoint, $\tau^4(false) = \tau^3(false)$. Since this fixpoint contains all the reachable states, the CTL formula holds for this model.

A.4 Model Checking Algorithm

In this section a model checking algorithm for CTL which uses OBDDs to represent the state transition graph will be described. The state of a concurrent

system is given by a vector of bits called the state vector. Assume this state vector consists of n boolean state variables $v_1, v_2, \ldots, v_n$. The transition relation $R(v, v')$ for the concurrent system will be given as a boolean formula in terms of the state variables: $v = (v_1, \ldots, v_n)$ which represent the current state and $v' = (v'_1, \ldots, v'_n)$ which represent the next state. The formula $R(v, v')$ is now converted to an OBDD.

The symbolic model checking algorithm is implemented by a procedure $MCheck$ that takes the CTL formula to be checked and the OBDD for $R(v, v')$ as its arguments and returns the OBDD representing the set of states that satisfy the formula. $MCheck$ is defined inductively over the structure of CTL formulas. If formula f is an atomic proposition v_i then $MCheck(f, R)$ is simply the OBDD for v_i. The formulas $EX\ f$, $E[f\ U\ g]$ and $EG\ f$ are handled as follows:

$$MCheck(EX\ f, R) = MCheckEX(MCheck(f, R))$$

$$MCheck(E[f\ U\ g], R) = MCheckEU(MCheck(f, R), MCheck(g, R))$$

$$MCheck(EG\ f, R) = MCheckEG(MCheck(f, R))$$

Since $AX\ f$, $A[f\ U\ g]$ and $AG\ f$ can all be rewritten using the above operators, this definition of $MCheck$ covers all CTL formulas.

The procedure for $MCheckEX$ is straightforward since the formula $EX\ f$ is true in a state if the state has a successor in which f is true.

$$MCheckEX(f(v)) = \exists v'[f(v') \wedge R(v, v')]$$

If we have OBDDs for f and R, then we can compute an OBDD for

$$\exists v'[f(v) \wedge R(v, v')]$$

by using the techniques described in section A.2 (figure A.5).

The procedure for $MCheckEU$ is based on the least fixpoint characterisation for the CTL operator EU that is given in the previous section.

$$MCheckEU(f(v), g(v)) = \mu Z(v)[g(v) \vee (f(v) \wedge MCheckEX(Z(v)))]$$

Here the function in Figure A.6 is used to calculate a sequence of approximations, $Q_0, Q_1, \ldots, Q_i, \ldots$ that converges to $E[f\ U\ g]$ in a finite number of steps. If we have OBDDs for f, g and the current approximation Q_i, then we can compute an OBDD for the next approximation Q_{i+1}. Since OBDDs provide a canonical form of boolean formulas, it is easy to test for convergence by comparing consecutive approximations. When $Q_i = Q_{i+1}$, the function for the least fixpoint terminates. The set of states corresponding to $E[f\ U\ g]$ will be represented by the OBDD for for Q_i.

MCheckEG is similar, but is based on the greatest fixpoint characterisation for the CTL operator EG:

$$MCheckEG(f(v)) = \nu Z(v)[f(v) \wedge MCheckEX(Z(v))]$$

If we have an OBDD for f, then the function for the greatest fixpoint from Figure A.6 can be used to compute an OBDD representation for the set of states that satisfy $EG\ f$.

The CTL model checker *SMV* [McM92a] is one of the best exponents of OBDD based symbolic model checking and is being used by Intel and IBM as the basis for some of their model checking efforts. In [CGH94] it is shown how the SMV model checker can also be used to do LTL model checking, by translating the LTL formula to a CTL formula with fairness constraints.

Bibliography

[Abr97] S. Abramsky. Semantics of Interaction. In *MATHFIT Instructional Meeting on Games and Computation*, University of Edinburgh, June 1997.

[AJ94] S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic. *Journal of Symbolic Logic*, 59(2):543–574, 1994.

[Ari] `http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html`.

[BBC+96] N. Bjorner, A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H. Sipma, and T. Uribe. STeP: Deductive-Algorithmic Verification of Reactive and Real-time Systems. In *CAV '96: 6th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, 1996.

[BC96] G. Bhat and R. Cleaveland. Efficient Local Model Checking for Fragments of the Modal μ-Calculus. In *TACAS '96: 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, 1996.

[BCDM86] M.C. Browne, E.M. Clarke, D.L. Dill, and B. Mishra. Automatic

Verification of Sequential Circuits Using Temporal Logic. *IEEE Transactions on Computers*, C-35(12):1035–1044, December 1986.

[BCG95] G. Bhat, R. Cleaveland, and O. Grumberg. Efficient On-the-fly Model Checking for CTL*. In *Symposium on Logic in Computer Science*, June 1995.

[BCL91] J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic Model Checking with Partitioned Transition Relations. In *Proceedings of the 1991 International Conference on Very Large Scale Integration*, August 1991.

[BCM+90] J.R. Burch, E.M. Clarke, K.L McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the 5-th IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, June 1990.

[Ber95] O. Bernholtz. *Model Checking for Branching Time Temporal Logics.* PhD thesis, The Technion, Haifa, Israel, June 1995.

[BF91] R.W. Butler and G.B. Finelli. The Infeasibility of Experimental Quantification of Life-Critical Software Reliability. In *Proceedings of the ACM SIGSOFT 91 Conference on Software for Critical System*, pages 66–76, New Orleans, Louisiana, December 1991.

[BFG89] H. Barringer, M. Fisher, and G. Gough. Fair SMG and Linear Time Model Checking. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite Systems.* Lecture Notes in Computer Science, 407, June 1989.

[BFG+97] H. Barringer, D. Fellows, G. Gough, P. Jinks, and A. Williams. Multi-View Design of Asynchronous Micropipeline Systems using Rainbow. In *Proceedings of VSLI'97. Gramado, Brazil.* Chapman & Hall, August 1997.

[BFGW97] H. Barringer, D. Fellows, G. Gough, and A. Williams. Abstract Modelling of Asynchronous Micropipeline Systems using Rainbow. In *Proceedings of CHDL'97. Toledo, Spain.* Chapman & Hall, April 1997.

[BG93] O. Bernholtz and O. Grumberg. Branching Time Temporal Logic and Amorphous Tree Automata. In *CONCUR '93: 4th International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, 1993.

[BG94] O. Bernholtz and O. Grumberg. Buy One, Get One Free !!! In *ICTL '94: 1st International Conference on Temporal Logic*, volume 827 of *Lecture Notes in Artificial Intelligence*, 1994.

[BL80] J. Brzozowksi and E. Leiss. Finite Automata and Sequential Networks. *Theoretical Computer Science*, 10:19–35, 1980.

[BLV95] N. Buhrke, H. Lescow, and J. Vöge. Strategy Construction in Infinite Games with Streett and Rabin Chain Winning Conditions. In *STACS '95: 12th Annual Symposium on Theoretical Aspects of Computer Science*, volume 900 of *Lecture Notes in Computer Science*, 1995.

[BMS95] J. Bern, C. Meinel, and A. Slobodova. Global Rebuilding of OBDDs — Avoiding memory Requirement Maxima. In *CAV '95: 5th International Conference on Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, 1995.

[Bry86] R.E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[Bry91] R. Bryant. On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Apllication

to Integer Multiplication. *IEEE Transactions on Computers*, 40(2):205–213, 1991.

[Bry92] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[BS90] J. Bradfield and C. Stirling. Verifying Temporal Properties of Processes. In *CONCUR '90: 1st International Conference on Concurrency Theory*, volume 458 of *Lecture Notes in Computer Science*, 1990.

[Büc62] J. Büchi. On a Decision Method in Restricted Second Order Arithmetic. In *Proceedings of the International Congress on Logic, Methodology and Philsophy of Science, 1960.* Stanford University Press, 1962.

[Bur88] A. Burns. *Programming in Occam 2.* Addison-Wesley, 1988.

[BVW94] O. Bernholtz, M. Vardi, and P. Wolper. An Automata-Theoretic Approach to Branching-Time Model Checking. In *CAV '94: 6th International Conference on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, 1994.

[CD88] E.M. Clarke and I.A. Draghicescu. Expressibility Results for Linear-Time and Branching-Time Logics. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 428–437. Lecture Notes in Computer Science, 354, Springer Verlag, 1988.

[CE81] E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Proceedings of IBM Workshop on Logic of*

Programs, pages 52–71. Lecture Notes in Computer Science, 131, 1981.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CFJ93] E.M. Clarke, T. Filkorn, and S. Jha. Exploiting Symmetry in Temporal Logic Model Checking. In *Proceedings of the Fifth International Conference for Computer-Aided Verification.* Lecture Notes in Computer Science, 697, July 1993.

[CGH94] E. Clarke, O. Grumberg, and K. Hamaguchi. Another Look at LTL Model Checking. In *CAV '94: 6th International Conference on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, 1994.

[CGH+95] E. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. Long, K. McMillan, and L. Ness. Verification of the Futurebus+ Cache Coherence Protocol. *Formal Methods in System Design*, 6:217–232, 1995.

[CGL92] E.M. Clarke, O. Grumberg, and D.E. Long. Model Checking and Abstraction. In *Proceedings of 19th Annual ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*, pages 343–354, Albuquerque, New Mexico, January 1992.

[CGL93] E. Clarke, O. Grumberg, and D. Long. Verification Tools for Finite-State Concurrent Systems. In *A Decade of Concurrency: Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*, 1993.

[CGS95] E. Clarke, O. Grumberg, and S.Jha. Verifying Parameterized

Networks using Abstraction and Regular Languages. In *CONCUR '95: 6th International Conference on Concurrency Theory*, volume 962 of *Lecture Notes in Computer Science*, 1995.

[CKS81] A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.

[CKS92] R. Cleaveland, M. Klein, and B. Steffen. Faster Model Checking for the Modal Mu-calculus. In *CAV '92: 4th International Conference on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, 1992.

[Cle90] R. Cleaveland. Tableau-Based Model Checking in the Propositional Mu-Calculus. *Acta Informatica*, 27:725–747, 1990.

[CMCHG96] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In *CAV '96: 6th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, 1996.

[CS91] R. Cleaveland and B. Steffen. A Linear Time model checking for Alternation Free Modal μ-Calculus. In *CAV '91: 3rd International Conference on Computer Aided Verification*, volume 575 of *Lecture Notes in Computer Science*, 1991.

[CVWY92] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, 1:275–288, 1992.

[Dam92] M. Dam. CTL* and ECTL* as Fragments of the Modal mu-Calculus. In *CAAP '92 : 17th Colloquium on Trees in Algebra and Programming*, volume 581 of *Lecture Notes in Computer Science*, 1992.

[DB95] A. Dsouza and B. Bloom. Generating BDD models for Process Algebra. In *CAV '95: 5th International Conference on Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, 1995.

[DGG93] D. Dams, O. Grumberg, and R. Gerth. Generation of reduced Models for Checking Fragments of CTL. In *CAV '93: 5th International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, 1993.

[Dil96] D. Dill. The murϕ verification system. In *CAV '96: 6th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, 1996.

[EFT91] R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for Symbolic Model Checking in CCS. In *Proceedings of the Third International Conference for Computer-Aided Verification.* Lecture Notes in Computer Science, 575, July 1991.

[EH86] E.A. Emerson and J.Y. Halpern. 'Sometimes' and 'Not never' Revisited: On Branching versus Linear Time Temporal Logic. *Journal of the ACM*, 33(1):151–178, January 1986.

[EJ88] E.A. Emerson and C.S. Jutla. Complexity of Tree Automata and Modal Logics of Programs. In *29th annual IEEE Symposium on Foundations of Computer Science*, 1988.

[EJS93] E. Emerson, C. Jutla, and A. Sistla. On Model Checking for Fragments of μ-Calculus. In *CAV '93: 5th International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, 1993.

[EL87] E.A. Emerson and C. Lei. Modalities for Model Checking:

Branching Time Strikes Back. *Science of Computer Programming*, 8:275–306, 1987.

[Eme85] E.A. Emerson. Automata, Tableaux and Temporal Logic. In *Logics of Programs*, pages 79–87. Lecture Notes in Computer Science, 193, 1985.

[Eme96] E.A. Emerson. Automated Temporal Reasoning about Reactive Systems. In F. Moller and G. Birtwistle, editors, *Logics for Concurrency*, pages 41–92. Lecture Notes in Computer Science, 1043, Springer Verlag, 1996.

[ES83] E.A. Emerson and A.P. Sistla. Deciding Full Branching Time Logic. In *Proceedings of the Workshop on Logics of Programs*, pages 176–192. Lecture Notes in Computer Science, 164, June 1983.

[ES93] E. Emerson and A. Sistla. Symmetry and Model Checking. In *CAV '93: 5th International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, 1993.

[Fis92] M. Fisher. A Model Checker for Linear Time Temporal Logic. *Formal Aspects of Computing*, 4(3):299–319, 1992.

[For98] 1998. `http://www.bell-labs.com/org/blda/product_formal.html`.

[Fra86] N. Francez. *Fairness*. Springer-Verlag, Inc., New York, 1986.

[FW91] P.G. Frankl and E.J. Weyuker. Assessing the Fault-Detecting Ability of Testing Methods. In *Proceedings of the ACM SIGSOFT 91 Conference on Software for Critical System*, pages 77–91, New Orleans, Louisiana, December 1991.

[GB96] B. Grahlmann and E. Best. PEP — More than Petri Net Tool. In *TACAS '96: 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, 1996.

[GH93] P. Godefroid and G.J. Holzmann. On the Verification of Temporal Properties. In *Participants Proceedings of the 13-th IFIP Symposium on Protocol Specification, Testing, and Verification*, Liège, Belgium, 25-28 May 1993.

[GHP92] P. Godefroid, G.J. Holzmann, and D. Pirottin. State Space Caching Revisited. In *Proc. 4th Computer Aided Verification Workshop*, Montreal, Canada, June 1992. also in: Formal Methods in System Design, Kluwer, 1995.

[GKPP95] R. Gerth, R. Kuiper, D. Peled, and W. Penczek. A partial order approach to branching time model checking. In *Proceedings of the 3rd Isreal Symposium on the Theory of Computing and Systems.*, pages 130–139. IEEE Computer Society Press, 1995.

[GL93] S. Graf and C. Loiseaux. A Tool for Symbolic Program Verification. In *CAV '93: 5th International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, 1993.

[God90] P. Godefroid. Using Partial Orders to Improve Automatic Verification Methods. In *Proceedings of the Second International Conference for Computer-Aided Verification.* Lecture Notes in Computer Science, 531, June 1990.

[Gou84] G. Gough. Decision Procedures for Temporal Logic. Master's thesis, Department of Computer Science, University of Manchester, 1984.

[GPVW95] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[Gré96] J.-Ch. Grégoire. State Space Compression with Graph Encoded Sets. In Jean-Charles Grégoire, Gerard J. Holzmann, and Doron Peled, editors, *Proceedings of the Second Workshop in the SPIN Verification System.* American Mathematical Society, DIMACS/39, August 1996.

[GW91] P. Godefroid and P. Wolper. A Partial Approach to Model Checking. In *6-th IEEE Symposium on Logic in Computer Science*, pages 406–414, Amsterdam, 15-18 July 1991.

[GW93] P. Godefroid and P. Wolper. Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. *Formal Methods in System Design*, 2(2):149–164, April 1993.

[HBK93] R. Hojati, R. Brayton, and R. Kurshan. BDD-Based Debugging of Designs Using Language Containment and Fair CTL. In *CAV '93: 5th International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, 1993.

[HD93] A. Hu and D. Dill. Efficient Verification with BDDs Using Implicit Conjoined Invariants. In *CAV '93: 5th International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, 1993.

[HDDY92] A.J. Hu, D.L. Dill, A.J. Drexler, and C.H. Yang. Higher-Level Specification and Verification with BDDs. In *CAV '92: 4th International Conference on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, 1992.

[HHK96] R. Hardin, Z. Har'El, and R. Kurshan. Cospan. In *CAV '96: 6th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, 1996.

[HKSV97] R. Hardin, R. Kurshan, S. Shukla, and M. Vardi. A New Heuristic for Bad Cycle Detection using BDDs. In *CAV '97: 6th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, 1997.

[Hol88] G.J. Holzmann. An Improved Protocol Reachability Analysis Technique. *Software, Practice & Experience*, 18(2):137–161, February 1988.

[Hol91] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[Hol92] G.J. Holzmann. Protocol Design: Redefining the State of the Art. *IEEE Software*, 9(1):17–22, January 1992.

[Hol95] G.J. Holzmann. An Analysis of Bitstate Hashing. In *15-th International Symposium on Protocol Specification, Testing, and Verification*, Warsaw,Poland, June 1995. North-Holland.

[Hol97a] G. Holzmann. Invited Presentation, November 1997. Formal Methods Day, Royal Holloway & Bedford NW College, University of London.

[Hol97b] G.J. Holzmann. State Compression in Spin. In *Proceedings of the Third Spin Workshop*, Twente University, The Netherlands, April 1997.

[Hol97c] G.J. Holzmann. The Model Checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[HP85] D. Harel and A. Pnueli. On the Development of Reactive Systems. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. Springer-Verlag, New York, 1985.

[HP94] G.J. Holzmann and Doron Peled. An Improvement in Formal Verification. In *Proc. FORTE94*, Berne, Switzerland, October 1994.

[HP96] G. Holzmann and D. Peled. The State of SPIN. In *CAV '96: 6th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, 1996.

[HP98] G. Holzmann and A. Puri. A Minimized Automaton Representation of Reachable States. To appear, 1998.

[HPY96] G.J. Holzmann, D. Peled, and M. Yannakakis. On Nested Depth First Search. In Jean-Charles Gregoire, Gerard J. Holzmann, and Doron Peled, editors, *Proceedings of the Second Workshop in the SPIN Verification System.* American Mathematical Society, DIMACS/39, August 1996.

[HWT95] P. Ho and H. Wong-Toi. Automated analysis of an audio control protocol. In *CAV '95: 5th International Conference on Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, 1995.

[ID93] C.W. Ip and D. Dill. Better verification through symmetry. In *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and their Application.* North Holland, April 1993.

[Ins98] 1998. `http://www.chrysalis.com/Design_INSIGHT/`.

[JJ89] C. Jard and T. Jéron. On-line Model Checking for Finite Linear Temporal Logic Specifications. In *Proceedings of the International*

Workshop on Automatic Verification Methods for Finite Systems. Lecture Notes in Computer Science, 407, June 1989.

[Kar96] P. Kars. The Application of PROMELA and SPIN in the BOS project. In Jean-Charles Grégoire, Gerard J. Holzmann, and Doron Peled, editors, *Proceedings of the Second Workshop in the SPIN Verification System.* American Mathematical Society, DIMACS/39, August 1996.

[KMMP93] Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli. A decision algorithm for full propositional temporal logic. In *CAV '93: 5th International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, 1993.

[Kur94] R. Kurshan. *Computer Aided Verification of Coordinating Processes.* Princeton Series in Computer Science, 1994.

[Kur95] R. Kurshan. Personal Communication, July 1995. CAV'95, Liege, Belguim.

[Lad] P. Ladkin. `http://www.rvs.uni-bielefeld.de/~ladkin/Incidents/FBW.html#Accidents`.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[Lam80] L. Lamport. Sometimes is sometimes "not never" — on the Temporal Logic of Programs. *Proceedings 7th ACM Symposium on Principles of Programming Languages*, pages 174–185, January 1980.

[Lei81] E. Leiss. Succint Representation of Regular Languages by Boolean Automata. *Theoretical Computer Science*, 13:323–330, 1981.

[Liu95] Y. Liu. *AMULET1: Specification and Verification in CCS.* PhD thesis, University of Calgary, 1995.

[Lon93] D.E. Long. *Model Checking, Abstraction and Compositional Verification.* PhD thesis, Carnegie Mellon University, July 1993.

[Low96] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In *TACAS '96: 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, 1996.

[LP85] Orna Lichtenstein and Amir Pnueli. Checking That Finite State Concurrent Programs Satisfy Their Linear Specification. *Proceedings 12th ACM Symposium on Principles of Programming Languages*, pages 97–107, January 1985.

[LR93] I. Lee and S. Rajasekaran. Fast Parallel Algorithms for Model Checking Using BDDs. In V. K. Prasanna, editor, *Proceedings of the 7th International Parallel Processing Symposium*, pages 444–448, Newport Beach, CA, April 1993. IEEE Computer Society Press.

[McC96] G. McCusker. Games and Full Abstraction for FPC. In *Symposium on Logic in Computer Science*, July 1996.

[McM92a] K.L. McMillan. *Symbolic Model Checking.* PhD thesis, Carnegie Mellon University, May 1992.

[McM92b] K.L. McMillan. The SMV System. DRAFT, February 1992.

[MH84] S. Miyano and T. Hayashi. Alternating Automata on ω-Words. *Theoretical Computer Science*, 32:321–330, 1984.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[MP87] Z. Manna and A. Pnueli. Specification and Verification of Concurrent Programs with ∀-automata. *Proceedings 14th ACM Symposium on Principles of Programming Languages*, 1987.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Concurrent and Reactive Systems - specification*. Springer-Verlag, 1992.

[MSS86] D.E. Muller, A. Saoudi, and P.E. Schupp. Alternating Automata, the Weak Monadic Theory of the Tree and its Complexity. In *13th International Colloquium on Automata, Languages and Programming*, volume 226 of *Lecture Notes in Computer Science*, 1986.

[MSS88] D.E. Muller, A. Saoudi, and P.E. Schupp. Weak Alternating Automata give a Simple Explanation of why Temporal and Dynamic Logics are Decidable in Exponential Time. In *Third Symposium on Logic in Computer Science*, pages 422–427, July 1988.

[Pav94] N.C. Paver. *The Design and Implementation of an Asynchronous Microprocessor*. PhD thesis, Manchester University, May 1994.

[Pel94] D. Peled. Combining Partial Order Reductions with On-the-fly Model Checking. In *CAV '94: 6th International Conference on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, 1994.

[PL90] D.K. Probst and H.F. Li. Using Partial-Order Semantics to Avoid the State Explosion Problem in Asynchronous Systems. In *Proceedings of the Second International Conference for Computer-Aided Verification*. Lecture Notes in Computer Science, 531, June 1990.

[Pnu77] A. Pnueli. The Temporal Logic of Programs. In *18th annual IEEE-CS Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[Pnu86] A. Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency*, pages 510–584. Lecture Notes in Computer Science, 224, Springer Verlag, 1986.

[PR89a] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 179–190, 1989.

[PR89b] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *ICALP '89: 21st International Colloquium on Automata, Languages and Programming*, volume 372 of *Lecture Notes in Computer Science*, 1989.

[QS82] J.P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, 1982.

[QS83] J P Queille and J Sifakis. Fairness and Related Properties in Transition Systems—A Temporal Logic to Deal with Fairness. *Acta Informatica*, pages 195–220, 1983.

[Rab69] M.O. Rabin. Decidability of Second Order Theories and Automata on Infinite Trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.

[Rab70] M.O. Rabin. Weakly Definable Relations and Special Automata.

In *Proceedings of the Symposium on Mathematical Logic and Foundations of Set Theory*, pages 1–23. North Holland, 1970.

[RL97] T. Ruys and R. Langerak. Validation of Bosch' Mobile Communication Network Architecture in SPIN. In *Proceedings of the Third Spin Workshop*, Twente University, The Netherlands, April 1997.

[Saf88] S. Safra. On the Complexity of Omega-automata. In *29th annual IEEE Symposium on Foundations of Computer Science*, 1988.

[SB96] T. Stornetta and F. Brewer. Implementation of an Efficient Parallel BDD package. In *In 33rd Design Automation Conference*, 1996.

[SC85] A.P. Sistla and E.M. Clarke. The Complexity of Propositional Linear Temporal Logics. *Journal of the ACM*, 32(3):733–749, July 1985.

[Sch97] K. Schneider. CTL and Equivalent Sublanguages of CTL*. In *13th IFIP WG 10.5 International Conference on Computer Hardware Description Languages and their Apllications (CHDL '97)*, 1997.

[SD95] U. Stern and D. Dill. Improved Probabilistic Verification by Hash Compaction. In *CHARME '95: Correct Hardware Design and Verification Methods*, volume 987 of *Lecture Notes in Computer Science*, 1995.

[SD96] U. Stern and D. Dill. Parallelizing the murϕ verifier. In *CAV '97: 6th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, 1996.

[SE84] R.S. Streett and E.A. Emerson. The Propositional μ-Calculus is Elementary. In *11th International Colloquium on Automata, Languages and Programming*, pages 465–472. Lecture Notes in Computer Science, 172, 1984.

[SECH98] F. Schneider, S. Easterbrook, J. Callahan, and G. Holzmann. Validating requirements for fault tolerant systems using model checking. In *Proc. International Conference on Requirements Engineering, ICRE*, Colorado Springs, Co., USA, April 1998. IEEE.

[SOR93] N. Shankar, S. Owre, and J.M. Rushby. The PVS Proof Checker: A Reference Manual (Beta Release). Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, March 1993. See also `http://www.csl.sri.com/pvs-bib.html`.

[SS98] P. Stevens and C. Stirling. Practical model checking using games. In *TACAS '98: 4th International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, April 1998.

[Sti95] C. Stirling. Local model checking games. In *CONCUR '95: 6th International Conference on Concurrency Theory*, volume 962 of *Lecture Notes in Computer Science*, 1995.

[Sti96] C. Stirling. Games and model mu-calculus. In *TACAS '96: 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, 1996.

[Sti97] C. Stirling. Bisimulation, Model Checking and Other Games. In *MATHFIT Instructional Meeting on Games and Computation*, University of Edinburgh, June 1997.

[SVW87] A.P. Sistla, A.P. Vardi, and P.L. Wolper. The Complementation Problem for Büchi Automata with Apllications to Temporal Logic. *Theoretical Computer Science*, 49:217–237, 1987.

[SW89] C. Stirling and D. Walker. Local Model Checking in the Modal

mu-Calculus. In *Proceedings of the 1991 International Joint Conference on Theory and Practice of Software Development*, volume 351 of *Lecture Notes in Computer Science*, 1989.

[SW91] C. Stirling and D. Walker. Local Model Checking in the Modal mu-Calculus. *Theoretical Computer Science*, 89(1):161–177, 1991.

[Tar55] A. Tarski. A Lattice-Theoretical Fixpoint Theorem and its Application. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[Tar72] Robert E. Tarjan. Depth-First Search and Linear Graph Algorithms. *Society for Industrial and Applied Mathematics*, 1(2):146–160, 1972.

[TB73] B. Trakhtenbrot and Y. Barzdin. *Finite Automata: Behaviour and Synthesis.* North-Holland, Amsterdam, 1973.

[tEM95] A. th. Eiríksson and K. McMillan. Using formal verification/analysis methods on the critical path in system design: A case study. In *CAV '95: 5th International Conference on Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, 1995.

[Tho90] W. Thomas. Automata on Infinite Objects. *Handbook of Theoretical Computer Science*, pages 165–191, 1990.

[Tho93] W. Thomas. On the Ehrenfeucht-Fraïssé Game in Theoretical Computer Science. In *TAPSOFT '93: 4th International Joint Conference CAAP/FASE*, volume 668 of *Lecture Notes in Computer Science*, 1993.

[Tho95] W. Thomas. On the Synthesis of Strategies in Infinite Games. In *STACS '95: 12th Annual Symposium on Theoretical Aspects of Computer Science*, volume 900 of *Lecture Notes in Computer Science*, 1995.

[Tho97] W. Thomas. Winning Strategies in Infinite Games. In *MATHFIT Instructional Meeting on Games and Computation*, University of Edinburgh, June 1997.

[THY93] S. Tani, K. Hamaguchi, and S. Yajima. The Complexity of the Optimal Variable Ordering of a Shared Binary Decision Diagram. In *Proceedings of the 4th ISAAC*, volume 762 of *Lecture Notes in Computer Science*, 1993.

[Val90] A. Valmari. A Stubborn Attack on State Explosion. In *Proceedings of the Second International Conference for Computer-Aided Verification.* Lecture Notes in Computer Science, 531, June 1990.

[Var96] M. Vardi. An Automata-Theoretic Approach to Linear Temporal Logic. In F. Moller and G. Birtwistle, editors, *Logics for Concurrency*, pages 238–266. Lecture Notes in Computer Science, 1043, Springer Verlag, 1996.

[VB96] W.C. Visser and H. Barringer. Memory Efficient State Storage in SPIN. In Jean-Charles Grégoire, Gerard J. Holzmann, and Doron Peled, editors, *Proceedings of the Second Workshop in the SPIN Verification System.* American Mathematical Society, DIMACS/39, August 1996.

[VBF+97] W. Visser, H. Barringer, D. Fellows, G. Gough, and A. Williams. Efficient CTL* Model Checking for the Analysis of Rainbow Designs. Proceedings of CHARME '97, Montreal, October 1997.

[Vis93] W.C. Visser. A Run-Time Environment for a Validation Language. Master's thesis, Department of Computer Science, University of Stellenbosch, Stellenbosch 7600, South Africa, December 1993.

[VL93] B. Vergauwen and J. Lewi. A Linear Local Model Checking Algorithm for CTL. In *CONCUR '93: 4th International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, 1993.

[VW83] M. Vardi and P. Wolper. Yet Another Process Logic. In *Proceedings of the Workshop on Logics of Programs*, pages 501–512. Lecture Notes in Computer Science, 164, June 1983.

[VW86a] M. Vardi and P. Wolper. Automata-theoretic Techniques for Modal Logics of Programs. *Journal of Computer and System Science*, 32(5), 1986.

[VW86b] M.Y. Vardi and P. Wolper. An Automata Theoretic Approach to Automatic Program Verification. In *First Symposium on Logic in Computer Science*, pages 322–331, June 1986.

[VW94] M. Vardi and P. Wolper. Reasoning about Infinite Computations. *Information and Computation*, 115(1), 1994.

[WL93] P. Wolper and D. Leroy. Reliable Hashing without Collision Detection. In *CAV '93: 5th International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, 1993.

[Wol89] P. Wolper. On the Relation of Programs and Computations to Models of Temporal Logic. In *Temporal Logic in Specification*, volume 398 of *Lecture Notes in Computer Science*, 1989.

[WVS83] P. Wolper, M. Vardi, and A. Sistla. Reasoning about Infinite Computation Paths. In *24th annual IEEE Symposium on Foundations of Computer Science*, pages 185–194, 1983.

[WW96] B. Willems and P. Wolper. Partial-Order Methods for Model

Checking: From Linear Time to Branching Time. In *Symposium on Logic in Computer Science*, July 1996.