Automata, Model Checking and Synthesis for Linear Time Temporal Logics

PRS-D.Phil. Transfer Thesis

Matthew Hague
Oxford University Computing Laboratory

November 21, 2005

Summary

The goal of Model Checking is to verify that a given program matches its specification without error. This task is difficult and frequently undecidable. Due to increasingly sophisticated algorithms and processing power, model checking has had many successes in industry.

Further to model checking, we may wish to produce a correct program from a specification, rather than evaluating a program that has already been constructed. This is the synthesis problem. As would be expected, this problem is more difficult than model checking and has remained a largely academic subject. However, as advances in model checking have produced practically viable results, it may be possible for further advances in synthesis to have industrial success.

These two problems form the basis of our research. In particular, we focus on the model checking and synthesis problems using Linear Temporal Logic as our specification language. We are also interested in particular types of automata, which are closely related to LTL.

In this document we describe the two problems in detail, and several approaches to them. We also describe our own research into model checking algorithms and automata simplification. We propose further research along these lines.

Document Structure

In chapter 1 we give an overview of the model checking problem. We discuss the fundamental concepts: representing programs formally as Kripke structures, program specification languages and automata. In particular we focus on Linear Temporal Logic.

In the next chapter we describe several model checking algorithms, including symbolic methods to combat the state-explosion problem. We detail two recent methods for efficient model checking: an "on the fly" method introduced by Merz, Hammer and Knapp [47] and a SAT-based speed up algorithm by McMillan [43]. We then discuss our own work which aims to exploit the complementary properties of these two approaches to produce a new, more efficient, model checking algorithm. We describe our encoding of model checking as a SAT-problem and discuss the implementation, which is still a work in progress.

In chapter 3 we expand the scope of the document to the synthesis problem. This chapter is a version of a survey paper produced during the Hilary term, 2005. It describes the differing notions of program synthesis alongside recent attempts to use games as a unifying framework. We finish the chapter with an overview of some of the open problems in synthesis.

Chapter 4 discusses automata simplification. Because automata play an important role in both model checking and synthesis, they can have a large effect on the efficiency of an algorithm. There are three main stages for minimisation: simplifying LTL formulae, producing smaller automata, and minimising an existing automata. We survey several techniques at each stage.

The main thrust of the chapter is quotienting with respect to simulation. Simulation is when one automata is able to mimic the actions of another. A quotient automata is produced by merging automata states that simulate each other. When the automata is alternating, Fritz and Wilke define minimax and semi-elective quotienting for two different types of simulation: direct and delayed [15].

We finish the chapter by discussing our initial work into simulation for Linear Weak Alternating Automata with a co-Büchi acceptance condition. We show that the analogue of Fritz and Wilke's results also hold for LWAA. Finally we discuss further ideas for research. In particular, we aim to exploit the structure of Linear Weak Alternating Automata to produce more efficient quotients and simulation algorithms. We would also like to relate this work to an alternate definition of LWAA that is frequently used in practice. A further avenue of research utilises the notion of strategy composition — used to prove the transitivity of simulation relations — to provide a categorical interpretation of alternating automata, which may be generalised to two-player games. This may provide a connection between two different strands of research into games: verification and semantics.

Finally, chapter 5 surveys LTL fragments. The study of LTL fragments aims to understand why LTL model checking works in practice despite it high upper bound complexity. We describe an automata characterisation of a class of LTL fragments that limits the nesting depth of LTL's temporal operators. We then describe Partially Ordered Deterministic Büchi automata, which are used by Alur and La Torre to prove the complexity bounds for synthesis with LTL fragments [73].

We also describe an NP-complete fragment of LTL studied by Muscholl and Walukiewicz [6]. We propose further research into this logic. In particular, we may seek an automata characterisation of the fragment and study its complexity when used for program synthesis.

Proposed Research Directions

There are several avenues of potential research following from this dissertation. These can be broken down into three main categories: model checking algorithms, automata simulation relations and the study of LTL fragments.

In joint work with William Blum and Luke Ong, we are currently implementing a new model checking algorithm. This algorithm is a combination of two complimentary algorithms published recently by Merz *et al* and McMillan. We intend to complete this implementation in the near future. If successful, this implementation may lead to a conference paper and provide a framework in which to test further optimisation work. For example, we may wish to test the effectiveness of automata minimisation using simulation quotienting.

The second direction of research is automata simulation. In [15] Fritz and Wilke introduce simulation quotienting for alternating automata with a Büchi acceptance condition. We have shown that these results still hold for Linear Weak Alternating Automata with a co-Büchi acceptance condition. We aim to exploit the structure of an LWAA to produce quotients tailored towards them. We hope to produce more efficient algorithms for calculating the simulation relations and minimising the automata. For example, we hope that quotienting by fair simulation — which cannot be applied to alternating automata in general — may be possible for LWAA.

Furthermore, transitivity of the simulation relations is shown via a notion of strategy composition in a simulation game. This notion of composition suggests a categorical interpretation of simulation. This may be generalised to strategies in two-player games, which can be thought of as alternating automata with appropriate winning conditions. Such an interpretation may enable us to bridge the gap between games for verification and game semantics, in which categories and strategy composition play a fundamental role.

Finally we aim to study LTL fragments. In particular an NP-complete fragment of LTL introduced by Muscholl and Walukiewicz. This fragment gains significant complexity improvements by augmenting the tomorrow operator with the next character of input. We intend to study the complexity of the logic for synthesis and model checking of paths. We will also seek an automata characterisation of the logic. Hopefully this will give a greater understanding of why the restriction on the tomorrow operator leads an exponential reduction in complexity.

Research Activities

Courses Attended

I have completed four assessed courses: Logic of Multi-Agent Information Flow (A-); Logic, Automata and Games (A-); Computer Aided Formal Verification (B); and Lambda Calculus and Types (A).

I have also attended courses on Category Theory and Axiomatic Set Theory.

Meetings/Conferences/Seminars

- Games Spring School, Bonn, 15-19 March 2005.

- The European Joint Conferences on Theory and Practice of Software (ETAPS) 2005, Edinburgh, 2-10 April 2005.
- Oxford Said Business School, Intellectual Property Program, 9 June 2005
- NATO Summer School on Foundations of Secure Computation, Marktoberdorf, 2-14 August 2005.
- Member of the organising committee for Computer Science Logic (CSL) 2005, Oxford, 22 - 25 August 2005.
- Games 2005 Annual Meeting, Paris, 21-24 August, 2005.
- FOCS lunchtime meetings; Cakes talks.

Talks Given

- "Weak Alternating Automata and Linear Time mu-Calculus" at a FOCS lunchtime meeting, 18 January 2005.
- "From Separation Logic to First Order Logic" at FOSSACS 2005, part of ETAPS 2005.

Papers Written

- Cristiano Calcagno, Philippa Gardner and Matthew Hague, "From Separation Logic to First Order Logic", Proceedings of FOSSACS 2005.
- Matthew Hague, "Distributed Games", unpublished survey paper.

Papers Studied

Below is a short list of the main papers I have studied during my first year:

- *An NP-complete Fragment of LTL*, A. Muscholl and I. Walukiewicz [6].
- *Simulation Relations for Alternating Büchi Automata*, C. Fritz and Th. Wilke [15].
- *Interpolation and SAT-based Model Checking*, K. L. McMillan [43].
- *Truly "on the Fly" LTL Model Checking*, M. Hammer, A. Knapp and S. Merz [47].
- *Weak Automata for the Linear Time μ-Calculus*, M. Lange [49].
- *An Automata Theoretic Approach to Linear Temporal Logic*, M. Y. Vardi [50].
- *Deterministic Generators and Games for LTL Fragments*, R. Alur and S. La Torre [73].
- *Distributed Games*, S. Mohalik and I. Walukiewicz [85].

Teaching

- Compilers, Geraint Jones, 2004, Michaelmas term. Assisted during programming sessions.
- Procedural Programming, Richard Bird, 2005, Hilary term. Assisted during programming sessions.

Contents

1 Model Checking **9**

1.1 The Model Checking Problem 9
1.2 Modelling Programs 10
1.3 Specification Languages 11
1.3.1 Linear Time 11
1.3.2 Branching Time Logics 13
1.3.3 Linear vs. Branching Time 14
1.4 Automata 15
1.4.1 Büchi Automata 15
1.4.2 Generalised Büchi Automata 16
1.4.3 Alternating Automata 16
1.5 Summary 18

2 Model Checking Algorithms **19**

2.1 The Basic Model Checking Algorithm 19
2.2 Model Checking as Reachability 20
2.3 The State-Explosion Problem 21
2.3.1 Representing Automata Symbolically 21
2.3.2 Model Checking with BDDs 22
2.3.3 Bounded Model Checking 23
2.4 Truly "on the Fly" LTL Model Checking 24
2.4.1 Run DAGs 25
2.4.2 Co-Büchi LWAA 25
2.4.3 LTL to LWAA 26
2.4.4 Simple LWAA 26
2.4.5 Acceptance of a Simple LWAA 27
2.4.6 Model Checking with Simple LWAA 27
2.5 Interpolation and SAT-based Model Checking 28
2.5.1 Interpolation and Over-approximation 28
2.5.2 Interpolation Algorithm 29
2.5.3 Model Checking with Interpolants 30
2.6 Combining the Approaches 31
2.6.1 Encoding LWAA as Boolean Formulae 32
2.7 Implementing the Combined Approach 34

2.7.1 Implementation Language . . . 34
2.7.2 Specification Language . . . 34
2.7.3 SAT Solvers . . . 35
2.7.4 Current Implementation . . . 35
2.8 Summary . . . 37

3 Program Synthesis and Games **38**
3.1 Program Synthesis . . . 38
3.2 Synthesis and Control . . . 39
3.2.1 Church's Problem . . . 39
3.2.2 Reactive Systems . . . 40
3.2.3 Implementations . . . 40
3.2.4 Distributed Systems . . . 41
3.2.5 Incomplete Information . . . 41
3.2.6 The Control of Discrete Event Systems . . . 42
3.2.7 Control and Synthesis . . . 42
3.2.8 Asynchronous Systems . . . 43
3.3 Games and Synthesis . . . 44
3.3.1 Two Player Games . . . 44
3.3.2 Multiplayer Games . . . 45
3.3.3 Games and Control . . . 46
3.4 Unification Using Games . . . 46
3.4.1 A Summary of Distributed Synthesis Problems . . . 46
3.4.2 A Games Model . . . 47
3.4.3 Solving Distributed Games . . . 48
3.4.4 Encoding Distributed Synthesis Problems . . . 49
3.5 Team Games . . . 50
3.5.1 Distributed Systems . . . 50
3.5.2 Encoding Mohalik and Walukiewicz . . . 52
3.6 Open Problems . . . 52
3.7 Different Models . . . 53
3.7.1 Timed Systems . . . 53
3.7.2 Probabilistic Systems . . . 53
3.7.3 Knowledge-Based Specifications . . . 54
3.8 Summary . . . 54

4 Automata Simplification **55**
4.1 Minimizing LTL Formulae . . . 55
4.2 Constructing Smaller Automata . . . 56
4.3 Reducing Automata . . . 57
4.4 Simulation . . . 58
4.4.1 Alternating Büchi Automata . . . 58
4.4.2 Simulation Games . . . 59
4.4.3 Quotienting Modulo Simulation . . . 61
4.4.4 Minimal and Maximal Successors . . . 61
4.4.5 Direct Simulation and Minimax Quotienting . . . 62

4.4.6 Delayed Simulation and Semi-elective Quotienting 62
4.4.7 Simulation Algorithms 63
4.5 Simulation and LWAA 64
4.5.1 Co-Büchi Acceptance Conditions 65
4.5.2 Further Results 65
4.5.3 Further Research 66
4.6 Summary 66

5 LTL Fragments 67
5.1 Fragments, Satisfiability and Synthesis 67
5.1.1 LTL Fragments 67
5.1.2 Automata 69
5.2 An NP-Complete Fragment of LTL 71
5.3 Future Work 71
5.4 Summary 72

A Encoding LWAA in CNF 80

B LTL Rewrite Rules 82
B.1 Pure Universality and Pure Eventuality Formulae 82
B.2 Rewrite Rules for LTL Formulae 83
B.3 Propositional Rewrite Rules 83

C Simulation Games for LWAA with a co-Büchi Acceptance Condition 84
C.1 Simulation Games 84
C.1.1 Game Positions 85
C.1.2 Game Moves 85
C.1.3 Simulation 86
C.1.4 Protoplays 86
C.1.5 Winning Conditions 86
C.2 Simulation Hierarchy 87
C.3 Strategy Composition 88
C.4 Transitivity 90
C.5 Language Containment 94
C.6 An Alternative Delayed Simulation 96
C.7 Naive Quotienting 97
C.8 Minimal and Maximal Successors 98
C.9 Minimax Strategies 99
C.10 Direct Simulation and Minimax Quotienting 100
C.11 Delayed Simulation and Minimax Quotienting 103
C.12 Delayed Simulation and Semi-elective Quotienting 103
C.12.1 A Simulates A^s_{de} 103
C.12.2 A^s_{de} Simulates A 106

Chapter 1

Model Checking

We begin by giving a description of the Model Checking problem. The first section of this chapter describes and motivates the problem in more detail. We then introduce a formal model of computer programs — namely Kripke Structures — and go on to describe the different languages we can use for reasoning about them.

Finally we overview some of the basic tools that are used to provide model checking algorithms. In particular we introduce several specification languages and automata over infinite words, including alternating automata.

1.1 The Model Checking Problem

An important property of any program is its correctness. Errors in a system can easily go undetected. Once deployed, the costs of bugs can be enormous. One basic method for checking correctness is straightforward testing: a user tries to break the system by inputting as many test cases as is feasible. However, to test every set of input conditions is almost impossible — especially in the concurrent case, where a degree of nondeterminism is present. Testing, therefore, cannot guarantee correctness. Furthermore, it can sometimes be difficult to determine precisely where a program goes wrong.

It would be ideal then if we were able to prove that a system is error free, or identify errors before they identify themselves. This is the goal of the Model Checking problem.

In general, this problem is undecidable. That is, for any program and any specification, we cannot automatically determine whether the specification is met. The halting problem is a classic example of undecidability. However, we can restrict the range of expressible properties via the method we use to represent the system and its specification. Once restricted, the problem often becomes decidable, although the complexities are high.

Despite the high complexity, efficient algorithms and implementations have been developed and Model Checking enjoys a number of successes in indus-

try [78].

1.2 Modelling Programs

To be able to reason about programs we require a formal representation of the system. There are a number of formalisms in the literature, ranging from the theoretical (lambda calculus, pi-calculus, CSP, etc.) to the more practically motivated (Promela, SMV). In this section we introduce the model most commonly associated with model checking: Kripke Structures. Although many model checkers take a more practical language as input, the internal workings of the system and the accompanying literature often use Kripke structures. Therefore, they are the right formalism for discussing model checking algorithms.

Definition 1.2.1 *A Kripke Structure S over a finite set of atomic propositions AP is a tuple $\langle \mathcal{Q}, \mathcal{R}, l, \mathcal{I} \rangle$. $\mathcal{Q}$ is a (possibly infinite) set of states, and $\mathcal{R} \subseteq \mathcal{Q} \times \mathcal{Q}$ is a transition relation. $l : \mathcal{Q} \rightarrow 2^{AP}$ labels each state of Q with the set of atomic propositions that are true at that state. Finally $I \subseteq Q$ is a set of initial states (usually a singleton) of S. S is a finite Kripke Structure when $\mathcal{Q}$ is finite. $\mathcal{R}$ is total iff $\forall q \in \mathcal{Q} \exists q' \in \mathcal{Q}. q\mathcal{R}q'$. That is, every state has a successor.*

Together $\mathcal{Q}$ and $\mathcal{R}$ form a directed graph. The program starts in an initial state. At each program step a transition is taken to a next state. This gives us the notion of a path and a run.

Definition 1.2.2 *Given a Kripke Structure $S = \langle \mathcal{Q}, \mathcal{R}, l, \mathcal{I} \rangle$ a sequence $q_0, q_1, q_2, \ldots$ of states $q_i \in \mathcal{Q}$ such that $q_i \mathcal{R} q_{i+1}$ is a path through S. If the sequence is maximal, it is a fullpath. A fullpath is a run when $q_0 \in \mathcal{I}$.*

An equivalent, definition of a Kripke Structure uses a finite alphabet Σ rather than a set of atomic propositions. In this alternative definition it is the transition relation that is labelled rather than the states. Conceptually the program will perform an action $a \in \Sigma$ to move from one state to another.

To translate from AP to Σ we simply set 2^{AP} as our alphabet and move the labelling of a state to the transition relation. In the opposite direction, we encode Σ using a number of atomic propositions and move the labelling to the states. If a state is reachable by both an $a \in \Sigma$ and a $b \in \Sigma$ action ($a \neq b$), then we divide it into two states — one if it is reached by an a, the other if it is reached by a b.

Consequently a sequence of states (and therefore a path or a run) can equivalently be represented as a word over the alphabet Σ.

For the the remainder of this dissertation, we will assume that $\mathcal{R}$ is total (we can easily augment a transition relation that is not total by adding a "dead" state) and consequently that all runs are infinite.

1.3 Specification Languages

We can reason about a Kripke Structure using several different languages. Two important areas of study are Linear Time and Branching Time languages. In the linear time paradigm we assume that each time step has one possible future, whereas in the branching time philosophy, many different futures may occur. We begin by introducing the logics and end with a discussion of their various advantages.

1.3.1 Linear Time

ω-Regular Languages

We observed in the last section that a program run may be considered to be a word over the alphabet Σ. Therefore, an important tool for discussing the expressivity of linear time logics are ω-Regular Languages. We can express almost all important properties of a system using these languages.

Definition 1.3.1 *The syntax of ω-Regular expressions is,*

$$\alpha \ ::= \ \epsilon \mid a \mid \alpha \cup \alpha \mid \alpha ; \alpha \mid \alpha^* \mid \alpha^\omega$$

where ϵ denotes the empty word, a the single character a, $\cup$ the union of two languages, ; the composition and $*$ and ω (respectively) finite and infinite repetition of a language.

An important subclass of ω-regular languages is the star-free languages. This is the subclass without $*$ or ω, but complementation ($\bar{\alpha}$) is allowed. The star-free languages are those that can be defined using first-order logic over strings [77].

Linear Temporal Logic

Linear Temporal Logic (LTL) was introduced by Pnueli in 1977 [8]. It is interpreted over a linear time structure, where proposition valuations $\pi : N \rightarrow 2^{AP}$ are parameterised by a natural number denoting the timestep. What may be true initially might not be true at a timestep greater than zero.

The logic is built from a set of atomic propositions AP, the boolean connectives and two binary temporal operators $\bigcirc$ (tomorrow) and U (until). The meaning of $\bigcirc$ is straightforward given the semantics given below. Until is more subtle. The formula $\phi U \psi$ asserts that ϕ holds at all timesteps from now until ψ holds, and, moreover, that ψ will eventually hold. We interpret U non-strictly, and allow ψ to hold immediately, meaning that ϕ may never need to be satisfied.

Definition 1.3.2 *For a set of atomic propositions AP and an LTL formula ϕ, given a valuation $\pi : N \rightarrow 2^{AP}$ and timestep $i \in N$, we interpret ϕ as follows ($p \in AP$):*

$$\begin{array}{lcl}
\pi, i \models p & \iff & p \in \pi(i) \\
\pi, i \models \phi \wedge \psi & \iff & \pi, i \models \phi \textit{ and } \pi, i \models \psi \\
\pi, i \models \neg\phi & \iff & \textit{not } \pi, i \models \phi \\
\pi, i \models \bigcirc\phi & \iff & \pi, i+1 \models \phi \\
\pi, i \models \phi U \psi & \iff & \textit{there is } j \geq i \textit{ such that } \pi, j \models \psi \textit{ and for all } i \leq k < j \\
& & \textit{we have } \pi, k \models \phi
\end{array}$$

The literature makes use of three important abbreviations when discussing LTL. Namely $\phi V \psi$, $F\phi$ and $G\phi$. $\phi V \psi$ is defined as the dual of $\phi U \psi$, that is $\neg(\neg\phi U \neg\psi)$. $F\phi$ (future ϕ) asserts that ϕ holds at some point in the future, and is encoded $\top U \phi$, where $\top$ is truth. Because of the non-strict interpretation of U, $F\phi$ is satisfied if ϕ holds immediately. $G\phi$ (globally ϕ) is the dual of F ($\neg F \neg \phi$) and asserts that ϕ is true from this timestep on.

LTL can express all star-free ω-regular languages, and therefore all first order properties of strings [88].

Linear Time μ-Calculus

Wolper first observed that LTL cannot express all ω-regular properties [69]. It has been shown that LTL can express only the star-free properties [88]. It was also shown that LTL is not adequate for modular verification because it cannot express the required assumptions about the environment [61]. That is, we cannot break large programs into modules and verify them independently.

In response to these observations, Banieqbal and Barringer proposed the use of fix-point operators [13], yielding the Linear Time μ-Calculus (μTL). μTL extends LTL with least (μ) and greatest (ν) fix-point operators. This requires the introduction of a set of fix-point variables $\mathcal{V}$ and an environment $\rho : \mathcal{V} \to 2^N$ interpreting these variables when they are not bound by a fix-point operator ($\sigma X.\phi(X)$ where $\sigma \in \{\mu, \nu\}$). ρ simply identifies the set of timesteps in which a variable can be considered true. A μTL formula is closed if all fix-point variables appearing in the formula are bound by a fix-point operator.

To aid in the definition of the semantics of μTL we introduce the notation $[\![\phi]\!]^{\pi}_{\rho}$ to denote the set of all timesteps at which ϕ holds.

Definition 1.3.3 *For a set of atomic propositions AP, a disjoint set of fix-point variables $\mathcal{V}$ and a μTL formula ϕ, given a valuation $\pi : N \to 2^{AP}$ and environment $\rho : \mathcal{V} \to 2^N$, we interpret ϕ as follows (in addition to the semantics of LTL)($X \in \mathcal{V}$):*

$$\begin{array}{lcl}
[\![X]\!]^{\phi}_{\rho} & := & \rho(X) \\
[\![\mu X.\phi]\!]^{\pi}_{\rho} & := & \bigcap\{M \subseteq N | [\![\phi]\!]^{\pi}_{\rho[X \mapsto M]} \subseteq M\} \\
[\![\nu X.\phi]\!]^{\pi}_{\rho} & := & \bigcup\{M \subseteq N | M \subseteq [\![\phi]\!]^{\pi}_{\rho[X \mapsto M]}\}
\end{array}$$

where $\rho[X \mapsto M]$ behaves like ρ in all cases except $\rho(X)$ is M.

The theory of fix-point operators is not a simple one. Intuitively we think of the least fix-point μ as a finite loop, whereas the greatest fix-point ν corresponds

to infinite loops. That is, we can "recurse" through a fix-point variable bound by μ only a finite number of times, whereas we can recurse through a variable bound by ν an infinite number of times. The least and greatest fix-points are used to encode the ω-regular language operators $*$ and ω respectively. Consequently μTL can express all ω-regular properties.

For example, suppose we have one atomic proposition p. Let 1 denote a timestep where p holds and 0 denote a timestep where p does not hold. If we bound the variable X with μ and require that p holds globally or that we have the sequence 110 followed by X, we would be expressing the language $(110)^*(1)^\omega$. This is because we can either settle in Gp now $((1)^\omega)$, or take the second route which requires (110) and then loops through X back to where X is bound. Since X is bound by μ, we can only take this second route a finite number of times. Eventually, we must choose Gp. If instead we had bound X with ν and removed Gp from the disjunction, we would require an infinite repetition of 110, that is, $(110)^\omega$.

Kripke Structures and Linear Specifications

To make sense of the model checking and synthesis problems we need to define what it means for a program to satisfy a specification.

Definition 1.3.4 *Given a Kripke Structure S and an LTL or μTL specification ϕ (and environment ρ), S satisfies ϕ iff for all runs $q_0, q_1, q_2, \ldots$ of S, we have $\pi, 0 \models_{(\rho)} \phi$ where π is the valuation $\pi(i) = l(q_i)$ for all i.*

1.3.2 Branching Time Logics

In a linear paradigm, we assume that each moment of time has a unique successor. An alternative model of time assumes many different possible futures. To reason in this model we use branching time logics.

Computational Tree Logic

Computational Tree Logic was introduced in 1981 by Emerson and Clarke [28]. It is interpreted over computation trees, rather than linear sequences. Intuitively, a Kripke Structure S can be unwound to form a tree structure, where the initial state is the root. The children of each node are derived from the successors of the corresponding state in S.

Definition 1.3.5 *Given a set of atomic propositions AP, the syntax of CTL is as follows ($p \in AP$),*

$$\phi \ := \ p \mid \phi \wedge \psi \mid \neg\phi \mid E(\phi U \psi) \mid A(\phi U \psi) \mid E \bigcirc \phi \mid A \bigcirc \phi$$

Intuitively, the semantics of a CTL assertion is similar to LTL. The temporal connectives are augmented with existential and universal quantifiers, E and A. The existential quantifier requires that the assertion holds on some path from the current node in the tree. Dually, the universal quantifier requires that it holds on all paths leading from the current node.

Computational Tree Logic*

CTL is a fragment of CTL*. CTL* is an extension of LTL with quantification over runs.

Definition 1.3.6 *Given a set of atomic propositions AP, the syntax of CTL is as follows* ($p \in AP$)*,*

$$\phi \ := \ p \mid \phi \wedge \psi \mid \neg\phi \mid E\phi \mid \phi U \psi \mid \bigcirc \phi$$

Universal quantification $A\phi$ *is defined as an abbreviation for* $\neg E \neg \phi$.

The semantics of CTL* is defined in terms of runs of a computation tree T. A run r of T is a sequence $s_0, s_1, s_2, \ldots$ of nodes of T, such that s_{i+1} is a child of s_i for all i. We write $r[0, \ldots, i]$ to denote the first $i+1$ nodes in the sequence r.

Definition 1.3.7 *Given a set of atomic assertions AP a CTL* assertion* ϕ, *a computation tree* T, *and run* r *of* T, *and a position* i *of* r, *we interpret* ϕ *as follows (in addition to the semantics of LTL):*

$$T, r, i \models E\phi \iff T, r', i \models \phi \text{ for some } r' \text{ in } T \text{ such that } r[0, \ldots, i] = r'[0, \ldots, i]$$

1.3.3 Linear vs. Branching Time

The model checking problem for both CTL* and LTL is PSPACE-complete, CTL, however, is P-complete. The respective synthesis problems are 2EXPTIME-complete for CTL* and LTL, and EXPTIME-complete for CTL (when we are synthesising a non-distributed system) [57]. It should be noted that CTL and LTL are expressively incomparable, but are both fragments of CTL*.

In practice, CTL has been widely successful, forming the primary specification language of tools such as SPIN [31] and SMV. This is not surprising given the complexities above. However, authors such as Vardi and Schnoebelen have argued the case for LTL [51, 68]. This recent interest has been accompanied by advances in LTL model checking algorithms (discussed in chapter 2) and the addition of LTL specifications to the tools mentioned above.

There are several arguments offered for linear time over the branching time. Firstly, branching time is somewhat unintuitive. (For example, $AXAF\phi \not\equiv AFAX\phi$ [51].) Consequently, the branching aspect of the language is rarely used in practice. This leads Vardi to state that, in practice, CTL is less expressive than LTL. Furthermore, Schnoebelen argues that the size of the program makes the most important contribution to the practical complexity of model checking, and hence the difference between LTL and CTL is negligible.

A further criticism of branching time is that it does not admit compositional reasoning. In concurrent systems we can reduce the state-space explosion by considering program modules individually and conclude (subject to certain side

conditions) that the complete system is correct. CTL is not adequate for this technique, and so tools such as SMV have adopted the linear time paradigm.

Because of the reasons outlined above we will focus our attention on the linear model of time, and in particular LTL.

1.4 Automata

We need to be able to reason automatically about linear temporal logics. Automata theory has provided several important tools to aid us in this task. In this section we describe two important classes of automata and discuss their relationship with LTL. (For a detailed description of the automata-theoretic approach to LTL, see Vardi's survey paper [50].)

1.4.1 Büchi Automata

Automata can be used to define languages over a given alphabet Σ. We observed in section 1.2 that a program run can be considered a word. In particular we are interested in infinite runs of programs, and hence infinite words. Büchi automata can be used to calculate whether a given word is in our language. That is, whether a program run meets our specification.

Definition 1.4.1 *A Büchi automaton A is a tuple $(\Sigma, S, S^0, \delta, F)$. Σ is a finite, non-empty, alphabet. S is a finite set of states, where $S^0 \subseteq S$ is the set of initial states. $\delta : S \times \Sigma \to 2^S$ is a transition relation. $F \subseteq S$ is a set of final states.*

To determine whether a Büchi automaton accepts a word w we proceed as follows. The automaton starts at an initial state. It reads the characters of w one by one (from left to right). At each character we take a transition allowed by the transition relation given the current state and character. We accept the word if we meet a state in F an infinite number of times.

More formally, given a word $w = a_0, a_1, a_2, \ldots$ we define a run of the automaton on w as a sequence of states $s_0, s_1, s_2, \ldots$ where $s_0 \in S^0$ and $s_{i+1} \in \delta(s_i, a_i)$ for all i. If $\delta(q, a)$ is a singleton for all q and a then the automaton is deterministic — for any given word there is only one possible run. If we have a choice of next states then the automaton is non-deterministic, and for any given word a number of runs may exist.

In order to define acceptance we define the limit of a run r as $lim(r) = \{s \mid s = s_i \text{ for infinitely many } i\}$. A word is accepted if there is a run r of the automaton over the given word such that $lim(r) \cap F \neq \emptyset$. That is, a run exists where a final state occurs infinitely often.

The language of an automaton A is written $\mathcal{L}(A)$ and is defined as the set of all words accepted by the automaton.

Proposition 1.4.1 *[52] Given an LTL formula ϕ, once can build a Büchi automaton $A_\phi = (\Sigma, S, S^0, \delta, F)$, where $\Sigma = 2^{AP}$ and $|S|$ is $2^{O(|\phi|)}$, such that $\mathcal{L}(A_\phi)$ is exactly the set of computations satisfying the formula ϕ.*

It is this relationship with automata that produces the basic LTL model checking algorithm discussed in chapter 2. Given a Kripke Structure S and a specification ϕ, we construct an automaton to check $\mathcal{L}(S) \cap \mathcal{L}(A_{\neg\phi}) = \emptyset$. That is, there are no runs of S that satisfy $\neg\phi$, violating the specification. Therefore, model checking can be reduced to a test for language emptiness for a Büchi automaton. This check can be done in linear time [24, 25].

Because we check non-emptiness of $\mathcal{L}(S) \cap \mathcal{L}(A_{\neg\phi})$, we refer to an accepting run as a counter-example to the correctness of S.

1.4.2 Generalised Büchi Automata

An important variation on Büchi automata are generalised Büchi automata. Generalised Büchi automata usually form an intermediate stage in the translation from LTL to Büchi automata. That is, we translate the LTL formula into a generalised Büchi automaton and then translate the result into Büchi automaton.

Definition 1.4.2 *A Büchi automaton A is a tuple $(\Sigma, S, S^0, \delta, \mathcal{F})$. Σ is a finite, non-empty, alphabet. S is a finite set of states, where $S^0 \subseteq S$ is the set of initial states. $\delta : S \times \Sigma \to 2^S$ is a transition relation. $\mathcal{F} \subseteq 2^S$ is a set of fair sets $F \subseteq S$.*

The winning condition, rather than being a set of final states, is a set of fair sets. A fair set is a set of states. A run is accepting iff for all fair sets F, there is a state $q \in F$ such that q appears infinitely often. That is, we have to meet each of the fair sets infinitely often.

A Büchi automaton can be encoded as generalised Büchi automaton by setting $\mathcal{F} = \{F\}$. The translation in the opposite direction is more involved. In essence we order the fair sets. A counter is maintained which is incremented when we meet the next fair set in the order. Initially we must meet the first fair set in the order. Once the counter reaches the number of fair sets, the next transition always resets it. The set of final states in the equivalent Büchi automaton is the set of all states where the counter is equal to the number of fair sets.

1.4.3 Alternating Automata

Alternating automata have been studied by Brzozowski and Leiss [37] and Chandra, Kozen and Stockmeyer [5]. As we noted earlier, a Büchi automaton may be non-deterministic. That is, at a given state we can choose (non-deterministically) which transition to take. The automaton will accept if any of these transitions results in an accepting path. That is, there exists a transition leading to an accepting path. Alternating automata go a step further, allowing universal as well as existential quantification. Two different, but equivalent, definitions of alternating automata have been introduced in the literature [37, 5].

Definition 1.4.3 *[5] An alternating Büchi automaton A is a tuple $(\Sigma, S, S^0, \delta, E, U, F)$. Σ is a finite, non-empty, alphabet. S is a finite set of states, where $S^0 \subseteq S$ is the set of initial states. $\delta : S \times \Sigma \to 2^S$ is a transition relation. $F \subseteq S$ is a set of final states. $\{E, U\}$ is a partition of S into existential and universal states.*

Informally, a word is accepted by an alternating automaton if, from the current state, either all transitions lead to an accepting run (if we are in a universal state), or there is a transition which leads to an accepting run (if we are in an existential state). A more formal definition of acceptance is given in section 4.4.1.

Definition 1.4.4 *[37] An alternating Büchi automaton (ABA) A is a tuple (AP, S, S^0, δ, F). AP is a set of atomic propositions. S is a finite set of states, where $S^0 \subseteq S$ is the set of initial states. $\delta : S \to \mathcal{B}(S \cup AP)$ is a transition relation, where $\mathcal{B}(S \cup AP)$ is the set of all boolean formulae over the atomic propositions in $S \cup AP$ and states in S only occur positively. $F \subseteq S$ is a set of final states.*

In this model of automata we introduce alternation via the transition relation. At each point in a run of the automaton we evaluate the transition formula for the current state. The value of the atomic propositions is given by the character of input that is being read. We are looking for models of $\delta(q)$ (where q is the current state). If we can find a model such that an accepting run can be found for all states in the model, then the automaton accepts. For example, if $\delta(q) = (q_1 \wedge q_2) \vee (q_3 \wedge q_4)$ we accept if accepting runs can be found from states q_1 and q_2, or if accepting runs can be found from states q_3 and q_4. Acceptance for this variant of alternating automata is treated more formally in section 2.4.1.

It is known that alternating automata are no more expressive than non-deterministic automata, but exponentially more succinct. In fact, there is a linear translation from LTL to alternating automaton. However, this does not improve the complexity of LTL satisfiability because the complexity of testing for language emptiness is exponential for an alternating automaton, as opposed to linear for a non-deterministic automaton.

Before we state the relationship with LTL more formally, we introduce two subclasses of alternating automata.

Weak Alternating Automata

Definition 1.4.5 *[19] A Weak Alternating Büchi Automaton (WABA) is an ABA $A = (\Sigma, S, S^0, \delta, F)$ where S can be partitioned into components $C_0, \ldots, C_n$ such that,*

- *for all $q \in S, i, j \in \{0, \ldots, n\}, a \in \Sigma$: if $q \in C_i$ and $q' \in C_j$ and $\delta(q, a) = f(, \ldots, q', \ldots)$ for some f, then $j \leq i$.*
- *for all $0 \leq i \leq n$: $C_i \subseteq F$ or $C_i \cap F = \emptyset$.*

That is, a WABA is an ABA whose states can be partitioned into a hierarchy. At each state a transition can either stay within the current level, or move to a lower partition. Eventually, then, every run must "get stuck" in a partition. If this partition is accepting, then the automaton accepts.

Linear Weak Alternating Automata

Linear Weak Alternating Automata (LWAA) are a further sub-class of Weak Alternating Automata. Intuitively, an LWAA is an ABA where the only cycles are self-loops. LWAA may also be refered to as Very Weak Alternating Automata.

Definition 1.4.6 *[79] A Linear Weak Alternating Büchi Automaton (LWAA) is an ABA* $A = (\Sigma, S, S^0, \delta, F)$ *such that the relation* $\preceq_a$ *is a partial order. We judge* $q' \preceq_a q$ *iff* $q \rightarrow^* q'$*, that is* q' *is reachable from* q*.*

Alternating Automata and Linear Time Logics

We present the following equivalences between alternating automata and linear time logics.

Proposition 1.4.2 *[49]*

- *For every* $\phi \in \mu TL$ *there is a WABA* A_ϕ *with* $\mathcal{L}(A_\phi) = \mathcal{L}(\phi)$*.*
- *For every WABA* A *there is a* $\phi_A \in \mu TL$ *with* $\mathcal{L}(\phi_A) = \mathcal{L}(A)$*.*

Similarly, for LTL:

Proposition 1.4.3

- *For every* $\phi \in LTL$ *there is an LWAA* A_ϕ *with* $\mathcal{L}(A_\phi) = \mathcal{L}(\phi)$ *[19].*
- *For every LWAA* A *there is a* $\phi_A \in LTL$ *with* $\mathcal{L}(\phi_A) = \mathcal{L}(A)$ *[16, 32].*

1.5 Summary

In this chapter we have described the basic model checking problem. We have discussed formalisms of both programs and specifications for non-terminating systems. In particular, we have argued for the study of Linear Temporal Logic. We have also discussed some of the tools for approaching LTL model checking. These include automata and their relationship with LTL and μTL.

In the next chapter we discuss model checking of LTL in more detail, explaining some algorithms that have been proposed in the literature. We also describe our own work on efficient LTL model checking and discuss further research directions in this field.

Chapter 2

Model Checking Algorithms

In this chapter we present several LTL model checking algorithms. We begin with the basic algorithm from which the more sophisticated procedures are derived. We discuss several problems that an efficient model checker must contend with. We then describe the symbolic approach to model checking using BDDs and a SAT-based methods that put a bound on the length of possible counter examples.

We go on to survey two particular model checking algorithms introduced by Hammer, Knapp and Merz [47] and McMillan [43]. We finish by describing on-going joint work with William Blum that combines these two algorithms in the search for greater efficiency.

2.1 The Basic Model Checking Algorithm

Proposition 1.4.1 states that every LTL assertion ϕ has an equivalent Büchi automaton A_ϕ with a number of states that is exponential in the size of ϕ. Given a Kripke Structure S and an LTL assertion ϕ, model checking S reduces to the following problem: $\mathcal{L}(S) \cap \mathcal{L}(A_{\neg\phi}) \neq \emptyset$. That is, if a run of S is also a run of $A_{\neg\phi}$ then it is a run that violates the specification ϕ. If such a run exists, then S contains an error.

To check $\mathcal{L}(S) \cap \mathcal{L}(A_{\neg\phi}) \neq \emptyset$ we construct a Büchi automaton that is the product $S \times A_{\neg\phi}$ of S and $A_{\neg\phi}$. Since $\mathcal{L}(S \times A_{\neg\phi}) = \mathcal{L}(S) \cap \mathcal{L}(A_{\neg\phi})$ the model checking problem becomes an emptiness test for $S \times A_{\neg\phi}$ — that is, can we find an accepting run of $S \times A_{\neg\phi}$.

Definition 2.1.1 *Given an alphabet Σ, an edge-labelled Kripke Structure $K = (\mathcal{Q}, (\rightarrow_a)_{a\in\Sigma}, \mathcal{I})$ and a Büchi automaton $A = (\Sigma, S, S^0, \delta, F)$, we define the product automaton $K \times A = (\Sigma, \mathcal{Q} \times S, I \times S^0, \delta', \mathcal{Q} \times F)$ where $(q', s) \in \delta'((q, s), a)$ iff $q \rightarrow_a q'$ and $s \in \delta(s, a)$ for $q, q' \in \mathcal{Q}, s, s' \in S$ and $a \in \Sigma$.*

In effect, the product construction runs the Kripke Structure and Büchi automaton in parallel. The runs of the automaton are restricted by the runs of

the Kripke structure. Acceptance occurs if the automaton is still able to find an accepting run under these restrictions.

Non-emptiness of a Büchi automaton is decidable in linear time [24, 25]. In particular, the problem is reducible to graph reachability. Any accepting run can be characterised as follows: $s_0 \in S^0 \rightarrow_* s \in F \rightarrow_* s \rightarrow \ldots$. That is, a state s in F is reachable from an initial state s_0, and moreover, s is reachable from itself. We can think of an accepting run as a lasso — a path that loops at the end.

To determine non-emptiness, we deconstruct the graph into nontrivial strongly connected components (a set of vertices such that all vertices are reachable from each other) using a linear time depth-first search (DFS) based algorithm [86]. We then look for a nontrivial SCC that contains a state in F that is reachable from an initial state. This just requires an application of the linear DFS algorithm also given in [86]. Therefore, non-emptiness of a Büchi automaton can be solved using several applications of reachability.

2.2 Model Checking as Reachability

In the last section we discussed the role of reachability analysis in determining non-emptiness of a Büchi automaton. In this section we discuss work by Biere, Artho and Schupan which reduces non-emptiness to reachability directly [3].

A Büchi automaton is non-empty if it contains a lasso with a final state in the loop. There is exactly one state in the lasso where the loop starts (and ends). To encode the presence of a loop as a reachability problem we increase the state space of the automaton so that (conceptually) each state carries a state variable. This state variable is initially null. Non-deterministically the automaton guesses when the start of the loop has been reached, and stores that state in the state variable. We know when we have reached the end of the loop by comparing the current state with the value of its state variable. If they match, we have completed the lasso.

For this lasso to be accepting, we require a final state to appear in the loop. To ensure that this property holds we add another variable to the states of the automaton. This variable is false if we are yet to start the loop, or we have started the loop but have not seen a final state. When we see a final state, the variable is set to true.

Thus, non-emptiness is reduced to reachability of a state s whose state variable is s and whose final variable is true. That is, the end of a lasso with a final state in the loop.

Definition 2.2.1 *Given a Büchi automaton $A = (\Sigma, S, S^0, \delta, F)$ we define the automaton A_R over finite words where $A_R = (\Sigma, S_R, S^0 \times \{\bot\} \times \{false\}, \delta_R, F_R)$. We define $S_R = S \times (S \cup \{\bot\}) \times \{true, false\}$, $F_R = \{(s, s, true) \mid s \in S\}$.*

Finally δ_R is defined as follows ($s, l \in S, f \in \{true, false\}$):

$$\delta_R(a,(s,l,f)) = \begin{cases} \delta(a,s) \times \{s,\perp\} \times \{false\} & \text{if } l = \perp \text{ and } s \notin F \\ \delta(a,s) \times \{s,\perp\} \times \{true\} & \text{if } l = \perp \text{ and } s \in F \\ \delta(a,s) \times \{l\} \times \{f\} & \text{if } l \neq \perp \text{ and } s \notin F \\ \delta(a,s) \times \{l\} \times \{true\} & \text{if } l \neq \perp \text{ and } s \in F \end{cases}$$

A_R accepts iff a state in F_R is reachable from an initial state. Furthermore, A_R has an accepting run iff A has an accepting run. It is easy to see that the size of A_R is quadratic in the size of A.

2.3 The State-Explosion Problem

The primary difficulty in LTL model checking is the state-explosion problem. The translation from LTL to Büchi automata is exponential. As specifications become more complex, the number of states in the automaton grows very large and can quickly become unmanageable. In the next sections we discuss two general methods that have had success in industry for tackling this problem. Both of these methods rely on a symbolic representation of automata. We begin by describing how an automaton can be represented symbolically.

2.3.1 Representing Automata Symbolically

The Stateset and Alphabet

Given an LTL formula of size n, its corresponding Büchi automata may contain $O(2^n)$ states. We can identify these states by numbering them. Given a binary representation of numbers, we will require $log(m)$ boolean variables to represent m states. That is, we will require $O(n)$ boolean variables to represent the states of our Büchi automata.

Similarly, the alphabet Σ of our automata can be represented by a (disjoint) set of boolean variables.

We will write $\tilde{s}$ to denote the assignment to the state variables representing the state $s \in S$. Similarly $\tilde{a}$ for the character $a \in \Sigma$.

The Transition Relation

To represent the transitions of our automata we introduce a transition formula R. The boolean variables of this formula are from the set $\{x \mid x \in var(S)\} \cup \{x' \mid x \in var(S)\} \cup \{a \mid a \in var(\Sigma)\}$, where $var(S)$ and $var(\Sigma)$ are the boolean variables used to represent the stateset and alphabet of the automata. $R(\tilde{a}, \tilde{s}, \tilde{s}')$ is constructed to hold only in case the state s' is reachable by an a transition from s.

Suppose we have a four state automata with $00 \rightarrow_a 01$ and $10 \rightarrow_a 11$. Let the variables $\{x, y\}$ represent a state, and the variable l represent an a when true. The automaton's transition relation is as follows,

$$R = l\overline{x}\overline{y}\overline{x}'y' + lxyx'\overline{y}'$$

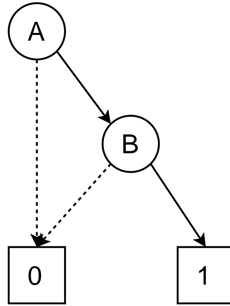


Figure 2.1: A BDD representing the formula $A \wedge B$

We can obviously minimise this formula.

Initial and Final States

We can represent the set of initial states using a similar encoding to that of the transition relation. That is $I(\tilde{s})$ is true iff s is an initial state. Similarly for $F(\tilde{s})$.

2.3.2 Model Checking with BDDs

Binary Decision Diagrams (BDDs) were introduced by Bryant in 1986 [74]. A BDD is a directed acyclic graph. Each node of the graph corresponds to a variable or a valuation. There are at most two nodes corresponding to valuations: 0 and 1. These are leaf nodes. A node that corresponds to a variable has two arcs — one if the variable is valued true, the other if it is valued false. A boolean formula represented as a BDD is evaluated by traversing the graph until a leaf node is reached. This gives a valuation of 0 or 1 — false or true respectively.

Figure 2.1 shows a BDD for the formula AB with the variable ordering $A > B$. If a variable evaluates to true, the opaque arrow is followed. Otherwise, the dotted arrow is followed.

In the worst case a BDD is exponential in size, although in practice they are often more reasonable.

A variable ordering specifies the order in which variables appear in the DAG. A variable higher in the ordering will appear towards the root of the DAG. Equivalent boolean formulae are always represented using a canonical BDD dependent on the variable ordering. In this sense BDDs are minimal representations of formulae, but this depends on the ordering. Finding the best ordering for a particular formula is NP-complete [14] and many algorithms have been proposed to tackle the problem (see [29], for example).

BDDs have been used in symbolic model checking since the late 80s [82, 56] and were used circa 1990 to improve practical limits on the number of

manageable states from 10^6 to 10^{20} [38]. They have since become standard in industry.

BDDs are used in the reachability analysis required for LTL model checking. To compute the set of reachable states we begin with a BDD representing I and a BDD representing R. We compute the set of reachable states by expanding I by an application of R. That is, we combine I and R to form the BDD representing all states reachable initially or after one step. Then we combine the result with R again to form the set of all states reachable within 2 steps, and so on. Eventually we reach a fix-point. If the intersection of this fix-point and F has satisfying assignments, then there are states in F reachable from an initial state.

2.3.3 Bounded Model Checking

A complementary approach to symbolic model checking was introduced by Biere, Cimatti, Clarke and Zhu in 1999 [2]. This approach takes advantage of the successes in SAT solving that allow large boolean formulae to be evaluated in increasingly viable times.

Bounded Model Checking (BMC) uses boolean formulae to encode n step runs of a Kripke Structure. We can then add constraints to this formula that are satisfied iff a run violating the specification is permitted. There are several encodings of the BMC problem, we present some of them below.

Although BMC is complete, we need to check paths that are exponential in length [27]. This length represents the longest simple path between two states, and is referred to as the completeness threshold. In industry BMC is generally used to find bugs, rather than prove correctness. Most bugs in a program usually occur within a relatively small number of steps. Speed-up algorithms have been proposed that more readily permit complete BMC. One such technique (for reachability analysis) is discussed in section 2.5.

The Original Encoding

The original encoding introduced by Biere *et al* takes the following form:

$$[\![M]\!]_k \ \wedge \ [\![\neg\phi]\!]_k$$

where $[\![M]\!]_k$ encodes k steps of the Kripke Structure and $[\![\neg\phi]\!]_k$ encodes violation of ϕ.

If s_i denotes the variables representing the state at timestep i, we write $I(s_0)$ to assert that the first state is initial and $T(s_i, s_{i+1})$ to assert a transition from s_i to s_{i+1}. Runs of length k of a Kripke Structure are encoded as follows:

$$[\![M]\!]_k := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$$

The encoding of $[\![\neg\phi]\!]_k$ is more involved and we defer the reader to the work of Biere et al for the full details [2]. The translation proceeds by considering two

cases — when the path contains a loop and when it does not. The final formula is a disjunction of these two cases expressing that, either we are considering a loop, and the loop encoding of $\neg\phi$ holds, or we are not, and the non-loop encoding of $\neg\phi$ holds.

For example, we present the encoding of $[\![\phi U \psi]\!]_k^i$ in the non-loop case. The superscript i denotes the timestep where the formula must hold. Initially we consider $[\![\neg\phi]\!]_k^0$.

$$[\![\phi U \psi]\!]_k^i := \bigvee_{j=i}^{k} \left([\![\psi]\!]_k^j \wedge \bigwedge_{n=i}^{j-1} [\![\phi]\!]_k^n \right)$$

that is, at some $j \geq i$ ψ holds, and at all steps in between ϕ holds. In the loop case we extend the formula to reflect the looping of time.

The Semantic Encoding

The semantic encoding presented by Clarke *et al* [27] is based on the work of Vardi and Wolper. Rather than encoding $\neg\phi$ directly, we encode runs of $A_{\neg\phi} \times S$. We encode accepting lassos of the resulting automaton as follows:

$$I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{l=0}^{k-1} \left((s_l = s_k) \wedge \bigvee_{j=l}^{k} F(s_j) \right)$$

The final conjunct asserts that there is a loop in the run, and it contains a final state.

The encoding presented above is quadratic in k. However, a linear translation can be achieved by sharing the fairness constraints (For example, $F(s_k)$ is repeated k times). Clarke *et al* argue that this encoding is more efficient than the original encoding and present results that show it requires fewer variables and shorter formulae.

2.4 Truly "on the Fly" LTL Model Checking

A new algorithm for LTL model checking has been introduced by Hammer, Knapp and Merz [47]. Traditional on-the-fly model checking constructs the product automaton $S \times A_{\neg\phi}$ as the algorithm proceeds. This means that we do not have to maintain $S \times A_{\neg\phi}$ in memory, expanding the scope of practically solvable LTL model checking problems. Hammer *et al* take this one step further — they avoid constructing the Büchi automaton by building an LWAA and constructing the product of an LWAA run and the Kripke Structure on the fly. This means that we do not have to store the exponentially sized Büchi automaton. It is for this reason that they call it *Truly "on the Fly" LTL model checking.*

2.4.1 Run DAGs

The alternating automata used by Hammer *et al* are described in definition 1.4.4. That is, the transition relation associates a boolean formula with each state.

Alternating automata permit universal branching. Therefore, a run of an alternating automaton can be considered a tree structure. At each universal branch the run of the automaton splits into several paths. Consequently we can imagine that there are several automata running in parallel, one for each of the separate paths. A more economic encoding recognises that two of these automata may be in the same state, and thus, one of them is redundant. In this way, trees become directed acyclic graphs rather than trees. Each level in the DAG can be considered a configuration $\{q_0, \ldots, q_n\}$ where q_i is the current state of the ith automata, for $i \in 1..n$.

Definition 2.4.1 *Given an alternating automaton $A = (AP, S, S^0, \delta, Acc)$, where Acc is any acceptance condition, and the sequence $\sigma = s_0, s_1, s_2, \ldots$ of assignments to AP, a run DAG of A over σ is a sequence $\Delta = e_0, e_1, e_2, \ldots$ of edges $e_i \subseteq S \times S$. Configurations $c_0, c_1, c_2, \ldots$ of Δ are defined $c_0 = \{q_0\}$ for some $q_0 \in S^0$ and $c_{i+1} = e_i(c_i)$, where $e_i(c_i)$ is the image of $c_i \subseteq S$ under $e_i \subseteq S \times S$. Furthermore, we require that for all i, $dom(e_i) \subseteq c_i$ and for all $q \in c_i$, $s_i \cup e_i(q_i)$ is a model of $\delta(q)$.*

A path is a sequence $\pi = p_0, p_1, p_2, \ldots$ of locations $p_i \in S$ such that $p_0 \in S^0$ and $(p_i, p_i + 1) \in e_i$ for all i. A run DAG is accepting iff $\pi \in Acc$ holds for all infinite paths in Δ.

2.4.2 Co-Büchi LWAA

The particular automata used by Hammer *et al* are LWAA with a co-Büchi acceptance condition. That is, rather than specifying which states must occur infinitely often, we specify which states must only occur finitely often.

Definition 2.4.2 *A co-Büchi LWAA is a tuple $A = (AP, S, s^0, \delta, F)$ where AP is a set of atomic propositions, S is a set of states and $F \subseteq S$, $s^0 \in S$ is the initial state and $\delta : S \to \mathcal{B}(AP \cup S)$ is a transition relation. Locations in S can only appear positively in $\delta(q)$, $q \in S$. The automaton also satisfies the following condition: the relation $\preceq_a$ is a partial order. We judge $q' \preceq_a q$ iff $q \to^* q'$, that is q' is reachable from q.*

Furthermore, $Acc = \{p_0, p_1, \ldots \in S^\omega \mid p_i \in F$ for only finitely many $i\}$

In section 1.4.3 we stated that every LTL formula has an equivalent LWAA with a Büchi acceptance condition. Because of the structure of an LWAA we can see that the two types of acceptance condition are easily interchangeable. Due to their structure, every run of an LWAA will reach a sink state. A Büchi acceptance condition ensures that the sink state (which is the only state that occurs infinitely often) is in the set of final states. Conversely, a co-Büchi acceptance condition requires that the sink state is not a final state. Therefor,

the two acceptance conditions can be exchanged by complementing the set of final states.

Hammer *et al* make use of a co-Büchi rather than Büchi winning condition because it simplifies the conditions under which an LWAA is non-empty (see section 2.4.5). All infinite run DAGs are lassos. If a state does not occur in at least one configuration in the loop, then it follows that this state does not occur infinitely often on any path of the LWAA. However, if a state in F appears in every state of the loop, this does not imply that all paths contain an infinitely recurring state in F, nor does it imply that there is a path whose sink state is in F.

2.4.3 LTL to LWAA

The translation from LTL to co-Büchi LWAA is very simple. We begin with an LTL assertion ϕ in positive normal form. That is, negations are only applied to propositions. Any formula can be translated into positive normal form by "pushing in" the negations using De Morgan laws and $\phi V \psi$ — the dual of $\phi U \psi$.

The states S of our automaton are the sub-formulae of ϕ. The number of states is therefore linear in the size of ϕ. We write q_ψ to denote the state associated with the sub-formula ψ. The initial state is q_ϕ.

The transition relation is defined as follows (a is a literal):

$$\begin{array}{rcl}
\delta(q_a) & := & a \\
\delta(q_{\psi \wedge \chi}) & := & \delta(q_\psi) \ \wedge \ \delta(q_\chi) \\
\delta(q_{\psi \vee \chi}) & := & \delta(q_\psi) \ \vee \ \delta(q_\chi) \\
\delta(q_{\bigcirc \psi}) & := & q_\psi \\
\delta(q_{\psi U \chi}) & := & \delta(q_\chi) \ \vee \ (\delta(q_\psi) \ \wedge \ q_{\psi U \chi}) \\
\delta(q_{\psi V \chi}) & := & \delta(q_\chi) \ \wedge \ (\delta(q_\psi) \ \vee \ q_{\psi U \chi})
\end{array}$$

Finally, we define $F := \{q_{\psi U \chi} \mid \psi U \chi \text{ is a sub-formula of } \phi\}$.

The transition relation for $q_{\psi U \chi}$ makes use of the recursive unfolding of the semantics of the until operator. We require that ψ holds at every timestep from now until χ holds. Therefore, at each timestep we can either choose to satisfy χ, or we can defer for a time step, as long as we can show that ψ holds. By defining F as the set of states corresponding to until formulae, we ensure that deference can only happen a finite number of times. That is, χ eventually holds. The correctness of the translation has been proved by Muller *et al* [19].

It is easy to see that the defined automaton is an LWAA.

2.4.4 Simple LWAA

Hammer *et al* identify a further subset of alternating automata called Simple LWAA.

Definition 2.4.3 *An LWAA $A = (AP, S, s_0, \delta, F)$ is simple if for all $q \in F$, all $q' \in S$, all valuations $s \subseteq AP$, and all $X, Y \subseteq S$ not containing q, we have that if $s \cup X \cup \{q\} \models \delta(q')$ and $s \cup Y \models \delta(q)$, then $s \cup X \cup Y \models \delta(q')$.*

An LWAA is simple if, given a valuation s, whenever a transition from q' goes (universally) to the states in $X \cup \{q\}$, and, under the same valuation we can move from q to the states in Y, then we can avoid q entirely by moving from q' to the states in $X \cup Y$.

We can ensure that the translation of a formula ϕ to an LWAA always produces a simple LWAA by rewriting any sub-formulae of the form $\bigcirc(\psi U \chi)$ to $(\bigcirc\psi) U (\bigcirc\chi)$.

Proposition 2.4.1 *For any LTL formula ϕ that does not contain any sub-formula $\bigcirc(\psi U \chi)$, the automaton A_ϕ is simple.*

This can be seen intuitively. In all cases except $\bigcirc\psi$, states inherit transitions from their sub-formulae. Therefore, by restricting the use of $\bigcirc(\psi U \chi)$ we prevent a state $q_{\psi U \chi}$ $(\in F)$ from having any transitions that are not matched by any state with a transition to $q_{\psi U \chi}$.

2.4.5 Acceptance of a Simple LWAA

The definition of simple LWAA is quite technical. However, it allows us to characterise acceptance without reference to the edges of a configuration DAG. This means that we only need to consider sequences of configurations.

Proposition 2.4.2 *Given a simple LWAA $A = (AP, S, s_0, \delta, F)$, $\mathcal{L}(A) \neq \emptyset$ iff there exists a finite run DAG $\Delta = e_0, e_1, \ldots, e_n$ with configurations $c_0, c_1, \ldots, c_{n+1}$ over a finite sequence of valuations $s_0, s_1, \ldots, s_n$ and some $k \leq n$ such that,*

1. *$c_k = c_{n+1}$, and*
2. *for every $q \in F$, one has $q \notin c_j$ for some $k \leq j \leq n$.*

This can be seen to characterise acceptance as follows. Because A is an LWAA, a state q can only occur infinitely often in a run if it is a terminal state with a self-loop. That is, the run finally settles in the state q. Condition 1 ensures that we have a lasso. Condition 2 asserts that in the loop of the lasso, there is no $q \in F$ that occurs in every configuration. This means that no state in $q \in F$ can occur infinitely often on a path. If q were to occur infinitely often, then it would occur uninterrupted. Because $q \notin c_j$ for some j, it must be the case that it is interrupted, and so it cannot occur infinitely often.

2.4.6 Model Checking with Simple LWAA

The model checking algorithm implemented by Hammer *et al* is an adaptation of Tarjan's algorithm for finding strongly connected components of a graph [75]. The algorithm operates on pairs (s, C) where s is a state of the Kripke Structure and C is a configuration. A depth first search is performed, which produces a tree spanning all reachable states from the initial (s, C). Tarjan shows that if two states are in an SCC, then their nearest common ancestor in the spanning tree is in the same SCC [75]. The algorithm takes advantage of this result to

expand SCCs inductively. At the same time, Hammer *et al* maintain a labelling stating which states of F are missing from at least one configuration in the SCC. If at any point in the algorithm an SCC is found to be labelled with F, then the automaton accepts.

Inevitably, determining non-emptiness of the product of an LWAA and a Kripke Structure is more expensive than determining non-emptiness of the product of a Büchi automaton and a Kripke Structure. This is because the non-emptiness checking with LWAA requires us to (conceptually) construct the Büchi automaton on-the-fly. Fortunately, we save time in the automata construction phase by producing an LWAA rather than a Büchi automaton.

Hammer *et al* have implemented their approach as part of the SPIN model checker [31]. They compare their implementation with SPIN and conclude that, for large LTL formulae, the increase in efficiency gained by avoiding the full Büchi automaton construction more than compensates for the extra time required to determine non-emptiness.

2.5 Interpolation and SAT-based Model Checking

In this section we discuss a speed-up algorithm introduced by McMillan for SAT-based model checking [43]. The method uses results from SAT-based BMC to produce an over approximation of the reachable states of a system. This over approximation is fed back into the algorithm until a fix-point is reached. This fix-point is an over-approximation of all reachable states in the system. Therefore, if the fix-point does not intersect with a bad state, then no bad states are reachable. If a bad state is reached, then the analysis becomes more fine-grained by increasing the bound on the original BMC problem. This process is iterated until an error is found without over-approximation, or the bound reaches the completeness threshold (section 2.3.3).

2.5.1 Interpolation and Over-approximation

For SAT-based model checking we assume that formulae are given in conjunctive normal form. That is, each formula is a set of clauses, where each clause is a set of literals (atomic propositions or their negations). A clause holds for a given model if at least one of its literals does. For a set of clauses to be satisfied, each of its clauses must hold.

Suppose we are given two sets of clauses, (A, B). If $A \cup B$ is unsatisfiable, we can produce a proof of unsatisfiability. From this proof we are able to derive an interpolant P. P has the properties,

- A implies P,
- $P \wedge B$ is unsatisfiable, and
- P only refers to the common variables of A and B.

Suppose we have a set of clauses,

$$\{I(\tilde{s_0}), T(\tilde{s_0}, \tilde{s_1}), T(\tilde{s_1}, \tilde{s_2}), \ldots, T(\tilde{s_{k-1}}, \tilde{s_k}), F(\tilde{s_k})\}$$

that represent an unsatisfiable BMC problem — that is, no error has been found. We observe that the common variables of the sets of clauses $A = \{I(\tilde{s_0}), T(\tilde{s_0}, \tilde{s_1})\}$ and $B = \{T(\tilde{s_1}, \tilde{s_2}), \ldots, T(\tilde{s_{k-1}}, \tilde{s_k}), F(\tilde{s_k})\}$ are those variables that represent the states reachable after one transition. Because it is implied by the initial condition and the first transition, the interpolant P of (A, B) is an over-approximation of the states reachable in one step. By replacing the initial condition with the over-approximation of states reachable in one step or fewer, and iterating the procedure, we can produce an over-approximation of the reachable states of the system.

2.5.2 Interpolation Algorithm

The interpolant P of the clauses (A, B) can be derived from a proof of the unsatisfiability of $A \cup B$.

A proof of unsatisfiability is an inverted (unbalanced) binary tree of clauses, with falsity at the unique leaf (FALSE). Each node corresponds to a clause that is the resolvent of its two predecessors. Two clauses have a resolvent $C_1 \vee C_2$ iff they are of the form $v \vee C_1$ and $\neg v \vee C_2$ and $C_1 \vee C_2$ is not tautological. The pivot variable of the resolution is v.

Because sub-trees may be shared, a proof is a DAG, rather than a tree.

Definition 2.5.1 *A proof of unsatisfiability Π for a set of clauses C is a directed acyclic graph (V_Π, E_Π), where V_Π is a set of clauses such that,*

- *for every vertex $c \in V_\Pi$, either*
  - *$c \in C$ and c is a root, or*
  - *c has exactly two predecessors, c_1 and c_2, such that c is the resolvent of c_1 and c_2, and*
- *the empty clause (falsity) is the unique leaf.*

We say that a variable is local to A if it occurs in A but not B. Conversely, a variable is global if it is shared by A and B. For a clause c, let $g(c)$ be the disjunction of global literals in c. The interpolant of (A, B) given the proof Π is defined below.

Definition 2.5.2 *Let (A, B) be a pair of clause sets, and Π be a proof of unsatisfiability with unique leaf* FALSE. *For all $c \in V_\Pi$ we define $p(c)$ such that,*

- *If c is a root, then*
  - *if $c \in A$ then $p(c) = g(c)$,*
  - *else $p(c)$ is the constant* TRUE.

- *else, let* c_1, c_2 *be the predecessors of* c *and* v *be their pivot variable,*
  - *if* v *is local to* A *then* $p(c) = p(c_1) \vee p(c_2)$*,*
  - *else,* $p(c) = p(c_1) \wedge p(c_2)$*.*

$p(\textsc{False})$ *is an interpolant of* (A, B)*.*

To extract an interpolant of (A, B) from a proof of unsatisfiability we essentially collect all clauses that are used to disprove $A \cup B$ and throw away the literals that refer to variables local to A. We perform any resolutions with a local pivot variable as the removal of local variables will make this resolution impossible at a later stage.

2.5.3 Model Checking with Interpolants

For the purposes of this section, the model checking problem can be phrased as follows. We define an automaton $M = (I, T, F)$ where I, T and F are boolean formulae denoting the initial conditions, the transition relation and the final conditions. The model checking problem is to return TRUE iff a state in F is reachable via the transition relation T from I.

We define,

$$\textsc{Pref}(M) := I(\tilde{s_0}) \wedge T(\tilde{s_0}, \tilde{s_1})$$

and

$$\textsc{Suff}^k(M) := \left(\bigwedge_{1 \leq i < k} T(\tilde{s_i}, \tilde{s_{i+1}}) \right) \wedge \left(\bigvee_{1 \leq i \leq k} F(\tilde{s_i}) \right)$$

The procedure FINITERUN constitutes the core of the algorithm. It takes an automaton and a bound as input and constructs the sets $\textsc{Pref}(M)$ and $\textsc{Suff}^k(M)$ in CNF. A SAT-check is made; if it returns true and we have performed no over-approximations then we have found an accepting run. Otherwise we have found an accepting run that may be spurious, so the procedure aborts and we try again with a larger k. If the SAT-solver returns unsatisfiable, the interpolant is generated. If this interpolant is a fix-point, then we know that there are no accepting runs. If it is not, we expand the set of initial (reachable) states with the states in the interpolant and repeat the procedure.

```
procedure FiniteRun(M = (I,T,F), k > 0)
  if I ∧ F is satisfiable, return True
  let R = I
  while true
    let M' = (R,T,F)
    let A = CNF(Pref(M))
    let B = CNF(Suff^k(M))
    Run SAT on A ∪ B. If satisfiable, then
       if R = I return True else abort
    else (if A ∪ B unsatisfiable)
      let P be an interpolant of (A,B)
      let R = P⟨s̃/s̃_0⟩
      if R' implies R return False
      let R = R ∨ R'
```

The remainder of the algorithm proceeds as follows: choose some bound k and call FiniteRun(M, k). If the result is True or False then this is the final result. If FiniteRun aborts, then increase k and iterate.

The completeness threshhold guarantees that eventually FiniteRun will return either True or False. This is because, at the completeness threshhold the SAT checker will either return "satisfiable" in the case of an accepting run, or "unsatisfiable" if there is no accepting run of length k. Since k is the completeness threshhold, it follows that there are no accepting runs of any number of steps. Therefore, FiniteRun iterates, the result of SAT always being "unsatisfiable" until a fixpoint is reached. It then returns False.We must always reach a fixpoint since there are only a finite number of states, and we always expand R.

In practice, McMillan observes a significant advantage of this interpolation method over several other algorithms. However, he also observes that when a property is false (has a counter-example), interpolation can have its disadvantages. This is because a negative result is final; however, in the case of an accepting run, we must ensure that the result is not spurious by a more fine-grained analysis.

2.6 Combining the Approaches

We now present joint work with William Blum and Luke Ong. In section 2.4 we discussed the on the fly model checking algorithm of Hammer *et al.* This algorithm avoided direct construction of a Büchi automaton by considering run DAGs of an LWAA. They showed that, for large formulae, a BDD based implementation outperformed SPIN. In this section we seek to discover whether similar improvements can be seen when run DAGs are applied to SAT-based model checking.

Hammer *et al* claim limited success with a SAT-based approach utilising run DAGs [84]. In the case when an accepting run exists, their SAT-based

approach found the run very quickly. However, when a run does not exist, the completeness threshhold is too high to obtain a negative result.

McMillan's speed-up method described in section 2.5 shows the opposite pattern: a negative result can be obtained quickly, but a positive result takes more time.

These two methods appear to be complementary. In the positive case, run DAGs may be small enough to ensure quick termination of McMillan's algorithm; and in the negative case, McMillan's algorithm may prevent the need to check up to the completeness threshhold. We therefore propose to use McMillan's speed-up algorithm with run DAGs as its input.

2.6.1 Encoding LWAA as Boolean Formulae

McMillan's method solves the reachability problem, rather than the full model checking problem. Therefore, we use a version of the method described in section 2.2 to reduce model checking to reachability.

Let $A = (AP, S, q_0, \delta, F)$.

Encoding Configurations

A configuration c is a subset of S. We can encode a configuration symbolically using a boolean variable q_i for each state of S. We say q_i is true iff state q_i is in the configuration. Since the size of the LWAA is linear in the size of the original formula $\neg\phi$, we only need a linear number of variables.

Encoding q_0

The initial condition I is simply,

$$I := q_0 \wedge \neg q_1 \wedge \ldots \wedge \neg q_n$$

where $n = |S| - 1$.

Encoding Transitions

$\delta(q)$ is a propositional formula over the variables $AP \cup S$. To represent the next configuration c' (after the transition) we prime the variables q_i – that is q_i'. We define $\delta(q)'$ as $\delta(q)$ with all state variables primed.

We then define,

$$T := (q_0 \rightarrow \delta(q_0)') \wedge \ldots \wedge (q_n \rightarrow \delta(q_n)')$$

That is, if state q is in the current configuration, then the next configuration must contain states prescribed by $\delta(q)$. Note that state variables never occur negatively in $\delta(q)$, and so the next configuration could contain all states. Hammer *et al* observe that adding arbitrary states to a configuration will not allow additional accepting runs — in fact, acceptance will become more difficult [47]. Therefore, the possibility of adding addition states to a configuration does not have an adverse effect on the existence of accepting runs.

Encoding Acceptance

Defining F is more involved. We begin by recalling the acceptance condition for configuration DAGs.

Proposition 2.6.1 *Given a simple LWAA $A = (AP, S, s_0, \delta, F)$, $\mathcal{L}(A) \neq \emptyset$ iff there exists a finite run DAG $\Delta = e_0, e_1, \ldots, e_n$ with configurations $c_0, c_1, \ldots, c_{n+1}$ over a finite sequence of valuations $s_0, s_1, \ldots, s_n$ and some $k \leq n$ such that,*

1. *$c_k = c_{n+1}$, and*
2. *for every $q \in F$, one has $q \notin c_j$ for some $k \leq j \leq n$.*

We need to encode the existence of a loop such that every final state does not appear in at least one configuration within the loop.

We introduce a set of variables $\hat{q}_i$ to represent the configuration at which the loop starts (c_k). We also introduce the variable l which is false if the loop has not begun, and true if we are in the loop. Further variables f_j are also required — one for each state in F — that indicate that a "q-free" configuration has been seen since starting the loop.

At each transition, the current configuration can "nominate" itself as the loop state, by setting $\hat{q}'_i$ accordingly. If it does so, it must also set all f'_j to false, since it has restarted the loop. If the state does not nominate itself it just copies the values of $\hat{q}_i$ across to $\hat{q}'_i$ as well as copying the value of f_j for all j, changing any to true if the corresponding state is not in the current configuration.

We then have,

$$T := \begin{array}{l} (q_0 \rightarrow \delta(q_0)') \wedge \ldots \wedge (q_n \rightarrow \delta(q_n)') \\ \wedge \left[\begin{array}{l} \left(l' \wedge \bigwedge_i (q_i \iff \hat{q}'_i) \wedge \bigwedge_j \neg f'_j\right) \\ \vee \left((l \iff l') \wedge \bigwedge_i (\hat{q}_i \iff \hat{q}'_i) \wedge \bigwedge_j (f_j \vee \neg q_j \iff f'_j)\right) \end{array} \right] \end{array}$$

The first line of T remains unchanged from the original definition. The second line describes the case when the current configuration nominates itself as the loop state. The third line describes the case when the current configuration doesn't change the loop state, but sets f_j for any states that it does not contain.

Finally we define F,

$$F := l \wedge \bigwedge_i (q_i \iff \hat{q}_i) \wedge \bigwedge_j f_j$$

That is, we've reached a loop and the required q-free states have been seen.

Encoding With Fewer Variables

The encoding given above has two variables for each state at each transition. We can do better by non-deterministically choosing the loop state at the beginning and storing it in the variables $\{\hat{q}_0, \ldots, \hat{q}_n\}$. Note that there is only one set of

these variables, not one set per transition. Then we set l as true iff we have started the loop. The variable is initially false and can only be switched to true if it is not already true and the current state is the state indicated by $\{\hat{q}_0, \ldots, \hat{q}_n\}$.

We then have,

$$T := \begin{array}{c} (q_0 \to \delta(q_0)') \wedge \ldots \wedge (q_n \to \delta(q_n)') \\ \wedge \left[\begin{array}{l} \left(\neg l \wedge l' \wedge \bigwedge_i (q_i \iff \hat{q}_i) \wedge \bigwedge_j \neg f'_j\right) \\ \vee \left((l \iff l') \wedge \bigwedge_j (f_j \vee \neg q_j \iff f'_j)\right) \end{array} \right] \end{array}$$

And we define F,

$$F := l \wedge \bigwedge_i (q_i \iff \hat{q}_i) \wedge \bigwedge_j f_j$$

That is, we've started the loop, we've re-reached the loop state and the required q-free states have been seen.

And finally, we redefine I,

$$I := \neg l \wedge q_0 \wedge \neg q_1 \wedge \ldots \wedge \neg q_n$$

Most SAT solvers use resolution-based proof strategies that require CNF input. In appendix A we give an encoding of these formulae in CNF.

2.7 Implementing the Combined Approach

The combined approach detailed above is currently being implemented with William Blum. The work is still very much "in progress" and so we only give a brief overview of the work so far.

2.7.1 Implementation Language

The system is being implemented in OCaml. This is because a functional paradigm aids significantly in the processing of syntactic structures. OCaml also provides a good range of imperative features which will be essential for the efficient implementation of the core algorithms. When OCaml's optimising compiler is used, its performance is comparable with C [90, 53]. It has also been argued that OCaml is only comparable to C when written imperatively [91]. However, the core of the model checking algorithm is the FiniteRun procedure, which shall be coded imperatively.

2.7.2 Specification Language

We will use NuSMV's specification language as our input. This is because SMV is widely used and a good range of benchmarks are available. This will help us compare our implementation with others. Furthermore, we are able to use

a modified version of NuSMV's parser files to generate the input parser for our tool.

The full SMV syntax allows concurrent program modules. However, our input will only consist of (sequential) Kripke Structures. This is because the "flattening" of a concurrent model into a Kripke Structure (required for model checking) is the subject of much research and several algorithms have been developed. In particular, Partial Order Reduction [22]. Instead, we use NuSMV as a state-of-the-art flattening tool so that all of our input will be sequential.

Promela — the input language used by SPIN — was also a candidate input language. However, SPIN does not provide flattening facilities. This means that to produce a competitive implementation, we would need to implement an efficient flattening tool ourselves. This both makes the implementation more difficult and limits the extent to which we can compare our algorithm with others: differences in performance may lie in the effectiveness of the flattening algorithms, rather than the core algorithm.

2.7.3 SAT Solvers

We have chosen MiniSat [54] as our SAT solver. MiniSat has performed exceptionally well in recent SAT solving competitions [92], and, moreover, provides a good Application-Program Interface (API) with proof-logging functionality. This is advantageous for two reasons. Firstly the API means that we do not have to communicate with the solver via the filesystem. Secondly, the proof-logging functionality allows easy construction/extraction of the refutation DAGs required to calculate interpolants.

zChaff [93] is also a competitive SAT-solver with proof logging facilities. However, we choose MiniSat rather than zChaff because zChaff communicates using the file system. This is obviously slower than direct communication through an API. Additionally, the proof-logging provided by zChaff requires the interpretation of a minimised representation of the refutation DAG.

2.7.4 Current Implementation

Currently our implementation takes an LTL formula as input, constructs an LWAA equivalent to the formula and creates its symbolic representation in CNF. It is then possible to send a bounded model checking problem to the SAT solver and print the answer on screen. It is also possible to produce a diagram of the constructed LWAA. Finally, we have implemented the extraction of interpolants from MiniSat and McMillan's reachability algorithm. Consequently we have a complete satisfiability checker for LTL formulae.

To complete our implementation we need to handle Kripke Structures. Additionally, in order for our implementation to be efficient, we will need to optimise the LWAA. We discuss automaton optimisation techniques in the next chapter.

The program flow is given in figure 2.2. Unimplemented modules are shown in grey.

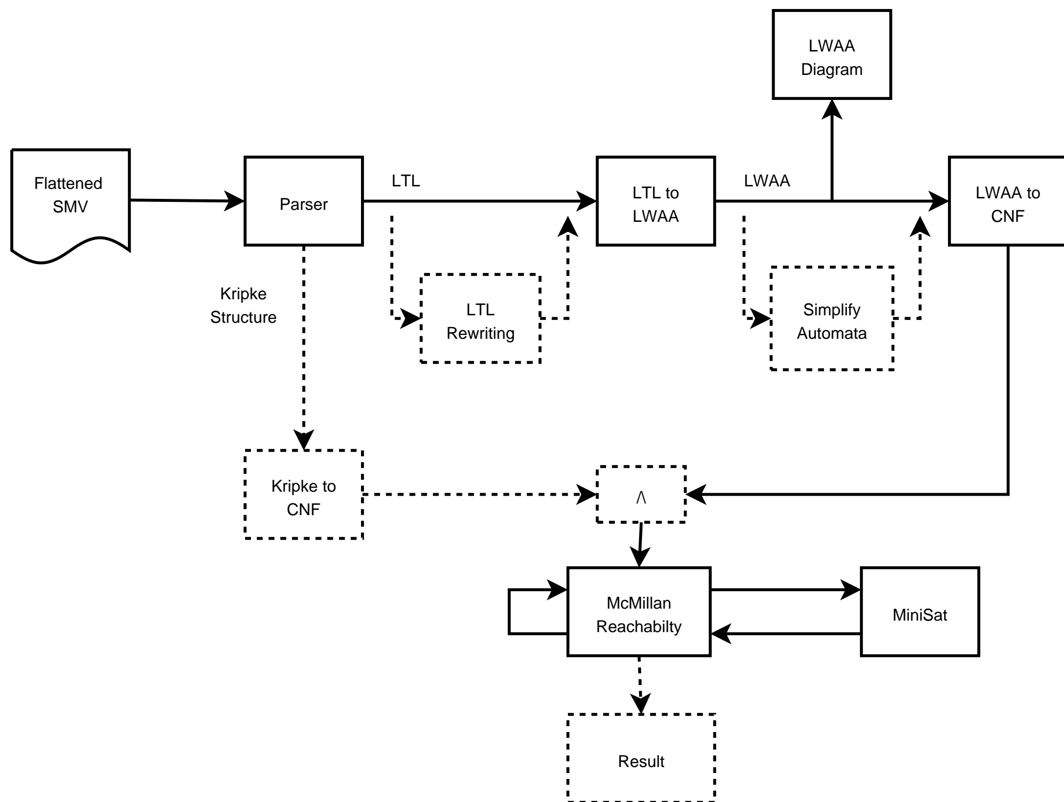


Figure 2.2: The program flow for our model checking tool

Processing of LTL formulae is done using the parser generator Yacc, and a modified version of the parser files distributed with NuSMV. The LTL formulae are translated into LWAA using the algorithm described in section 2.4.3. We store LWAA as a record, containing an array of boolean formulae indexed by integers corresponding to states. These formulae represent the transitions of the automata.

To produce LWAA diagrams the transition formulae are translated into Disjunctive Normal Form. A Line-type definition file (`.dot`) file is then produced. Transitions between states are drawn as hyper-edges. That is, each disjunct has an edge leaving the source state, which splits into several branches representing the conjuncts.

An LWAA is translated into CNF using the encoding described in section 2.6.1 and the CNF representation given in appendix A. This translation initially creates a "template" CNF representation, containing I, T and F. We then produce bounded model checking problems by expanding T to the appropriate length. In this way, existing BMC problems can be extended further. This means that we do not have to recreate the full BMC problem each time the bound k is increased.

To perform BMC on the encoded LWAA, we create a MiniSat proof object and add the CNF clauses to it using the API. Then, using the API, we check whether the clauses are satisfiable — indicating that the LWAA is non-empty.

To extract the interpolants from unsatisfiable BMC instances we create a proof logger object which is attached to the solver. This object collects information on the proof DAG which is then used to construct the interpolant. Because we need to maintain information on which clauses are in the set A and the set B we modified MiniSat to carry a flag with its clauses. We chose to modify MiniSat because all clauses that are added to the solver object are converted to a normal form and simplified. Hence, it is difficult to relate the clauses added to the solver with the clauses used in the proof DAG. By modifying MiniSat we were able to avoid maintaining sets of clauses which must be searched to determine which set a clause belongs to.

2.8 Summary

We have discussed several LTL model checking algorithms and the problems they are designed to overcome. In particular, we discussed the basic model checking algorithm and the state-explosion problem. We then showed how model checking can be reduced to reachability and described the symbolic approach to managing large state-spaces. We gave two of the most popular and complementary symbolic model checking paradigms; namely Binary Decision Diagrams and Bounded Model Checking. We then discussed two state of the art model checking algorithms and finished with a presentation of our own work which aims to overcome the shortcomings of the two algorithms through a hybrid approach.

Chapter 3

Program Synthesis and Games

Critics of model checking argue that we can do more than prove a program correct. If a system has been specified precisely, then we should be able to generate the program automatically — avoiding the costs of separately developing a possibly incorrect system. This is the Synthesis Problem.

Predictably, synthesis is more difficult than model checking and the complexities are such that it remains a largely academic undertaking. However, research continues, with some success, into producing practically viable synthesis algorithms [4]. Research also continues on a more theoretical level.

In this chapter we present a survey of synthesis techniques and paradigms. In particular we describe the synthesis of distributed reactive systems and the use of multiplayer games as a unifying methodology. We present some open problems in the area and look briefly at further models such as knowledge-based or timed systems.

3.1 Program Synthesis

Given a specification, program synthesis is the automatic construction of a design that is guaranteed to be correct. The synthesis problem has as many variants as there are system paradigms. One such distinction lies between *open* and *closed* systems. Classical synthesis has concentrated on closed systems, where both program and user work together to find the required output. In contrast, an open system assumes a hostile environment. A correct design is one where the program is able to handle a user who may throw a spanner in the works. That is, for all possible inputs, the program is able to produce a correct output.

In the open setting the notion of a two-player game arises naturally. The two players being the environment ($\forall$) and the system ($\exists$); the environment wins if the specification is violated and the system wins otherwise. A correct design

can be considered to be a winning strategy for the system. In other words, no matter how the environment acts, the system will not violate the specification.

Reactive systems constitute a further variation of program synthesis. In this model systems are intended to run forever, reacting to user input. For example, a web-server is designed to serve user requests (whatever they may be) and to do so indefinitely.

Finally, reactive systems are frequently distributed — that is, a number of modules working together against a single environment. The notion of a game extends to an analogous multiplayer setting.

3.2 Synthesis and Control

3.2.1 Church's Problem

In 1963 Church published a summary of then recent work concerning mathematics and finite automata [17]. In this paper he described automata, restricted recursive arithmetic and regular languages. He then presented three problems connecting automata and languages: the simplification problem, the decision problem and the synthesis problem.

The decision problem is known as the model checking problem today: given a specification and a proposed solution, determine whether the solution implies the specification. The synthesis problem, which is the focus of this survey, requires that, given a specification, if it is possible to construct an automaton satisfying the specification, that automaton is to be constructed; otherwise a negative answer should be returned. The simplification problem is to find the "simplest" equivalent specification for a given notion of simplicity.

Church summarises several results concerning the synthesis and decision problems that use different fragments of restricted recursive arithmetic or regular properties as their specification language. Church identifies several open problems, the last of which is known as "Church's Solvability Problem" and is described succinctly in [58].

Given an S1S relation $R \subseteq (2^I)^\omega \times (2^O)^\omega$, where I and O are sets of input and output signals respectively, find a function $f : (2^I)^\omega \to (2^O)^\omega$ such that for all $x \in (2^I)^\omega$ we have $R(x, f(x))$. If such a function does not exist, a negative result must be returned.

In 1969, Büchi and Landweber presented the first solution to this problem in the finite-state case [40]. They also link the problem to mathematical games, crediting McNaughton for the initial observation. In the game paradigm the environment (responsible for the inputs) plays against the system (responsible for outputs). The required f is then a winning strategy for the system. Although the notion of games is secondary to their result, Büchi and Landweber note that the game paradigm in this context adds "appealing flavours" to both automata theory and game theory.

However, Büchi and Landweber's solution to Church's Solvability problem is not straightforward and has a high computational complexity. In 1972, Hossley

and Rackoff [76] and Rabin [81] provided new solutions to the problem using tree automata. A winning strategy is a tree whose branches correspond to the different possible inputs and prescribed outputs; therefore, an automaton can be constructed that accepts all trees whose paths ("computations") satisfy the specification. The synthesis problem is then reduced to finding a witness to the non-emptiness of the constructed tree automata.

3.2.2 Reactive Systems

In the early 1980s authors such as Clark and Emerson [28] and Manna and Wolper [89] considered the synthesis problem in a distributed setting. The delay in the consideration of this problem is, in part, attributable to the search for an appropriate specification language [9].

Early methods of synthesis in a concurrent setting extracted the program from a proof that a solution *could* be derived [89, 28]. That is, it is possible for the modules to work together to produce the correct answer. Such an approach does not consider the effect of a hostile environment and, as such, applies to *closed* rather than *open* systems.

In 1985 Pnueli and Harel [21] proposed the reactive/transformational dichotomy: whereas transformational systems work towards a final solution, reactive systems are designed to run forever, reacting to environment input. For example, web servers, operating systems and microwave ovens are all reactive systems. These systems are therefore *open* and Pnueli and Harel used a "black-cactus" analogy — with many inputs and outputs – as opposed to the more traditional "black-box".

Because many concurrent programs fall into the reactive category, the work of Pnueli and Hoel helped establish temporal logic as the principle specification language for both concurrent and reactive systems [9].

In 1989 Pnueli and Rosner [9] provided a solution to the synthesis problem for single processor reactive systems with specifications given in Linear Temporal Logic. In their setting a linear specification $\phi(x, y)$ — where x refers to the input and y to the output at each time step — is augmented to form the branching time formula $(\forall x)(\exists y)A\phi(x, y)$, where "A" means "on all paths". Their solution reduces the problem to a non-emptiness test for tree automata that accepts infinite strategy trees. The procedure is doubly exponential in the size of the specification.

The problem they consider is closely related to Church's Solvability problem. However, their solution is more general than the solutions provided by Büchi *et al* and Rabin in the sense that they allow infinite state solutions rather than only finite state solutions.

3.2.3 Implementations

Because of its high complexity, the method of Pnueli and Rosner has performed poorly in practice. Recently, Harding, Ryan and Schobbens have presented a

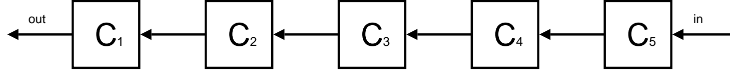


Figure 3.1: A five processor pipeline.

new algorithm that avoids determinising Büchi automata [4]. Although incomplete and still 2EXPTIME-complete, an implementation of their algorithm has shown promising results.

There have also been attempts to find synthesis problems with lower complexities. In [73], Alur and La Torre give complexity results for various fragments of LTL. In the case of LTL with only conjunction and diamond, complexity becomes PSPACE-complete.

3.2.4 Distributed Systems

Pnueli and Rosner have extended their work in [9] to a synchronous distributed setting [11]. In this setting the synthesis problem is to find finite state programs $f_1, \ldots, f_k$ for the architecture built from processes $C_1, \ldots, C_k$ and their interconnection scheme. The joint (synchronous) behaviour of the system must then satisfy the linear temporal logic specification ϕ.

Their first result — derived from the work on multiple person alternation by Peterson and Reif [33] — is that in this distributed model the synthesis problem is undecidable. The proof is given by a reduction of the halting problem. However, in the case of hierarchical architectures such as a strict pipeline (as shown in Figure 3.2.4 for the five processor case) the problem is shown to be decidable in non-elementary time.

Pnueli and Rosner present two different methods for distributed synthesis. The first method involves the synthesis of each processor in turn. The algorithm is recursive and constructs a tree automaton that accepts a strategy for the tail of a pipeline iff there exists a strategy for the head. The second method constructs a single processor strategy which is then decomposed into a distributed strategy. This method uses similar techniques to the first. The decomposition result can be extended to all acyclic architectures, however, it is not complete in the sense that a distributed strategy may exist where a decomposition does not.

3.2.5 Incomplete Information

Until 1997, research into program synthesis had only considered the case of complete information. This is when the program sees all inputs. In the case of incomplete information the environment may execute "hidden moves" and our strategy must operate independently.

Kupferman and Vardi addressed this problem for sequential systems and showed that for LTL, CTL and CTL* specifications incomplete information did

not make the problem harder – that is, the sequential synthesis problems remain 2EXPTIME, EXPTIME and 2EXPTIME-complete respectively [57].

In the case of LTL the problem of incomplete information can be solved by non-deterministically guessing the hidden moves. However, in the branching case we require that nodes of the tree whose input paths co-coincide (i.e. when hidden events are removed from the history) prescribe the same moves. This is a non-regular consistency property, and cannot be evaluated directly using automata-theoretic methods. Kupferman and Vardi in effect sidestep this problem by constructing an automaton that accepts all strategies that, when padded with the possible hidden moves, satisfy the specification.

Kupferman and Vardi extended these techniques in 2001 to solve the synthesis problem in a distributed setting for a CTL* specification, with or without complete information, and with or without communication delay, in a one or two way communication paradigm and in a pipeline or ring architecture [59]. The problems are shown to be of non-elementary complexity.

3.2.6 The Control of Discrete Event Systems

A closely related problem to program synthesis is the control of discrete event systems. This problem was introduced by Ramadge and Wodham at the end of the eighties [70]. The control problem consists of a plant and a specification.

A plant is an automaton that describes all possible sequential behaviours of the system. Behaviours are a sequence of events, some of which are uncontrollable or unobservable. Uncontrollable and unobservable events may correspond to external input and hidden moves of the environment respectively. For example, in a microwave system, button presses may be uncontrollable events and cooking may be controllable. An example of an unobservable event may be a user placing a metallic object into the oven.

The specification is given as a set of admissible behaviours of the system. For example, we may not allow a cook event to occur whilst the microwave door is open. Additionally, we may require that it is always possible to reset the microwave to its initial state.

The controller synthesis problem is then to construct a controller, if one exists, that restricts the plant such that all of its restricted behaviours are admissible. Such a system is called a supervised system. A controller is a map from sequences of observable events to sets of allowed events. Intuitively, the controller presents a selection of possible next moves for any given history of events that can be observed. Uncontrollable events can never be restricted.

3.2.7 Control and Synthesis

In 2000, Kupferman, Madhusudan, Thiagajaran and Vardi presented a reduction of the program synthesis problem to the control problem [60]. This reduction is simply a case of constructing a universal plant which describes all possible sequences of inputs, outputs and hidden events. Inputs are uncontrollable and

hidden events are unobservable. A controller for the plant corresponds to a program strategy for the synthesis problem.

In their paper, Kupferman *et al* consider a more general synthesis problem, where the environment may enable or disable its moves. These are called reactive environments. The solution is by reduction to the control problem, which is solved using automata-theoretic methods. The problem for reactive environments is shown to be 3EXPTIME-complete for CTL* specifications and 2EXPTIME-complete for CTL specification. This means that reactive environments lead to an exponential blow-up over the EXPTIME-complete and 2EXPTIME-complete results for maximal (non-reactive) environments.

Many results from program synthesis can be applied to the control problem. However, whilst distributed synthesis is decidable for hierarchical structures, the result only carries through for pipeline architectures. Madhusudan and Thiagajaran argue that the global nature of the specifications is unreasonably expressive [66]. Instead they consider local specifications and show that the controller synthesis problem is decidable iff the architecture is a clean (or doubly-flanked) pipeline, or a sub-architecture of a clean pipeline. A clean pipeline is similar to a pipeline except that the environment may provide input to both extremal nodes.

3.2.8 Asynchronous Systems

So far we have only discussed distributed synchronous settings. The asynchronous setting was first considered by Pnueli and Rosner [10]. However, they only considered global plants with asynchronous environments and their decision procedure was of non-elementary complexity. Madhusudan and Thiagarajan revisited this problem in 2002 using local plants with local environments who communicate via handshakes [67].

To obtain decidability results Madhusudan and Thiagarajan impose three restrictions on the problem. Firstly, the specification must be *robust*. That is, independent events may occur in any order without affecting satisfaction. Secondly the program strategy can only keep track of the time, rather than the complete history. Finally, at each time step the strategy may only recommend events that involve communication between the same set of processes.

It should be noted that Madhusudan and Thiagarajan impose very few restrictions on the architecture of the system. The model they consider requires deterministic plants and single readers and writers [66]. The authors, however, conjecture that such restrictions can be removed, and, indeed, when non-deterministic plants are allowed, a finite memory may be encoded.

The decision procedure provided runs in doubly exponential time, improving the non-elementary results of Pnueli and Rosner. The algorithm involves creating an automaton that feeds all possible linearisations of program traces into another; this second automaton checks the traces against the specification. Program strategies are those accepted by the resulting automaton.

3.3 Games and Synthesis

In Section 3.2 we gave an overview of the history of program synthesis. The techniques that have been used traditionally vary depending upon the exact nature of the problem being solved. However, in almost all cases automata-theoretic techniques are used. Recent work on synthesis has used game models. In this section we introduce the notion of mathematical games and explain some of the applications they have in program synthesis.

We begin by presenting some of the basic concepts of games and how they relate to program synthesis. We then describe recent attempts to unify the many variants of the synthesis problem using games.

3.3.1 Two Player Games

A game is played between a number of players. In this section we consider two-player games as a metaphor for sequential systems. A program is a strategy for the system in a game played against the environment. A correct program is a winning strategy.

A game consists of a game graph — which describes the possible moves in the game — and a winning condition. A two-player game graph is a tuple $G = (V, V_0, V_1, \gamma)$, where V is a finite or countable set of vertices partitioned into two sets, V_0 and V_1. $\gamma : V \rightarrow 2^V$ assigns to each vertex a set of successors. At each vertex in V_i for $i \in \{0, 1\}$ player i can move. γ describes the vertices that player i can move to. A play of a game graph is a sequence of vertices $v_0, v_1, v_2 \ldots$ such that for all j, $v_{j+1} \in \gamma(v_j)$.

A strategy for player i from a vertex u is a total function f mapping a sequence starting at u and ending in V_i to V. That is, at each vertex where player i can move, f tells him which move to make. If f depends only on the last vertex of a play, it is a memoryless strategy. A play $v_0, \ldots, v_n$ is played according to a strategy f if, whenever $v_j \in V_i$ for $j \in \{0, \ldots, n-1\}$, then $f(v_0, \ldots, v_j) = v_{j+1}$.

We consider asymmetric games between a protagonist and an antagonist. A game (G, W) consists of a game graph G and a winning condition W. A protagonist's strategy f is winning from a vertex u iff all plays from u played according to f satisfy the winning condition. That is, no matter how the antagonist plays, the protagonist will always win.

Definition 3.3.1 *A LTL game is a pair (G, ϕ) where G is a game graph with a labelling $\mu : V \rightarrow 2^{AP}$ and ϕ is an LTL formulae over the atomic propositions in AP. A strategy f is winning strategy from u if ϕ is satisfied on all plays from u corresponding to f.*

Definition 3.3.2 *A Büchi game is a pair (G, F) where G is a game graph and $F \subseteq V$ is a set of final vertices. The protagonist has a winning strategy f from u if on every play according f contains a vertex in F that occurs infinitely often.*

The synthesis problem can be considered in terms of games. We can view a reactive system as playing a game against its environment. Input from the environment constitutes a move by the antagonist. The system has to respond to the environment's moves in such a way that the specification is never violated. In this sense, a program is a strategy for the protagonist. Moreover, a winning strategy is a correct program — the environment will never be able to find a bug.

A game graph in an LTL game is labelled, and analogies with Kripke Structures are easily seen. In the synthesis problem we can set the game graph to allow all possible moves. The strategy restricts these possibilities to form a coherent program. Recalling the correspondence between LTL and Büchi automata, a Büchi can be seen as a graph resulting from the a Kripke Structure and an automaton being run in parallel with the final set F being derived from automaton's set of final states.

Synthesis, therefore, requires us to find a winning strategy for the protagonist in the game (G, W). In the case of Büchi games, a memoryless winning strategy can be found in quadratic time. LTL games do not always have memoryless winning strategies, and the complexity of finding a winning strategy is 2EXPTIME-complete [9].

The link between two player games and program synthesis was first observed by McNaughton and reported by Büchi and Landweber as an aside to their solution to Church's problem [40]. However, this link was not significantly developed in a sequential setting. In the distributed case, however, there has been more interest in the link between games and synthesis.

3.3.2 Multiplayer Games

In a distributed setting two player games may not provide a suitable model. A more general notion of games has many players rather than just two.

Alternating multi-player games were introduced by Peterson and Reif [33]. In a multiplayer game the players are divided into two teams, $\exists$ and $\forall$. They show that, in general, deciding whether a team has a winning strategy is undecidable. However, they also show that in the case that the game is hierarchical – that is, all resources available to player i are available to player $i-1$ – the problem is non-elementary decidable.

Intuitively, we can interpret $\exists$ as a representation of the system, who has to find a way to win. $\forall$ represents the environment who is trying to defeat the system. The synthesised program must give a correct response for all behaviours of the environment.

This work by Peterson and Reif formed the basis of the initial work by Pnueli and Rosner on distributed synthesis [11]. They showed that for non-hierarchical architectures, the synthesis problem is undecidable, and that the problem is non-elementary decidable for hierarchical architectures. However, the notion of games was not used directly.

3.3.3 Games and Control

In 2002, Arnold, Vincent and Walukiewicz presented a solution to the distributed control problem with partial observation [1]. Their solution involves transforming the specification into a μ-calculus formula whose models are exactly those controllers that satisfy the problem. However, the standard μ-calculus is not expressive enough to formulate the property that the controller cannot take into account unobservable events. To overcome this problem the μ-calculus is extended with a loop proposition ($\circlearrowleft_a$) that holds at state q iff there is a transition from q to q labelled with an a.

The decision procedure for the modal-loop μ-calculus is given via a translation to "loop automata" whose semantics is given in terms of a two player game. It is then shown that an automaton has a model only in case that there is a winning strategy for player 0.

The control problem considered by Walukiewicz *et al* consists of a plant, an automaton giving the global specification and an automaton giving the local specification for each processor. A solution for each processor is found by constructing a controller that satisfies its local specification and satisfies the global specification when "quotiented" by the local specifications for the other processors, any processors that have already been created and the behaviour of the plant. In order to obtain decidability, only one local specification is allowed to use the loop proposition.

3.4 Unification Using Games

In section 3.2 we discussed several synthesis problems and their various solutions. This work has shown that synthesis is a hard and frequently undecidable task. However, because these techniques and proofs are specific to the precise nature of the problem being considered, it is difficult to see what makes synthesis so difficult and draw general conclusions. This was the motivation behind recent work (2003) by Mohalik and Walukiewicz who proposed a game framework for the formalisation of distributed synthesis [85].

We will begin by summarising the main synthesis problems. We will then introduce the games framework of Mohalik and Walukiewicz and describe how to solve the synthesis problem in this setting. Finally we will give an overview of how this framework can encode the various synthesis problems.

3.4.1 A Summary of Distributed Synthesis Problems

The four synthesis problems considered by Mohalik and Walukiewicz are as follows:

- **Pipeline Synthesis** — A pipeline is a sequence of processes $C_1, \ldots, C_n$ each communicating with their neighbour in a single direction. The environment provides input to one end of the pipeline, and output occurs

at the other. Execution is synchronous and proceeds in communication rounds.

A pipeline controller is a tuple $\langle f_1, \ldots, f_n \rangle$ where f_i controls process C_i. The synthesis problem is, given a specification, determine $\langle f_1, \ldots, f_n \rangle$ such that the behaviour of the pipeline meets the specification.

- **Local Specifications** — The synthesis problem with local specifications is similar to the pipeline problem. However, a specification is given for each individual process instead of a single specification for the system as a whole. As a result we are able to consider doubly flanked pipelines. That is, pipelines whose external processes both take input from the environment.

- **Communicating Machines** — In the communicating machines paradigm we fix a set of n processes, each with an alphabet of environment and controllable actions. We assume that no two processes share the same environment actions, but that they may share controllable ones. These common actions provide a method of synchronisation between processes. Each process alternates between taking environment input and producing controllable output.

  To make the problem decidable only *trace closed* specification are considered. That is, independent actions may occur in any order without changing the satisfaction of the specification. We also require that programs only take the clock time into account when choosing their next output, and that at each time step all next actions that the program may perform are shared by the same set of processes.

  The synthesis problem is then to find a controller for each process such that all possible execution paths satisfy the given specification.

- **Discrete Event Systems** — In the control problem for discrete event systems we are given a plant over a set of actions Σ. A plant is a deterministic finite state automaton that identifies every possible sequence of actions that a process may perform. Plants have controllable and unobservable actions. A controller can use the execution history to determine which controllable actions to restrict at a given stage, however, the controller cannot take unobservable actions into account.

  The distributed control problem is, given a plant P and languages $M, N \subseteq \Sigma^*$, find controllers $C_1, \ldots, C_n$ such that,

$$M \subseteq \mathcal{L}(P \times C_1 \times \ldots \times C_n) \subseteq N$$

3.4.2 A Games Model

Mohalik and Walukiewicz introduce the following notion of distributed games. A distributed game is constructed from a number of local games $\langle P, E, T \rangle$ without a winning condition, where P is the set of player positions, E the set of

environment positions, and T the set of game moves. It is assumed that player turns alternate. The distributed game has a global winning condition.

For local games $G_i = \langle O_i, E_i, T_i \rangle$ for $i = 1, \ldots, n$, a distributed game is $\mathcal{G} = \langle P, E, T, Acc \subseteq (E \cup P)^\omega \rangle$, where:

1. $E = E_1 \times \ldots \times E_2$
2. $P = (P_1 \cup E_1) \times \ldots \times (P_n \cup E_n) \setminus E$
3. From a player's position we have $(x_1, \ldots, x_n) \to (x'_1, \ldots, x'_n) \in T$ iff $x_i \to x'_i \in T_i$ for all $x_i \in P_i$ and $x_i = x'_i$ for all $x_i \in E_i$.
4. From an environment position, if we have $(x_1, \ldots, x_n) \to (x'_1, \ldots, x'_n) \in T$, then for every x_i, either $x_i = x'_i$ or $x_i \to x'_i \in T_i$.
5. Acc is any winning condition.

There are several points to notice about distributed games. The first is that the environment does not need to move all positions during its turns, however, a player must always move immediately. In this way, environment moves may be blocked by global restrictions. Because of this we are able to assume that the environment may move to any player's position in the local games, putting any restrictions on moves into the definition of the global game.

A solution to the synthesis problem in this setting is a *distributed strategy.* A distributed strategy is a tuple of local strategies, one for each local subgame. In this sense the players do not work together against the environment, and any communication between them must go through the environment. As a result it is possible that a global strategy may exist where a distributed one does not.

3.4.3 Solving Distributed Games

Two tools are provided by Mohalik and Walukiewicz that enable us to determine a winning strategy for a distributed game. These are DIVIDE and GLUE. The essence of the decision procedure is that we DIVIDE and GLUE the game until we are left with a two player parity game. Decidability follows from results on two-player games.

The DIVIDE operation is predicated on players 0 and n being able to determine the global state of the game from their local view — this property is called i-deterministic, for player i. If this is the case, then, for the game $\mathcal{G}$ built from $\mathcal{G}_0, \ldots, \mathcal{G}_n$, we can build an equivalent game $\tilde{\mathcal{G}}$. This game has one less player and is built by combining $\mathcal{G}_0$ and $\mathcal{G}_n$ to form $\mathcal{G}'$ and then constructing $\tilde{\mathcal{G}}$ from $\mathcal{G}', \mathcal{G}_1, \ldots, \mathcal{G}_{n-1}$. The function $flat$ is defined to translate between positions of the two games; that is $flat((x_0, x_n), x_1, \ldots, x_{n-1}) = (x_0, x_1, \ldots, x_{n-1}, x_n)$. We then have the following theorem.

Theorem 3.4.1 *Let $\mathcal{G}$ be a 0-deterministic and n-deterministic distributed game of $n+1$ players. For every position η of $\mathcal{G}$: there is a distributed winning strategy from η iff there is one from $flat^{-1}(\eta)$ in $\tilde{\mathcal{G}}$.*

The GLUE operation is used to determinise the game for a particular player. The operation is analogous to the standard subset construction method for determinising automata. Mohalik and Walukiewicz identify two sets of conditions — I-gluable and II-gluable — under which the GLUE operation is possible; we omit the details. As shown by the following theorem, gluing a game results in an equivalent game, $\tilde{\mathcal{G}} = \text{GLUE}(\mathcal{G})$.

Theorem 3.4.2 *Let $\mathcal{G}$ be a I-gluable or II-gluable game. There is a distributed winning strategy from a position η in $\mathcal{G}$ iff there is a distributed winning strategy from the position $\tilde{\eta}$ in $\tilde{\mathcal{G}}$.*

We are then able to GLUE a game – to ensure determinism for players 0 and n – before applying DIVIDE, provided all preconditions are met along the way. This process can be repeated until we are left with a two-player game. Then we can apply known techniques to solve the game.

Theorem 3.4.3 ([18, 26, 12]) *Every (two-player) game with regular winning conditions is determined, i.e., every vertex is winning for the player or for the environment. In a parity game a player has a memoryless winning strategy from each of his winning vertices. Similarly for the environment. It is decidable to check if a given vertex of a finite game with a regular winning condition is winning for the player.*

3.4.4 Encoding Distributed Synthesis Problems

We now describe briefly how several synthesis problems can be encoded in this game setting.

- **Pipeline Synthesis** — we assume that the specification is given as an automaton over an alphabet that is the disjoint union of alphabets $A_1, \ldots, A_n$. A_n and A_0 represent the environment inputs and outputs respectively, and A_i is used for communication between processes C_{i+1} and C_i.

  The game is constructed as follows. There are $n + 1$ players; the first controls the automaton – making any non-deterministic choices – while the others represent the n processes. The environment provides the input A_n and "passes on" output from one process to its neighbour and the automaton player. The winning condition is simply a translation of the automaton acceptance condition.

- **Local Specifications** — The encoding for a doubly-flanked pipeline with local specifications is very similar to the encoding of the pipeline synthesis problem. To account for the local specifications we do not have an automaton player. Instead, given the regular languages $L_1, \ldots, L_n$ we define our acceptance condition to be that the projection of player i's inputs and outputs is in L_i.

Additionally, we need to define player one so that the extra input is taken into account. Furthermore, to meet the conditions for *gluability* we encode the deterministic parity automaton that accepts L_1 into player one's game. The winning condition for player one is that the automaton's acceptance condition has been met.

- **Communicating Machines** — A similar technique is used to encode Madhusudan and Thiagajaran's communicating machines model. In this model each process is given as a plant that needs to be controlled. Players one to n are defined to represent these plants, with the appropriate restrictions for decidability as discussed in section 3.4.1.

  Player zero encodes the parity automaton representing the trace closed specification. Because, in this encoding, the environment may decide to block certain processes forever, the automaton maintains a component specifying which processes are active. It can then simply ignore the inactive processes.

  The environment then decides which actions should be executed, to which the active processes must respond. The winning condition is given by the acceptance condition of the specification automata.

- **Discrete Event Systems** — unfortunately Mohalik and Walukiewicz were unable to provide an adequate encoding of Ramadge and Wodham's discrete events systems in their games framework. However, they do provide a presentation of the problem as a game. The details are omitted.

3.5 Team Games

In 2004, Gastin, Lerman and Zeitoun [64] introduced a notion of games for asynchronous systems. This model provides a solution to the distributed control problem that subsumes much of the previous work on program synthesis. In particular, the authors cite [59, 66, 67, 85] and [11]. At the end of the paper it is shown how the work of Mohalik and Walukiewicz — described in section 3.4 — can be encoded in their model.

We begin by describing the notion of games put forward by Gastin *et al*, and then describe how the work of Mohalik and Walukiewicz is subsumed by the approach.

3.5.1 Distributed Systems

Rather than representing each process as a player of a distributed game, Gastin *et al* view a distributed system as a single asynchronous model, where actions are players divided into two teams: the actions of the system and the actions of the environment. An architecture has a finite set of actions (Σ) and a finite set of processes ($\mathcal{P}$). Each action has a read and write domain (R and W) that specify its set of read and write processes.

Previous work that the team games subsume has shown that the ability to observe properties of particular linearisations of process actions often leads to undecidability. Hence, Gastin *et al* allow only trace recognisable specifications. That is, a dependency relation between actions is given, and independent actions may occur in any order without affecting satisfaction. This results in the following restrictions on architectures,

$$\begin{array}{ll} \forall a \in \Sigma, & \emptyset \neq W(a) \subseteq R(a) \\ \forall a, b \in \Sigma, & R(a) \cap W(b) = \emptyset \iff R(b) \cap W(a) = \emptyset \end{array}$$

A distributed game over an architecture is defined via a disjoint set of states for each process and a transition relation per action. Each relation describes a transition from a product of reader process states to a product of writer process states for the action. A global state is a product of a local state for each process.

An action moves play from one global state to another iff the move preserves the local states of all processes not in the write domain of the action, and the transition relation for the action admits the product of local reader states to the product of local writer states.

A play is then a sequence of global states and action labelled transitions between them. The (trace equivalent) winning condition is defined as a set of words over tuples. Each tuple contains the action played and the projection of writer states for that action after the move had taken place.

For program strategies, Gastin *et al* use a slightly different notion of memory than other models. Instead of a history of local states, processes are able to remember information collected from other processes throughout the play. However, it can also be shown that, for all games $\mathcal{G}$ there exists an equivalent game $\mathcal{G}^\mu$ which has memoryless winning strategies.

Proposition 3.5.1 *Let $\mathcal{G}$ be a distributed game and let μ be a distributed memory on $\mathcal{G}$. One can construct a distributed games $\mathcal{G}^\mu$ such that there exists a μ-WDS (Winning Distributed Strategy) for $\mathcal{G}$ iff there exists a WMDS (Winning Memoryless Distributed Strategy) in $\mathcal{G}^\mu$. Moreover, if $\mathcal{G}$ is finite and μ is realised by a finite asynchronous automaton, then $\mathcal{G}^\mu$ is finite.*

Decidability is obtained via a translation into a "global" two-player game $\tilde{\mathcal{G}}$. In this game player 0 mimics a distributed strategy and player 1 "tests" the strategy. At each stage, player 1 can ask player 0 to play a particular action. In this way, all possible linearisations of plays may be investigated. Player 0 can either play the desired action, or refuse. To ensure that player 0 is following a distributed strategy, player 1 can reset the game. In the continuing play, player 0 must act in a manner that is consistent with the previous plays. Additionally, player 1 must act fairly; that is, he may only reset a finite number of times, and he must request that player 0 performs each action an infinite number of times — this ensures that player 0 has the opportunity to act out any strategy. Player 0 wins if the trace induced by the actions he played meet the winning condition of the original game.

Theorem 3.5.1 *The following conditions are equivalent for a distributed game $\mathcal{G}$:*

1. *There exists a WMDS for team* 0 *in the distributed game* $\mathcal{G}$.
2. *There exists a WMS for player* 0 *in the global game* $\tilde{\mathcal{G}}$.
3. *There exists a WS for player* 0 *in the global game* $\tilde{\mathcal{G}}$.

The global game $\tilde{\mathcal{G}}$ can be transformed into a parity game. Known methods can be used from then on.

3.5.2 Encoding Mohalik and Walukiewicz

Gastin *et al* provide a somewhat elegant encoding of the Mohalik and Walukiewicz's distributed games. The idea is to associate an action i with each player i of a Walukiewicz game. The players form one team, whilst the environment forms a team on its own. The set of processes in the team game matches the set of processes in the Walukiewicz game, similarly for the local states of each process. The reader and writer processes of an action i is simply $\{i\}$ and the transition relation for each process is encoded verbatim. The reader and writer sets for the environment are the set of all players and its transitions are all transitions in the Walukiewicz game that move from an environment to a player position.

The main difference between the two notions of games is the memory. Team games allow the memory to contain all information gathered from all processes throughout the history. Walukiewicz games, however, only allow the observation of local states. Consequently, the notion of memory for the corresponding team game needs to be restricted to a local view. With this restriction it follows that a Walukiewicz game is equivalent to its team game encoding. Given a Walukiewicz game $\mathcal{G}$ and its equivalent team game $\bar{\mathcal{G}}$, we have the following result.

Theorem 3.5.2 *The players have a WS in* $\mathcal{G}$ *iff team* 0 *has a local WDS in* $\bar{\mathcal{G}}$.

The absence of a satisfactory encoding of the distributed control problem was a shortcoming of the work of Mohalik and Walukiewicz. Gastin *et al* state briefly at the end of their paper that the control problem [67] has a straightforward translation into team games. This is because transitions are local (in the case of the environment) or the product of local transitions.

3.6 Open Problems

The aim of the Walukiewicz *et al* in presenting a unifying approach to Distributed Synthesis is to identify classes of architecture-specification pairs for which the synthesis problem is decidable. The two mathematical tools, DIVIDE and GLUE, are enough to solve the problems considered. However, further tools may be required for different architecture-specification pairs. Additionally, the

current decidability results are specific to particular problems; we may use the framework to identify more general decidability results.

Presently, the Team Games approach of Gastin *et al* has been shown to be decidable for trace recognisable winning conditions and undecidable for rational winning conditions. Classical winning conditions, such as Büchi, liveness and parity conditions have not been studied in general [63]; however, recent work by Gastin *et al* has shown that for series-parallel systems, *controlled reachability conditions* are decidable. These include reachability and safety conditions as well as recognisable ones [63].

Additionally, the proof of decidability given by Gastin *et al* does not present a clear method of strategy construction. Therefore, a more direct method of constructing program strategies for a specification is an open problem.

Finally, Gastin *et al* identify the distributed synthesis problem for non-co-graph alphabets. Broadly speaking, in a co-graph alphabet, if an action a is dependent on an action b, then b is also dependent on a (the formal definition is more subtle). The problems considered in the team paradigm so far have all used a co-graph model of traces.

In addition to these problems, the work by Alur and La Torre into LTL fragments [73] may be extended to a distributed setting. This work is described in more detail in chapter 5

3.7 Different Models

The synthesis problem has many different flavours. In this section we will briefly discuss the synthesis of timed systems, probabilistic systems and knowledge based systems.

3.7.1 Timed Systems

The controller synthesis problem for a given plant can be augmented with a notion of time. This can be achieved by presenting the plant using the timed automata of Alur and Dill [71, 72]. We can then reduce the synthesis problem, in the sequential setting, to a timed game between two players. We then search for a winning strategy for the controller.

Even in the sequential setting, many problems are undecidable. (See [62] for an introduction to the techniques and decidability results). Decidability can be obtained by imposing restrictions on the specification or by limiting the resources (such as the number of clocks) available to the controller. However, there is very little work available that studies the synthesis of timed systems in a distributed setting.

3.7.2 Probabilistic Systems

In [36] Walukiewicz observes that some distributed communication problems are well-known to only have probabilistic solutions; therefore we may wish to

study randomised strategies in a distributed setting.

Recent work by Luca de Alfaro *et al* [45, 44] considers this problem in a two player setting. In this model, we extend the usual two player game with probabilistic transitions and probabilistic strategies — that is, the next state or move is given by a probability distribution.

3.7.3 Knowledge-Based Specifications

In 1998, van der Meyden and Vardi argued that temporal logics are not enough to specify certain program properties [23]. In particular they identify properties reflecting that a program may not have complete information about its environment. They give the following example, "send an acknowledgement as soon as you *know* that the message has been received". Consequently they study specifications given in a linear temporal logic with epistemic operators.

Automata-based techniques are used to show that the synthesis problem in this setting the single agent case is solvable. In the multi-agent case the problem is decidable in broadcast environments or when the specification is restricted to positive knowledge. Settings including asynchronous observation, branching time or conservative approximations have not been studied [87]. Additionally, there is currently no game interpretation of the knowledge-based synthesis problem.

3.8 Summary

In this chapter we surveyed the synthesis problem, particularly in a distributed reactive setting. We described several variants of the problem for different systems and notions of synthesis. We then described recent work by Mohalik and Walukiewicz [85] and Gastin *et al* [64] which attempts to generalise the problem and its solutions through the use of multiplayer games. We finished with a discussion of some of the open problems in synthesis and an overview of further synthesis paradigms.

Chapter 4

Automata Simplification

In chapter 2 we described several model checking algorithms. Automata are fundamental to these algorithms, and naturally contribute significantly to their complexity. Consequently, optimizing the generated automata can greatly affect the efficiency of a model checking tool.

Similarly, in chapter 3 we discussed the synthesis problem. Automata again played a vital role: a program specification is represented by its equivalent automaton when formulating a synthesis game.

To produce efficient automata, we need a method for determining whether one automata is more "optimal" than another. The most common heuristic used in the literature is the number of states. An automaton with fewer states will require less memory and should be easier to check for non-emptiness. Another notion, introduced in [80], is the degree of non-determinism. The more non-deterministic an automaton is, the more difficult it will be to check all possible runs.

We begin by surveying some simple techniques for simplification. We then discuss a notion of "simulation" — when one state can in some sense "mimic" another. We finish with some of our own work and suggestions for future research on simulation, particularly with respect to LWAA.

4.1 Minimizing LTL Formulae

One simple heuristic for producing smaller automata is to start with a smaller LTL formula. Therefore, most model checkers will attempt to minimize the given specification. This is done by the repeated application of a number of rewrite rules, surveyed in appendix B.

In [20], the LTL syntax used when applying the rewrite rules does not allow abbreviations (F, G, etc.). This means, for example, that a rewrite rule for $F\phi$ will also apply to a formula of the form $\top U\phi$. Consequently, there is a potential increase the number of situations where a rewrite rule may apply.

Reducing non-determinism in [80] is also done at the LTL level. These rewrite rules are described in appendix B.

4.2 Constructing Smaller Automata

There are two ways of obtaining a smaller automaton from a (minimized) LTL formula. The first is by optimising the construction algorithm, the second is to work post-hoc, minimizing a given automaton. In this section we overview several methods used in the literature to produce smaller Büchi automata from an LTL formula.

Two elementary simplification techniques are described in [35]. The first is to assign an order (lexicographic, say) to the top-level subformulae ϕ_1, $\phi_2, \ldots$ of a formula constructed with a commutative binary connective (For example, $\phi_1 \wedge \phi_2 \wedge \ldots$). This means we can avoid translating equivalent subformulae of a formula. Although this does not reduce the size of the automata in the worst case, it means that we can avoid repeating possibly expensive minimisation steps on equivalent sub-automata. Secondly we can simplify transition guards – in the case of LWAA or a Büchi automaton over a set of atomic propositions – by applying propositional rewrite rules.

In [30] LTL formulae are translated into Büchi automata using a tableaux method. In this method *elementary subformulae* (constants, atomic propositions, $X\phi$) denote the states of the automaton, and an *elementary cover* is a disjunction of elementary subformulae equivalent to the initial formula. There are infinitely many elementary covers, each resulting in different automata.

The authors propose an optimisation method that poses the selection of the elementary cover (and therefore the number of states) as an Integer Linear Programming problem, and approximates the solution. Because the size of the automata is only a heuristic, the cost of calculating an exact solution may outweigh the benefits of a smaller automaton.

In [20] an extra stage is added to the translation process. Instead of translating directly to Büchi automata, they translate to an intermediate automaton — a Transition Based Generalised Büchi automaton – and then to a Büchi automaton. A transition based automaton operates over the alphabet (q, a, q') — the transitions of an automaton. It is claimed that in a TGBA more states tend to be merged during simplification than in a Büchi automaton.

In [46], Daniele, Giunchiglia and Vardi attempt to produce small automata by detecting redundancies and contradictions as soon as possible during cover generation. It also manages its data structures to aid detection of redundancies that occur when the conjuncts of a conjunction appear in the cover, but the conjunction itself does not. An improved version of this algorithm is suggested in [20].

4.3 Reducing Automata

Once an automaton has been constructed we can analyse its structure and attempt simplification. There are several techniques available, we discuss some of them here. It should be noted that, in some cases, these techniques may be applied during the construction phase.

Several reduction methods are given in [41]. Firstly, if there are no accepting runs leading from a state, then we may remove the state from the automaton. To calculate which states can be removed we compute the strongly connected componants of the automaton and then compute the set of states that cannot reach an SCC that contains an accepting state.

Secondly, if an automaton has an SCC such that there are no exiting transitions, all transitions have the same label, and the the SCC contains an accepting state, then the componant can be replaced by a single accepting state with a reflexive transition.

In [30] several more reduction methods utilising SCCs are presented. Several techniques rely on simulation and will be discussed here. The remaining reduction methods are as follows:

- A state q can be removed from a fair set if it does not appear in any Strongly Connected Componants.
- For an SCC γ, and the fair sets F and F' restricted to only the states in γ (F^γ and F'^γ), if $F^\gamma \subseteq F'^\gamma$ then we can remove F'^γ from F'.
- For a fair set F, if we have $p \in F$ such that all cycles through p meet another state also in F, then we can remove p from F.
- Finally, if a Generalised Büchi Automaton is weak, we can replace the set of fair sets with the set of all states in a fair SCC. (A fair SCC is an SCC that contains a state in a fair set.)

Further simplification techniques are described in [65]. The most basic is that inaccessible states can be removed. We can also remove a transition if it is "implied" by another transition from the same state. One transition implies another if they go to the same state and – assuming the transitions are labelled by propositional formulae – the labelling on the first implies the labelling on the second. Finally, we can merge equivalent states. That is, states with matching transition relations that occur in exactly the same fair sets.

In [35] a number of optimisations are given that are predicated by $\mathcal{L}(\phi_1) \subseteq \mathcal{L}(\phi_2)$. For example, if we have transitions t_1 and t_2 from a given state such that the conditions on t_1 imply those on t_2 and the accepted language from the next states of t_1 is a subset of the accepted language from the next states of t_2, then we can remove t_1.

In general, checking language containment is expensive. Some simple cases of language containment are given in [35] as well as some optimisation techniques. However, as we will see in the following sections, simulation implies language

containment and is easily checkable (polynomial time). Therefore, implifications based on language containment may prove complimentary to simulation-based reductions.

4.4 Simulation

The notion of simulation is widespread in computing. Intuitively, a system simulates another if it can mimic every move it makes. In the case of automata, simulation is easy to check and implies language containment. If an automaton contains two states q_1, q_2, and the automaton starting from q_1 simulates the automaton starting from q_2, and vice versa, then the language accepted from these two states is equivalent and the states can be merged. This provides a useful method for reducing large state spaces – an important task in automata-based verification.

Simulation for Büchi automata has been studied in [42]. This work has been extended by Fritz and Wilke to alternating Büchi automata [15]. Since the alternating case is more relevent to our work, and to avoid repetition, we will forgo a description of simulation for Büchi automata and concentrate on alternating automata instead. A Büchi automata can be thought of as an alternating Büchi automata constructed entirely from existential states.

4.4.1 Alternating Büchi Automata

Fritz and Wilke use the partition-based definition of alternating automata.

Definition 4.4.1 *[5] An alternating Büchi automaton A is a tuple $(\Sigma, S, s_0, \delta, E, U, F)$. Σ is a finite, non-empty, alphabet. S is a finite set of states, where s_0 is the initial state. $\delta : S \times \Sigma \to 2^S$ is a transition relation. $F \subseteq S$ is a set of final states. $\{E, U\}$ is a partition of S into existential and universal states.*

For an alternating Büchi automaton A and an input word $w \in \Sigma^\omega$, acceptance is defined via the game $G(A, w) = (P, P_0, P_1, p_I, Z, W)$, where,

- $P = S \times \omega, P_1 = U \times \omega, P_2 = E \times \omega$,
- $p_I = (s_I, 0)$,
- $Z = \{((s,i),(s',i+1)) \mid s' \in \delta(s, w(i))\}$, and
- $W = (P^*(F \times \omega))^\omega$.

We refer to player 1 as *Automaton* and player 0 as *Pathfinder*. w is accepted by the automaton A iff *Automaton* has a winning strategy in $G(A, w)$. That is, from an existential state, it must be possible to choose a transition that leads to an accepting path (where a state in F occurs infinitely often). Conversly, from a universal state, it must be the case that an accepting path can be constructed no matter which transition is taken. For $q \in S$ we write $A(q)$ to denote the automaton that is a copy of A, with q as its initial state.

4.4.2 Simulation Games

Simulation is defined in terms of games. The three main types of simulation — direct, delayed and fair – operate over the same game graph, which we present below.

Given automata $A^0 = (\Sigma, S^0, s_I, \delta^0, E^0, U^0, F^0)$ and $A^1 = (\Sigma, S^1, s_I, \delta^1, E^1, U^1, F^1)$, the simulation game $G(A^0, A^1)$ is played between two players: the *Spoiler* and the *Duplicator*. Between them, *Spoiler* and *Duplicator* run the two automata in parallel. *Spoiler*'s task is to push the automata into a situation where A^0 cannot be matched by A^1. In the non-alternating case *Spoiler* moves A^0 and *Duplicator* has to match him in A^1. In the alternating case, the presence of universal states necessitates a more subtle definition.

Consider A^0. From an existential state, the language accepted is the union of all transitions — the automaton can take any path. If *Duplicator* were to choose the next state, he would be able to cherry-pick the transition that he knows can be mimicked in A_1. Conversely *Spoiler* can choose a transition that may lead to a situation that can't be mimicked, should one exist. Since we want to make sure $\mathcal{L}(A^0) \subseteq \mathcal{L}(A^1)$ in all cases (no matter how the situation proceeds), we allow *Spoiler* to choose the transition.

From a universal state in A^0 the situation is reversed. The language accepted is the intersection of all transitions — the automaton must take all paths. It follows that if any of the transitions leads to a run that can be mimicked, then the intersection of all transitions must also lead to a run that can be mimicked. Therefore, we let the *Duplicator* choose the next move.

Similar intuition can be applied in A^1. At an existential state, *Duplicator* chooses the next move since the automaton can take any transition to mimick A^0. From a universal state *Spoiler* chooses the next move since the automaton must take all transitions.

During each round *Spoiler* always moves first. This is because *Spoiler* represents the main restrictions that must be satisfied, whilst *Duplicator* is free, once all restrictions are satisfied, to make the best move possible.

$G(A^0, A^1)$ is played in an possibly infinite number of rounds. At the beginning of each round we are at a state (p, q), where $p \in S^0$ and $q \in S^1$. The round proceeds as follows:

1. *Spoiler* chooses a letter $a \in \Sigma$.
2.
   - If $(p, q) \in E^0 \times E^1$, then *Spoiler* chooses a transition $p' \in \delta^0(p, a)$. *Duplicator* then chooses a transition $q' \in \delta^1(q, a)$.
   - If $(p, q) \in U^0 \times U^1$, then *Spoiler* chooses a transition $q' \in \delta^1(q, a)$. *Duplicator* then chooses a transition $p' \in \delta^0(p, a)$.
   - If $(p, q) \in E^0 \times U^1$, then *Spoiler* chooses a transition $p' \in \delta^0(p, a)$ and a transition $q' \in \delta^1(q, a)$.
   - If $(p, q) \in U^0 \times E^1$, then *Duplicator* chooses a transition $p' \in \delta^0(p, a)$ and a transition $q' \in \delta^1(q, a)$.
3. The next round begins with the pair (p', q').

At the beginning of each round *Spoiler* chooses the input $a \in \Sigma$. This is because automata operate over an input word, and we require that A_1 can simulate A_0 for all inputs. Therefore, *Spoiler* is able to pick an input that refutes simulation, should one exist.

If, at any point, a player is unable to make a move, he loses the game. In the case of an infinite game the winning condition is determined by the type of simulation we are interested in. There are three types of simulation: direct, delayed and fair. Over an infinite play $(p_0, q_0), (p_1, q_1), \ldots$ we have,

Definition 4.4.2 (Direct Simulation (di)) *Duplicator wins if for every i with $p_i \in F^0$, we have $q_i \in F^1$.*

Definition 4.4.3 (Delayed Simulation (de)) *Duplicator wins if for every i with $p_i \in F^0$, there is $j \geq i$ such that $q_j \in F^1$.*

Definition 4.4.4 (Direct Simulation (f)) *Duplicator wins if there are infinitely many j with $q_j \in F^1$ whenever there are infinitely many i with $p_i \in F^0$.*

We say that A_1 x-simulates A_0, for $x \in \{di, de, f\}$ if *Duplicator* has a winning strategy in the game $G(A^0, A^1)$ with the appropriate winning condition. We write $A_0 \leq_x A_1$ to denote that A_1 x-simulates A_0.

Proposition 4.4.1 *For $x \in \{di, de, f\}$, the relation $\leq_x$ is reflexive and transitive (a preorder). Furthermore, the following relationship holds,*

$$\leq_{di} \subset \leq_{de} \subset \leq_f$$

Transitivity is shown via strategy composition. Suppose $A_0 \leq A_1$ and $A_1 \leq A_2$. That is, *Duplicator* has a winning strategy σ_1 in $G(A_0, A_1)$ and a winning strategy σ_2 in $G(A_1, A_2)$. We can define the strategy $\sigma = \sigma_1 \bowtie \sigma_2$ such that σ is a winning strategy in $G(A_0, A_2)$ and hence $A_0 \leq_x A_2$.

Intuitively, the strategy σ is obtained by *Duplicator* playing mock games on $G(A_0, A_1)$ and $G(A_1, A_2)$. He uses σ_1 and σ_2 to determine any moves he has to make in the respective mock games, and determines the *Spoiler puppets*' moves in either game from *Spoiler*'s moves in $G(A_0, A_2)$, or from the moves of A_1 played in $G(A_0, A_1)$ or $G(A_1, A_2)$ (as appropriate). Whenever *Duplicator* has to move in $G(A_0, A_2)$ he looks at the mock games to see how play should proceed.

Finally, we state that simulation implies language containment.

Proposition 4.4.2 *Let $x \in \{di, de, f\}$ and A^0 and A^1 be alternating Büchi automata. If $A^0 \leq_x A^1$, then $\mathcal{L}(A^0) \subseteq \mathcal{L}(A^1)$.*

It is easy to see that A^1 must not get stuck over any word that A^0 does not get stuck over. Also, for all three simulation relations it can be seen that, if a final state occurs infinitely often in A^0, then a final state must also occur infinitely often in A^1. That is, A^1 must accept all words accepted by A^0.

4.4.3 Quotienting Modulo Simulation

We define the equivalence relation $\equiv_x$ where $p \equiv_x q$ (that is $A(p) \equiv_x A(q)$ for automaton A and p, q are states of A) iff $p \leq_x q$ and $q \leq_x p$. Since $\leq_x$ implies language containment we have $\mathcal{L}(p) = \mathcal{L}(q)$. In this sense, p and q are the same. $[p]_{\equiv_x}$ denotes the set of all states $\equiv_x$-equivalent to p. We introduce the notion of quotienting, which merges all simulation equivalent states.

The basic structure of a quotient automata is as follows.

Definition 4.4.5 *Given an alternating Büchi automaton $A = (\Sigma, S, s_0, \delta, E, U, F)$, the quotient $A/\equiv_x$ has the form,*

$$(\Sigma, S/\equiv_x, [s_0]_{\equiv_x}, \delta', E', U', F/\equiv_x)$$

where $M/\equiv_x = \{[q]_{\equiv_x} \mid q \in M\}$ for all $M \subseteq S$ and $[q]_{\equiv_x} = \{q' \in S \mid q \equiv_x q'\}$. The following minimal constraints must also hold:

1. *If $[q]_{\equiv_x} \in \delta'([p]_{\equiv_x}, a)$, then there exists $p' \equiv_x p$ and $q' \equiv_x q$ such that $q \in \delta(p, a)$.*
2. *If $[q]_{\equiv_x} \subseteq E$, then $[q]_{\equiv_x} \in E'$, and*
3. *If $[q]_{\equiv_x} \subseteq U$, then $[q]_{\equiv_x} \in U'$.*

Notice that this is not a complete definition. In the case that $[q]_{\equiv_x}$ is a mixed set — containing both universal and existential states — then it is not clear whether the state is universal or existential. In the case of non-alternating Büchi automata this problem does not arise because all states are existential.

For direct and delayed simulation the definition is completed by introducing notions of x-maximal and x-minimal a-successors of a state q. In the case of fair simulation, we cannot define a quotient which preserves the language of the automaton. This follows from the same result shown for Büchi automata in [42].

We begin by describing x-minimal and x-maximal a-successors of a state q. We then describe the complete definition of a quotient for direct and delayed simulation.

4.4.4 Minimal and Maximal Successors

Given an automaton A, a state q of A and character a, an x-maximal a-successor of q is a state $q' \in \delta(q, a)$ such that for every state $q'' \in \delta(q, a)$ with $q' \leq_x q''$, we have $q'' \leq_x q'$. That is, q' is simulation equivalent to any successor state that can simulate it.

Conversely q' is an x-minimal a-successor of q just in case for all $q'' \in \delta(q, a)$ with $q'' \leq_x$ we have $q' \leq_x q''$. That is, q' is simulated by every successor state that it can simulate.

We define the sets $min_a^x(q)$ and $max_a^x(q)$,

$$\begin{aligned} min_a^x(q) &:= \{q' \in \delta(q, a) \mid q' \text{ is an } x\text{−minimal } a\text{−successor of } q\} \\ max_a^x(q) &:= \{q' \in \delta(q, a) \mid q' \text{ is an } x\text{−maximal } a\text{−successor of } q\} \end{aligned}$$

4.4.5 Direct Simulation and Minimax Quotienting

In the case of direct simulation we define the minimax quotient,

Definition 4.4.6 *An x-minimax quotient of an alternating Büchi automaton A is a quotient whose transition relation is,*

$$\begin{aligned}\delta \quad := \quad & \{([p]_{\equiv_x}, a, [q]_{\equiv_x}) \mid a \in \Sigma, p \in E, q \in max^x_a(p)\} \cup \\ & \{([p]_{\equiv_x}, a, [q]_{\equiv_x}) \mid a \in \Sigma, p \in U, q \in min^x_a(p)\}\end{aligned}$$

A mixed class can be either universal or existential.

That is, we choose maximal successors from an existential state and minimal successors from a universal state. This follows the intuition that, from an existential state, it is the least restrictive paths that dominate acceptance, whereas, from a universal state, it is the most restrictive paths that give acceptance.

The following result justifies the assertion that a mixed class can be existential or universal.

Proposition 4.4.3 *For a mixed class $M \in Q/\equiv_x$ and $a \in \Sigma$,*

$$\begin{aligned}&\{[q]_{\equiv_x} \mid \exists p(p \in M \cap E \ \wedge \ q \in max^x_a(p))\} \\ &\quad = \{[q]_{\equiv_x} \mid \exists p(p \in M \cap U \ \wedge \ q \in min^x_a(p))\}\end{aligned}$$

and these sets are singletons.

That is, all maximal successors of an existential state in M are simulation equivalent to each other, and, furthermore, they are simulation equivalent to all minimal successors of a universal state in M. M, therefore, has one successor: it does not matter if it is universal or existential.

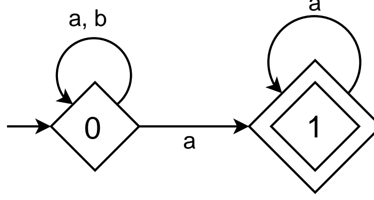
From the definition of direct simulation we have that, if $p \leq_{di} q$ and $p \in F$, then $q \in F$, and so, if $p \equiv_{di} q$ then $p \in F$ iff $q \in F$. It is therefore easy to see that for $q \in S$, $[q]_{\equiv_{di}} \cap F \neq \emptyset$ iff $[q]_{\equiv_{di}} \subseteq F$. It follows that,

Proposition 4.4.4 *Let A be an alternating Büchi automaton and B^m any di-minimax quotient of A.*

1. *For all $p, q \in S$ such that $p \leq_{di} q$, $A(q)$ di-simulates $B^m([p]_{\equiv_{di}})$ and $B^m([q]_{\equiv_{di}})$ di-simulates $A(p)$.*
2. *$A \equiv_{di} B^m$ and $\mathcal{L}(A) = \mathcal{L}(B^m)$.*

4.4.6 Delayed Simulation and Semi-elective Quotienting

For delayed simulation, minimax quotienting does not work. Consider the following automata A, where diamond boxes represent existential states, and double boxes represent states in F.



Notice that $1 \leq_{de} 0$ but not $0 \leq_{de} 1$, therefore $max_a^{de}(0) = \{0\}$ and so, in the minimax quotient automata B, there is no a-transition from 0 to 1. Consequently, it is not the case that $B(1) \leq_{de} B(0)$.

Definition 4.4.7 *A x-semi-elective quotient A_x^s of an alternating Büchi automaton A is a quotient whose transition relation is,*

$$\begin{aligned} \delta \;:=\; & \{([p]_{\equiv_x}, a, [q]_{\equiv_x}) \mid a \in \Sigma, p \in E\} \cup \\ & \{([p]_{\equiv_x}, a, [q]_{\equiv_x}) \mid a \in \Sigma, p \in U, q \in min_a^x(p)\} \end{aligned}$$

A mixed class is defined to be existential.

That is, all mixed classes are declared existential and inherit all transitions from their existential members. Because the maximal successor of the existential states is equivalent to the minimal successor of the universal states, it does not affect acceptance — in the case of direct simulation — if we also inherit the non-maximal successors from the existential states. Inheriting all existential successors does, however, circumvent the problem highlighted in the above example. That is, existential states no longer lose transitions which rendered delayed simulation impossible in the quotient.

Proposition 4.4.5 *For every alternating Büchi automata A, the automata A and A_{de}^s de-simulate each other. In particular $\mathcal{L}(A) = \mathcal{L}(A_{de}^s)$.*

4.4.7 Simulation Algorithms

In this section we give a brief overview of the algorithms used to determine $A(p) \leq_x B(q)$ for some automata A, B and states p, q.

A polynomial time algorithm for determining simulation is given in [42] for Büchi automata and modified slightly in [15] for alternating Büchi automata. The core of the algorithm uses $A(p) \leq_x B(q)$ iff *Duplicator* has a winning strategy in the game $G_x(A(p), B(p))$. The problem of deciding whether *Duplicator* has a winning strategy is reduced to finding winning strategies in parity games.

Definition 4.4.8 *A parity game graph is a tuple $G = (V_0, V_1, E, p)$ where V_0 are the vertices where player 0 can move and V_1 a disjoint set of vertices where player 1 moves. When $V = V_0 \cup V_1$, $E \subseteq V \times V$ are the edges of the graph. $p : V \to \{0, \ldots, d-1\}$ is a mapping assigning a parity to each vertex. A player wins the game if his opposition gets stuck or the lowest infinitely occuring priority in the play is even (for player 0 to win), or odd (for player 1 to win).*

In the case of fair simulation, $G(A(p), B(q))$ has a Büchi winning condition. We can assign priorities to the vertices of $G(A(p), B(q))$ as follows: $p(p, q) = 0$ if $q \in F_B$; $p(p, q) = 1$ if $p \in F_A$ and $q \notin F_B$; and p(p, q) = 2 otherwise.

It is easy to see that player 0 will win in this game iff B hits an infinite number of final states whenever A does. This gives us fair simulation.

For delayed simulation $G(A(p), B(q))$ does not have a Büchi winning condition. We can modify the game by adding a flag b. This flag is set to 1 if we have encountered a final state in A and we are still waiting to encounter one in B. Otherwise it is 0. It is then the case that *Duplicator* wins if b is 0 an infinite number of times. This is a Büchi winning condition and reduction to parity games proceeds as above.

Jurdzinski's algorithm [48] is used to solve the parity games, yielding a complexity $O(n^3 m)$ where n is the number of states and m the number of transitions.

Direct simulation is easier to solve. We reduce to a parity game as described, but, we simplify the problem by taking advantage of the straightforward winning condition. If play encounters a state (p, q) where $p \in F_A$ and $q \notin F_B$, then *Duplicator* loses. Therefore, we remove all transitions that allow *Duplicator* to move to a losing state, and all transitions allowing *Spoiler* to move from states where $p \notin F_A$ and $q \in F_B$. The set of states from which *Duplicator* can win can now be computed using AND/OR graph accessibility, decidable in linear time. This gives a time complexity of $O(nm)$.

In case the alternating Büchi automaton is weak, these complexities can be improved further. Because of the partitioning of a weak alternating automata, any SCC is either accepting or rejecting. Therefore, in the game graph of $G(A, A)$ where A is a weak alternating automata, all positions (p, q) of an SCC have SCCs C_p and C_q in A such that p is in C_p and q is in C_q. Therefore, it is easy to determine for all SCCs from which no other SCC is reachable whether that SCC is winning for *Duplicator*. For example, if $C_p \subseteq F$ and $C_q \subseteq F$, *Duplicator* wins.

The full set of winning positions can be calculated inductively by ordering the SCCs of $G(A, A)$ topologically. SCCs that cannot reach another form the base cases. In the inductive case, computing the winning states for *Duplicator* is AND/OR reachability of a winning SCC.

There are a linear number of SCCs, computable in linear time. AND/OR reachability is also linear time. Therefore, in the weak case, computing the simulation relations $\leq_{di}$, $\leq_{de}$, $\leq_f$ is $O(nm)$.

4.5 Simulation and LWAA

In this section we present our ongoing work into simulation and automata reduction. In chapter 2 we detail work into new LTL model checking algorithms. The first stage in this algorithm is a translation of LTL to LWAA. The remainder of the model checking algorithm is exponential in the size of the LWAA. Therefore, reducing the size of this automaton can lead to significant gains in efficiency.

In this chapter we have discussed several methods for state-space reduction. In particular, we have focussed on simulation. Simulation is complementary to the other reduction algorithms, and has proved successful in experimental results. Direct simulation achieved significant state-space reductions in [30] and in [42] it is shown that delayed simulation can gain greater reductions than direct simulation.

The work of Fritze and Wilke applies simulation to alternating automata. They also show that, in the case of weak alternating automata, the complexity of computing the simulation relations can be reduced to linear time.

In this section we begin to investigate the applicability of simulation to LWAA with a co-Büchi acceptance condition.

We begin by redefining direct, delayed and fair simulation to reflect the co-Büchi acceptance condition. We then discuss possibilties for further research.

4.5.1 Co-Büchi Acceptance Conditions

We define the game $G(A, B)$, for LWAA A and B, as given in section 4.4.2. To define direct, delayed and fair simulation, we define the winning conditions, over the play $(p_0, q_0), (p_1, q_1), \ldots$ as follows:

Definition 4.5.1 (Direct simulation (di))

$$\forall i.p_i \notin F_A \Rightarrow q_i \notin F_B$$

Definition 4.5.2 (Delayed simulation (de))

$$\forall i.p_i \notin F_A \Rightarrow \exists j \geq i.q_j \notin F_B$$

Definition 4.5.3 (Fair simulation (f))

$$(\exists i_0 \forall i \geq i_0.p_i \notin F_A) \Rightarrow (\exists j_0 \forall j \geq j_0.q_j \notin F_B)$$

In appendix C.2 we prove the following relationship: $\leq_{di} \subset \leq_{de} \subset \leq_f$.

Because all runs of an LWAA must end with a single repeating state, we considered an alternative definition of delayed simulation. In this definition we require that if automaton A reaches a repeating state not in F_A, then there must be a later point where automaton B reaches a repeating state not in F_B. We prove in appendix C.6 that this definition of delayed simulation is equivalent to fair simulation.

4.5.2 Further Results

Also, in appendix C we prove the following results:

Corollary 4.5.1 *For $x \in \{di, de, f\}$, $\leq_x$ is a pre-order.*

Property 4.5.1 *Let $x \in \{di, de, f\}$ and A^0 and A^1 be alternating Büchi automata. If $A^0 \leq_x A^1$, then $\mathcal{L}(A^0) \subseteq \mathcal{L}(A^1)$.*

That is, the simulation relation is both transitive and reflexive. Furthermore, simulation implies language containment. The proofs for these properties can be found in appendix C.4 and C.5 respectively.

Finally, we show that we can quotient LWAA. The proofs for these properties can be found in appendices C.10 and C.12.

Theorem 4.5.1 *Let A be an alternating Büchi automaton and B^m any di-minimax quotient of A, $A \equiv_{di} B^m$ and $\mathcal{L}(A) = \mathcal{L}(B^m)$.*

Theorem 4.5.2 *For every LWAA A with a co-Büchi acceptance condition, the automata A and A^s_{de} de-simulate each other. In particular, $\mathcal{L}(A) = \mathcal{L}(A^s_{de})$.*

4.5.3 Further Research

Although the minimax and semi-elective quotients are adequate it may be possible to exploit the linear structure of an LWAA to define more efficient quotients. Also, since a reasonable alternative definition of delayed simulation is equivalent to fair simulation, it is worth investigating whether LWAA admit fair quotients, which may prove more fruitful than delayed simulation in state-space reduction. Finally, we need to redesign the simulation relation calculation algorithms to represent the new co-Büchi acceptance conditions.

A further avenue of research is to relate the work to the second definition of LWAA, where transitions are determined by boolean formulae. This second definition is often more convenient in a practical setting. Hence it is desirable for an implementation of the simulation algorithms to use this representation, rather than the partioned presentation.

We may also study simulation for weak alternating automata. Fritz and Wilke exploit the structure of a weak alternating automata to reduce the complexity of the simulation algorithms. It may also be possible to redefine the notion of a quotient to produce smaller equivalent automata when the original automata is weak.

Finally, the notion of strategy composition suggests a categorical interpretation of automata and simulation. This may provide us with a better understanding of the simulation relation and may also provide a link between two of the main interpretations of games in computing: verification and semantics. This is because we can view two-player verification/synthesis games as alternating automata, with an existential and universal player.

4.6 Summary

In this chapter we reviewed several techniques for automata minimization. These included LTL rewrite rules, algorithms for producing smaller automata and algorithms for reducing a pre-constructed automata. We described state-space reduciton via simulation quotients in detail. Finally, we discussed our own initial work into simulation-based reduction in the particular case of LWAA with a co-Büchi acceptance condition.

Chapter 5

LTL Fragments

In the previous chapters we introduced the LTL model checking and synthesis problems. We discussed several algorithms aimed at solving the model checking problem efficiently. We also discussed automata simplification techniques that can be used in conjunction with these algorithms.

Despite its PSPACE-complexity, LTL model checking has proven feasible in practice. Part of this success can be attributed to the efficiency of the implemented algorithms. However, part of this success stems from the way LTL is used. The high complexity bound is a worst-case scenario. It can be argued that, in practice, we do not require particularly complex LTL specifications.

The desire for a more rigorous understanding of this situation has motivated the study of LTL fragments. In this chapter we overview the different fragments studied in the literature and the complexities of the model checking and synthesis problems. We then discuss a recent NP-complete fragment introduced by Muscholl and Walukiewicz [6]. Finally we discuss possibilities for further research.

5.1 Fragments, Satisfiability and Synthesis

LTL fragments have been well-studied in the literature [7, 79, 83, 73, 6]. In this section we discuss the main definitions of LTL fragments, and their complexities for model checking and synthesis. We then discuss some of the automata that characterise these fragments.

5.1.1 LTL Fragments

There are several main techniques for restricting LTL formulae to produce fragments. The simplest is the removal of certain temporal operators. We may, for example, disallow the until operator U. We may choose to compensate for this restriction by allowing the use of F.

The second technique places restrictions on the temporal nesting depth of a formula. That is, the longest chain of temporal operators that occur within the scope of each other. For example, the formula $\bigcirc\phi$ has a temporal depth of one, whilst the formula $\bigcirc(\bigcirc\phi \vee \bigcirc\psi)$ has a temporal depth of two. In practice, we rarely require a temporal depth of greater than two or three.

A third technique for defining LTL fragments is to limit the number of atomic propositions.

Because there are many ways in which to define a fragment, there are several different notations used in the literature.

Definition 5.1.1 ([83]) *$L_n^k(H_1, \ldots)$ denotes the fragment of LTL with at most n atomic propositions, a temporal height of at most k and only the temporal operators $H_1, \ldots$. n and k may be omitted, or set to ω when there are no restrictions on the number of atomic propositions or temporal nesting depth respectively. Negations are allowed; thus $L(F)$ is equivalent to $L(G)$.*

Definition 5.1.2 ([79]) *$\mathcal{L}(U^m, \bigcirc^n, F^k)$ denotes the fragments of LTL allowing a depth of m nested until operators, n nested tomorrow operators and k nested F operators. We may omit an operator if m, n or k is set to 0 — that is, the operator cannot be used. For example, $\mathcal{L}(F^1)$ allows only F operators without nesting.*

Definition 5.1.3 ([73]) *$LTL_+(op_1, \ldots, op_n)$ is the fragment of LTL built only from atomic propositions and the boolean connectives and temporal operators in $op_1, \ldots, op_n$. $LTL(op_1, \ldots, op_n)$ denotes the fragment of LTL constructed from boolean combinations of the formulae in $LTL_+(op_1, \ldots, op_n)$. For example, the syntax of $LTL_+(\bigcirc, \wedge)$ is ($p \in AP$),*

$$\phi \ := \ p \mid \phi \wedge \psi \mid \ \bigcirc \phi$$

whilst the fragment $LTL(\bigcirc, \wedge)$ has the following grammar ($\chi \in LTL_+(\bigcirc, \wedge)$):

$$\phi \ := \ \chi \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi$$

For the sake of brevity, we do not include a full list of complexities. We summarise some of the results below.

Fragment	Model Checking	Synthesis
$L_\omega^\omega(F)$	NP-complete [7]	—
$L_n^{k+1}(F)$	NL-complete	—
$L_\omega^\omega(U)$	PSPACE-complete [7]	—
$L_n^{k+1}(U)$	NL-complete	—
$L_\omega^\omega(\bigcirc)$	NP-complete	—
$L_\omega^\omega(F, \bigcirc)$	PSPACE-complete	—
$LTL(F, \wedge)$	NP-complete	PSPACE-complete
$LTL(F, \bigcirc, \wedge, \vee)$	PSPACE-complete	EXPSPACE-complete
$LTL(F, G)$	NP-complete	2EXPTIME-complete [39]
LTL	PSPACE-complete	2EXPTIME-complete

5.1.2 Automata

As we have seen, LTL and automata are closely related. For example, the linear time μ-calculus can be characterised by weak alternating automata, and LTL can be characterised by LWAA. It is therefore natural to study the relationship between automata and fragments of LTL. In this section we discuss two different notions of automata used in the literature. The first restricts LWAA just as we restrict the nesting depth of temporal operators in LTL. The second, partially ordered deterministic Büchi automata, are instrumental in proving some of the complexity bounds given above.

Automata Characterisation of LTL Fragments

Pelánek and Strejček have given an automata characterisation of the LTL fragments $\mathcal{L}(U^m, \bigcirc^n, F^k)$ [79]. Naturally, this characterisation is quite subtle, and will require the introduction of several automata properties. In this section we use the definition of LWAA where boolean formulae define the transitions. A transition $p \rightarrow_a S_p$, where $a \in \Sigma$ and $S_p \subseteq S$, is a transition from p on the input character a, which moves to the configuration S_p. $Succ(p)$ denotes the set of successors of p.

Definition 5.1.4 ($\bigcirc$-Free) *Let $p \rightarrow_a S_p$ be a transition of an automaton A. A set $X \subset S_p \setminus \{p\}$ is said to be $\bigcirc$-free for $p \rightarrow_a S_p$ if,*

1. *For each $q \in X$ there is $S_q^{'} \subseteq S$ such that $q \rightarrow_a S_q^{'}$.*
2. *Let $Y \subseteq X$ and for each $q \in Y$ let $S_q^{'} \subseteq S$ be a set satisfying $q \rightarrow S_q^{'}$ and $q \notin S_q^{'}$. Then there exists a set $S'' \subseteq (S_p \setminus Y) \cup \bigcup_{q \in Y} S_q^{'}$ satisfying $p \rightarrow_a S''$.*

The conditions for $\bigcirc$-freeness are similar to the conditions for a simple LWAA. All states in the set Y can in some sense be avoided. There are a-transitions from p to the states in Y, but there is also an a-transition to the set S'', which is disjoint from Y, but may contain some of the a-successors of Y. This is "$\bigcirc$-free" because a transition – or tomorrow step – can be avoided when moving to S''.

Next, we introduce the notion of F- and G-type states. Intuitively, these states correspond to sub-formulae of the form $F\phi$ and $G\phi$.

Definition 5.1.5 (F-type, G-type) *A state p is F-type if there is a transition $p \rightarrow_a \{p\}$ for all $a \in \Sigma$. A state p is G-type if every transition of the form $p \rightarrow_a S$ satisfies $p \in S$.*

The loop-height ($lh(p)$) of a state p counts the maximum number of loops along any path of the automaton. The $\bigcirc$-height ($\bigcirc h(p)$) counts the number of $\bigcirc$ operators that would be required to represent any path of the automata.

Definition 5.1.6 ($lh(p)$, $\bigcirc h(p)$) *The loop-height and* $\bigcirc$*-height of a state* p *are defined inductively,*

$$lp(p) = \begin{cases} max\{lh(q) \mid q \in Succ(p) \setminus \{p\}\} + 1 & \text{if } p \in Succ(p) \\ max\{lh(q) \mid q \in Succ(p) \setminus \{p\}\} & \text{otherwise} \end{cases}$$

$$\bigcirc h(p) = max_{p \to_a S}\{min_{X \text{ is } \bigcirc\text{-free for } p \to_a S}\{need\bigcirc\,(p \to_a, X)\}\}$$

where the maximum over the empty set is 0 *and,*

$$need\bigcirc\,(p \to_a, X) = max(\{\bigcirc h(q) \mid q \in X\} \cup \{\bigcirc h(q) + 1 \mid q \in S \setminus X, q \neq p\})$$

Finally, we define the U-height of an automaton. This is similar to the loop-height of an automaton. Since all loops correspond to U, F or G formulae, the U-height is the number of loops on any path of the automata which do not meet the criteria for F- or G-type states.

Definition 5.1.7 ($Uh(p)$) *The* U*-height* $Uh(p)$ *of a state* p *is defined as,*

$$Uh(p) = \begin{cases} max\{Uh(q) \mid q \in Succ(p) \setminus \{p\}\} + 1 & \text{if } p \in Succ(p) \text{ and} \\ & p \text{ is neither } F- \text{ or } G-type. \\ max\{Uh(q) \mid q \in Succ(p) \setminus \{p\}\} & \text{otherwise} \end{cases}$$

We are now able to characterise automata in terms of their loop-height, $\bigcirc$-height and U-height.

Definition 5.1.8 *Let* $m, n, k \in N \cup \{\omega\}$*,* $LWAA(m, n, k)$ *is the set* $\{\mathcal{L}(A) \mid A$ *is an automaton with initial state* s_0 *and* $Uh(s_0) \leq m$*,* $\bigcirc h(s_0) \leq n$ *and* $lh(s_0) \leq m + k\ \}$*.*

We require that $lh(s_0) \leq m+k$ since loops correspond to U, F or G operators. Therefore, the loop-height will need to be constrained by the number of U operators that can be nested, plus the number of F operators than can be nested. Finally, we state the main result of this section:

Proposition 5.1.1 *For* $m, n, k \in N \cup \{\omega\}$*,* $LWAA(m, n, k) = \mathcal{L}(U^m, \bigcirc^n, F^k)$*.*

Partially Ordered Deterministic Büchi Automata

Alur and La Torre provide an in-depth analysis of the complexities of the model checking and synthesis problems for several fragments of LTL [73]. This work uses Partially Ordered Deterministic Büchi Automata (PODB). A PODB is a deterministic Büchi automata whose transition graph contains no cycles except for self-loops.

An important property of a PODB is its longest distance. That is, the longest distance from a start node to a sink node. Additionally, PODBs are closed under the boolean operations: $A \cap B, A \cup B$ and $\overline{B}$.

Proposition 5.1.2 *For $i = 1, 2$, let A_i be PODBs of size n_i and longest distance d_i. There exists a PODB $A_1 \cap A_2$ (resp. $A_1 \cup A_2$) accepting the language $\mathcal{L}(A_1) \cap \mathcal{L}(A_2)$ (resp. $\mathcal{L}(A_1) \cap \mathcal{L}(A_2)$), whose size is $O(n_1 \cdot n_2)$ and longest distance is not greater than $d_1 + d_2$. Moreover, for $i = 1, 2$, there exists PODB $\overline{A_i}$ of size n_i and longest distance d_i accepting $\Sigma^\omega \setminus \mathcal{L}(A_i)$.*

In section 3.3.1 we described Büchi games and their relationship with the synthesis problem. Alur and La Torre prove the following result:

Proposition 5.1.3 *Given a Büchi game G with n vertices and longest distance d, the game can be solved in space $O(d \cdot log\ n)$.*

This result is used to calculate the complexity of the synthesis problem for several fragments of LTL. For example, every formulae of $LTL(F, \wedge)$ is equivalent to a PODB with an exponential number of vertices and linear longest distance. This yields a polynomial space algorithm for the $LTL(F, \wedge)$ synthesis problem. In fact, synthesis for $LTL(F, \wedge)$ is PSPACE-complete.

5.2 An NP-Complete Fragment of LTL

An NP-complete fragment of LTL, recently introduced by Muscholl and Walukiewicz [6] has the following syntax ($b \in \Sigma$),

$$\phi \ := \ \text{tt} \mid \text{ff} \mid \bigcirc_b \phi \mid F\phi \mid G\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2$$

This fragment of LTL is very similar to the fragment $L(F, \bigcirc)$. However, whilst $L(F, \bigcirc)$ is PSPACE-complete, $subLTL$ is only NP-complete. This improvement in complexity is due to the definition of the tomorrow operator. $subLTL$ uses a restricted form of tomorrow $\bigcirc_b \phi$ that requires $b \in \Sigma$ to be the next character of input.

The proof that $subLTL$ lies in NP is complicated and will not be reproduced here. Instead we refer the reader to the work of Muscholl and Walukiewicz [6]. However, we can see the result intuitively since $L(F, \bigcirc)$ can be encoded in LTL by replacing sub-formulae of the form $\bigcirc \phi$ with the disjunction,

$$\bigvee_{b \in \Sigma} \bigcirc_b \phi$$

This translation causes an exponential blow up in the size of ϕ, bridging the gap between NP-complete and PSPACE-complete complexities.

5.3 Future Work

The exponential improvement in complexity between $\bigcirc \phi$ and $\bigcirc_b \phi$ suggests further avenues for research. We propose to investigate this logic at the automata level. An automata characterisation of this logic may help explain how the full

tomorrow operator contributes to complexity, and why the restriction causes such a significant improvement. Also, the precise complexity of $subLTL$ is unknown. An automata characterisation may provide more precise complexity bounds.

Furthermore, we may investigate whether the complexity improvements carry through to the synthesis problem. It is likely that the work of Alur and La Torre will provide a suitable framework within which this problem may be approached.

Finally, the path model checking problem applies model checking to individual runs of a program. This is useful for pinpointing bugs in an error trace. The path model checking problem is significantly easier than full model checking. For example, LTL model checking over a path can be done in linear time [55]. We may study the complexity of LTL fragments over a path. In particular, the NC hierarchy may be suitable for this purpose.

5.4 Summary

In this chapter we gave an overview of some of the fragments of LTL that have been studied in the literature. We presented several complexity results and described the kinds of automata which can be used to study the fragments. We then described a recently proposed fragment of LTL, $subLTL$. This fragment is NP-complete due to an interesting restriction in the tomorrow operator. Finally we discussed possibilities for future work in $subLTL$, such as an automata characterisation and the complexity of $subLTL$ for program synthesis.

Bibliography

[1] A. Arnold, A. Vincent, and I. Walukiewicz. Games for synthesis of controllers with partial observation. In *Theoretical Computer Science*, volume 303(1), pages 7–34. 2003.

[2] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, London, UK, 1999. Springer-Verlag.

[3] A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking, 2002. A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. In FMICS workshop 2002, 2002.

[4] A. Harding, M. Ryan, and P.-Y. Schobbens. A new algorithm for strategy synthesis in ltl games. In *Proceedings of TACAS'05*. Lecture Notes in Computer Science, Springer, 2005.

[5] A. K. Chandra, D. Kozen, and L. J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.

[6] A. Muscholl and I. Walukiewicz. An NP-complete fragment of LTL. In *Proceedings of DLT'04*, LNCS, 2004. To appear.

[7] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, 1985.

[8] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.

[9] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the Sixteenth ACM Symposium on Principles of Programming Languages*. Austin, January 1989.

[10] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Proceedings of 16th ICALP*, volume 372, pages 652–671. Lecture Notes in Computer Science, 1989.

[11] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proceedings of the 31st IEEE Symposium on Foundations of Computer Science*, pages 746–757, 1990.

[12] A. W. Mostowski. Games with forbidden positions. Technical report, University of Gdansk, 1991.

[13] B. Banieqbal and H. Barringer. Temporal logic with fixed points. In *Temporal Logic in Specification*, pages 62–74, 1987.

[14] B. Bollig and I. Wegener. Improving the variable ordering of obdds is np-complete. *IEEE Trans. Comput.*, 45(9):993–1002, 1996.

[15] C. Fritz and Th. Wilke. Simulation relations for alternating Büchi automata. *Theoretical Computer Science*, 338(1–3):275–314, 2005.

[16] C. Löding and W. Thomas. Alternating automata and logics over infinite words. In *TCS '00: Proceedings of the International Conference IFIP on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics*, pages 521–535, London, UK, 2000. Springer-Verlag.

[17] A. Church. Logic, arithmetic and automata. In *Proceedings of the International Congress of Mathematicians*, pages 23–35, 1962.

[18] D. A. Martin. Borel determinacy. *Ann. of Math. (2)*, 102(2):363–371, 1975.

[19] D. E. Muller, A. Saoudi, and P. E. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In Y. Gurevich, editor, *Proceedings of the Third Annual IEEE Symp. on Logic in Computer Science, LICS 1988*, pages 422–427. IEEE Computer Society Press, July 1988.

[20] D. Giannakopoulou and F. Lerda. Efficient translation of ltl formulae into buechi automata. Research report, Research Insititute for Advanced Computer Science, June 2001.

[21] D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and Models of Concurrent Systems*, pages 477–498. Springer, 1985.

[22] D. Peled. Ten years of partial order reduction. In *CAV*, pages 17–28, 1998.

[23] D. Sangiorgi and R. de Simone, editors. *Synthesis from Knowledge-Based Specifications (Extended Abstract)*, volume 1466 of *Lecture Notes in Computer Science*. Springer, 1998.

[24] E. A. Emerson and C.-L. Lei. Modalities for model checking (extended abstract): branching time strikes back. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96, New York, NY, USA, 1985. ACM Press.

[25] E. A. Emerson and C.-L. Lei. Temporal model checking under generalized fairness constraints. In *Proceedings of the 18th Hawaii International Conference on System Sciences*, North Hollywood, CA, 1985. Western Periodicals Company.

[26] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In *Proceedings of the 32nd annual symposium on Foundations of computer science*, pages 368–377, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[27] E. Clarke, D. Kroening, J. Ouaknine, and Ofer Strichman. Computational challenges in bounded model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(2):174–183, April 2005.

[28] E.M. Clark and E.A. Emerson. Design and synthesis of synchronisation skeletons using branching time temporal logic. In *Proceedings IBM Workshop on Logics of Programs*, volume 131, pages 52–71. Lecture Notes in Computer Science, Springer, 1981.

[29] F. Somenzi. Cudd: Cu decision diagram package release, 1998.

[30] F. Somenzi and R. Bloem. Efficient buchi automata from LTL formulae. In *Computer Aided Verification*, pages 248–263, 2000.

[31] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[32] G. S. Rohde. *Alternating automata and the temporal logic of ordinals.* PhD thesis, 1997. Adviser-Paul E. Schupp.

[33] G.L. Peterson and J.H. Reif. Multiple person alternation. In *Proceedings of the 20th IEEE Symposium on Foundations of Computer Science*, pages 348–363, 1979.

[34] H. K. Buning and T. Letterman. *Propositional Logic: Deduction and Algorithms.* Cambridge University Press, New York, NY, USA, 1999.

[35] H. Tauriainen. On translating linear temporal logic into alternating and nondeterministic automata. Research Report A83, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, December 2003.

[36] I. Walukiewicz. A landscape with games in the background. In *Proceedings of IEEE LICS*, pages 356–366, 2004.

[37] J. A. Brzozowski and E. L. Leiss. On equations for regular languages, finite automata, and sequential networks. *Theor. Comput. Sci.*, 10:19–35, 1980.

[38] J. Burch, E. Clarke, K. Mcmillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[39] J. Marcinkowski and T. Truderung. Optimal complexity bounds for positive ltl games. In *CSL '02: Proceedings of the 16th International Workshop and 11th Annual Conference of the EACSL on Computer Science Logic*, pages 262–275, London, UK, 2002. Springer-Verlag.

[40] J.R. Buchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. In *Transactions of the American Mathematics Society*, volume 138, pages 295–311. 1969.

[41] K. Etessami and G. J. Holzmann. Optimizing buchi automata. In *International Conference on Concurrency Theory*, pages 153–167, 2000.

[42] K. Etessami, T. Wilke, and R. A. Schuller. Fair simulation relations, parity games, and state space reduction for büchi automata. In *ICALP*, pages 694–707, 2001.

[43] K. L. McMillan. Interpolation and sat-based model checking. In *CAV*, pages 1–13, 2003.

[44] L. de Alfaro. Quantitative verification and control via the mu-calculus. In *CONCUR'03, volume 2761 of Lecture Notes in Computer Science*, pages 142–156, 2003.

[45] L. de Alfaro and R. Majumdar. Quantitative solution of omega-regular games. In *STOC'01: 33rd Anual ACM Symposium on Theory of Computer Science*, pages 675–683, 2001.

[46] M. Daniele, F. Giunchiglia, and M. Y. Vardi. Improved automata generation for linear temporal logic. In *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*, pages 249–260, London, UK, 1999. Springer-Verlag.

[47] M. Hammer, A. Knapp, and S. Merz. Truly on-the-fly ltl model checking. In *TACAS*, pages 191–205, 2005.

[48] M. Jurdzinski. Small progress measures for solving parity games. In *STACS '00: Proceedings of the 17th Annual Symposium on Theoretical Aspects of Computer Science*, pages 290–301, London, UK, 2000. Springer-Verlag.

[49] M. Lange. Weak automata for the linear time μ-calculus. In R. Cousot, editor, *Proc. 6th Int. Conf. on Verification, Model Checking and Abstract Interpretation, VMCAI'05*, volume 3385 of *LNCS*, pages 267–281, 2005.

[50] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Banff Higher Order Workshop*, pages 238–266, 1995.

[51] M. Y. Vardi. Branching vs. linear time: Final showdown. *Lecture Notes in Computer Science*, 2031:1–??, 2001.

[52] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 15 1994.

[53] Kevin Murphy. Why ocaml?
http://www.cs.ubc.ca/~murphyk/Software/Ocaml/why_ocaml.html.

[54] N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT*, pages 502–518, 2003.

[55] N. Markey and Ph. Schnoebelen. Model checking a path. In *CONCUR*, pages 248–262, 2003.

[56] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*, pages 365–373, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[57] O. Kupferman and M.Y. Vardi. Synthesis with incomplete informatio. In *2nd International Conference on Temporal Logic*, pages 91–106, 1997.

[58] O. Kupferman and M.Y. Vardi. Church's problem revisited. In *The Bulletin of Symbolic Logic, 5(2)*, pages 245–263. June 1999.

[59] O. Kupferman and M.Y. Vardi. Synthesizing distributed systems. In *Proceedings of the 16th IEEE Syposium on Logic in Computer Science*, July 2001.

[60] O. Kupferman, P. Madhusudan, P.S. Thiagajaran, and M.Y. Vardi. Open systems in reactive environments: Control and synthesis. In *Proceedings of CONCUR'00, 11th International Conference*, volume 1877. Lecture Notes in Computer Science, September 2000.

[61] O. Lichtenstein, A. Pnueli, and L. D. Zuck. The glory of the past. In *Proceedings of the Conference on Logic of Programs*, pages 196–218, London, UK, 1985. Springer-Verlag.

[62] P. Bouyer. Synthesis of timed systems. Talk given at the GAMES Spring School on Infinite Games and Their Applications, Bonn, 15–19 March 2005, http://www.games.rwth-aachen.de/Events/, 2005.

[63] P. Gastin, B. Lerman, and M. Zeitoun. Causal memory distributed games are decidable for series-parallel systems. In *Proceedings of FSTTCS'04*, LNCS, pages 257–286. Springer, 2004.

[64] P. Gastin, B. Lerman, and M. Zeitoun. Distributed games and distributed control for asynchronous systems. In M. Farach-Colton, editor, *Proceedings of the 6th Latin American theoretical Informatics Symposium (LATIN'04)*, volume 2976 of *Lect. Notes in Comp. Science*. Springer, 2004.

[65] P. Gastin and D. Oddoux. Fast ltl to büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, number 2102 in LNCS, pages 53–65, London, UK, 2001. Springer-Verlag.

[66] P. Madhusudan and P.S. Thiagajaran. Distributed control and synthesis for local specifications. In *Proceedings of ICALP'01*, volume 2076. Lecture Notes in Computer Science, July 2001.

[67] P. Madhusudan and P.S. Thiagajaran. A decidable class of asynchronous distributed controllers. In *Proceedings of CONCUR'02*, volume 2421, pages 145–160. Lecture Notes in Computer Science, Springer, 2002.

[68] P. Schnoebelen. The complexity of temporal logic model checking, 2002.

[69] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1/2):72–99, 1983.

[70] P.J.G. Ramadge and W.M. Wonham. The control of discrete event systems. In *Proceedings of the IEEE*, volume 77, 1989.

[71] R. Alur and D. L. Dill. Automata for modeling real-time systems. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*, pages 322–335, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[72] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[73] R. Alur and S. La Torre. Deterministic generators and games for ltl fragments. *ACM Trans. Comput. Logic*, 5(1):1–25, 2004.

[74] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[75] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[76] R. Hossley and C. Rackoff. The emptiness problem for automata on infinite trees. In *Proceedings of the 13th IEEE Symposium on Switching and Automata Theory*, pages 121–124, 1972.

[77] R. Mc Naughton and S. Papert. *Counter-Free Automata*. M.I.T. Press, Cambridge, Mass., 1971.

[78] R. P. Kurshan. Formal verification in a commercial setting. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 258–262, New York, NY, USA, 1997. ACM Press.

[79] R. Pelánek and J. Strejček. Deeper connections between ltl and alternating automata. In *Proc. of Conference on Implementation and Application of Automata(CIAA 2005)*, LNCS. Springer, 2005.

[80] R. Sebastiani and S. Tonetta. "More deterministic" vs. "smaller" Büchi automata for efficient LTL model checking. In *12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'03)*, 2003.

[81] M. Rabin. Automata on infinite objects and church's problem. In *Regional Conference Series in Mathematics*, volume 13. American Mathematics Society, 1972.

[82] S. Bose and A. Fisher. Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic. In *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, 1989.

[83] S. Demri and P. Schnoebelen. The complexity of propositional linear temporal logics in simple cases. *Inf. Comput.*, 174(1):84–103, 2002.

[84] S. Merz and A. Sezgin. Emptiness of linear weak alternating automata. Technical report, LORIA, December 2003.

[85] S. Mohalik and I. Walukiewicz. Distributed games. In *FSTTCS'03*, pages 145–160. Lecture Notes in Computer Science, Springer, 2003.

[86] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[87] T. Wilke. Synthesis from knowledge-based specifications. Talk given at the GAMES Spring School on Infinite Games and Their Applications, Bonn, 15–19 March 2005, `http://www.games.rwth-aachen.de/Events/`, 2005.

[88] W. Thomas. A combinatorial approach to the theory of ω-automata. In *Information and Computation*, volume 48, pages 261–283, 1981.

[89] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. In *ACM Transactions on Programming Language Systems*, volume 6, pages 68–93. 1984.

[90] The computer language shootout benchmarks. `http://shootout.alioth.debian.org/`.

[91] Ocaml vs. c++ for dynamic programming. `http://developers.slashdot.org/article.pl?sid=05/03/14/2258219`.

[92] SAT competition 2005. `http://www.satcompetition.org/`.

[93] zchaff. `http://www.princeton.edu/~chaff/zchaff.html`.

Appendix A

Encoding LWAA in CNF

We present an encoding of LWAA in CNF. We assume that $\delta(q)$ is given in CNF. If $\delta(q)$ is not given in CNF we can use the standard linear translation of a boolean formulae into CNF [34].

In the following, all variables with a bar ($\bar{A}$) are fresh variables introduced during the translation.

$$
\begin{array}{lcl}
[I]_{CNF} & = & I \\
{[T]_{CNF}} & = & \bigwedge_i [q_i \Rightarrow \delta(q_i)']_{CNF} \\
 & & \wedge \left[\begin{array}{l} \left(\neg l \wedge l' \wedge \bigwedge_i (q_i \iff \hat{q}_i) \wedge \bigwedge_j (\neg f'_j)\right) \vee \\ \left((l \iff l') \wedge \bigwedge_j \left(f_j \vee \neg q_j \iff f'_j\right)\right) \end{array} \right]_{CNF} \\
{[F]_{CNF}} & = & l \wedge \bigwedge [q_i \iff \hat{q}_i]_{CNF} \wedge \bigwedge_j f_j
\end{array}
$$

$$
\begin{array}{lcl}
[q_i \Rightarrow \delta(q_i)']_{CNF} & = & (\neg q_i \vee C_1) \wedge \ldots \wedge (\neg q_i \vee C_n) \\
 & & \text{where} \\
 & & \quad \delta(q_i)' = C_1 \wedge \ldots \wedge C_n
\end{array}
$$

$$
\left[\begin{array}{l} \left(\neg l \wedge l' \wedge \bigwedge_i (q_i \iff \hat{q}_i) \wedge \bigwedge_j (\neg f'_j)\right) \vee \\ \left((l \iff l') \wedge \bigwedge_j (f_j \vee \neg q_j \iff f'_j)\right) \end{array} \right]_{CNF} = \begin{array}{l}
(\overline{P}_1 \vee \overline{P}_2) \wedge \\
\left[\overline{P}_1 \Rightarrow \neg l \wedge l' \wedge \bigwedge_i (q_i \iff \hat{q}_i) \wedge \bigwedge_j (\neg f'_j)\right]_{CNF} \wedge \\
\left[\neg l \wedge l' \wedge \bigwedge_i (q_i \iff \hat{q}_i) \wedge \bigwedge_j (\neg f'_j) \Rightarrow \overline{P}_1\right]_{CNF} \wedge \\
\left[\overline{P}_2 \Rightarrow (l \iff l') \wedge \bigwedge_j (f_j \vee \neg q_j \iff f'_j)\right]_{CNF} \wedge \\
\left[(l \iff l') \wedge \bigwedge_j (f_j \vee \neg q_j \iff f'_j) \Rightarrow \overline{P}_2\right]_{CNF}
\end{array}
$$

$$
\left[\overline{P}_1 \Rightarrow \neg l \wedge l' \wedge \bigwedge_i (q_i \iff \hat{q}_i) \wedge \bigwedge_j (\neg f'_j)\right]_{CNF} = \begin{array}{l}
\left[\overline{P}_1 \Rightarrow \neg l\right]_{CNF} \wedge \\
\left[\overline{P}_1 \Rightarrow l'\right]_{CNF} \wedge \\
\bigwedge_i \left[\overline{P}_1 \Rightarrow (q_i \iff \hat{q}_i)\right]_{CNF} \wedge \\
\bigwedge_i \left[\overline{P}_1 \Rightarrow \neg f'_j\right]_{CNF}
\end{array}
$$

$$\left[\neg l \wedge l' \wedge \bigwedge_i (q_i \iff \hat{q}_i) \wedge \bigwedge_j (\neg f'_j) \Rightarrow \overline{P}_1\right]_{CNF} = \begin{array}{l} \left(l \vee \neg l' \vee \bigvee_i (\overline{Q}_i) \vee \bigvee_j (f'_j) \vee \overline{P}_1\right) \wedge \\ \bigwedge_i \left[\overline{Q}_i \iff (\neg q_i \iff \hat{q}_i)\right]_{CNF} \end{array}$$

$$\left[\overline{P}_2 \Rightarrow (l \iff l') \wedge \bigwedge_j (f_j \vee \neg q_j \iff f'_j)\right]_{CNF} = \begin{array}{l} \left[\overline{P}_2 \Rightarrow (l \iff l')\right]_{CNF} \wedge \\ \bigwedge_j \left[\overline{P}_2 \Rightarrow (f_j \vee \neg q_j \iff f'_j)\right]_{CNF} \end{array}$$

$$\left[(l \iff l') \wedge \bigwedge_j (f_j \vee \neg q_j \iff f'_j) \Rightarrow \overline{P}_2\right]_{CNF} = \begin{array}{l} \left(\overline{L} \vee \bigvee_j (\overline{F}_j) \vee \overline{P}_2\right) \wedge \\ \left[\overline{L} \iff (\neg l \iff l')\right]_{CNF} \wedge \\ \bigwedge_J \left[\overline{F}_j \iff (f_j \vee \neg q_j \iff \neg f_j)\right]_{CNF} \end{array}$$

Finally, we present some general translations over the literals, A, B, C and P.

$$\begin{array}{rcl} [A \iff B]_{CNF} & = & (\neg A \vee B) \wedge (\neg B \vee A) \\ [A \iff (B \iff C)]_{CNF} & = & [A \Rightarrow (B \Rightarrow C)]_{CNF} \wedge \\ & & [A \Rightarrow (C \Rightarrow B)]_{CNF} \wedge \\ & & [\neg A \Rightarrow (\neg B \Rightarrow C)]_{CNF} \wedge \\ & & [\neg A \Rightarrow (C \Rightarrow \neg B)]_{CNF} \\ [A \Rightarrow (B \Rightarrow C)]_{CNF} & = & \neg A \vee \neg B \vee C \\ [P \iff (A \vee B \iff C)]_{CNF} & = & [P \Rightarrow (A \vee B \iff C)]_{CNF} \wedge \\ & & [\neg P \Rightarrow (A \vee B \iff \neg C)]_{CNF} \\ [P \Rightarrow (A \vee B \iff C)]_{CNF} & = & (\neg P \vee \neg A \vee C) \wedge \\ & & (\neg P \vee \neg B \vee C) \wedge \\ & & (\neg P \vee \neg C \vee A \vee B) \end{array}$$

Appendix B

LTL Rewrite Rules

B.1 Pure Universality and Pure Eventuality Formulae

In [41], the following rewrite rules are identified:

1. $(\phi U \psi) \wedge (\gamma U \psi) \equiv (\phi \wedge \gamma) U \psi$
2. $(\phi U \psi) \vee (\phi U \delta) \equiv \phi U (\phi \vee \delta)$
3. $\Diamond(\phi U \psi) \equiv \psi$
4. If ψ is a pure eventuality formula, then $(\phi U \psi) \equiv \psi$ and $\Diamond \psi \equiv \psi$
5. If ψ is a pure universality formula, then $(\phi V \psi) \equiv \psi$ and $\Box \psi \equiv \psi$

A pure eventuality formula is:

1. Any formula of the form $\Diamond \phi$
2. If ϕ_1 and ϕ_2 are pure eventuality, then so are $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $\phi_1 U \phi_2$, $\Box \phi_1$, $\phi_1 V \phi_2$ and $X \phi_1$

A pure universality formula is:

1. Any formula of the form $\Box \phi$
2. If ϕ_1 and ϕ_2 are pure universality, then so are $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $\phi_1 U \phi_2$, $\Box \phi_1$, $\phi_1 V \phi_2$ and $X \phi_1$

B.2 Rewrite Rules for LTL Formulae

In [30] a collection of LTL rewrite rules are given, along with some simple cases of $\phi \leq \psi$ (that is, ϕ imples ψ).

$\phi \leq \psi \Rightarrow (\phi \wedge \psi) \equiv \phi$
$\phi \leq \neg\psi \Rightarrow (\phi \wedge \psi) \equiv F$
$(X\phi)U(X\psi) \equiv X(\phi \wedge \psi)$
$(\phi R\psi) \wedge (\phi Rr) \equiv \phi R(\psi \wedge r)$
$(\phi Rr) \vee (\psi Rr) \equiv (\phi \vee \psi)Rr$
$(X\phi) \wedge (X\psi) \equiv X(\phi \wedge \psi)$
$XT \equiv T$
$\phi UF \equiv F$
$\phi \leq \psi \Rightarrow (\phi U\psi) \equiv \psi$
$\neg\psi \leq \phi \Rightarrow (\phi U\psi) \equiv (TU\psi)$

$GF\phi \vee GF\psi \equiv GF(\phi \vee \psi)$
$FX\phi \equiv XF\phi$
$\phi \leq \psi \Rightarrow \phi U(\psi Ur) \equiv \psi Ur$
$GGF\phi \equiv GF\phi$
$FGF\phi \equiv GF\phi$
$XGF\phi \equiv GF\phi$
$F(\phi \wedge GF\psi) \equiv (F\phi) \wedge (GF\psi)$
$G(\phi \vee GF\psi) \equiv (G\phi) \vee (GF\psi)$
$X(\phi \wedge GF\psi) \equiv (X\phi) \wedge (GF\psi)$
$X(\phi \vee GF\psi) \equiv (X\psi) \vee (GF\psi)$

Simple cases of $\phi \leq \psi$:

$\phi \leq \phi$
$\phi \leq T$
$(\phi \leq \psi) \vee (\psi \leq \chi) \Rightarrow (\phi \wedge \psi) \leq \chi$
$(\phi \leq \chi) \vee (\psi \leq \chi) \Rightarrow (\phi \wedge \psi) \leq \chi$

$\chi \leq \psi \Rightarrow \chi \leq (\phi U\psi)$
$(\phi \leq \chi) \wedge (\psi \leq \chi) \Rightarrow (\phi U\psi) \leq \chi$
$(\phi \leq \chi) \wedge (\psi \leq s) \Rightarrow (\phi U\psi) \leq (\chi Us)$

B.3 Propositional Rewrite Rules

Several LTL rewrite rules are given in [20]. Most of these rules have been presented above, so we only present the propositional rewrite rules here.

$p \wedge p \equiv p$
$p \wedge T \equiv p$
$p \wedge F \equiv F$
$p \wedge \neg p \equiv F$
$p \vee p \equiv p$
$p \vee T \equiv T$
$p \vee F \equiv p$
$p \vee \neg p \equiv T$

Appendix C

Simulation Games for LWAA with a co-Büchi Acceptance Condition

Closely following the work of Fritz and Wilke [15] we formally define simulation games for LWAA with a co-Büchi acceptance condition. We show that the three types of simulation are strict subsets of each other, and that an alternative definition of delayed simulation is equivalent to fair simulation. We then define strategy composition, and show transitivity of the simulation relation for direct, delayed and fair simulation. We also show that simulation implies language containment, and the correctness of minimax and semi-elective quotienting for direct and delayed simulation.

The main differences between this chapter and Fritz and Wilke are in the proofs of Lemma C.2.1, Lemma C.4.2, Property C.5.1 and the failure of naive quotienting for direct simulation. We also describe an alternative definition of delayed simulation and show that it is equivalent to fair simulation.

C.1 Simulation Games

Given LWAA $A^0 = (Q^0, \Sigma, p_I, \Delta^0, E^0, U^0, F^0)$ and $A^1 = (Q^1, \Sigma, q_I, \Delta^1, E^1, U^1, F^1)$ we define the simulation game,

$$G^x(A^0, A^1) = (P, P_0, P_1, (p_I, q_I), Z, W^x)$$

where P is a set of game positions, P_0 and P_1 form a disjoint partition of P into player 0 (*Spoiler*) and player 1 (*Duplicator*) positions, (p_I, q_I) is the starting position, Z is the transition relation and W^x the winning condition.

C.1.1 Game Positions

Play proceeds in rounds. At the start of a round the game is at a position (p, q). *Spoiler* chooses a letter $a \in \Sigma$. Then, depending on certain conditions, *Spoiler* or *Duplicator* will make a move in each automata.

The act of *Spoiler* choosing $a \in \Sigma$ moves play to state of the form (p, q, a, S, b, S', b'). That is, the round started at (p, q), *Spoiler* chose a and now player $S \in \{s, d\}$ must move automata $A^b, b \in \{0, 1\}$. Player S's move will be followed by player $S' \in \{s, d\}$ making a responding move in auomaton $A^{b'}, b' \in \{0, 1\}$.

Player S makes a move to a state of the form (p, q, a, S', b') from which player S' makes the remaining move, completing the round.

States are therefore defined by the sets,

$$R = Q^0 \times Q^1$$

$$U_s = Q^0 \times Q^1 \times \Sigma \times \{s\} \times \{0,1\} \times \{s,d\} \times \{0,1\}$$
$$U_d = Q^0 \times Q^1 \times \Sigma \times \{d\} \times \{0,1\} \times \{s,d\} \times \{0,1\}$$

$$V_s = Q^0 \times Q^1 \times \Sigma \times \{s\} \times \{0,1\}$$
$$V_d = Q^0 \times Q^1 \times \Sigma \times \{d\} \times \{0,1\}$$

where,

$$P = R \cup U_s \cup U_d \cup V_s \cup V_d$$
$$P_0 = R \cup U_s \cup V_s$$
$$P_1 = U_d \cup V_d$$

C.1.2 Game Moves

The protocol for each round is as follows: at the beginning of each round we are at a state (p, q), where $p \in S^0$ and $q \in S^1$. The round proceeds as follows:

1. *Spoiler* chooses a letter $a \in \Sigma$.
2.
   - If $(p, q) \in E^0 \times E^1$, then *Spoiler* chooses a transition $p' \in \delta^0(p, a)$. *Duplicator* then chooses a transition $q' \in \delta^1(q, a)$.
   - If $(p, q) \in U^0 \times U^1$, then *Spoiler* chooses a transition $q' \in \delta^1(q, a)$. *Duplicator* then chooses a transition $p' \in \delta^0(p, a)$.
   - If $(p, q) \in E^0 \times U^1$, then *Spoiler* chooses a transition $p' \in \delta^0(p, a)$ and a transition $q' \in \delta^1(q, a)$.
   - If $(p, q) \in U^0 \times E^1$, then *Duplicator* chooses a transition $p' \in \delta^0(p, a)$ and a transition $q' \in \delta^1(q, a)$.
3. The next round begins with the pair (p', q').

More formally, we define $Z \subseteq P \times P$ containing all moves of the form,

$$\begin{array}{ll}
((p,q),(p,q,a,s,0,d,1)) & \text{for } p \in E^0, q \in E^1.a \in \Sigma \\
((p,q),(p,q,a,s,0,s,1)) & \text{for } p \in E^0, q \in U^1.a \in \Sigma \\
((p,q),(p,q,a,d,0,d,1)) & \text{for } p \in U^0, q \in E^1.a \in \Sigma \\
((p,q),(p,q,a,s,1,d,0)) & \text{for } p \in U^0, q \in U^1.a \in \Sigma \\
\\
((p,q,a,x,0,y,1),(p',q,a,y,1)) & \text{for } (p,a,p') \in \Delta^0, x,y \in \{s,d\} \\
((p,q,a,s,1,d,0),(p,q',a,d,0)) & \text{for } (q,a,q') \in \Delta^1 \\
\\
((p,q,a,d,0),(p',q)) & \text{for } (p,a,p') \in \Delta^0 \\
((p,q,a,x,1),(p,q')) & \text{for } (q,a,q') \in \Delta^1
\end{array}$$

C.1.3 Simulation

We say A^1 x-simulates A^0 for $x \in \{di, de, f\}$ when *Duplicator* has a winning strategy in the game $G^x(A^0, A^1)$. We write $A^0 \leq_x A^1$ to denote A^1 x-simulates A^0. We also write $G^x(p,q)$ instead of $G^x(A(p), A(q))$ for some LWAA A. The winning conditions for a simulation game are described below.

C.1.4 Protoplays

A play of a simulation game is a sequence $T = t_0 t_0^0 t_0^1 t_1 t_1^0 t_1^1 \ldots$ with $t_i \in R$, $t_i^0 \in U_s \cup U_d$ and $t_i^1 \in V_s \cup V_d$. A play is completely determined by the sequence $t_0 t_1 \ldots$ and a word $w \in \Sigma$ (chosen by *Spoiler*). Similarly, for a partial play. We define $((p_i, q_i)_{i<n}, w)$ where $n \in \omega \cup \{\omega\}$, $w \in \Sigma^{n-1}$ and for all $i+1 < n$, $(p_i, w(i), p_{i+1}) \in \Delta^0$ and $(q_i, w(i), q_{i+1}) \in \Delta^1$. Such a structure is called a protoplay. Protoplays can be mapped to their corresponding play in $G^x(A^0, A^1)$ by the partial mapping,

$$\mathcal{E} : R^\infty \times \Sigma^\infty \to \text{set of } G^x(A^0, A^1) \text{ plays}$$

C.1.5 Winning Conditions

We define the winning conditions for direct, delayed and fair simulation over protoplays $((p_i, q_i)_{i<\omega}, w)$.

Direct Simulation (di)

$$\forall i. p_i \notin F^0 \Rightarrow q_i \notin F^1$$

Delayed Simulation (de)

$$\forall i. p_i \notin F^0 \Rightarrow \exists j \geq i. q_j \notin F^1$$

Fair (f)

$$(\exists i_0 \forall i \geq i_0. p_i \notin F^0) \Rightarrow (\exists j_0 \forall j \geq j_0. q_j \notin F^1)$$

C.2 Simulation Hierarchy

In this section we prove the following lemma:

Lemma C.2.1

$$\leq_{di} \quad \subset \quad \leq_{de} \quad \subset \quad \leq_f$$

Proof. We prove four cases separately:

- $A_0 \leq_{di} A_1 \Rightarrow A_0 \leq_{de} A_1$: suppose $A_0 \leq_{di} A_1$. Then, in all protoplays $((p_i, q_i)_{i<\omega}, w)$ we have the following property:

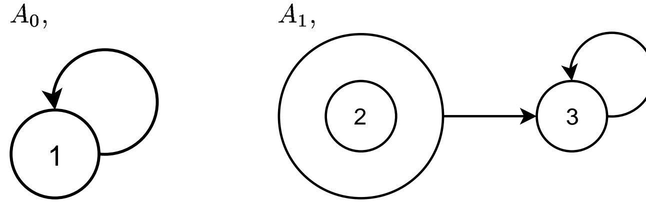
  $$\forall i.p_i \notin F^0 \Rightarrow q_i \notin F^1$$

  Taking $j = i$ as our witness, we observe the following property also holds:

  $$\forall i.p_i \notin F^0 \Rightarrow \exists j \geq i.q_j \notin F^1$$

  That is, $A_0 \leq_{de} A_1$.

- $\leq_{di} \neq \leq_{de}$: consider the following automata, where 2 is a final state.

  There are no choices to be made in either automata, hence all plays are of the form, $(1, 2), (1, 3), (1, 3), \ldots$. Since $1 \notin F^0$ and $2 \in F^1$ it is not the case that $A^0 \leq_{di} A^1$. However, because $\forall j \geq 1.q_j = 3$ and $3 \notin F^1$ we have $A_0 \leq_{de} A_1$. That is, $\leq_{di} \neq \leq_{de}$.

- $A_0 \leq_{de} A_1 \Rightarrow A_0 \leq_f A_1$: suppose $A_0 \leq_{di} A_1$. Then, in all protoplays $((p_i, q_i)_{i<\omega}, w)$ we have the following property:

  $$\forall i.p_i \notin F^0 \Rightarrow \exists j \geq i.q_j \notin F^1$$

  We show that this strategy also has the following property,

  $$(\exists i_0 \forall i \geq i_0.p_i \notin F^0) \Rightarrow (\exists j_0 \forall j \geq j_0.q_j \notin F^1)$$

  That is, $A_0 \leq_f A_1$.

  Assume $(\exists i_0 \forall i \geq i_0.p_i \notin F^0)$, then, from the condition for delayed simulation we have,

  $$\forall i \geq i_0 \exists j \geq i.q_j \notin F^1 \qquad (*)$$

Since A_1 is an LWAA, there exists j_0 such that for all $j' \geq j_0$, $q_{j'} = q_{j_0}$. That is, we have a sink state q_{j_0}.

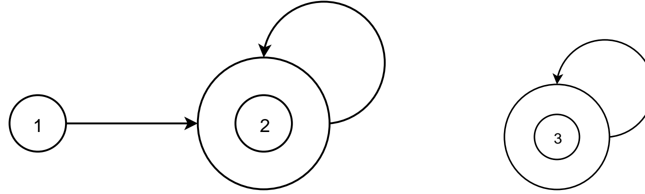
Suppose $q_{j_0} \in F^1$, then for $i \geq j_0$, there is no $j \geq i$ such that $q_j \notin F^1$. This contradicts (*). Consequently, it must be the case that $q_{j_0} \notin F^1$. Therefore,

$$(\exists j_0 \forall j \geq j_0 . q_j \notin F^1)$$

That is, $A_0 \leq_f A_1$.

- $\leq_{de} \neq \leq f$: consider the following automata.

A_0, A_1,



Since there are no choices, all plays are as follows: $(1,3)(2,3)(2,3)\ldots$. Since it is not the case that $(\exists i_0 \forall i \geq i_0 . p_i \notin F^0)$, the fair simulation winning condition is satisfied vacuously. That is $A_0 \leq_f A_1$. However, $p_0 \notin F^0$, but there is no $j \leq 0$ such that $q_j \notin F^1$. Therefore, it is not the case that $A_0 \leq_{de} A_1$.

Therefore, $\leq_{di} \subset \leq_{de} \subset \leq_f$, as required. □

C.3 Strategy Composition

We define strategy composition exactly as Fristz and Wilke. Given LWAA A^0, A^1 and A^2, states $k \in Q^0, p \in Q_1, q \in Q^2$ and strategies σ_0, σ_1 for the games $G^x(k,p)$ and $G^x(p,q)$ respectively, we define the joint strategy $\sigma = \sigma_0 \bowtie \sigma_1$ inductively. We simultaneously define the *intermediate p-sequence*. Intuitively, any combination of the plays $(k_i, p_i)_i$ and $(p_i, q_i)_i$ must agree on the p_i.

The definition is constructed around the following property,

Property C.3.1 *If $((k_j, q_j)_{j<n+1}, w)$ is a partial $(\sigma_0 \bowtie \sigma_1)$-conform protoplay and $(p_j)_{j<n+1}$ is the intermediate p-sequence for this protoplay, then $((k_j, p_j)_{j<n+1}, w)$ is a partial σ_0-conform $G^x(k,p)$-protoplay and $((p_j, q_j)_{j<n+1}, w)$ is a partial σ_1-conform $G^x(p,q)$-protoplay*

We now define strategy composition inductively. For the initial position – $G^x(k,q)$-protoplay $((k,q), \epsilon)$ – the intermediate p-sequence is p. Since no moves have been made, we have a $(\sigma_0 \bowtie \sigma_1)$-conform protoplay.

Inductively, assume we have the $(\sigma_0 \bowtie \sigma_1)$-conform protoplay $T = ((k_i, q_i)_{i<n+1}, w)$ and intermediate p-sequence $(p_i)_{i<n+1}$ satisfying property C.3.1. Let $T' = ((k_i, p_i)_{i<n+1}, w)$ and $T'' = ((p_i, q_i)_{i<n+1}, w)$.

$(\sigma_0 \bowtie \sigma_1)$ and p_{n+1} for the next round of play are defined depending on the modes of k_n, p_n and q_n.

- Case EEE. Assume *Spoiler* chooses $G^x(k,q)$-positions $t_n^0 = (k_n, q_n, a, s, 0, d, 1)$ and $t_n^1 = (k_{n+1}, q_n, a, d, 1)$ and,

$$\sigma_0(\mathcal{E}(T')(k_n, q_n, a, s, 0, d, 1)(k_{n+1}, q_n, a, d, 1)) = (k_{n+1}, p_{n+1})$$
$$\sigma_0(\mathcal{E}(T'')(p_n, q_n, a, s, 0, d, 1)(p_{n+1}, q_n, a, d, 0)) = (p_{n+1}, q_{n+1})$$

  We define,

$$\sigma_0 \bowtie \sigma_1(\mathcal{E}(T)t_n^0 t_n^1) = (k_{n+1}, q_{n+1})$$

  and $(p_i)_{i \le n+1}$ as the intermediate p-sequence for the partial protoplay $((k_i, q_i)_{i \le n+1}, wa)$. (This pattern for the intermediate p-sequence follows throughout, and shall be ommitted from the remainer of the definition.)

- Case EUE. Assume *Spoiler* chooses $t_n^0 = (k_n, q_n, a, s, 0, d, 1)$ and $t_n^1 = (k_{n+1}, q_n, a, d, 1)$, and

$$\sigma_1(\mathcal{E}(T'')(p_n, q_n, a, d, 0, d, 1)) = (p_{n+1}, q_n, a, d, 1)$$
$$\sigma_1(\mathcal{E}(T'')(p_n, q_n, a, d, 0, d, 1)(p_{n+1}, q_n, a, d, 1)) = (p_{n+1}, q_{n+1})$$

  We define,

$$\sigma_0 \bowtie \sigma_1(\mathcal{E}(T)t_n^0 t_n^1) = (k_{n+1}, q_{n+1})$$

- Case UEU. Assume the next positions are $t_n^0 = (k_n, q_n, a, s, 1, d, 0)$ and $t_n^1 = (k_n, q_{n+1}, a, d, 0)$, and

$$\sigma_0(\mathcal{E}(T')(k_n, p_n, a, d, 1, d, 0)) = (k_{n+1}, p_n, a, d, 1)$$
$$\sigma_0(\mathcal{E}(T')(k_n, p_n, a, d, 1, d, 0)(k_{n+1}, p_n, a, d, 1)) = (k_{n+1}, p_{n+1})$$

  We define,

$$\sigma_0 \bowtie \sigma_1(\mathcal{E}(T)t_n^0 t_n^1) = (k_{n+1}, q_{n+1})$$

- Case UUU. Assume the next positions are $t_n^0 = (k_n, q_n, a, s, 1, d, 0)$ and $t_n^1 = (k_n, q_{n+1}, a, d, 0)$ and

$$\sigma_1(\mathcal{E}(T'')(p_n, q_n, a, s, 1, d, 0)(p_n, q_{n+1}, a, d, 0)) = (p_{n+1}, q$$
$$\sigma_0(\mathcal{E}(T'')(k_n, p_n, a, s, 1, d, 0)(k_n, p_{n+1}, a, d, 0)) = (k_{n+1}, p_{n+1})$$

  We define,

$$\sigma_0 \bowtie \sigma_1(\mathcal{E}(T)t_n^0 t_n^1) = (k_{n+1}, q_{n+1})$$

- Case UEE. Assume *Spoiler* chooses the position $t_n^0 = (k_n, q_n, a, d, 0, d, 1)$ and

$$\sigma_0(\mathcal{E}(T')(k_n, p_n, a, d, 0, d, 1)) = (k_{n+1}, p_n, a, d, 1)$$
$$\sigma_0(\mathcal{E}(T')(k_n, p_n, a, d, 0, d, 1)(k_{n+1}, p_n, a, d, 1)) = (k_{n+1}, p_{n+1})$$
$$\sigma_1(\mathcal{E}(T'')(p_n, q_n, a, s, 0, d, 1)(p_{n+1}, q_n, a, d, 1)) = (p_{n+1}, q_{n+1})$$

We define,

$$\sigma_0 \bowtie \sigma_1(\mathcal{E}(T)t_n^0) = (k_{n+1}, q_n, a, d, 1)$$
$$\sigma_0 \bowtie \sigma_1(\mathcal{E}(T)t_n^0(k_{n+1}, q_n, a, d, 1)) = (k_{n+1}, q_{n+1})$$

- Case UUE. Assume *Spoiler* chooses $t_n^0 = (k_n, q_n, a, d, 0, d, 1)$ and

$$\sigma_1(\mathcal{E}(T'')(p_n, q_n, a, d, 0, d, 1)) = (p_n + 1, q_n, a, d, 1)$$
$$\sigma_1(\mathcal{E}(T'')(p_n, q_n, a, d, 0, d, 1)(p_{n+1}, q_n, a, d, 1)) = (p_{n+1}, q_{n+1})$$
$$\sigma_0(\mathcal{E}(T')(k_n, p_n, a, s, 1, d, 0)(k_n, p_{n+1}, a, d, 0)) = (k_{n+1}, p_{n+1})$$

  We define

$$\sigma_0 \bowtie \sigma_1(\mathcal{E}(T)t_n^0) = (k_{n+1}, q_n, a, d, 1)$$
$$\sigma_0 \bowtie \sigma_1(\mathcal{E}(T)t_n^0(k_{n+1}, q_n, a, d, 1)) = (k_{n+1}, q_{n+1})$$

- Case EEU. Duplicator cannot move in this case. For the purposes of the intermediate p-sequence, we assume *Spoiler* makes the following moves, $t_n^0 = (k_n, q_n, a, s, 0, s, 1)$ and $t_n^1 = (k_{n+1}, q_n, a, s, 1)$ and $t_n + 1 = (k_{n+1}, q_{n+1})$, with,

$$\sigma_0(\mathcal{E}(T')(k_n, p_n, a, s, 0, d, 1)(k_{n+1}, p_n, a, d, 1)) = (k_{n+1}, p_{n+1})$$

- Case EUU. Duplicator cannot move in this case. For the purposes of the intermediate p-sequence, we assume *Spoiler* makes the following moves, $t_n^0 = (k_n, q_n, a, s, 0, s, 1)$ and $t_n^1 = (k_{n+1}, q_n, a, s, 1)$ and $t_n + 1 = (k_{n+1}, q_{n+1})$, with,

$$\sigma_1(\mathcal{E}(T')(k_n, p_n, a, s, 1, d, 0)(p_n, q_{n+1}, a, d, 0)) = (p_{n+1}, q_{n+1})$$

C.4 Transitivity

In this section we follow the proof of Fritz and Wilke to show that $\leq_x$ is a pre-order. (Reflexivity is obvious.)

First we need the following lemma:

Lemma C.4.1 *Let A^0, A^1 be alternating Büchi automata and let p, q be states of A^0 and A^1, respectively, such that $p \leq_x q$. Let $a \in \Sigma$.*

1. *If $(p, q) \in E^0 \times E^1$, for every $p' \in \Delta^0(p, a)$ there is $q' \in \Delta^1(q, a)$ such that $p' \leq_x q'$.*
2. *If $(p, q) \in E^0 \times U^1$, for all $p' \in \Delta^0(p, a)$ and for all $q' \in \Delta^1(q, a)$ we have $p' \leq_x q'$.*
3. *If $(p, q) \in U^0 \times E^1$, there exists $p' \in \Delta^0(p, a)$ and $q' \in \Delta^1(q, a)$ such that $p' \leq_x q'$.*

4. *If* $(p,q) \in U^0 \times U^1$*, for every* $q' \in \Delta^1(q,a)$ *there is* $p' \in \Delta^0(p,a)$ *such that* $p' \leq_x q'$*.*

Proof. We prove each of the cases individually.

1. Assume $(p,q) \in E^0 \times E^1$. Since $p \leq_x q$, *Duplicator* has a winning strategy σ in $G^x(p,q)$. For all transitions $p' \in \Delta^0(p,a)$ *Spoiler* chooses, we have a play $T_0 = (p,q)(p,q,a,s,0,d,1)(p',q,a,d,1)$. Let $\sigma(T_0) = (p',q')$. Since σ is winning, $\sigma'(T) = \sigma(T_0T)$ must be winning from (p',q') for some $q' \in \Delta^1(q,a)$. That is, *Duplicator* has a winning strategy in $G^x(p',q')$ and thus $p' \leq_x q'$.

2. Assume $(p,q) \in E^0 \times U^1$. *Duplicator* has a winning strategy σ in $G^x(p,q)$. From (p,q) *Spoiler* chooses both transitions $p' \in \Delta^0(p,a)$ and $q' \in \Delta^1(q,a)$. Let T_0 be the play from (p,q) to (p',q'). Because σ is winning from (p,q), $\sigma'(T) = \sigma(T_0T)$ must be winning from (p',q'). That is, *Duplicator* has a winning strategy in $G^x(p',q')$ for all transitions $p' \in \Delta^0(p,a)$ and $q' \in \Delta^1(q,a)$ and thus $p' \leq_x q'$.

3. Assume $(p,q) \in U^0 \times E^1$. *Duplicator* has a winning strategy σ in $G^x(p,q)$. From (p,q) *Duplicator* chooses both transitions $p' \in \Delta^0(p,a)$ and $q' \in \Delta^1(q,a)$. Let T_0 be the play from (p,q) to (p',q'). Because σ is winning from (p,q), $\sigma'(T) = \sigma(T_0T)$ must be winning from (p',q'). That is, *Duplicator* has a winning strategy in $G^x(p',q')$ for some transitions $p' \in \Delta^0(p,a)$ and $q' \in \Delta^1(q,a)$ and thus $p' \leq_x q'$.

4. Assume $(p,q) \in U^0 \times U^1$. Since $p \leq_x q$, *Duplicator* has a winning strategy σ in $G^x(p,q)$. For all transitions $q' \in \Delta^1(q,a)$ *Spoiler* chooses, we have a play $T_0 = (p,q)(p,q,a,s,1,d,0)(p,q',a,d,0)$. Let $\sigma(T_0) = (p',q')$. Since σ is winning, $\sigma'(T) = \sigma(T_0T)$ must be winning from (p',q') for some $p' \in \Delta^0(p,a)$. That is, *Duplicator* has a winning strategy in $G^x(p',q')$ and thus $p' \leq_x q'$.

Thus, in all cases Lemma C.4.1 holds. □

We now need to show that the composition of winning strategies is a winning strategy.

Lemma C.4.2 *Let* $k \in Q^0$*,* $p \in Q^1$ *and* $q \in Q^2$ *such that* $k \leq_x p$ *and* $p \leq_x q$*. Let* σ_0 *be a Duplicator strategy for* $G^x(k,p)$ *and* σ_1 *be a winning Duplicator strategy for* $G^x(p,q)$*. If* σ_0 *and* σ_1 *are winning, then* $\sigma_0 \bowtie \sigma_1$ *is winning in* $G^x(k,q)$*. That is,* $k \leq_x q$*.*

Proof. Assume σ_0 and σ_1 are winning strategies, and let $T = ((k_i,q_i)_{i<\omega}, w)$ be an arbitrary $(\sigma_0 \bowtie \sigma_1)$-conform play with intermediate p-sequence $(p_i)_{i<\omega}$. The plays $T' = ((k_i,p_i)_{i<\omega}, w)$ and $T'' = ((p_i,q_i)_{i<\omega}, w)$ are σ_0- and σ_1-conform plays respectively. We require that T is a win for *Duplicator*. There are three cases:

Direct Simulation T' is σ_0-conform and, since σ_0 is winning for *Duplicator*, for all i with $k_i \notin F^0$ we have $p_i \notin F^1$. Since T'' is σ_1-conform and a win for *Duplicator*, $p_i \notin F^1$ implies $q_i \notin F^2$. Therefore, T is a win for *Duplicator*.

Delayed Simulation Since T' is a win for *Duplicator*, we have for all i such that $k_i \notin F^0$ there is $j_0 \geq i$ with $p_{j_0} \notin F^1$. Similarly, since T'' is a win for *Duplicator*, we have $j_1 \geq j_0$ such that $q_{j_1} \notin F^2$. Hence, for all i with $k_i \notin F^0$ we have $j_1 \geq i$ such that $q_{j_1} \notin F^2$. That is, T is winning for *Duplicator*.

Fair Simulation Assume there are not infinitely many i such that $k_i \in F^0$. Since T' is winning for *Duplicator*, this implies that there are only finitely many j with $p_j \in F^1$, and, since T'' is also winning for *Duplicator*, it follows that there are only finitely many j' with $k_{j'} \in F^2$. Consequently, T is a win for *Duplicator*.

Hence, $\sigma_0 \bowtie \sigma_1$ is winning in $G^x(k, q)$ as required. □

A *Duplicator* strategy σ is $\leq_x$-respecting if $p \leq_x q$ holds at every reachable position in the game $G^x(p_0, q_0)$ whenever *Duplicator* follows σ. It is easy to see that all *Duplicator* winning strategies are $\leq_x$-respecting. We prove that a joint strategy is $\leq_x$-respecting if its componant strategies are. The proof is exactly as in the work of Fritz and Wilke (here, the missing cases are elaborated).

Lemma C.4.3 *If σ_0 and σ_1 are $\leq_x$-respecting strategies, then $\sigma_0 \bowtie \sigma_1$ is a $\leq_x$-respecting strategy.*

Proof. The proof is by induction.

In the base case, let τ be a *Spoiler* strategy for $G^x(k, q)$, and let $T = ((t_j)_{j<\omega}, w)$ be the $(\tau, \sigma_0 \bowtie \sigma_1)$-conform protoplay. We have $k \leq_x p \leq_x q$ and, by Lemma C.4.2, $k \leq_x q$.

Now, let $i \in \omega$ and $T_i = ((t_j)_{j \leq i}, w[0..i-1])$ be the prefix of T of length $i+1$. Let $t_i = (k_i, q_i)$ and let $(p_j)_{j \leq i}$ be the intermediate p-sequence of T_i. Assume $k_i \leq_x p_i \leq_x q_i$. We show that $k_{i+1} \leq_x p_{i+1} \leq_x q_{i+1}$ holds for the next $(Q^0 \times Q^2)$-position $t_{i+1} = (k_{i+1}, q_{i+1})$ of T and the next state of the intermediate p-sequence. There are four cases:

- Assume $(k_i, q_i) \in U^0 \times U^2$. Let $t_i^0 := \tau(\mathcal{E}(T_i)) = (k_i, q_i, a, s, 1, d, 0)$ and $t_i^1 := \tau(\mathcal{E}(T_i)t_i^0) = (k_i, q_{i+1}, a, d, 0)$. Let $\sigma_0 \bowtie \sigma_1(\mathcal{E}(T_i)t_i^0 t_i^1) = (k_{i+1}, q_{i+1})$, and p_{i+1} be the next state of the intermediate p-sequence.

  If $p_i \in E^1$, the definition of $\sigma_0 \bowtie \sigma_1$ implies $k_{i+1} \leq_x p_{i+1}$, since $\mathcal{E}(T')$ is σ_0-conform (both k_{i+1} and p_{i+1} are chosen according to σ_0). Also, $p_{i+1} \leq_x q_{i+1}$ by Lemma C.4.1, since $p_i \leq_x q_i$ and $(p_i, q_i) \in E^1 \times U^2$. Hence $k_{i+1} \leq_x q_{i+1}$.

  If $p_i \in U^1$, the definition of $\sigma_0 \bowtie \sigma_1$ also implies $k_{i+1} \leq_x q_{i+1}$, since $\mathcal{E}(T'')$ is σ_1-conform (p_{i+1} is chosen according to σ_1, hence $p_{i+1} \leq_x q_{i+1}$).

Because $\mathcal{E}(T^{'}_{i+1})$ is σ_0-conform (that is, k_{i+1} is chosen according to σ_0), we have $k_{i+1} \leq_x p_{i+1} \leq_x q_{i+1}$.

- Assume $(k_i, q_i) \in E^0 \times U^2$. Let $t_i^0 := \tau(\mathcal{E}(T_i)) = (k_i, q_i, a, s, 0, s, 1)$ and $t_i^1 := \tau(\mathcal{E}(T_i)t_i^0) = (k_{i+1}, q_i, a, s, 1)$ and $t_{i+1} := \tau(\mathcal{E}(T_i)t_i^0 t_i^1) = (k_{i+1}, q_{i+1})$. Let p_{i+1} be the next state of the intermediate p-sequence.

  If $p_i \in E^1$ then *Spoiler* chooses k_{i+1} and q_{i+1} and p_{i+1} is chosen according to σ_0. Since $p_i \leq_x q_i$, we have that $p_{i+1} \leq_x q_{i+1}$ follows from Lemma C.4.1. $k_{i+1} \leq_x p_{i+1}$ follows because $\mathcal{E}(T')$ is σ_0-conform (p_{i+1} is chosen according to σ_0). We therefore have $k_{i+1} \leq_x p_{i+1} \leq_x q_{i+1}$.

  If $p_i \in U^1$ then *Spoiler* chooses k_{i+1} and q_{i+1} and p_{i+1} is chosen according to σ_1. $\mathcal{E}(T')$ is σ_0 conform because *Duplicator* has no choice from an EU position. Therefore $k_{i+1} \leq_x p_{i+1}$. $\mathcal{E}(T'')$ is σ_1-conform because p_{i+1} was chosen according to σ_1. Hence $p_{i+1} \leq q_{i+1}$ and thus $k_{i+1} \leq_x p_{i+1} \leq q_{i+1}$

- Assume $(k_i, q_i) \in U^0 \times E^1$. Let $t_i^0 := \tau(\mathcal{E}(T_i)) = (k_i, q_i, a, d, 0, d, 1)$, $t_i^1 := \sigma_0 \bowtie \sigma_1(\mathcal{E}(T_i)t_i^0) = (k_{i+1}, q_i, a, d, 1)$ and $t_{i+1} := \sigma_0 \bowtie \sigma_1(\mathcal{E}(T_i)t_i^0 t_i^1) = (k_{i+1}, q_{i+1})$. Let p_{i+1} be the next state of the intermediate p-sequence.

  If $p_i \in E^1$ then k_{i+1} and p_{i+1} are chosen according to σ_0 and q_{i+1} is chosen according to σ_1. Therefore $\mathcal{E}(T')$ is σ_0-conform and so $k_{i+1} \leq_x p_{i+1}$. Since q_{i+1} is chosen according to σ_1, we have that $\mathcal{E}(T'')$ is σ_1-conform and so $p_{i+1} \leq_x q_{i+1}$. Thus $k_{i+1} \leq_x p_{i+1} \leq_x q_{i+1}$.

  If $p_i \in U^1$ then p_{n+1} and q_{n+1} are chosen according to σ_1 and k_{n+1} is chosen according to σ_0. Hence $\mathcal{T}'$ is σ_0-conform and $k_{i+1} \leq_x p_{i+1}$. Similarly $\mathcal{T}''$ is σ_1-conform and $p_{i+1} \leq_x q_{i+1}$. Thus $k_{i+1} \leq_x p_{i+1} \leq_x q_{i+1}$.

- Assume $(k_i, q_i) \in E^0 \times E^2$. Let $t_i^0 := \tau(\mathcal{E}(T_i)) = (k_i, q_i, a, s, 0, d, 1)$ and $t_i^1 := \tau(\mathcal{E}(T_i)t_i^0) = (k_{i+1}, q_i, a, d, 1)$. Let $\sigma_0 \bowtie \sigma_1(\mathcal{E}(T_i)t_i^0 t_i^1) = (k_{i+1}, q_{i+1})$, and p_{i+1} be the next state of the intermediate p-sequence.

  If $p_i \in E^1$ then *Spoiler* chooses k_{i+1}, p_{i+1} is chosen according to σ_0 and q_{i+1} according to σ_1. Therefore $\mathcal{E}(T')$ and $\mathcal{E}(T'')$ are σ_0- and σ_1-confrom respectively, and hence $k_{i+1} \leq_x p_{i+1} \leq_x q_{i+1}$.

  If $p_i \in U^1$ then *Spoiler* chooses k_{i+1}. p_{i+1} and q_{i+1} are chosen according to σ_1. $k_{i+1} \leq_x p_{i+1}$ follows from $k_i \leq_x p_i$ and Lemma C.4.1. Since $\mathcal{T}''$ is σ_1-conform, we also have $p_{i+1} \leq_x q_{i+1}$. We have $k_{i+1} \leq_x p_{i+1} \leq_x q_{i+1}$ as required.

Since $k_{i+1} \leq_x p_{i+1} \leq_x q_{i+1}$ we have $k_{i+1} \leq_x q_{i+1}$ by application of Lemma C.4.2. Hence $\sigma_0 \bowtie \sigma_1$ is $\leq_x$-respecting. □

Corollary C.4.1 *For $x \in \{di, de, f\}$, $\leq_x$ is a pre-order.*

We can therefore define the equivalence relation $\equiv_x$:

$$p \equiv_x q \iff p \leq_x q \text{ and } q \leq_x p$$

C.5 Language Containment

We now show that simulation implies language containment. We begin by presenting a definition of acceptance for an LWAA with a co-Büchi acceptance condition.

Definition C.5.1 *An LWAA with a co-Büchi acceptance condition is a tuple* $A = (\Sigma, S, s_0, \delta, E, U, F)$. Σ *is a finite, non-empty, alphabet.* S *is a finite set of states, where* s_0 *is the initial state.* $\delta : S \times \Sigma \to 2^S$ *is a transition relation.* $F \subseteq S$ *is a set of final states.* $\{E, U\}$ *is a partition of* S *into existential and universal states.*

For an LWAA A and an input word $w \in \Sigma^\omega$, acceptance is defined via the game $G(A, w) = (P, P_0, P_1, p_I, Z, W)$, where,

- $P = S \times \omega, P_1 = U \times \omega, P_2 = E \times \omega,$
- $p_I = (s_I, 0),$
- $Z = \{((s, i), (s', i+1)) \mid s' \in \delta(s, w(i))\}$, and
- $W = P^*(P \setminus (F \times \omega))^\omega$.

We refer to player 1 as *Automaton* and player 0 as *Pathfinder*. w is accepted by the automaton A iff *Automaton* has a winning strategy in $G(A, w)$. Over a game graph, a state in F must occur only finitely often on all paths leading from a universal state, and on some path leading from an existential state. For $q \in S$ we write $A(q)$ to denote the automaton that is a copy of A, with q as its initial state.

Property C.5.1 *Let* $x \in \{di, de, f\}$ *and* A^0 *and* A^1 *be alternating Büchi automata. If* $A^0 \leq_x A^1$*, then* $\mathcal{L}(A^0) \subseteq \mathcal{L}(A^1)$.

Proof. Assume σ is a winning strategy for *Duplicator* in $G^x(A^0, A^1)$, $w \in \mathcal{L}(A^0)$ and σ' is a winning strategy for *Automaton* in $G(A^0, w)$. We need to show that *Automaton* has a winning strategy σ'' in the game $G^x(A^1, w)$.

To define σ'' we need to map $G^x(A^0, A^1)$-protoplays to prefixes of $G(A^0, w)$- and $G(A^1, w)$-plays. For $T = ((p_i, q_i)_{i \leq n}, w[0..n-1])$, we define,

$$pr^0(T) = (p_0, 0) \dots (p_n, n)$$
$$pr^1(T) = (q_0, 0) \dots (q_n, n)$$

Given a partial σ-conform protoplay $T = ((p_i, q_i)_{i \leq n+1}, w[0..n])$ such that $pr^0(T)$ is σ'-conform and $q_n \in E^1$, we define,

$$\sigma''((q_0, 0) \dots (q_n, n)) = (q_{n+1}, n+1)$$

We now need to show that σ'' is well-defined, a strategy for *Automaton*, and winning.

1. To show σ'' is well-defined we need to prove that, for every partial σ-conform play $T = ((p_0, q_0) \dots (p_n, q_n), w[0..n-1])$ with $(p_0, 0) \dots (p_n, n)$ σ'-conform, we have that for all $q \in Q^1$, there is at most one $p \in Q^0$ such that $((p_0, q_0) \dots (p_n, q_n)(p, q), w[0..n])$ is a partial σ-conform protoplay, and $(p_0, 0) \dots (p_n, n)(p, n+1)$ is σ'-conform. Observe that, if $p_n \in E^0$ then σ' determines p. If $p_n \in U^1$ then p is determined by σ.

   Assume, σ'' is not well-defined. That is, there are two protoplays $T = ((p_0, q_0) \dots (p_n, q_n)(p, q), w[0..n])$ and $\hat{T} = ((\hat{p}_0, q_0) \dots (\hat{p}_n, q_n)(\hat{p}, \hat{q}), w[0..n])$ with $q \neq \hat{q}$, $q_n \in E^1$ and $pr^0(T)$ and $pr^0(T')$ are σ'-conform. Since $p_0 = \hat{p}_0$ and the next p is determined by σ or σ', we have $p_i = \hat{p}_i$ for $i \leq n$. Since $q_n \in E^1$, σ determines both q and $\hat{q}$, and so $q = \hat{q}$. This is a contradiction; σ'' is well-defined.

2. For σ'' to be a strategy for *Automaton* its domain must contain all σ''-conform plays $T = (q_0, 0) \dots (q_n, n)$ with $q_n \in E^1$. The proof is by induction. The base case occurs when $q_i \in U^1$ for all $i < n$. In this case σ does not restrict any of the q_i for $i \leq n$ (*Spoiler* chooses q_{i+1} in $G^x(A^0, A^1)$, so T is in the domain of σ''.

   Now, suppose there is a maximal $i < n$ such that $q_i \in E^1$. By induction we know that $(q_0, 0) \dots (q_i, i)$ is in the domain of σ''. By a simlar argument used in the base case, we know that σ does not restrict q_j for $i < j \leq n$. Hence, T is in the domain of σ''.

3. We now show that σ'' is winning for *Automaton*. Given a σ''-conform play V. We know, by construction of σ'', that there is a σ-conform protoplay $T = ((p_0, q_0)(p_1, q_1) \dots, w)$ such that $pr^0(T)$ is σ'-conform and $pr^1(T) = V$. σ' is winning and so there is a i_0 such that for all $i \geq i_0$ we have $p_i \notin F^0$. σ is winning. In the case of direct simulation it follows that for all $i \geq i_0$ we have $q_i \notin F^1$. Thus, V is a win for *Automaton*. In the case of fair simulation it follows from the winning condition that there is $j_0 \geq i_0$ with $q_j \notin F^1$ for all $j \geq j_0$. Hence, V is a win for *Automaton*. In the case of delayed simulation, we need to exploit the structure of an LWAA.

   All paths $s_0 s_1 s_2 \dots$ of an LWAA have k_0 such that for all $k \geq k_0$, $s_k = s_{k_0}$. This is because LWAA are finite structures with no cycles except for self-loops (all infinite paths of a finite automaton are ultimately periodic, hence LWAA must settle in one state (assuming a total transition relation)). Assume V is not winning. That is, for sink state q_{k_0}, we have $q_{k_0} \in F^1$. Take i' such that $i' \geq i_0$ and $i' \geq k_0$. We know that $p_{i'} \notin F^0$. Therefore, by the winning condition for delayed simulation, there exists $j' \geq i'$ with $q_{j'} \notin F^1$. But, since for all $k \geq k_0$, $q_k = q_{k_0}$ and $q_{k_0} \in F^1$, it must be the case that $q_{j'} \in F^1$. This is a contradiction. V must be winning for *Automaton*.

σ'' is therefore a winning strategy for *Automaton* in $G(A^1, w)$, and so $w \in \mathcal{L}(A^1)$. Finally, $\mathcal{L}(A^0) \subseteq \mathcal{L}(A^1)$, as required. □

C.6 An Alternative Delayed Simulation

We also considered an alternative definition of delayed simulation, which took into account the structure of an LWAA.

Delayed Simulation* (de^*)

If there exists i_0 such that $\forall i \geq i_0.p_i \notin F^0$, then $\exists j_0 \geq i_0 \forall j \geq j_0.q_j \notin F^1$

That is, if A_0 reaches a sink state that is accepting, A_1 must eventually reach a sink state that is accepting. The intuition behind this definition is that accepting states (not in F) that are not sink states in a run do not contribute to the acceptance of the run, hence they can be ignored.

However, it can be shown that this definition of delayed simulation is equivalent to fair simulation.

Proposition C.6.1 *Given LWAA A_0 and A_1, we have,*

$$A_0 \leq_{de^*} A_1 \quad \Longleftrightarrow \quad A_0 \leq_f A_1$$

Proof. In the only-if direction, assume $A_0 \leq_{de^*} A_1$, that is, over all protoplays $((p_i, q_i)_{i<\omega}, w)$, we have that if there exists i_0 such that $\forall i \geq i_0.p_i \notin F^0$, then $\exists j_0 \geq i_0 \forall j \geq j_0.q_j \notin F^1$. To show $A_0 \leq_f A_1$ we suppose $\forall i \geq i_0.p_i \notin F^0$ and show $\exists j' \forall j \geq j'.q_j \notin F^1$. This is immediate: we take $j' = j_0$.

In the if direction, assume $A_0 \leq_f A_1$. That is, over all protoplays $((p_i, q_i)_{i<\omega}, w)$, we have $\exists i_0 \forall i \geq i_0.p_i \notin F^0 \Rightarrow \exists j_0 \forall j \geq j_0.q_j \notin F^1$. To show $A_0 \leq_{de^*} A_1$ we assume there exists i_0 such that $\forall i \geq i_0.p_i \notin F^0$ and show $\exists j' \geq i_0 \forall j \geq j'.q_j \notin F^1$. Let $j' = max(i_0, j_0)$. Since $j' \geq i_0$ and for all $j \geq j'$ we have $j \geq j_0$ and therefore $q_j \notin F^1$, we have $A_0 \leq_f A_1$. □

Definition C.6.1 *Given an equivalence relation $\equiv$ on the state space of an LWAA, we define quotients of A to be automata of the following form,*

$$A/\equiv \;:=\; (Q/\equiv, \Sigma, [q_I]_\equiv, \Delta', E', U', F/\equiv)$$

where $M/\equiv = \{[q]_\equiv \mid q \in M\}$ for $M \subseteq Q$ and $[q]_\equiv = \{q' \in Q \mid q \equiv q'\}$.

Furthermore,

1. *If $([p]_\equiv, a, [q]_\equiv) \in \Delta'$, then there exists $p' \equiv p$ and $q' \equiv q$ such that $(p', a, q') \in \Delta$,*
2. *if $[q]_\equiv \subseteq E$, then $[q]_\equiv \in E'$, and*
3. *if $[q]_\equiv \subseteq U$, then $[q]_\equiv \in U'$.*

C.7 Naive Quotienting

A naive quotient is defined as follows,

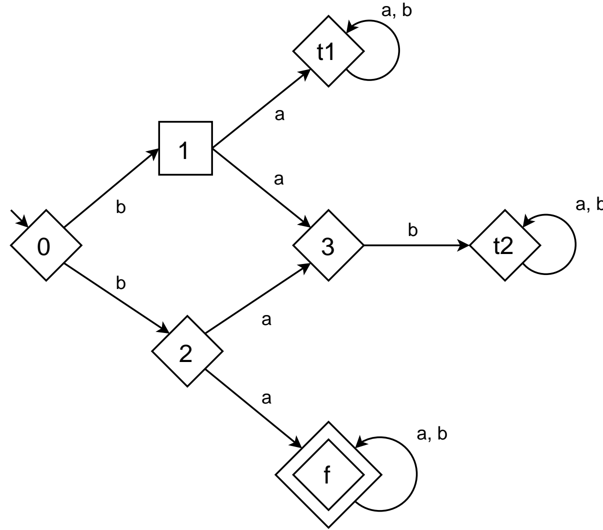
Definition C.7.1 *Given an equivalence relation $\equiv$ on the state space of an LWAA, we define quotients of A to be automata of the following form,*

$$A/\equiv \;:=\; (Q/\equiv, \Sigma, [q_I]_\equiv, \Delta', E', U', F/\equiv)$$

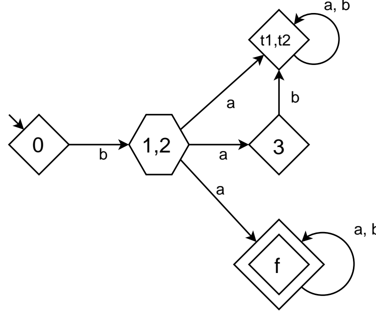
where $M/\equiv\, = \{[q]_\equiv \mid q \in M\}$ for $M \subseteq Q$ and $[q]_\equiv = \{q' \in Q \mid q \equiv q'\}$.
Furthermore,

1. *$([p]_\equiv, a, [q]_\equiv) \in \Delta'$ iff there exists $p' \equiv p$ and $q' \equiv q$ such that $(p', a, q') \in \Delta$,*
2. *if $[q]_\equiv \subseteq E$, then $[q]_\equiv \in E'$, and*
3. *if $[q]_\equiv \subseteq U$, then $[q]_\equiv \in U'$.*

Naive quotients run into problems when an equivalence class contains both existential and universal states. Consider the following automaton A:



Observe $\mathcal{L}(A) = bab(a \cup b)^\omega$. Also observe that $t1 \equiv_{di} t2$ and $1 \equiv_{di} 2$. The result A' of a naive quotient by the direct simulation equivalence relation is as follows:



State $\{1, 2\}$ has been declared neither existential nor universal. We consider both interpretations and conclude that neither results in a satisfactory quotient. That is $\mathcal{L}(A) \neq \mathcal{L}(A')$.

Suppose $\{1, 2\}$ is universal. Then, from state $\{1, 2\}$ we require an 'a' as input. Since $\{1, 2\}$ is universal, *Pathfinder* can choose f as the next state. Since $f \in F$ and f is a sink node, it follows that $\mathcal{L}(A') = \emptyset$.

If, however, we declare $\{1, 2\}$ to be existential, then from $\{1, 2\}$ *Automaton* can choose state $\{t1, t2\}$, therefore $\mathcal{L}(A') = ba(a \cup b)^\omega$.

C.8 Minimal and Maximal Successors

Given an automaton A, a state q of A and character a, an x-maximal a-successor of q is a state $q' \in \delta(q, a)$ such that for every state $q'' \in \delta(q, a)$ with $q' \leq_x q''$, we have $q'' \leq_x q'$. That is, q' is simulation equivalent to any successor state that can simulate it.

Conversely q' is an x-minimal a-successor of q just in case for all $q'' \in \delta(q, a)$ with $q'' \leq_x$ we have $q' \leq_x q''$. That is, q' is simulated by every successor state that it can simulate.

We define the sets $min_a^x(q)$ and $max_a^x(q)$,

$$
\begin{aligned}
min_a^x(q) &:= \{q' \in \delta(q, a) \mid q' \text{ is an } x\text{-minimal } a\text{-successor of } q\} \\
max_a^x(q) &:= \{q' \in \delta(q, a) \mid q' \text{ is an } x\text{-maximal } a\text{-successor of } q\}
\end{aligned}
$$

Corollary C.8.1 *Let $p \in Q^0$, $q \in Q^1$ be states of LWAA A^0 and A^1 such that $p \equiv_x q$. Let $a \in \Sigma$.*

1. *If $(p, q) \in E^0 \times E^1$ and $p' \in max_a^x(p)$, then there is a $q' \in max_a^x(q)$ such that $p' \equiv_x q'$.*

2. *If $(p, q) \in U^0 \times U^1$ and $p' \in min_a^x(p)$, then there is a $q' \in min_a^x(q)$ such that $p' \equiv_x q'$.*

3. *If $(p, q) \in E^0 \times U^1$, then all x-maximal a-successors of p and all x-minimal a-successors of q are x-equivalent.*

Furthermore, $q_0 \equiv_x q_1$ for all $q_0, q_1 \in min_a^x(q) \cup max_a^x(p)$.

Proof.

1. Let $(p,q) \in E^0 \times E^1$ and $p' \in max_a^x(p)$. By Lemma C.4.1 and $p \leq_x q$, we have a $q' \in \Delta(q,a)$ such that $p' \leq_x q'$. Suppose we have $q'' \in \Delta(q,a)$ with $q' \leq_x q''$. By Lemma C.4.1 and $q \leq_x p$, there is $p'' \in \Delta(p,a)$ such that $q'' \leq_x p''$. Since $p' \leq_x q' \leq_x q'' \leq_x p''$ we have $p' \leq_x p''$ and since $p' \in max_a^x(p)$ we have $p'' \leq_x p'$. It then follows that $p' \leq_x q' \leq_x q'' \leq_x p'' \leq_x p' \leq_x q'$. Consequently, q' is an x-maximal a-successor of q. Furthermore, $p' \equiv q'$.

2. Let $(p,q) \in U^0 \times U^1$ and $p' \in min_a^x(p)$. By Lemma C.4.1 and $q \leq_x p$, we have a $q' \in \Delta(q,a)$ such that $q' \leq_x p'$. Suppose we have $q'' \in \Delta(q,a)$ with $q'' \leq_x q'$. By Lemma C.4.1 and $p \leq_x q$, there is $p'' \in \Delta(p,a)$ such that $p'' \leq_x q''$. Since $p'' \leq_x q'' \leq_x q'$ we have $p'' \leq_x q' \leq_x p'$. That is $p' \leq_p''$ and since $p' \in min_a^x(p)$ we have $p' \leq p'' \leq q'$. Hence $p' \equiv_x q'$. Furthermore, since $q' \leq_x p' \leq_x p'' \leq_x q''$ we have $q \in min_a^x(q)$.

3. Let $(p,q) \in E^0 \times U^1$, $p' \in max_a^x(p)$ and $q' \in min_a^x(q)$. We require $p' \equiv q'$. By Lemma C.4.1 and $p \leq_x q$ we have $p' \leq_x q'$.

   In the opposite direction, by Lemma C.4.1 and $q \leq_x p$ we have that there are $p'' \in \Delta(p,a)$ and $q'' \in \Delta(q,a)$ with $q'' \leq_x p''$. By Lemma C.4.1 and $p \leq q$ we know that $p' \leq_x q''$ and $p'' \leq_x q'$. Since $p' \in max_a^x(p)$ we have $p'' \leq_x p'$, and since $q' \in min_a^x(q)$, $q' \leq_x q''$. Hence $q' \leq_x q'' \leq_x p'' \leq p'$. Consequently $p' \equiv q'$.

$q_0 \equiv_x q_1$ for all $q_0, q_1 \in min_a^x(q) \cup max_a^x(p)$ follows from the transitivity of $\equiv_x$. □

C.9 Minimax Strategies

Remark C.9.1 *Given an LWAA $A = (Q, \Sigma, q_I, \Delta, E, U, F)$, and the relations $\leq_{di} \subseteq Q \times Q$ and $\equiv_{di}$ we have,*

1. *For all $p, q \in Q$, if $p \leq_{di} q$ and $p \notin F$, then $q \notin F$.*

2. *For all $p, q \in Q$, if $p \equiv_{di} q$, then $p \notin F$ iff $q \notin F$.*

If $((p_i, q_i), w)$ is a protoplay in an x-game that is σ-conform for some winning *Duplicator* stratgy σ, then $p_i \leq_x q_i$ holds for all $i \geq 0$. For direct simulation, the converse is also true.

Lemma C.9.1 *Let $p_0 \leq_{di} q_0$. A $\leq_{di}$-respecting strategy for Duplicator in $G^{di}(p_0, q_0)$ is a winning strategy.*

Proof. Let $p_0 \leq_{di} q_0$, and let σ be a di-respecting *Duplicator* strategy in $G^{di}(p_0 . q_0)$. Let $T = ((p_i, q_i)_{i<\omega}, w)$ we a σ-conform $G^{di}(p_0, q_0)$-protoplay. By assumption we have $p_i \leq_{di} q_i$ for all $i \geq 0$. Therefore, from Remark C.9.1 we have $q_i \notin F$ whenever $p_i \notin F$. Hence T is a win for *Duplicator* and σ is a

winning strategy. □

Hence, the di-respecting strategies are exactly the winning strategies for *Duplicator* .

Definition C.9.1 *A minimax strategy σ is a strategy such that for every σ-conform protoplay $T = ((p_i, q_i)_{i<\omega}, w)$ it is the case that,*

1. *If $(p_i, q_i) \in U^0 \times Q^1$, then $p_{i+1} \in min^x_{w(i)}(p_i)$, and*
2. *If $(p_i, q_i) \in E^0 \times Q^1$, then $q_{i+1} \in max^x_{w(i)}(q_i)$.*

Lemma C.9.2 *Let $p_0 \leq_{di} q_0$. Then, there exists a $\leq_x$-respecting minimax strategy for Duplicator in $G^x(p_0, q_0)$.*

Proof. Defined inductively using Lemma C.4.1. □

C.10 Direct Simulation and Minimax Quotienting

In the case of direct simulation we define the minimax quotient,

Definition C.10.1 *An x-minimax quotient of an alternating Büchi automaton A is a quotient whose transition relation is,*

$$\begin{array}{rcl}\delta & := & \{([p]_{\equiv_x}, a, [q]_{\equiv_x}) \mid a \in \Sigma, p \in E, q \in max^x_a(p)\} \cup \\ & & \{([p]_{\equiv_x}, a, [q]_{\equiv_x}) \mid a \in \Sigma, p \in U, q \in min^x_a(p)\}\end{array}$$

A mixed class can be either universal or existential.

The following result justifies the assertion that a mixed class can be existential or universal.

Remark C.10.1 *For a mixed class $M \in Q/\equiv_x$ and $a \in \Sigma$,*

$$\begin{array}{l}\{[q]_{\equiv_x} \mid \exists p(p \in M \cap E \ \wedge\ q \in max^x_a(p))\} \\ \quad = \{[q]_{\equiv_x} \mid \exists p(p \in M \cap U \ \wedge\ q \in min^x_a(p))\}\end{array}$$

and these sets are singletons.

Proof. First we prove the two sets are equal. Take $[q']_x \in \{[q]_{\equiv_x} \mid \exists p(p \in M \cap E \ \wedge\ q \in max^x_a(p))\}$ and the associated p. Also, take $p' \in M \cap U$ (there is at least one, since M is a mixed class). Since $p \equiv p'$ and $(p, p') \in E \times U$, it follows from Corollary C.8.1 that for $q'' \in min_a(p')$ we have $q' \equiv q''$, that is $[q']_x = [q'']_x$. Since W.l.o.g. we assume a total transition relation, $min_a(p')$ cannot be non-empty (easy proof). Hence $[q']_x \in \{[q]_{\equiv_x} \mid \exists p(p \in M \cap U \ \wedge\ q \in min^x_a(p))\}$.

Conversely, take $[q']_x \in \{[q]_{\equiv_x} \mid \exists p(p \in M \cap U \ \wedge \ q \in min^x_a(p))\}$ and the associated p. Take $p' \in M \cap E$. Since $p \equiv p'$ and $(p', p) \in E \times U$ we have by Corollary C.8.1 that for $q'' \in max_a(p')$ it is the case that $q' \equiv q''$. That is $[q']_x = [q'']_x$, therefore $[q']_x \in \{[q]_{\equiv_x} \mid \exists p(p \in M \cap E \ \wedge \ q \in max^x_a(p))\}$.

Finally, we show these sets are singletons. Since the two sets are equivalent, it is sufficient to show $\{[q]_{\equiv_x} \mid \exists p(p \in M \cap E \ \wedge \ q \in max^x_a(p))\}$ is a singleton. Take any $[q]_x, [q']_x \in \{[q]_{\equiv_x} \mid \exists p(p \in M \cap E \ \wedge \ q \in max^x_a(p))\}$. We show $q \equiv_x q'$, hence $[q]_x = [q']_x$. We have that there exists $p_e \in M \cap E$ with $q \in max_a(p_e)$. Since it is also the case that $[q]_x, [q']_x \in \{[q]_{\equiv_x} \mid \exists p(p \in M \cap U \ \wedge \ q \in min^x_a(p))\}$, there exists $p^{'}_u \in M \cap U$ and $q^{'}_{min} \in min_a(p^{'}_u)$ with $q' \equiv_x q^{'}_{min}$. Observe that it is also the case that $p_e \equiv_x p^{'}_u$ since both states are in M. Hence, since $(p_e, p^{'}_u) \in E \times U$ and $q \in max_a(p_e)$ and $q^{'}_{min} \in min_a(p^{'}_u)$, it follows by Corollary C.8.1 that $q \equiv_x q^{'}_u \equiv q'$. Hence $[q]_x = [q']_x$, as required. □

Remark C.10.2 *For a mixed class $M \in Q/\equiv_x$, for every $q \in Q$, $[q]_{di} \cap F \neq \emptyset$ iff $[q]_{di} \subseteq F$.*

Proof. From Remark C.9.1. □

Theorem C.10.1 *Let A be an alternating Büchi automaton and B^m any di-minimax quotient of A.*

1. *For all $p, q \in S$ such that $p \leq_{di} q$, $A(q)$ di-simulates $B^m([p]_{\equiv_{di}})$ and $B^m([q]_{\equiv_{di}})$ di-simulates $A(p)$.*
2. *A and B^m di-simulate each other. That is, $A \equiv_{di} B^m$.*
3. *$A \equiv_{di} B^m$ and $\mathcal{L}(A) = \mathcal{L}(B^m)$.*

Proof. Because all mixed classes are deterministic, it is enough to consider B^m where mixed classes are declared existential. Furthermore, since the second and third properties follow from the first, we only prove property one.

To show $A(q_0)$ *di*-simulates $B^m([p_0]_{di})$ we define a winning strategy σ of *Duplicator* for $G^{di}([p_0]_{di}, q_0)$. By Lemma C.9.2, for all (q, q') with $q \leq_{di} q'$, there exists a $\leq_{di}$-respecting minimax strategy $\sigma_{q,q'}$ for *Duplicator*.

Let T be the prefix of a $G^{di}([p_0]_{di}, q_0)$-play whose last position t is in P_1 (a *Duplicator* move) and whose last $(Q_{di} \times Q)$-position $([p]_{di}, q)$ satisfies $p \leq_{di} q$. There are three cases:

1. The suffix of the partial protoplay T is of the form,

$$([p]_{di}, q)([p]_{di}, q, a, s, 0, d, 1)([p']_{di}, q, a, d, 1)$$

that is, $p, q \in E$.

There are $\hat{p} \in [p]_{di}$ and $\hat{p}' \in [p']_{di}$ such that $(\hat{p}, a, \hat{p}') \in \Delta$. We define $\sigma(T) = ([p']_{di}, q')$ where,

$$q' = pr_2(\sigma_{\hat{p},q}((\hat{p}, q)(\hat{p}, q, a, s, 0, d, 1)(\hat{p}', q, a, d, 1)))$$

Since $\sigma_{\hat{p},q}$ is minimax, and $q \in E$, we have $q' \in max_a(q)$. Furthermore, since $\hat{p}' \equiv_{di} p'$ and since $\sigma_{\hat{p},q}$ is $\leq_{di}$-respecting, we have $p' \equiv_{di} \hat{p}' \leq_{di} q'$.

2. The suffix of the partial protoplay T is of the form,

$$([p]_{di}, q)([p]_{di}, q, a, d, 0, d, 1)$$

that is, $(p, q) \in U \times E$.
We define $\sigma(T) = ([p']_{di}, q, a, d, 1)$ and $\sigma(\sigma(T)) = ([p']_{di}, q')$ where,

$$p' = pr_1(\sigma_{p,q}((p,q)(p,q,a,d,0,d,1)))$$
$$q' = pr_2(\sigma_{p,q}(\sigma_{p,q}((p,q)(p,q,a,d,0,d,1)))$$

Since $\sigma_{p,q}$ is minimax, and $p \in U$, we have $q' \in min_a(q)$. Furthermore, since $\sigma_{p,q}$ is $\leq_{di}$-respecting, we have $p' \leq_{di} q'$.

3. The suffix of the partial protoplay T is of the form,

$$([p]_{di}, q)([p]_{di}, q, a, s, 1, d, 0)([p]_{di}, q', a, d, 0)$$

that is, $(p, q) \in U \times U$. We define $\sigma(T) = ([p'], q')$ where,

$$p' = pr_1(\sigma_{p,q}((p,q)(p,q,a,s,1,d,0)(p,q',a,d,0)))$$

By choice of $\sigma_{p,q}$, we have $p' \in min_a(p)$ and $p' \leq_{di} q'$.

Since σ has the property that for any σ-conform play, every position $([p_i]_{di}, q_i) \in Q_{di} \times Q$ contains $p_i \leq_{di} q_i$. Consequently, it cannot be the case that $([p_i]_{di}, q) \in (Q_{di} \setminus F_{di}) \times F$, since, by Remark C.10.2, this would imply $(p_i, q_i) \in (Q \setminus F) \times F$, contradicting $p_i \leq_{di} q_i$. Hence, σ is winning for *Duplicator* in $G^{di}([p_0]_{di}, q_0)$.
The converse direction is symmetrical. □

Since the above proof does not use the the set of transitions is minimal, we may allow more transitions. However, we may only allow extra transitions if mixed classes are declared existential, and, from a universal state, no non-minimal transitions are considered for mised classes.

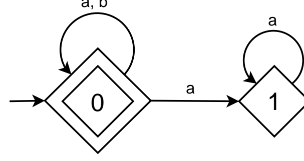
Corollary C.10.1 *Let $A = (Q, \sigma, q_I, \Delta, E, U, F)$ be an LWAA. Let $B_{di} = (Q/\equiv_{di}, \Sigma, [q_I]_{di}, \Delta', E'.U', F/\equiv_{di})$ be a direct quotient of A, such that,*

1. *$\Delta^m_{di} \subseteq \Delta'$,*
2. *If $[q]_{di} \cap E \neq \emptyset, then [q]_{di} \in E'$,*
3. *For every $q_u \in [q]_{di} \cap U$ with $[q]_{di} \cap E \neq \emptyset$, we have the property that, if $([q_u]_{di}, a, [q']_{di}) \in \Delta'$, then there are $\hat{q} \in [q_u]_{di} \cap E$, $\hat{q}' \in [q']_{di}$ such that $(\hat{q}, a, \hat{q}') \in \Delta$.*

Then $A \equiv_{di} B_{di}$.

C.11 Delayed Simulation and Minimax Quotienting

Minimax quotienting does not work for delayed simulation. Consider the following LWAA A:



For $x \in \{de, f\}$, we have $0 \geq_x 1$ but not $0 \equiv_x 1$. That is, $max_a(0) = \{0\}$. Consequently, there is no transition from $[0]_{de}$ to $[1]_{de}$ in any minimax quotient of A. That is, the language of any minimax quotient of A is empty.

C.12 Delayed Simulation and Semi-elective Quotienting

A semi-elective quotient is defined as follows:

Definition C.12.1 *Given LWAA A, we define the semi-elective quotient A^s_x of A as the quotient whose transition relation is given by,*

$$\begin{aligned}\Delta^s_x = &\{([p]_x, a, [q]_x) \mid (p, a, q) \in \Delta, p \in E\} \\ &\cup \{([p]_x, a, [q]_x) \mid a \in \Sigma, [p] \subseteq U, q \in min_a(p)\}\end{aligned}$$

Corollary C.12.1 *For every LWAA A, $A \equiv_{di} A^s_{di}$.*

Proof. Follows immediately from Corollary C.10.1. □

C.12.1 A Simulates A^s_{de}

We require several lemmas before we can prove A simulates A^s_{de}.

Definition C.12.2 *Given a simulation game $G(K_0, p_0)$ where K_0 is a state of the quotient automaton A^s_{de},*

$$K_0 \sqsubseteq_{de} p_0 \iff \exists k_0 \in K_0 . k_0 \leq_{de} p_0$$

Corollary C.12.2 *For all $K_0 \in Q/\equiv_{de}$ and for all $p_0 \in Q$ such that $K_0 \sqsubseteq_{de} p_0$, there is a minimax strategy σ of Duplicator for $G(K_0, p_0)$ such that, for all Spoiler strategies τ for $G(K_0, p_0)$, the (τ, σ)-conform protoplay $((K_i, p_i)_{i<\omega}, w)$ satisfies $K_i \sqsubseteq_{de} p_i$ for all $i < \omega$.*

Proof. Let $K_0 \in Q_{de}, p_0 \in Q$ and T_i be a prefix of a $G(K_0, p_0)$-play such that the last position of T_i is a P_1 position. There are three cases:

1. $(K_i, p_i)(K_i, p_i, a, s, 0, d, 1)(K_{i+1}, p_i, a, d, 1)$ is a suffix of T. That is, $K_i \in E_{de}$. Since all transitions from an existential quotient state have a corresponding transtion from an existential state in Q (by Remark C.10.1), we can take $k_i \in K_i \cap E$ and $k_{i+1} \in \Delta(k_i, a) \cap K_{i+1}$.

   By Lemma C.4.1 we know $P' = \{p' \in \Delta(p_i, a) \mid k_{i+1} \leq_{de} p'\}$ is non-empty. We choose $P_{i+1} \in P' \cap max_a^{de}(p_i)$. Define $\sigma(T) = (K_{i+1}, p_{i+1})$. Hence $k_{i+1} \in K_{i+1}$ and $k_{i+1} \leq_{de} p_{i+1}$, therefore $K_{i+1} \sqsubseteq_{de} p_{i+1}$, as required.

2. $(K_i, p_i, a, d, 0, d, 1)$ is a suffix of T or (since *Duplicator* makes the next two moves) $(K_i, p_i, a, d, 0, d, 1)(K_{i+1}, p_i, a, d, 1)$ is a suffix of T'. That is, $(K_i, p_i) \in U_{de} \times E$.

   We take $k_i \in K_i$ (and $k_i \in U$). By Lemma C.4.1, there are $k_{i+1} \in \Delta(k_i, a)$ and $p_{i+1} \in \Delta(p_i, a)$ such that $k_{i+1} \leq_{de} p_{i+1}$. We choose $k_{i+1} \in min_a^{de}(k_i)$, hence $K_{i+1} = [k_{i+1}]_{de} \in \Delta_{de}^s$. Thus, $K_{i+1} \sqsubseteq p_{i+1}$. We define,

$$\sigma(T) = (K_{i+1}, p_i, a, d, 1)$$
$$\sigma(T') = (K_{i+1}, p_{i+1})$$

3. $(K_i, p_i, a, s, 1, d, 0)(K_i, p_{i+1}, a, d, 0)$ is a suffix of T. Hence $(K_i, p_i) \in U_{de} \times U$.

   We take $k_i \in K_i$ such that $k_i \leq_{de} p_i$. By Lemma C.4.1 there is $k_{i+1} \in \Delta(k_i, a)$ such that $k_{i+1} \leq_{de} p_{i+1}$. We take the minimal such k_{i+1} (hence $k_{i+1} \in min_a^{de}(k_i)$) and define $K_{i+1} = [k_{i+1}]_{de} \in \Delta(K_i, a)$. Since $k_{i+1} \leq_{de} p_{i+1}$ it follows that $K_{i+1} \sqsubseteq p_{i+1}$. We define,

$$\sigma(T) = (K_{i+1}, p_{i+1})$$

Hence σ is a $\sqsubseteq_{de}$-respecting minimax strategy. □

We now show that $\sqsubseteq_{de}$-respecting minimax strategies can be composed.

Corollary C.12.3 *Let $K_0 \in Q/\equiv_{de}, p_0 \in Q$ such that $K_0 \sqsubseteq_{de} p_0$, and $q_0 \in Q$ such that $p_0 \leq_{de} q_0$. Let σ be a $\sqsubseteq_{de}$-respecting minimax strategy for Duplicator on $G(K_0, p_0)$ and let σ^{de} be a Duplicator winning strategy for $G^{de}(p_0, q_0)$.*

Then $\sigma \bowtie \sigma^{de}$ is a $\sqsubseteq_{de}$-respecting strategy.

Proof. The proof is by induction.

In the base case, let τ be a *Spoiler* strategy for $G^x(K_0, q_0)$, and let $T = ((t_j)_{j<\omega}, w)$ be the $(\tau, \sigma \bowtie \sigma^{de})$-conform protoplay. We have $k \in K_0$ such that $k \leq_{de} p_0 \leq_{de} q_0$ and, by Lemma C.4.2, $k \leq_{de} q_0$, therefore $K_0 \sqsubseteq_{de} q_0$.

Now, let $i \in \omega$ and $T_i = ((t_j)_{j\leq i}, w[0..i-1])$ be the prefix of T of length $i+1$. Let $t_i = (K_i, q_i)$ and let $(p_j)_{j\leq i}$ be the intermediate p-sequence of T_i. Assume $K_i \sqsubseteq_{de} p_i \leq_{de} q_i$. We show that $K_{i+1} \sqsubseteq_{de} p_{i+1} \leq_{de} q_{i+1}$ holds for the next $(Q/\equiv_{de} \times Q)$-position $t_{i+1} = (K_{i+1}, q_{i+1})$ of T and the next state of the intermediate p-sequence. There are four cases:

- Assume $(K_i, q_i) \in U^{de} \times U$. Let $t_i^0 := \tau(\mathcal{E}(T_i)) = (K_i, q_i, a, s, 1, d, 0)$ and $t_i^1 := \tau(\mathcal{E}(T_i)t_i^0) = (K_i, q_{i+1}, a, d, 0)$. Let $\sigma \bowtie \sigma^{de}(\mathcal{E}(T_i)t_i^0 t_i^1) = (K_{i+1}, q_{i+1})$, and p_{i+1} be the next state of the intermediate p-sequence.

  If $p_i \in E$, the definition of $\sigma \bowtie \sigma^{de}$ implies $K_{i+1} \sqsubseteq_{de} p_{i+1}$, since $\mathcal{E}(T')$ is σ-conform (both K_{i+1} and p_{i+1} are chosen according to σ). Also, $p_{i+1} \leq_{de} q_{i+1}$ by Lemma C.4.1, since $p_i \leq_{de} q_i$ and $(p_i, q_i) \in E \times U$. Hence $K_{i+1} \sqsubseteq_{de} p_{i+1} \leq_{de} q_{i+1}$.

  If $p_i \in U^1$, the definition of $\sigma \bowtie \sigma^{de}$ also implies $K_{i+1} \sqsubseteq_{de} q_{i+1}$, since $\mathcal{E}(T'')$ is σ^{de}-conform (p_{i+1} is chosen according to σ^{de}, hence $p_{i+1} \leq_{de} q_{i+1}$). Because $\mathcal{E}(T'_{i+1})$ is σ-conform (that is, K_{i+1} is chosen according to σ), we have $K_{i+1} \sqsubseteq_{de} p_{i+1} \leq_{de} q_{i+1}$.

- The remaining three cases are simlarly analogous to Lemma C.4.3

Since $K_{i+1} \sqsubseteq_{de} p_{i+1} \leq_{de} q_{i+1}$ we have $k_{i+1} \in K_{i+1}$ such that $k_{i+1} \leq_x p_{i+1}$, and therefore $k_{i+1} \leq_{de} q_{i+1}$ by application of Lemma C.4.2 and $K_{i+1} \sqsubseteq_{de} q_{i+1}$. Hence $\sigma \bowtie \sigma^{de}$ is $\sqsubseteq_{de}$-respecting. □

Lemma C.12.1 *Let $K_0, p_0, q_0, \sigma, \sigma^{de}$ be chosen like Corollary C.12.3.*

For every Spoiler strategy τ in $G^{de}(K_0, q_0)$, $p_0 \notin F$ implies that the $(\tau, \sigma \bowtie \sigma^{de})$-conform play contains a position $(K_j, q_j) \in Q_{de} \times \overline{F}$. That is, $\sigma \bowtie \sigma^{de}$ is a winning strategy for Duplicator in $G(K_0, q_0)$ with winning set $\{u \in P^\omega \mid \exists i (u_i \in Q_{de} \times \overline{F}\}$.

Proof. For a *Spoiler* strategy τ in $G^{de}(K_0, q_0)$, let $p_0 \notin F$, and $T = ((t_i)i < \omega, w)$ be the $(\tau, \sigma \bowtie \sigma^{de})$-protoplay.

For contradiction, assume there is no $i \in \omega$ with $t_i = (K_i, q_i) \in Q_{de} \times \overline{F}$. Take $T'' = ((p_i, q_i)_{i<\omega}, w)$ for $(p_i)_{i<\omega}$, the intermediate p-sequence of T. Since T is $(\sigma \bowtie \sigma^{de})$-conform, T'' is σ^{de} conform. However, since there is no $q_i \notin F$, T'' is not a win for *Duplicator* . This contradicts the fact that σ^{de} is a winning strategy for *Duplicator*. Hence, there is a position $t_i \in Q_{de} \times \overline{F}$. □

Finally,

Theorem C.12.1 *Let A be an LWAA, and p, q be states such that $p \leq_{de} q$. $A(q)$ de-simulates $A^s_{de}([p]_{de})$. That is, there is a winning Duplicator strategy in $G^{de}([p]_{de}, q)$.*

Proof. We fix,

1. For every $K \in Q_{de}$, a representative $r(K)$ such that, if $K \cap \overline{F} \neq \emptyset$, then $r(K) \notin F$.

2. For every $(K, p) \in Q_{de} \times Q$ such that $K \sqsubseteq_{de} p$, a $\sqsubseteq_{de}$-respecting mini-max strategy $\sigma^o_{K,p}$ of *Duplicator* for $G(K, p)$. (Such a strategy exists by Corollary C.12.3.)

3. For every $(p,q) \in Q \times Q$ such that $p \leq_{de} q$, a winning *Duplicator* strategy $\sigma^{de}_{p,q}$ for $G^{de}(p,q)$.

Given a prefix T_n of a $G^{de}([p]_{de}, q)$-play T, let $(t_i)_{i \leq n}$ be the subsequence of $(Q_{de} \times Q)$-positions in T_n. Take,

$$j = min\{i \leq n \mid (K_i, q_i) \in \overline{F}_{de} \times F \wedge \forall i'(i \leq i' \leq n \rightarrow q_{i'} \in F)\}$$

or $j = 0$ if this set is empty. Let $T_{[j,i]}$ be the suffix of T_i starting with t_j. Define,

$$\sigma(T_i) = \sigma^{o}_{K_j, r(K_j)} \bowtie \sigma^{de}_{r(K_j), q_j}(T_{[j,i]})$$

σ is $\sqsubseteq_{de}$-respecting by Corollary C.12.3. Therefore, if $t_i = (K_i, q_i)$ is the first $(\overline{F}_{de} \times F)$-position after the last $(Q_{de} \times \overline{F})$-position (or the first at all), we have $K_i \sqsubseteq_{de} q_i$. We then update σ to $\sigma^{o}_{K_j, r(K_j)} \bowtie \sigma^{de}_{r(K_j), q_j}$, where $r(K) \in K_i \cap \overline{F}$, and only the suffix starting with (K_i, q_i) of the play is taken into account when determining the *Duplicator* moves.

This strategy σ forces play to reach a position $(K_j, q_j) \in Q_{de} \times \overline{F}$ (and $K_j \sqsubseteq_{de} q_j$) by Lemma C.12.1. Thus, every $\overline{F}_{de} \times F$ position is followed by a $Q_{de} \times \overline{F}$ in a σ-conform protoplay. That is, σ is a winning *Duplicator* strategy in $G^{de}([p]_{de}, q)$. □

C.12.2 A^s_{de} Simulates A

Corollary C.12.4 *Let $q_0^{'} \in [q_0]_{de}$. There is a Duplicator strategy $\sigma^{\equiv}$ for $G^{de}(q_0^{'}, [q_0]_{de})$ such that, for every $Q \times Q_{de}$-position $(q_i^{'}, [q_i]_{de})$ of a $\sigma^{\equiv}$-conform play, $q_i^{'} \in [q_i]_{de}$.*

Such a strategy is $\equiv_{de}$-respecting.

Proof. Follows immediately from the construction of A^s_{de} and Corollary C.8.1. □

Theorem C.12.2 *Let A be a Büchi automaton with states p_0, q_0 such that $p_0 \leq_{de} q_0$. The automaton $A^s_{de}([q_0]_{de})$ de-simulates $A(p_0)$. That is, there is a winning strategy for Duplicator in $G^{de}(p_0, [q_0]_{de})$.*

Proof. For σ^{de}, a winning *Duplicator* strategy for $G^{de}(p_0, q_0)$, and $\sigma^{\equiv}$, a $\equiv_{de}$-respecting *Duplicator* strategy for $G^{de}(q_0, [q_0]_{de})$, we show $\sigma^{de} \bowtie \sigma^{\equiv}$ is a winning strategy in $G^{de}(p_0, [q_0])$ for *Duplicator*.

Given a *Spoiler* strategy τ for $G^{de}(p_0, [q_0]_{de})$, let $T = (t_i)_{i<\omega}$ be a $(\tau, \sigma^{de} \bowtie \sigma^{\equiv})$-conform protoplay with intermediate q-sequence $(q_i^{'})_{i<\omega}$.

For $T' = ((p_i, q_i^{'}))_{i<\omega}$, we know $\mathcal{E}(T')$ is *de*-conform. Therefore, for every $i < \omega$ with $p_i \notin F$ we have $j \geq i$ such that $q_j^{'} \notin F$. Similarly, for $T'' = ((q_i^{'}, [q_i]_{de}))_{i<\omega}$ we know that $\mathcal{E}(T'')$ is $\sigma^{\equiv}$-conform. Consequently $q_j^{'} \in [q_j]_{de}$. That is, $[q_j]_{de} \notin F_{de}$. $\sigma^{de} \bowtie \sigma^{\equiv}$ is a winning *Duplicator* strategy. □

Theorem C.12.3 *For every LWAA A with a co-Büchi acceptance condition, the automata A and A^s_{de} de-simulate each other. In particular, $\mathcal{L}(A) = \mathcal{L}(A^s_{de})$.*

Proof. Follows from Theorem C.12.1 and Theorem C.12.2. □