THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

Controlling the Generation of Multiple Counterexamples in LTL Model Checking

Sentot Kromodimoeljo

Bachelor of Engineering

A thesis submitted for the degree of Doctor of Philosophy at

The University of Queensland in 2014

School of Information Technology and Electrical Engineering

Abstract

The focus of traditional model checking has been on the verification problem where counterexamples play a secondary role. In many potential uses of model checkers, however, counterexamples play a primary role. For example, in safety analysis, achieving perfect safety in the system being analysed is often impossible or too expensive. In such a case, the analyst is interested in discovering all of the situations that can lead to unsafe conditions in order to assess their likelihood. These situations appear as counterexamples to a system safety property expressed as a temporal logic formula in model checking. This thesis proposes an approach to model checking when counterexample generation is the primary goal. Model checking is viewed as a search for counterexamples rather than simply ensuring that a specification is satisfied by a model. The temporal logic used is Linear Temporal Logic (LTL).

Most existing model checkers stop after the first counterexample is found. The few that can generate multiple counterexample paths typically generate too many counterexample paths that are slight variations of each other. For LTL, a counterexample path is an infinite sequence of states, and the number of counterexample paths for a model checking problem can be infinite. Typically, the analyst is interested in a finite number of classes of counterexample, with each class represented by a single counterexample path. However, the classes of interest are often specific to the problem domain. An approach explored in this thesis is to control the generation of counterexample paths by allowing the analyst to direct the search for a counterexample path to rule in or rule out certain classes of counterexamples. The counterexample paths generated are of the so-called *lasso* form, each consisting of a prefix part (a possibly empty finite sequence of states) and a cycle part (a non-empty finite sequence of states that is repeated forever).

The main technique proposed for controlled generation of counterexamples within a symbolic framework is called *directed counterexample generation*. The search for a counterexample path is directed using two kinds of constraints: a *global constraint* which is a state property that must be satisfied by all states in the counterexample path, and a *cycle constraint* which is a state property that must be satisfied by at least one state in the cycle part of a counterexample path. While global constraints can be easily integrated with existing techniques for counterexample path generation, cycle constraints entail a search technique different from the existing techniques. As well as controlling the generation of multiple counterexample paths, the use of constraints can greatly reduce the search space in generating individual counterexample paths. The framework together with directed counterexample generation provide an infrastructure for exploring the counterexample space in a model checking problem.

Model checking, and thus counterexample generation, suffers from the state explosion problem. Although many techniques have been developed to mitigate the state explosion problem in model checking, including symbolic model checking, no best single combination of techniques for model checking and counterexample generation has been found and it is unlikely that one will be found, since the state explosion problem cannot in general be eliminated. The approach in this thesis is to develop a framework for LTL model checking and counterexample generation where different techniques and strategies can be mixed and matched, including fixpoint and on-the-fly techniques. The proposed framework is intended to support the analysis of finite-state asynchronous systems with interleaving semantics.

The framework is independent of the modelling notation, but the Behavior Tree (BT) notation is used as a concrete example notation for modelling finite-state asynchronous systems. A method for translating models in a substantial subset of the BT notation into objects in the framework is provided. As a proof of concept, a prototype that incorporates the proposed techniques within the framework has been developed. The prototype includes a translator from the BT notation. Experiments with the prototype were performed to demonstrate the advantages of the proposed approach and assess the effects of model checking strategies on directed counterexample generation.

Declaration by author

This thesis is composed of my original work, and contains no material previously published or written by another person except where due reference has been made in the text. I have clearly stated the contribution by others to jointly-authored works that I have included in my thesis.

I have clearly stated the contribution of others to my thesis as a whole, including statistical assistance, survey design, data analysis, significant technical procedures, professional editorial advice, and any other original research work used or reported in my thesis. The content of my thesis is the result of work I have carried out since the commencement of my research higher degree candidature and does not include a substantial part of work that has been submitted to qualify for the award of any other degree or diploma in any university or other tertiary institution. I have clearly stated which parts of my thesis, if any, have been submitted to qualify for another award.

I acknowledge that an electronic copy of my thesis must be lodged with the University Library and, subject to the General Award Rules of The University of Queensland, immediately made available for research and study in accordance with the Copyright Act 1968.

I acknowledge that copyright of all material contained in my thesis resides with the copyright holder(s) of that material. Where appropriate I have obtained copyright permission from the copyright holder to reproduce material in this thesis.

Publications during candidature

1. P.A. Lindsay, K. Winter, and S. Kromodimoeljo. Model-based Safety Risk Assessment using Behavior Tress, In *Proceedings of the 6th Asia Pacific Conference on System Engineering*. Systems Engineering Society of Australia, 2012.

2. S. Kromodimoeljo. A Framework for Symbolic LTL Model Checking. Technical Report 2013-12-03, University of Queensland School of Information Technology and Electrical Engineering, 2013.

Publications included in this thesis

[2] describes an early version of the framework proposed in this thesis. Parts of [2] are incorporated into this thesis as follows:

- A substantially revised Chapter 2 of [2] is incorporated as Chapter 3 of this thesis.
- A substantially revised Chapter 3 of [2] is incorporated as Chapter 5 of this thesis.
- A substantially revised Chapter 5 of [2] is incorporated as Chapter 8 of this thesis.

Contributor	Statement of contribution
Sentot Kromodimoeljo (Candidate)	Researched and wrote the technical report (100%)

Contributions by others to the thesis

No contributions by others.

Statement of parts of the thesis submitted to qualify for the award of another degree

None.

Acknowledgements

I would like to express my gratitude to my supervisors Peter Lindsay and Ian Hayes for their guidance. Peter Lindsay introduced me to model checking in safety analysis, which became the motivation for this thesis. Ian Hayes taught me the importance of having concepts be formalised as simply as possible.

My gratitude to Kirsten Winter for many fruitful discussions on model checking and Behavior Trees, and for providing useful feedback on this thesis. Graeme Smith provided useful feedback early in my study. Other researchers in model checking and Behavior Trees at UQ and Griffith University with whom I have had the pleasure to interact include, in alphabetical order, Kushal Ahmed, Robert Colvin, Irene Havsa, Toby Myers, Abdul Sattar, Kaile Su and Larry Wen.

Abdul Sattar and Natalie Dunstan were very helpful in administering the linkage project that funded my APAI scholarship.

I would also like to thank the non-academic staff at UQ, especially Karen Kinnear and Steve Fick, for making me feel at home on campus.

This work was supported by the following scholarships:

- Australian Postgraduate Award Industry (2010-2014) under Linkage Project grant LP0989363 from the Australian Research Council and Raytheon Australia.
- UQ International Scholarship (2010-2014).

Keywords

linear temporal logic, symbolic model checking, multiple counterexamples, behavior trees

Australian and New Zealand Standard Research Classifications (ANZSRC)

ANZSRC code: 080203, Computational Logic and Formal Languages, 50%
ANZSRC code: 080204, Mathematical Software, 50%

Fields of Research (FoR) Classification

FoR code: 0802, Computation Theory and Mathematics 100%

Contents

List of Figures

List of Tables

List of Symbols and Abbreviations

List of Symbols

φ	-	LTL formula whose satisfiability is being checked
C_φ	-	set of fairness constraints used in checking the satisfiability of φ
$F_{C\varphi}$	-	proposition characterising the set of counterexample states
F_φ	-	proposition characterising the set of fair states

List of Abbreviations

BA	-	Büchi Automaton
BT	-	Behavior Tree
CTL	-	Computation Tree Logic
DFS	-	Depth-First Search
ECS	-	Eager Counterexample Strategy
ERS	-	Eager Reachability Strategy
GBA	-	Generalised Büchi Automaton
LCS	-	Lazy Counterexample Strategy
LRS	-	Lazy Reachability Strategy
LTL	-	Linear Temporal Logic
NNF	-	Negation Normal Form
OBDD	-	Ordered Binary Decision Diagram
SCC	-	Strongly-Connected Component
TGBA	-	Transition-based Generalised Büchi Automaton

Chapter 1

Introduction

1.1 Motivation

In real life, there may be several different ways for something to go wrong. As an example, the failure of an ATM to dispense 500 dollars requested by a person A may be caused by one of the following:

- An automatic payment was made from A's account to a utility company the day before, causing insufficient funds for the requested withdrawal.
- The ATM's keypad malfunctioned, causing the PIN number entered by A to be garbled and not recognised.
- A withdrawal was made by A earlier that day, and withdrawing 500 dollars would exceed A's daily withdrawal limit.

In general, an undesired outcome in real life is caused by a sequence of events (i.e., a behaviour). Suppose we have a model that includes the ATM and A, and allows the above behaviours. The behaviours that lead to an undesired outcome may be viewed as a counterexample to the conjecture that a desired outcome is always achieved in the model. The different causes of an undesired outcome are represented by multiple counterexamples. Often, one would like to know the different behaviours — each represented by a counterexample — that can lead to an undesired outcome.

Similar to the above real life situation where one wants to know the different possible behaviours that can lead to an undesired outcome, in many applications of Linear Temporal Logic (LTL) [Pnu77] model checking [CE81, QS82], the analyst may want to generate multiple counterexamples. For example, in safety analysis, achieving complete safety in the system being analysed is impossible or too expensive [Per84], but one would like to know all of the cases that can lead to unsafe situations, where the sequence of events

that leads to an unsafe situation is represented by a counterexample to a safety property expressed in LTL. Another example application is in the area of coverage-based test case generation [GH99, RH01] where test sequences are derived from counterexamples to trap properties: each trap property is an LTL formula asserting that the negation of some condition of interest holds forever (in this case, a counterexample path leads to a desired outcome whereby the condition of interest is reached). In these and many other types of applications, the interest is in counterexamples that are linear paths (representing sequences of events), thus the focus on LTL.

Most existing model checkers, including NuSMV [CCGR99] and SAL [dMOR+04], stop after finding the first counterexample. The few that can generate multiple counterexamples (e.g., SPIN [Hol04]) often generate too many counterexamples that are slight variations of each other [JD03].

With a model checker that stops after finding the first counterexample, to make the model checker find other counterexamples, one must modify either the temporal logic formula or the model so that the counterexample found is eliminated from the set of behaviours analysed, and then rerun the model checker. This is a time-consuming and error-prone process, and in the case where an LTL formula is modified, the required model checking time can grow exponentially with the size of the modified formula. Having the model checker find other counterexamples without needing to modify the temporal logic formula or the model is preferable.

This thesis proposes a capability to direct the search for a counterexample away from certain parts of the model or towards certain parts of the model. Counterexample paths can then be classified based on parts of the model that are excluded and parts of the model that are included. This provides some control over generation of multiple counterexample paths, with the granularity of the control as coarse or as fine as the parts excluded and/or the parts included. The proposed capability is not to be tied to a specific modelling notation to enable it to be used with various modelling notations for finite-state asynchronous systems.

In addition to being independent of the modelling notation, the proposed capability is to be compatible with as many model checking approaches as possible. This is important because many techniques and strategies have been developed to mitigate the state explosion problem in model checking, but no single combination of techniques and strategies has been found to be best for all model checking problems. Still, symbolic techniques operating on sets of states have been found to be more efficient than techniques that operate on explicit states (see for example, [RV07], in the context of LTL satisfiability checking). Thus the proposed capability is to be symbolic-oriented.

1.2 Research Questions

This thesis addresses the following research questions:

1. Can the search for an LTL counterexample path be directed away from certain parts of a model or towards certain parts of a model?

2. Can a method for such a directed counterexample path generation be made independent of the modelling notation?

3. Can a method for directed counterexample path generation be incorporated into a symbolic framework?

4. How do model checking strategies, such as when reachability is determined, affect directed counterexample path generation?

5. Can directed counterexample path generation be mixed and matched with existing techniques?

1.3 Approach

To answer the research questions, a general symbolic framework for LTL model checking and counterexample generation is first developed. Within the framework, a method for directed counterexample path generation is proposed.

The proposed symbolic framework is intended to support the analysis of asynchronous finite state systems with interleaving semantics. The framework proposed consists of two levels:

- a state machine modelling level that is not specific to any modelling notation, and
- an LTL level that can accommodate various LTL encoding schemes.

The Behavior Tree (BT) notation [Dro03,Dro06] is used as an example modelling notation for asynchronous systems, and examples involving Behavior Trees are used throughout this thesis. Two example LTL encoding schemes are used in this thesis: the classic LTL encoding scheme developed by Lichtenstein and Pnueli [LP85] and the Transition-based Generalised Büchi Automaton (TGBA) encoding scheme of Rozier and Vardi [RV11].

A state machine model in the proposed framework is essentially a Kripke structure plus "elementary blocks," which are intended to represent structures inherited from the modelling notation. By maintaining structures from the modelling notation in elementary blocks, analyses can take advantage of the extra structures and counterexample paths

can be easily mapped back to the modelling notation. Elementary blocks and the use of transfer functions to compute images and pre-images under transition relations for the elementary blocks are concepts borrowed from the field of program analysis.

The LTL level provides a unified framework for encoding an LTL formula φ into a tableau for checking the satisfiability of φ, using any of the well-known LTL encoding schemes (including the classic and TGBA encoding schemes). The framework does not confine symbolic model checking to the traditional fixpoint approach [McM92]. It is intended to allow approaches, techniques and strategies for LTL model checking and counterexample generation to be mixed and matched.

Several novel techniques are developed within the framework, including

- An algorithm for computing reachable states that mimics Kildall's algorithm for program flow analysis [Kil73].

- A novel adaptation into the symbolic framework, of on-the-fly LTL model checking [GPVW95] that is usually associated with an explicit approach.

However, the main technique developed for the controlled generation of multiple counterexamples is:

- A novel technique called *directed counterexample path generation*, in which the cycle part of a counterexample path is searched before the prefix part (the counterexample paths considered are of the so-called *lasso* type consisting of a prefix followed by a cycle that is repeated forever). Direction for the search for a counterexample path is provided through a *global constraint* (a constraint that must be satisfied by all states in the path) and a *cycle constraint* (a constraint that must be satisfied by at least one of the states in the cycle part of the path).

A prototype symbolic LTL model checker that incorporates the proposed techniques has been developed. The prototype includes a translator from a significant subset the BT notation into elementary blocks. Experiments with the prototype were conducted to assess the effectiveness of the techniques proposed in this thesis and to investigate the effects of different model checking strategies on the proposed directed counterexample path generation.

1.4 Thesis Contributions

The main contributions of this thesis are as follows:

- The symbolic framework for LTL model checking and counterexample generation. The unified framework captures essential concepts common to many existing approaches to LTL model checking, allowing different approaches, strategies and techniques to be mixed and matched.

- Generic proof plans for showing the soundness of an LTL encoding scheme within the framework. The proof plans are instantiated for the classic LTL encoding scheme and the TGBA encoding scheme.

- The formulation of transitions as collections of guarded updates, allowing images and pre-images under transition relations to be computed without performing relational products using transition relations represented as entire propositions. Instead, images and pre-images are computed using updates, existential quantification in the logic of quantified boolean formulas [AHU74] and propositional operations.

- An algorithm for computing reachable states that mimics Kildall's algorithm for program flow analysis [Kil73]. The algorithm can be used in cases where computing reachability of states and using the reachability information in model checking is a good strategy. The algorithm provides an alternative to a naive fixpoint computation and in some cases is significantly more efficient than the naive fixpoint computation.

- A novel adaptation of the on-the-fly explicit automata approach to LTL model checking into a symbolic framework. This provides an alternative to the fixpoint approach for symbolic LTL model checking.

- A novel approach for generating symbolic LTL counterexample paths called *directed counterexample generation* whereby the cycle part of a counterexample path is searched before the prefix part (the counterexample paths considered are of the *lasso* type consisting of a possibly empty finite sequence of states called the *prefix* part, followed by a non-empty finite sequence of states that is repeated forever called the *cycle* part). The resulting mechanism facilitates the controlled generation of multiple counterexamples.

1.5 Thesis Structure

- **Chapter 2 Background and Literature Review** — This chapter provides essential background material on temporal logic and Behavior Trees and provides a literature review on model checking and LTL counterexample generation.

- **Chapter 3 The State Machine Framework** — This chapter describes the state machine level of the proposed framework. A state machine in the framework is essentially a Kripke structure with additional structure in the form of elementary blocks.

- **Chapter 4 Translating Behavior Trees** — This chapter describes a method to translate Behavior Trees to objects in the framework. It is intended to illustrate how objects of a source notation can be translated into elementary blocks.

- **Chapter 5 Reachability** — This chapter discusses the concept of reachability in symbolic LTL model checking within the framework. An algorithm for computing reachable states is proposed in this chapter.

- **Chapter 6 Foundation for LTL Model Checking** — This chapter identifies essential concepts in LTL model checking that are common to many LTL model checking approaches and are independent of the LTL encoding scheme. Theorems that need to be proved to demonstrate the soundness of an encoding scheme are presented in a manner independent of the encoding scheme. Proof plans for the theorems are presented (their applications to the classic LTL encoding scheme and the Transition-based Generalised Büchi Automaton (TGBA) encoding scheme appear in Appendix A).

- **Chapter 7 LTL Model Checking within the Framework** — This chapter describes the application of the concepts in Chapter 6 to the state machine framework of Chapter 3, resulting in a symbolic framework for LTL model checking and counterexample generation. The chapter also discusses how various approaches and strategies for LTL model checking and counterexample generation might be implemented within the symbolic framework.

- **Chapter 8 Directed Counterexample Path Generation** — This chapter proposes a novel approach to LTL counterexample path generation. The search for a counterexample path is directed by a *cycle constraint* — a state formula that must be satisfied by a state in the cycle — and the search can be constrained using a *global constraint* — a state formula that must be satisfied by all states in the counterexample path. The cycle part of a counterexample path is searched first, based on the cycle constraint, and a prefix to the cycle is searched after a cycle is found.

- **Chapter 9 Experiments with a Prototype** — This chapter presents results of experiments with a prototype LTL model checker that incorporates the techniques for directed counterexample path generation proposed. The purpose of the

experiments is twofold: to demonstrate the need for multiple strategies and to demonstrate the advantages of directed counterexample path generation when generating multiple counterexample paths. The prototype is a proof of concept rather than a production model checker, and the timing comparisons are mostly between different strategies for model checking and counterexample path generation in the prototype. Nevertheless, some timing comparisons with SAL (Symbolic Analysis Laboratory) [dMOS03] are provided to demonstrate the practicality of the proposed approach.

- **Chapter 10 Conclusion** — This chapter explains how the research questions in Section 1.2 are answered, summarises the main contributions of the thesis, and discusses possible future work.

Chapter 2

Background and Literature Review

This chapter provides background material on temporal logic and Behavior Trees and provides a literature review on model checking and LTL counterexample generation. The Behavior Tree notation is a notation for asynchronous multi-threaded systems used in this thesis to illustrate the concepts that are used in the proposed approach.

2.1 Temporal Logic

The full propositional temporal logic, called CTL* [EH86], adds *path quantifiers* and *temporal operators* to propositional logic. A path in temporal logic is an infinite sequence of states and a temporal operation describes a property on a path. The path quantifiers are **A** (for all paths) and **E** (for some path). The temporal operators are **X** ("next time"), **F** ("eventually"), **G** ("always"), **U** ("strong until") and **R** ("release").

There are two types of formulas in CTL*: state formulas (to be interpreted as propositions about states) and path formulas (to be interpreted as propositions about paths).

Definition 2.1. Given a set AP of atomic propositions, the syntax of CTL* is inductively defined as follows:

- If $p \in AP$ then p is a state formula.
- If f and g are state formulas, then $\neg f$, $f \vee g$ and $f \wedge g$ are state formulas.
- If f is a path formula, then $\mathbf{E}\, f$ and $\mathbf{A}\, f$ are state formulas.
- If f is a state formula, then f is also a path formula.
- If f and g are path formulas, then $\neg f$, $f \vee g$, $f \wedge g$, $\mathbf{X}\, f$, $\mathbf{F}\, f$, $\mathbf{G}\, f$, $f\, \mathbf{U}\, g$ and $f\, \mathbf{R}\, g$ are path formulas.

The propositional operators $\neg$, $\vee$ and $\wedge$ are inherited from standard propositional logic.

Since a state formula is also a path formula, all CTL* formulas can be interpreted as path formulas. But some CTL* formulas cannot be interpreted as state formulas. A temporal operation or a propositional operation with a temporal operation appearing as a factor (e.g., $f \vee (g \wedge \mathbf{F}h)$) can only be interpreted as a path formula.

The semantics of temporal logic is described in terms of a Kripke structure $M = (S, R, L)$.

Definition 2.2. A Kripke structure M is a triple (S, R, L) where S is a finite set of states, R is a relation on $S \times S$ called the transition relation, L is a labelling on S ($L : S \to \mathcal{P}(AP)$) and L determines the atomic propositions that hold in each state ($p \in L(s)$ means the atomic proposition p holds in state s). S together with R determine the set of paths in M. To ensure that all paths are infinite, it is often required that the transition relation R be total on the left argument (there is a transition from every state). Without loss of generality, to simplify the proposed framework, L is required to be injective so that a state is completely determined by its label.

A path π is an infinite sequence of states. The notation π_i is used to denote the $(i+1)$th state in path π (i starts at 0). The notation π^i is used to denote the suffix of π starting from the ith state, e.g., $\pi^2 = \langle \pi_2, \pi_3, ... \rangle$. For each path π in a Kripke structure $M = (S, R, L)$, the following holds:

$$\forall n \geq 0 : (\pi_n, \pi_{n+1}) \in R.$$

The notation $M, s \models f$ is used to indicate that the state formula f holds in state s in Kripke structure M. The notation $M, \pi \models f$ is used to indicate that the path formula f holds for path π in Kripke structure M.

Definition 2.3. The semantics of CTL* is determined by the $\models$ relation, inductively

defined as follows:

$$
\begin{aligned}
M, s \models p &\Leftrightarrow p \in L(s), \text{ if } p \text{ is an atomic proposition,} \\
M, s \models \neg p &\Leftrightarrow M, s \not\models p, \\
M, s \models p \vee q &\Leftrightarrow (M, s \models p) \vee (M, s \models q), \\
M, s \models p \wedge q &\Leftrightarrow (M, s \models p) \wedge (M, s \models q), \\
M, s \models \mathbf{E}\, p &\Leftrightarrow \exists \pi : \pi_0 = s \wedge (M, \pi \models p), \\
M, s \models \mathbf{A}\, p &\Leftrightarrow \forall \pi : \pi_0 = s \Rightarrow (M, \pi \models p), \\
M, \pi \models p &\Leftrightarrow M, \pi_0 \models p, \text{ if } p \text{ is an atomic proposition,} \\
M, \pi \models \neg p &\Leftrightarrow M, \pi \not\models p, \\
M, \pi \models p \vee q &\Leftrightarrow (M, \pi \models p) \vee (M, \pi \models q), \\
M, \pi \models p \wedge q &\Leftrightarrow (M, \pi \models p) \wedge (M, \pi \models q), \\
M, \pi \models \mathbf{X}\, p &\Leftrightarrow M, \pi^1 \models p, \\
M, \pi \models \mathbf{F}\, p &\Leftrightarrow \exists i \geq 0 : (M, \pi^i \models p), \\
M, \pi \models \mathbf{G}\, p &\Leftrightarrow \forall i \geq 0 : (M, \pi^i \models p), \\
M, \pi \models p\, \mathbf{U}\, q &\Leftrightarrow \exists i \geq 0 : (M, \pi^i \models q) \wedge \forall 0 \leq j < i : (M, \pi^j \models p), \\
M, \pi \models p\, \mathbf{R}\, q &\Leftrightarrow \forall j \geq 0 : (\forall i < j : (M, \pi^i \not\models p)) \Rightarrow (M, \pi^j \models q).
\end{aligned}
$$

Computation tree logic (CTL) [Eme81] is a fragment of CTL* in which all temporal operations must be immediately quantified thus producing state formulas. As a result, one can work entirely with state formulas in CTL model checking. However, strong fairness assumptions cannot be expressed in CTL (the $\mathbf{GF}p$ in $(\neg\mathbf{GF}p) \vee q$ is a strong fairness assumption asserting that p is satisfied infinitely often in an infinite path).

Linear temporal logic (LTL) [Pnu77] is the path quantifier free fragment of CTL*. Strong fairness assumptions can be expressed in LTL, but the CTL formula $\mathbf{AG}\,\mathbf{EF}p$ (always able to go from any state to a state satisfying p) cannot be expressed in LTL. Because LTL is a fragment of the full propositional temporal logic, the semantics of LTL can be inferred from the semantics of the full propositional temporal logic. An LTL specification is implicitly quantified over all paths.

2.2 Model Checking

This section provides a quick survey on model checking. For an in-depth treatment of model checking, the reader can consult a textbook on model checking such as [CGP99] or [BK08].

Model checking is a process whereby the satisfiability of a specification expressed in a modal logic is checked against a model that can be translated into a Kripke structure. Although parts of the specification may be interpreted over infinite paths, model checking does not operate on infinite paths directly. Instead, model checking operates on states and transitions.

2.2.1 CTL Model Checking

The field of model checking started in the early 1980s with pioneering work by Clarke and Emerson [CE81] on CTL model checking. At around the same time, and independently, Queille and Sifakis [QS82] developed a CTL model checking algorithm operating on Petri nets. CTL is simple to model check because one can work entirely with state formulas and a bottom up processing of subformulas of the formula being checked can be performed.

CTL model checking is based on computing states that satisfy each of the state subformulas of the formula being checked. The states that satisfy a state subformula p are completely determined by states that satisfy each of p's immediate state subformulas and the transition relation R of the model. For example, if $p \equiv \mathbf{EG}q$, then the states that satisfy p are completely determined by states that satisfy q and the transition relation R. States that satisfy a subformula that is an atomic proposition can be determined from the labelling function L of the Kripke structure. In an explicit CTL model checker such as [CE81], the computation of states satisfying p may involve the construction of strongly connected components (SCCs). For example, if $p \equiv \mathbf{EG}q$ and $S' = \{s \in S \mid M, s \models q\}$, then one can form $M' = (S', R', L')$ where $R' = R|_{S' \times S'}$ and $L' = L|_{S'}$ and construct the SCCs in the graph of states and transition in M'. The states that satisfy $\mathbf{EG}q$ are those that belong to a non-trivial SCC plus states in S', each of which has a path (via R') to a non-trivial SCC.

2.2.2 Fairness Constraints

A fairness constraint is a state formula that must be satisfied infinitely often in an infinite path. To enable the checking of properties under strong fairness assumptions, many CTL model checkers allow the addition of a set C of fairness constraints to the model. A path π is fair with respect to C if each fairness constraint in C is satisfied infinitely often in π. In CTL model checking with fairness constraints C, only paths that are fair with respect to C are considered.

In the explicit CTL model checker of [CE81], determining fairness involves the construction of strongly connected components (SCCs) in the graph of states (nodes) and transitions (edges), where each fairness constraint is fulfilled by a state in each of the

SCCs. In the symbolic CTL model checker of [McM92], in the presence of fairness constraints, computations of states satisfying certain temporal operations involve nested fixpoint operations, e.g.,

$$\mathbf{EG}p \triangleq \nu Z.p \wedge \bigwedge_{c \in C} \mathbf{EX}(\mu Y.(Z \wedge c) \vee (p \wedge \mathbf{EX}\, Y))$$

where the outer greatest fixpoint operation (νZ) interacts with the inner least fixpoint operations (μY), and C is the set of fairness constraints.

Fairness constraints add complexity to CTL model checking. Whereas the complexity of pure CTL model checking is $O(|f|\cdot(|S|+|R|))$ where $|f|$ is the size of the formula f being checked (the number of symbols in f), $|S|$ is the number of states in the model and $|R|$ is the size of the transition relation in the model (as a set of ordered pairs), the complexity of CTL model checking with a set of fairness constraints C is $O(|f| \cdot (|S| + |R|) \cdot |C|)$ (see, e.g., [CGP99]).

2.2.3 LTL Model Checking

Unlike CTL which is state oriented, LTL is very much path oriented. An LTL specification f is implicitly quantified over all paths, and thus is taken to be the CTL* formula $\mathbf{A}f$. Model checking is performed by checking the satisfiability of φ where $\varphi \equiv \neg f$. Since φ is a path formula, satisfiability checking amounts to finding a path π such that $M, \pi \models \varphi$. Usually it is understood that $\pi_0 \in S_0$ where $S_0 \subseteq S$ is a designated set of initial states. To check path formulas in a states-and-transitions setting, the states are augmented with *path commitments*, and *transition constraints* are added based on the path commitments, producing an augmented model M'. A path commitment to a path formula p in an augmented state s' may be viewed as a commitment for s' to start only suffix paths that satisfy p. We will denote the commitment to satisfy p, $S_\varphi(p)$. Thus an augmented state in which $S_\varphi(p)$ holds is commited to start suffix paths that satisfy p.

Path commitments and transition constraints determine a tableau, which is a decision graph for checking the satisfiability of φ (the use of tableaux to decide satisfiability of formulas was first proposed by Beth [Bet55]). The tableau guarantees that the paths in M', when projected to M, cover all the paths that need to be considered. However, it does not guarantee that a considered path that starts at an augmented state committed to φ satisfies φ. A subformula may require that some condition eventually holds. As an example, if $\mathbf{F}g$ holds in a suffix π^i where $i \geq 0$, then g needs to hold in some suffix π^j where $j \geq i$. The formula g is called an eventual condition. To guarantee that a path that starts at a state committed to φ does in fact satisfy φ, eventual conditions in the

path must be checked. An *LTL encoding scheme* determines how the tableau for φ is generated and how eventual conditions can be checked.

The first LTL model checking algorithm was developed by Lichtenstein and Pnueli [LP85]. Their algorithm operates on explicit states and they call their augmented states *atoms*. Their check for eventual conditions involves constructing strongly connected components (SCCs) which are *self-fulfilling* in terms of the eventual conditions: if a node (an atom) in an SCC is committed to a suffix path in which an eventual condition q is required, e.g., if $S_\varphi(p \mathbf{U} q)$ holds in the atom, then there is an atom in the SCC in which $S_\varphi(q)$ holds. The SCCs are constructed in a reachability graph involving atoms (instead of unaugmented states) where the transitions are constrained by the tableau for φ as well as the transition relation R. However, instead of keeping track of when $S_\varphi(p \mathbf{U} q)$ holds and when $S_\varphi(q)$ is fulfilled, Lichtenstein and Pnueli introduced

$$\neg S_\varphi(p \mathbf{U} q) \vee S_\varphi(q) \tag{2.1}$$

as a fairness constraint. For each subformula of φ of the form $p \mathbf{U} q$, a fairness constraint as in 2.1 is constructed. Lichtenstein and Pnueli showed that an SCC is self-fulfilling if and only if each fairness constraint holds in at least one atom in the SCC. φ is satisfiable if and only if there is a prefix path starting at an initial atom in which $S_\varphi(\varphi)$ holds to a self-fulfilling SCC. The complexity of the algorithm of Lichtenstein and Pnueli is $O((|S| + |R|) \cdot 2^{O(|\varphi|)})$, i.e., it is exponential in the size of φ.

An alternative approach for LTL model checking is the automata approach, typically performed on-the-fly [CVWY92,GPVW95]. As with the original approach for LTL model checking, the on-the-fly explicit automata approach uses fairness constraints to track the fulfillment of eventual conditions. Each fairness constraint must be fulfilled infinitely often in a path that satisfies φ. The tableau for φ (equivalent to $\neg f$) imposed on the model together with the fairness constraints for φ may be viewed as forming a generalised Büchi automaton (GBA) which can be transformed into a Büchi automaton (BA). To avoid the exponential complexity of precise tracking of fairness constraints, in many implementations, the BA essentially uses imprecise tracking of fairness constraints, thus some paths that are accepted by the GBA are not accepted by the BA. However, if there is a path that is accepted by the GBA, then there is a path that is accepted by the BA. A depth first search is performed to find a path that is accepted by the BA. If a path is found, then it satisfies φ and represents a counterexample for f in the model. If the search properly terminates without finding a path, then f holds in the model.

2.2.4 Symbolic Model Checking

Symbolic model checking was first advocated by McMillan [McM92]. Symbolic model

checking operates on sets of states rather than individual states. Typically, a set is constructed for states that satisfy a state subformula. A set of states is characterised using a proposition, and ordered binary decision diagrams (OBDDs) [Bry86] are used to represent propositions. Set operations are performed using OBDD operations. In symbolic model checking as advocated by [McM92], computation of strongly connected components in explicit model checking is replaced by fixpoint computation based on the approach of Emerson and Lei [EL86], using the fixpoint operations of μ-calculus [Koz83]. As an example,

$$\mathbf{EG}q \triangleq \nu Z.q \wedge \mathbf{EX}Z.$$

As with explicit CTL model checking, symbolic CTL model checking proceeds bottom up on the state subformulas of the formula being checked.

Symbolic model checking was the first breakthrough in mitigating the state explosion problem. Burch et al [BCM+92] were able to apply symbolic model checking on a system with 10^{20} states. However, the performance of OBDDs can be highly sensitive to the variable ordering, where an OBDD variable represents an uninterpreted atomic proposition (not necessarily an atomic proposition of the Kripke structure) and in a direct traversal from an OBDD "root" to any leaf, the variable nodes in the traversal must be ordered according to the variable ordering. In some cases there is no variable ordering that can prevent an explosion in the size of the OBDD.

The first symbolic LTL model checking algorithm was proposed in [BCM+92]. It is based on encoding atomic path commitments as extra state variables, adding transition constraints based on the atomic path commitments, and using fairness constraints to track the fulfillment of eventual conditions, in a CTL model checker with fairness constraints. Although in practice the algorithm is often much faster than the explicit algorithm of [LP85], the complexity of the algorithm is the same as [LP85].

2.2.5 State Space Reduction Techniques

In addition to symbolic model checking, state space reduction techniques have been developed to mitigate the state explosion problem. These techniques include partial order reduction, compositional reasoning and abstraction. Partial order reduction is used in the case of parallel programs with an interleaving model where some interleavings are eliminated from the model because they are already represented by other interleavings that have the same effects [PWW96]. In compositional reasoning, a system is decomposed into smaller parts and the *assume-guarantee* paradigm is used [MC81]. Abstraction techniques include those that do not change the meaning of the analysis such as removing variables that do not affect the properties being checked (often called variables outside the

cone of influence [CGP99]), and those that potentially change the meaning of the analysis such as data abstraction [CGL94] and predicate abstraction [GS97]. An approach called *counterexample-guided abstraction refinement* [CGJ+00] performs refinements to an abstracted model to remove "spurious" counterexamples (a "spurious" counterexample is a counterexample found for an abstracted model that is not a counterexample for the original unabstracted model), to automatically arrive at an appropriate level of abstraction.

Most of the state space reduction techniques are orthogonal to "core model checking," and can be applied in conjunction with the techniques and strategies proposed in this thesis. A notable exception is partial order reduction. To be effective, partial order reduction must be applied in an on-the-fly approach to model checking.

2.3 Counterexample Generation

Although there has been much work on state space reduction and efficient LTL model checking, there has been little work on LTL counterexample generation. Traditionally, the focus of model checking has been on the verification problem with counterexamples playing a secondary role.

2.3.1 Basic LTL Counterexample Path Generation

The two main techniques for LTL counterexample path generation in use are:

- The technique of Clarke et al [CGMZ95] for symbolic CTL model checking with fairness constraints. The technique searches for a counterexample path guided by states that satisfy fairness constraints. The technique is efficient and tends to generate counterexample paths with relatively short prefixes and cycles.

- On-the-fly LTL model checking using explicit automata [GPVW95]. Because the technique used to generate a Büchi automaton from a generalised Büchi automaton uses imprecise tracking of fairness constraints (thus the BA accepts a subset of the counterexamples paths accepted by the GBA) and the search is a depth-first search, on-the-fly LTL model checking as proposed by [GPVW95] tends to produce counterexample paths with unnecessarily long prefixes and cycles.

Additionally, there is the technique of [SB05] that generates the shortest counterexample paths for LTL with past operators. None of the cited techniques for generating LTL counterexample paths address the issue of controlling the generation of multiple counterexamples.

2.3.2 Generating Multiple Counterexamples

Most model checkers stop after finding the first counterexample. SPIN [Hol04] can generate multiple counterexamples, but the multiple counterexamples tend to contain too many that are slight variations of each other. In fact, the difficult part of generating multiple counterexamples is how to generate "meaningful" variations. In general, what constitutes a meaningful variation is domain-specific. Nevertheless, a model checker ought to support the generation of meaningful variations by providing appropriate mechanisms.

Without the appropriate mechanisms, the generation of meaningful multiple counterexamples requires multiple runs of the model checker. After each run, if the run produced a counterexample, the counterexample is examined and either the model or the temporal logic formula is modified for the next run, based on the examination. For some applications, the process can be automated (see e.g., [JD03]). Rerunning the model checker is expensive, and in the case of LTL, the modification to the temporal formula can increase the running time significantly. The problem is compounded if the modification is manual, since it is error-prone.

In contrast, with the appropriate mechanisms, only the counterexample path generator is rerun, and no modification to the temporal formula nor the model is required. This can significantly reduce the running time compared to rerunning the model checker, and it avoids potential errors associated with modifying the temporal formula or the model.

Chechik and Gurfinkel [CG05] developed a framework for counterexample generation and exploration, but only for pure CTL without fairness constraints. There are other frameworks for counterexamples/witnesses[1] but most of them are for certifying the results of model checking. They include a proof system for certifying counterexamples/witnesses for CTL* developed by Namjoshi [Nam01], and a framework for generating certifiable symbolic counterexamples/witnesses for pure CTL (without fairness constraints) developed by Shankar and Sorea [SS04]. Except for the framework of Chechik and Gurfinkel, which does not apply to LTL, none of the frameworks address the issue of generating multiple counterexamples.

Techniques that may provide some control when generating multiple counterexamples are those for counterexample generation in probabilistic model checking for discrete-time Markov chains [HKD09, SVV09]. Some measure of control is provided by the probabilistic techniques, e.g., by generating only counterexamples that exceed some threshold probability. However, the techniques only apply in a probabilistic model, specifically a discrete-time Markov chain model. Applications in safety analysis often treat likelihood more qualitatively than probabilities in a discrete-time Markov chain model. In addition,

[1] in CTL/CTL* model checking, a witness proves the validity of an existential specification.

the control (filtering effect) obtained by using a probabilistic technique does not address coverage issues in safety analysis and test case generation.

2.4 Behavior Trees

The Behavior Tree (BT) notation is a graphical notation for specifying system behaviours, originally developed by Dromey to support the process of capturing requirements and transforming them into designs [Dro03, Dro06]. Dromey adopted the tree structure notation to help cope with complexity because humans have more difficulty coping with an arbitrary graph structure. Other issues that strongly influenced the design of the Behavior Tree notation include:

- traceability to requirements, and
- integrating several trees (from several requirements) into a single tree.

Figure 2.1 shows an example BT that specifies the operation of a simple, perhaps unrealistic, vending machine. The system starts with the vending machine in a ready state, and waits for a customer to insert a coin. After a customer inserts a coin, the system waits for the customer to select either candy or chips. Once the customer selects either candy or chips, the vending machine dispenses the appropriate selection, and the cycle is repeated.

A BT consists of nodes (drawn as boxes) representing actions and arrows between nodes representing control flow. The tree structure of a BT represents the basic control flow structure, which consists of sequencing and branching. In the example, there is an alternative branching after a customer inserts a coin. The branch taken depends on whether the customer selects candy or chips. In the BT notation, the selection of alternatives is based on guards. There is another kind of branching in the BT notation called parallel branching. Unlike alternative branching, all branches in a parallel branching are taken and each branch starts a new subthread. Sequential nodes may be merged to form an atomic composition whereby the execution of the sequence of nodes becomes indivisible and uninterruptible.

In addition to the basic control flow structure, there are other control flows in the BT notation, specified using control flow directives. In the example, there are nodes with the reversion ($\wedge$) directive. The reversion directive is the most important of all BT directives, as it is the directive for constructing a loop. It kills and restarts the execution of a subtree rooted at its "matching" node. Killing the execution of a subtree causes all threads activated in the subtree to be killed as well. Since the kill and restart are intended

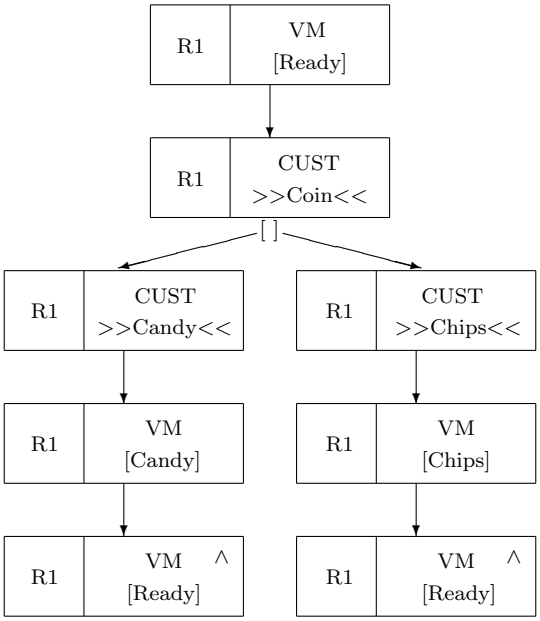

Figure 2.1: Simple Vending Machine BT

to produce a loop, the subtree must include the node with the reversion directive, thus the matching node must be an ancestor of the node with the reversion directive. In the example, the matching node for both reversions is the root of the entire BT. Other BT directives include the reference (=>) directive (similar to "go to"), the kill (−−) directive for killing the execution of a subtree, and the synchronise (=) directive for synchronising threads.

The BT notation allows a sequence of nodes to be composed into an atomic composition, drawn as a contiguous sequence of boxes without arrows. The nodes in an atomic composition is executed atomically with the effect of the execution the same as though the nodes are executed in sequence.

Each BT node refers to a component (VM or CUST in the example), and has a *behaviour* that is either a state realisation (e.g., [Ready]), a guard (e.g., ???Ready???), a selection (e.g., ?Ready?), an event (e.g., ??Coin??), an internal output (e.g., <Coin>), an internal input (e.g., >Coin<), an external output (<<Coin>>), or an external input (e.g., >>Coin<<). A state realisation node indicates that a component enters a particular state or that an attribute of a component gets a particular value. In the example, we have state realisation nodes where VM enters the "ready," "candy" and "chips" states, modelling the corresponding behaviours abstractly. A guard node waits until its guard expression holds before proceeding. A selection node is like a guard node except it does not wait until its guard holds, instead its thread is killed if the guard does not hold. An

event node is also like a guard node, except it waits for an event. Inputs and outputs are similar to events, however, an internal input can only occur simultaneously with the corresponding internal output (e.g., >Coin< waits for <Coin> and proceeds simultaneously with <Coin>), but internal output does not have to wait for the corresponding input to be ready. The example BT has three external input events modelling the cases where the system waits for a customer (CUST) to insert a coin, select candy and select chips respectively.

Each BT node can have a requirement tag, which provides traceability to requirements. The example BT has all of the nodes tagged R1 since it is assumed that there is only a single requirement named R1. For a large BT that models a large set of requirements, each node can be tagged according to which requirements are relevant. Since requirements traceability is not relevant for this thesis, we will instead use the tags as node identifiers by tagging each node differently.

A BT node may be viewed as a guarded command [Dij75] that is executed atomically. For an atomic composition of nodes, guards of all the nodes in the composition must be enabled for the atomic composition to be executed. The execution of BT nodes from parallel threads are interleaved, unless they are synchronised. BT nodes that are synchronised with each other are executed simultaneously. The BT notation allows for non-deterministic choice between internal behaviours in an alternative branching. In addition, when a BT is not coupled with an environment, events and external I/O may be viewed as occurring non-deterministically.

The graphical nature of the BT notation makes it natural to have tools for BTs with visual effects. An execution path of a BT (e.g., a counterexample path) may be animated by highlighting the BT nodes as the "execution" steps through the nodes. The status of active threads may also be shown in the animation, e.g., by highlighting (in a different fashion) "active" nodes. Some of the structural constraints of BTs may ease the comprehension of animations, e.g., we need not worry about a node and its ancestor being active at the same time.

The full syntax of the BT notation is described in [Beh07] and a formal semantics for the BT notation is described in [CH11]. Although the BT notation is part of a larger methodology called Behavior Engineering [BE], it is a useful in its own right as a notation for modelling behaviour.

Several tools are available for BTs. They include BESE [PLTP08], Integrare [WLC+07] and TextBE [tex]. However, none of the tools support model checking directly, instead, the tools generate code for the Symbolic Analysis Laboratory (SAL) [dMOS03], where model checking can be performed.

BTs have been used by Raytheon Australia on large systems [Bos08]. In the application area of failure modes and effects analysis, BTs have been used on several case studies including analysis of the Airbus A320 hydraulic system [LWY10].

2.5 Summary

The necessary background material for the thesis has been presented in this chapter. Section 2.1 described temporal logic, essentially as defined in [CGP99]. Section 2.2 provided a brief literature survey on model checking. Section 2.3 on counterexample generation provided context for the main contribution of the thesis: directed counterexample path generation. Finally, Section 2.4 briefly introduced Behavior Trees: the modelling notation used as an example in this thesis.

Chapter 3

The State Machine Framework

This chapter describes the state machine level of the proposed framework. Although in theory Kripke structures already provide a sufficient framework for model checking, in practice it is convenient and beneficial to have additional structures in the framework based on the structures in the modelling notation. Potential benefits of having the additional structures include:

- The mapping of a model in some modelling notation into the framework and the mapping back of the result of model checking into the modelling notation can both be made straightforward.
- Model checking and other analyses can take advantage of the additional structures.

The proposed framework is intended to support symbolic analyses of asynchronous finite-state systems with interleaving semantics, but is otherwise a general framework. The framework includes basic concepts that are present in modelling notations for asynchronous finite-state transition systems, without the excess baggage of a modelling notation. Elements of the framework include:

- *guarded updates*: restricted forms of Dijkstra's *guarded commands* [Dij75],
- *transfer functions*: functions to compute images and pre-images under transition relations,
- *elementary blocks*: components of a model, each of which is executed atomically with the effect of its execution being deterministic, and
- *program counters*: state variables that represent execution states of threads.

Although transfer functions are derived from models, they are included in the framework because of their important role in the proposed approach for symbolic analyses.

Section 3.1 on states and guarded updates and Section 3.2 on elementary blocks describe the concepts in an explicit setting. Section 3.3 describes how the concepts map to a symbolic domain where sets are characterised by propositions, and how transfer functions in the symbolic domain can be implemented efficiently using ordered binary decision diagrams (OBDDs) [Bry86].

3.1 States and Guarded Updates

Since the system being modelled is a finite state transition system, there is a finite set V of state variables:

$$V = \{v_1, v_2, ..., v_n\}, \tag{3.1}$$

where $type(v_i)$ is a non-empty finite enumeration for $1 \leq i \leq n$. A *state* assigns each state variable with a value from its type ($v_i \leftarrow d_i$ with $d_i \in type(v_i)$ for each $1 \leq i \leq n$): i.e., a state is a set of assignments where each state variable is assigned exactly once. Thus, a state can be viewed as a set of pairs $\{(v_1, d_1), (v_2, d_2), ..., (v_n, d_n)\}$ (a relation and a function) or simply an n-tuple $(d_1, d_2, .., d_n)$, with $d_i \in type(v_i)$ for each $1 \leq i \leq n$, and the set S of possible states is

$$S \triangleq type(v_1) \times type(v_2) \times ... \times type(v_n). \tag{3.2}$$

The atomic propositions of the system being modelled are simple equalities of the form $l = r$ where l is v_i and $r \in type(v_i)$ for some $1 \leq i \leq n$. From a Kripke structure point of view, if S is to become part of a Kripke structure (S, R, L), then the truth value of an atomic proposition in a state must be able to be determined from the labelling function L of the Kripke structure i.e., we have $\forall s \in S : (l = r) \in L(s) \Leftrightarrow (l \leftarrow r) \in s$.

In addition to states, a finite state transition system also has a finite set of *transitions* between states. The set of transitions can be viewed as forming a single monolithic transition relation. However, operationally, it is often convenient to view transitions as updates to states. To that end, we add to the framework *guarded updates* which are restricted forms of guarded commands [Dij75]. A guarded update is a pair (g, u) where *guard* g is a propositional formula and u is a simple *update*. The idea is that a state s in which guard g holds can transition to a state t that is the result of updating s with u. Thus a guarded update may be viewed as representing a set of transitions.

A guard g holds in a state s (denoted $s \models g$) if, by replacing occurrences of state variables in g with their assigned values in s, g evaluates to *true*. The evaluation is performed using simple equality checks and propositional calculus. As an example, the following is the evaluation of $g \equiv v_1 = a_1 \vee v_2 = b_3$ in the state $s = \{v_1 \leftarrow a_2, v_2 \leftarrow$

$b_3, v_3 \leftarrow c_1\}$ where $type(v_1) = \{a_1, a_2\}$, $type(v_2) = \{b_1, b_2, b_3, b_4\}$, and $type(v_3) = \{c_1, c_2\}$:

$$\begin{aligned} g &\equiv v_1 = a_1 \vee v_2 = b_3 \\ &\equiv a_2 = a_1 \vee b_3 = b_3 \text{ (substituting variables with values assigned in } s) \\ &\equiv \mathit{false} \vee \mathit{true} \text{ (evaluating equalities)} \\ &\equiv \mathit{true} \text{ (propositional calculus).} \end{aligned}$$

A simple update u assigns a subset of the state variables with values in their types, i.e., each state variable is assigned at most once. Like a state, an update can be viewed as a relation or a function. Unlike a state, however, not all state variables need to be assigned, i.e., the relation or function need not be total. The state variables that are not assigned retain their values from before the update (they are unchanged by the update). The result of an update u on a state s is written $s \oplus u$ and is defined as follows:

$$s \oplus u \triangleq \left\{ v \leftarrow d \;\middle|\; (v \leftarrow d) \in u \vee (v \leftarrow d) \in s \wedge \neg(\exists e : (v \leftarrow e) \in u) \right\} \tag{3.3}$$

i.e., $\oplus$ is the override operator if we treat s and u as functions. As an example, if $s = \{v_1 \leftarrow a, v_2 \leftarrow b, v_3 \leftarrow c\}$ and $u = \{v_1 \leftarrow d, v_3 \leftarrow e\}$, then $s \oplus u = \{v_1 \leftarrow d, v_2 \leftarrow b, v_3 \leftarrow e\}$.

A guarded update represents a set of transitions, i.e., there is a transition relation associated with a guarded update. For a guarded update (g, u), its transition relation, denoted $R_{(g,u)}$, is defined as follows:

$$R_{(g,u)} \triangleq \{(s, t) \mid (s \models g) \wedge t = s \oplus u\}. \tag{3.4}$$

Because the framework is intended for symbolic analyses, operations in the framework use transfer functions for computing images and pre-images under transition relations rather than the transition relations themselves. For a guarded update (g, u), the transfer functions for computing the image and pre-image under its transition relation are denoted $f_{(g,u)}$ and $r_{(g,u)}$ respectively:

$$f_{(g,u)}(I) \triangleq \{t \mid (s, t) \in R_{(g,u)} \wedge s \in I\} = \{s \oplus u \mid s \in I \wedge (s \models g)\}. \tag{3.5}$$

$$r_{(g,u)}(O) \triangleq \{s \mid (s, t) \in R_{(g,u)} \wedge t \in O\} = \{s \mid (s \oplus u) \in O \wedge (s \models g)\}. \tag{3.6}$$

Although guarded updates seem overly restrictive, for a finite-state transition system, an arbitrary atomic guarded command can always be encoded as a finite collection of guarded updates. The use of guarded updates simply makes the cases of an atomic guarded command more explicit. Arbitrary atomic guarded commands are to be modelled as elementary blocks, described in the next section.

3.2 Elementary Blocks

The concept of *elementary blocks* — taken from the field of program flow analysis [NNH99] — plays a central role in the framework. The use of elementary blocks in the framework provides a way of maintaining structures from the source notation. In addition to providing a straightforward bidirectional mapping between objects in the source notation and objects in the framework, elementary blocks allow analyses to take advantage of the structures inherited from the source notation (some of which may have been lost had the source notation been translated into a "flat" Kripke structure). The use of elementary blocks and image and pre-image transfer functions reflects the view that model checking is a special case of program analysis.

Elementary blocks provide a mechanism for "abstract transitions": sets of transitions, each of which is a grouping of transitions. The set of transitions for an elementary block may be viewed as a partition of the transition relation for the system. In the framework, elementary blocks are required to be deterministic (i.e., if (s_1, s_2) and (s_1, s_3) are possible transitions through a particular elementary block, then $s_2 = s_3$). The reasons for requiring elementary blocks to be deterministic are:

- It simplifies the concept of symbolic (abstract) counterexample path, defined in Section 7.1.3.

- It allows partial order reduction [PWW96] — which uses the concept of an abstract transition as a deterministic set of transitions — to use elementary blocks as the abstract transitions.

Although each elementary block is deterministic, the overall system is in general nondeterministic.

An elementary block is constructed from a finite collection of guarded updates. A transition through an elementary block means one of the guarded updates is chosen. The guards of the guarded updates must be mutually exclusive (their pairwise conjunctions are $false$), which ensures that they give rise to a deterministic choice, and thus ensures that the elementary block is deterministic.

Because each guarded update represents a choice, the image transfer function f_b for an elementary block b is defined as follows:

$$f_b(I) \triangleq \bigcup_{(g,u) \in GU(b)} f_{(g,u)}(I) \tag{3.7}$$

where $GU(b)$ is the set of guarded updates for block b. Similarly, the pre-image transfer

function r_b is defined as follows:

$$r_b(O) \triangleq \bigcup_{(g,u)\in GU(b)} r_{(g,u)}(O). \tag{3.8}$$

A system is modelled as a finite collection of elementary blocks where each elementary block represents a transition choice. Whereas the collection of guarded updates for an elementary block gives rise to a deterministic choice, the collection of elementary blocks for a system may give rise to a non-deterministic choice. In addition, multiple elementary blocks may be synchronised to "execute" in parallel (i.e., multiple elementary blocks are chosen), in which case the elementary blocks must update the system state identically.

The image transfer function for a system modelled as a set of elementary blocks B is defined as follows:

$$f_B(I) \triangleq \bigcup_{b\in B} f_b(I). \tag{3.9}$$

Similarly, the pre-image transfer function for the system is defined as follows:

$$r_B(O) \triangleq \bigcup_{b\in B} r_b(O). \tag{3.10}$$

Although the framework adopts the concept of elementary blocks from program analysis, it does not also adopt the static control flow graph of program analysis, since a finer concept of flows possibly involving multiple threads is needed. Instead, the framework uses program counters (PCs) to represent states of thread executions, with each thread having its own PC.

Each elementary block belongs to a specific thread, and thus has a specific PC-value pair specifying the value of the thread's PC at the entrance to the elementary block. For each elementary block b, its PC is denoted $\mathrm{PC}(b)$ and its PC value is denoted $\mathrm{PCval}(b)$. The PC-value pair must be satisfied by each guard:

$$\forall b \in B : \forall (g,u) \in GU(b) : g \Rightarrow \mathrm{PC}(b) = \mathrm{PCval}(b). \tag{3.11}$$

Figure 3.1 shows an example in the form of a Behavior Tree (BT) [Dro03, Dro06], that is used to illustrate the concepts. Each BT node that is not in an atomic composition becomes an individual elementary block. An atomic composition (e.g., that involving BT nodes tagged N3 and N4) becomes a single elementary block.

There are 3 threads in the BT example, thus 3 PCs are used. The choice of PC names (PC1, PC2 and PC3) and the assignment of PC values (e.g., PC1=1 at the entrance of the elementary block for N1) are somewhat arbitrary. What is important is that different points in a thread execution path are assigned different values. However, a PC value of 0 is reserved for indicating that a thread is inactive. The use of PCs in encoding parallel processes is common and dates as far back as 1978 in a paper by Flon and Suzuki [FS78].

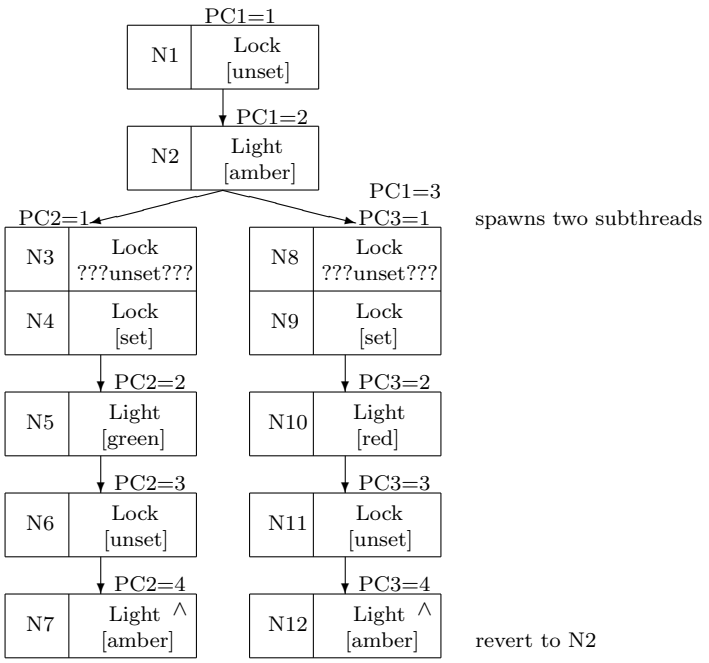

Figure 3.1: A BT Example

In the rest of this thesis, M and M' refer to models in the framework or their corresponding Kripke structures. A model in the framework easily maps to its Kripke structure as follows:

- S of the model becomes S in the Kripke structure.
- $R_B \triangleq \bigcup_{b \in B} \bigcup_{(g,u) \in GU(b)} R_{(g,u)}$ becomes R in the Kripke structure.
- Since atomic propositions are simple equalities, the labelling L in the Kripke structure can be defined such that for each state s, each state variable v, and each $d \in type(v)$, $(v = d) \in L(s) \Leftrightarrow (v \leftarrow d) \in s$.

Note that although temporal logic is defined only for non-terminating systems (i.e., R in the Kripke structure is left-total), the framework is neutral with respect to how to handle termination. Model checking as described in Chapter 7 will work in the presence of termination in M, although some care must be taken in interpreting the result: a terminating run which violates a safety property might not produce a counterexample path.

3.3 Symbolic Approach to Model Checking

Instead of operating on individual states, symbolic model checking operates on sets of states, characterised using propositions. Propositional operations are used to implement

Equality	Equality Encoding	Assignment	Assignment Encoding
$v_1 = a_1$	$\neg v_{10}$	$v_1 \leftarrow a_1$	$\langle v_{10} \leftarrow \mathit{false} \rangle$
$v_1 = a_2$	v_{10}	$v_1 \leftarrow a_2$	$\langle v_{10} \leftarrow \mathit{true} \rangle$
$v_2 = b_1$	$\neg v_{20} \wedge \neg v_{21}$	$v_2 \leftarrow b_1$	$\langle v_{20} \leftarrow \mathit{false}, v_{21} \leftarrow \mathit{false} \rangle$
$v_2 = b_2$	$v_{20} \wedge \neg v_{21}$	$v_2 \leftarrow b_2$	$\langle v_{20} \leftarrow \mathit{true}, v_{21} \leftarrow \mathit{false} \rangle$
$v_2 = b_3$	$\neg v_{20} \wedge v_{21}$	$v_2 \leftarrow b_3$	$\langle v_{20} \leftarrow \mathit{false}, v_{21} \leftarrow \mathit{true} \rangle$
$v_2 = b_4$	$v_{20} \wedge v_{21}$	$v_2 \leftarrow b_4$	$\langle v_{20} \leftarrow \mathit{true}, v_{21} \leftarrow \mathit{true} \rangle$
$v_3 = c_1$	$\neg v_{30}$	$v_3 \leftarrow c_1$	$\langle v_{30} \leftarrow \mathit{false} \rangle$
$v_3 = c_2$	v_{30}	$v_3 \leftarrow c_2$	$\langle v_{30} \leftarrow \mathit{true} \rangle$

Table 3.1: Encoding equalities and assignments using OBDD variables

set operations: logical disjunctions for set unions and logical conjunctions for set intersections. Image and pre-image transfer functions become functions that operate on propositions.

The main purpose of a symbolic approach is to delay the enumeration of cases until necessary, to avoid, if possible, enumerating some of the cases. Thus, symbolic model checking does not need to imply that a transition system must be represented as a single proposition describing the transition relation for the system. Nor does it need to imply that a fixpoint approach such as that of Emerson and Lei [EL86] is used. It simply means operations are performed on propositions representing sets of states (sets of transitions are not necessarily represented entirely using propositions). Using this generalised notion of a symbolic approach allows strategies commonly associated with the explicit automata approach to be used.

The use of ordered binary decision diagrams (OBDDs) [Bry86] to represent propositions and speed up propositional operations is common in symbolic model checking (see, e.g., [BCM+92]). If no inference mechanism (e.g., equational reasoning) other than propositional reasoning is used for non-temporal reasoning, then equalities and assignments in guarded updates must be encoded using OBDD variables. The equalities and assignments in the example in Section 3.1 might be encoded as in Table 3.1. The encoding for $g \equiv v_1 = a_1 \vee v_2 = b_3$ would be

$$g \equiv v_1 = a_1 \vee v_2 = b_3 \equiv \neg v_{10} \vee \neg v_{20} \wedge v_{21}.$$

For convenience, an update is represented as a sequence (since in the context of OBDD variable ordering, an update may be viewed as a totally ordered set). Since several OBDD variables may be required to encode a single state variable, a single assignment of a state variable is represented as a sequence of assignments of OBDD variables. To get the encoding for an update, the sequences for the assignments must be merged. The

corresponding equality encodings represent post conditions that hold after the updates. For example, if $u = \{v_2 \leftarrow b_2\}$ then, assuming the OBDD variables are ordered as $v_{10}, v_{20}, v_{21}, ...,$ u is represented as the sequence $u = \langle v_{20} \leftarrow true, v_{21} \leftarrow false \rangle$ and $v_{20} \wedge \neg v_{21}$ holds immediately after the update u is applied. The function *post* that produces the post condition for an update is defined as follows:

$$\begin{aligned} post(\langle v \leftarrow true \rangle \frown u) &\equiv v \wedge post(u), \\ post(\langle v \leftarrow false \rangle \frown u) &\equiv \neg v \wedge post(u), \\ post(\langle\rangle) &\equiv true \end{aligned}$$

where $\frown$ is the sequence append operation.

A cautionary implementation note: some combinations of values for OBDD variables may correspond to undefined values for state variables. Depending on the intended semantics of the model, uninitialised state variables may or may not be assumed to have defined values. If they are assumed to have defined values, then undefined values can be precluded by excluding them from the set of initial states. On the other hand, if uninitialised state variables are not assumed to have defined values, the type of each state variable must include an undefined value.

Transfer functions operate on propositions rather than sets. Thus, the definition of the image transfer function for a guarded update (g, u) in (3.5) is replaced by:

$$f_{(g,u)}(I) \triangleq (I \wedge g) \oplus u \tag{3.12}$$

where I is a proposition that characterises a set of states and $\oplus$ is now a symbolic override operation. The correctness of (3.12) is easy to see: $(I \wedge g)$ characterises the set $\{s \mid s \in I \wedge s \models g\}$.

The post condition holds after a transition through the guarded update, thus

$$f_{(g,u)}(I) \Rightarrow post(u)$$

for any I. The symbolic version of the override function $\oplus$ is defined as follows:

$$\begin{aligned} P \oplus (\langle v \leftarrow true \rangle \frown u) &\equiv v \wedge ((\exists v : P) \oplus u), \\ P \oplus (\langle v \leftarrow false \rangle \frown u) &\equiv \neg v \wedge ((\exists v : P) \oplus u), \\ P \oplus (\langle\rangle) &\equiv P \end{aligned}$$

where $\exists v : P$ is existential quantification in the logic of quantified boolean formula (QBF) [AHU74]:

$$\exists v : P \triangleq P|_{v \leftarrow true} \vee P|_{v \leftarrow false}$$

where $P|_{v \leftarrow true}$ is a restriction operation: restricting v to *true* in P (i.e., replacing v with *true* in P). In effect, $\exists v : P$ ignores v in the proposition P. The symbolic override operation can have an efficient OBDD implementation with the assignments performed in a single traversal (assuming u is ordered according to the OBDD ordering).

The symbolic version of the pre-image transfer function for a guarded update uses the *ignore* function defined using following rules:

$$\begin{aligned} ignore(\langle\rangle, P) &\equiv P, \\ ignore(\langle v \rangle \frown V, P) &\equiv ignore(V, (\exists v : P)) \end{aligned}$$

where the first argument is a sequence of OBDD variables and the second argument is a proposition. For an update u represented as a sequence of OBDD variable assignments, the OBDD variables assigned in u, represented as a sequence (ordered according to the OBDD variable ordering), is denoted $variables(u)$, and can be constructed as follows:

$$\begin{aligned} variables(\langle v \leftarrow d \rangle \frown u) &= \langle v \rangle \frown variables(u), \\ variables(\langle\rangle) &= \langle\rangle. \end{aligned}$$

The definition of the pre-image transfer function in (3.6) is replaced by:

$$r_{(g,u)}(O) \triangleq ignore(variables(u), O \wedge post(u)) \wedge g. \tag{3.13}$$

The correctness of (3.13) is slightly more difficult to see than that of (3.12). $O \wedge post(u)$ characterises the biggest subset of O such that each element of the subset is a state that can be the result of applying the update u ($\{t \in O \mid \exists s : t = (s \oplus u)\}$). The set of states that can be updated by u to states in the subset is characterised by $ignore(variables(u), O \wedge post(u))$. Finally, of the states that can be updated by u to states in O, only those satisfying the guard g can transition, thus the conjunction with g.

For an elementary block b, the symbolic version of the image transfer function is as follows:

$$f_b(I) \triangleq \bigvee_{(g,u) \in GU(b)} f_{(g,u)}(I) \tag{3.14}$$

where $GU(b)$ is the set of guarded updates for block b. Similarly, the symbolic version of the pre-image transfer function for b is as follows:

$$r_b(O) \triangleq \bigvee_{(g,u) \in GU(b)} r_{(g,u)}(O). \tag{3.15}$$

Note that although the transfer functions for an elementary block are defined in terms of transfer functions for guarded updates, they can be implemented directly using relational

products, where the transition relation for each elementary block is represented by a proposition (see [McM92]).

For the overall system defined by a set of elementary blocks B, the symbolic version of the image transfer function is defined as follows:

$$f_B(I) \triangleq \bigvee_{b \in B} f_b(I). \tag{3.16}$$

Similarly, the symbolic version of the pre-image transfer function for the system is defined as follows:

$$r_B(O) \triangleq \bigvee_{b \in B} r_b(O). \tag{3.17}$$

All symbolic image and pre-image transfer functions are functions from propositions to propositions (they are not functions from Boolean to Boolean). It is not difficult to show that they distribute over disjunctions ($\vee$) and conjunctions ($\wedge$).

Symbolic model checking was initially developed with a fixpoint approach, first proposed by Emerson and Lei [EL86], and made practical by McMillan [McM92] using an OBDD implementation. The fixpoint approach uses the fixpoint operations of μ-calculus [Koz83]. The fixpoint operations may be viewed as operating on a finite complete lattice that is the powerset of a set of states S where the ordering relation is the subset relation ($\subseteq$), the join and meet operations are $\cup$ and $\cap$ respectively, the bottom element is the empty set, and the top element is S. The least fixpoint operator is μ and the greatest fixpoint operator is ν. In both

$$\mu Z.f(Z) \text{ and } \nu Z.f(Z),$$

$f(Z)$ is a proposition characterising a set of states, and the result of the operation is also a proposition characterising a set of states. The function f is a function from propositions to propositions that is monotonic with respect to the lattice. As an example, if S_0 characterises the set of initial states, then

$$\mu Z.S_0 \vee f_B(Z)$$

characterises the set of reachable states. Since f_B distributes over $\vee$, the function f defined as

$$f(Z) = S_0 \vee f_B(Z)$$

is monotonic with respect to the appropriate lattice. Since the lattice is finite, fixpoints can be computed based on Kleene's fixpoint theorem.

The framework does not restrict symbolic analysis to a fixpoint approach. Elementary blocks and their transfer functions support search strategies normally associated with

explicit model checking. Paths can be searched symbolically without first computing fixpoints. However, there are differences between a search for a path with explicit states and a search with symbolic states.

In a search with explicit states, e.g., using a depth-first strategy, one starts at an initial state and at each choice point chooses a transition that determines the next state. The next state is then checked to see if it has been "visited" to determine if the search has reached a dead end or a "fulfilling" cycle has been found. If a dead end is reached, then the search backtracks to a previous choice point that still has an alternative and the search continues from there. If a fulfilling cycle is found then the search terminates successfully. Otherwise the search continues forward by choosing the next transition. The sequence of states chosen that "terminates" in a fulfilling cycle corresponds to a path in the sense of temporal logic (it is an infinite path when the cycle is repeated forever).

In a forward search with symbolic states, the search starts at an initial symbolic state, say SS_0. At each step i, an elementary block b_i is chosen and the next symbolic state SS_i is computed using b_i's image transfer function:

$$SS_i \equiv f_{b_i}(SS_{i-1}).$$

One needs to ensure that $SS_i \not\equiv false$. Note that since a transition through an elementary block is deterministic, $|SS_i| \leq |SS_{i-1}|$, where $|SS_i|$ is the cardinality of the set characterised by SS_i.

Checking if a symbolic state has been visited can be performed naively by checking whether the exact symbolic state has been visited (using equality). However, dead-ends can be detected sooner if subsumption testing is used instead. Using subsumption testing, a symbolic state SS_i is said to have been visited if there is a j such that $0 \leq j < i$ and $(SS_i \wedge SS_j) \equiv SS_i$. Subsumption testing can also be used to detect a fulfilling cycle, but the cycle found needs to be verified.

In a search with explicit states, it is guaranteed that a state in the path found can transition to the next state in the path. The corresponding guarantee in a symbolic path is each explicit state in a symbolic state SS_{i-1} must be able to transition through elementary block b_i to an explicit state in SS_i, and each state in SS_i must be the result of transitioning from a state in SS_{i-1} through b_i. Thus we require that

$$\forall j : 0 \leq j < i \Rightarrow (SS_{j+1} \equiv f_{b_{j+1}}(SS_j)) \wedge (SS_j \equiv r_{b_{j+1}}(SS_{j+1})). \tag{3.18}$$

However, during a symbolic search, (3.18) can be violated, i.e.,

$$\exists j : 0 \leq j < i \wedge (SS_j \not\equiv r_{b_{j+1}}(SS_{j+1})).$$

Such a violation occurs when some states in SS_j cannot transition through elementary block b_{j+1}. To ensure that a symbolic path found satisfies (3.18) for all $i \geq 0$, the symbolic

states may need to be narrowed. Suppose a cycle is found that starts at SS_k and ends at SS_i. Then the following operation needs to be performed iteratively with j from $i-1$ to k:

$$SS_j \leftarrow SS_j \wedge r_{b_{j+1}}(SS_{j+1}). \tag{3.19}$$

The resulting cycle is verified by ensuring that $SS_k \equiv SS_i$. Once the cycle is verified, the rest of the symbolic states are narrowed by iteratively performing (3.19) with j from $k-1$ to 0.

3.4 Summary

A state machine framework for symbolic model checking and counterexample generation has been presented that adds additional structure to the standard Kripke structure. The concept of guarded update was introduced to simplify the exposition and lead to simple implementations. The main structuring mechanism beyond the Kripke structure is that of elementary blocks. The purpose of elementary blocks is to enable analyses to take advantage of the structures from the modelling notation and simplify the mapping of analysis results back to the modelling notation. A salient feature of the framework is the use of image and pre-image functions under transition relations rather than the transition relations themselves, thus avoiding the use of relational products. Examples of how the concepts can be implemented with the use of OBDDs were provided, but the framework does not preclude the use of other techniques for propositional reasoning.

Chapter 4

Translating Behavior Trees

This chapter describes a method for translating BTs into objects in the proposed framework. A substantial subset of the BT notation [Beh07] is handled by the translation, based on the formal semantics defined in [CH11]. The purpose of this chapter is to show how objects in a modelling notation can be translated into objects in the framework.

4.1 Overview

Some features of the BT notation result in BT models that cannot be model-checked without abstraction or inductive reasoning. For example, the semantics of arithmetic expressions in the BT notation requires Presburger arithmetic extended (with negation) to integers. In addition, there are features of the BT notation in [Beh07] that do not have formal semantics in [CH11]. As a result, the following features of the BT notation are not handled by the translation:

- the use of operators in expressions (they may involve infinite domains),
- the *may* and *start new* directives (no formal semantics),
- condition operations on nodes (no formal semantics), and
- multiple component instances (no formal semantics).

The omission of the above features allows the translation to focus on behavioural aspects of BTs. The translation assumes that each component or attribute is of an enumerated type, whose values can be finitely enumerated.

A translator from the BT notation to the SAL notation [dMOS03] that uses translation rules described in [GWY08] already exists. There are important differences, in addition to having different targets, between the translation here and the translation used in the BT-to-SAL translator. They are:

- The translation of reference BT nodes. The BT-to-SAL translator restricts a reference BT node to reference only a BT node in the same thread, and uses a "goto" semantics for references. The translation here allows references to BT nodes in other threads, and uses the "copy" semantics as specified in [CH11]. As an option in the translation here, references to BT nodes in the same thread can use the goto semantics. Note that in general the goto semantics produces a set of behaviours that can be different from that produced by the copy semantics, even for a reference to a BT node in the same thread (the latter was pointed out by Peter Lindsay in a private communication).

- The translation of external input/output event BT nodes. The BT-to-SAL translator defines a Boolean SAL input variable for an external input event and requires the variable to have the value *true* to enable a transition through a BT node with the event, and defines a Boolean SAL output variable for an external output event and sets the variable to *true* upon transition through a BT node with the event. The translation here does not associate a state variable with an external input/output event.

- The translation of internal input/output event BT nodes. The BT-to-SAL translator provides many options for the semantics of internal input/output, including whether to buffer and whether output waits for a corresponding input to be ready. The translation here simply uses the non-blocking semantics specified in [CH11].

The translation of BTs to elementary blocks with their collections of guarded updates is performed in stages in the following order.

1. **Expansion of references.** This is needed because of the copy semantics of references as specified in [CH11]. However there is an option to not expand references to BT nodes in the same thread and treating them as gotos.

2. **Assigning PCs.** The main requirements are that each thread must have a unique PC and the value of a thread's PC must be different at different execution points in the thread (see Section 3.2).

3. **BT node level mapping.** The guarded updates are constructed incrementally. The "behaviour" part of a BT node determines a default guarded update for the node. Because of branches, directives, atomic compositions, internal I/Os, synchronisations and selections, the collections of guarded updates may need to be modified. The modifications are performed incrementally. The effects of branching

and directives (except the synchronisation directive) are processed at the BT node level. The rest must be processed after the processing of atomic compositions.

4. **Creating elementary blocks.** The execution model for an atomic composition is that it is really atomic: not only is it uninterruptible, but we can never see the intermediate states. As a result, BT nodes in an atomic composition are merged into a single elementary block. For optimisation purposes, PC values are reassigned after the elementary blocks have been created, so that the range of values for each PC is contiguous (since atomic compositions may cause some PC values to become unused).

5. **Elementary block level mapping.** BT nodes that are synchronised with each other can only "execute" if they are all "ready." Because of the atomicity of atomic compositions, "readiness" applies at the elementary block level, thus synchronisations and internal I/Os are processed after the elementary blocks are created. Note that the synchronisation for internal I/O is only partial synchronisation, but "readiness" applies at the elementary block level here as well. Finally, because of the peculiarity of the semantics of selections, they must be processed last.

Sections 4.2 through 4.6 describe the stages of the translation in detail.

4.2 Expansion of References

The first stage in a BT translation is the expansion of references (BT nodes with the => directive). The semantics of BTs described in [CH11] specifies that a reference is to be replaced by a copy of the subtree rooted at the referenced node. This process is called expansion since it is similar to macro expansion in some programming languages.

The semantics of BTs described in [CH11] actually restricts references to be to BT nodes in sibling alternative branches. If the goto semantics is used, such cases can be handled as in Section 4.4.3. The well-formedness requirements of references can be relaxed to allow a reference to a BT node in another thread (i.e., a BT node in a different parallel branch), since the copy semantics correctly handles such a case. This necessitates the "expansion" of a reference.

Without loss of generality, references are restricted so that a target is strictly on the left of the reference, i.e., the target node occurs earlier in a depth-first preorder traversal than the source node and the target node must not be an ancestor of the source node. The target of a reference is also restricted to be a non-leaf node. The expansion of references is performed in a left-to-right fashion (one may use a depth-first left-to-right traversal to find and replace the references, since references must be from leaf nodes).

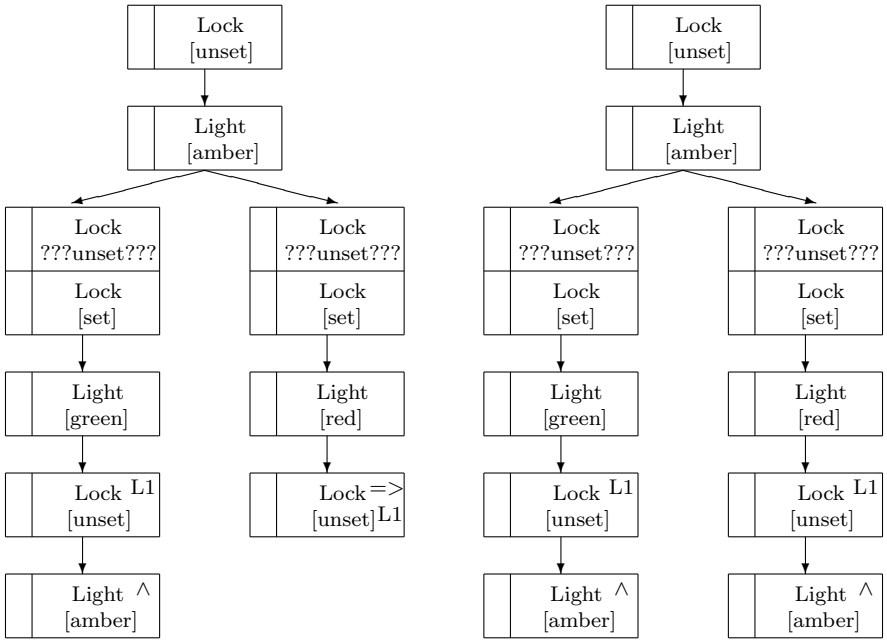

Figure 4.1: Expansion of a reference

The target of a reference must match the source in that the component and "behaviour" parts of the target node must be the same as the corresponding parts of the source node. Additionally, the source and target must either both be unlabelled or be labelled identically. There may be multiple BT nodes that qualify as a target for a reference (i.e., the BT nodes match the source and satisfy the restrictions above). In such a case, we apply the tie-breaking rules in Section 4.4.3 to choose the target. Figure 4.1 shows the expansion of a reference to a BT node labelled L1, with the expanded BT on the right.

If the goto semantics is used for references whose targets are in the same thread, such references are not expanded. (Two nodes are in the same thread if their nearest common ancestor is not a parallel branching node and there is no parallel branching node in the paths — excluding the ends — from the nearest common ancestor to the nodes.) Instead, they are treated as a gotos and handled as in Section 4.4.3.

4.3 Assigning PCs

The second stage in the translation of a BT is assigning PCs. The main role of a PC is to indicate the state of a thread in an execution in terms of where it is. Each thread in a BT is assigned a PC unique to that thread. The value of the thread's PC indicates the status of the thread. A PC value of 0 indicates that the thread is inactive. A non-zero natural number value indicates that the thread is either active or suspended (after it spawned

subthreads) and indicates where in the BT the thread is active or suspended.

When a thread is about to make a choice among alternative branches, it is simultaneously at the entry points of the alternative branches. In such a case, a single execution point for the thread (indicated by the thread's PC value) corresponds to multiple BT points that are entrances to BT nodes, each of which is the start of an alternative branch.

Associated with a node is an entry point and an exit point. The exit point is connected to the entry points of the node's children (if any). The connection is reflected in the consistency between the value of the node's PC at the node's exit and the value at the entrances to the node's children. However, if the node is a parallel branching node, each of the children will have its own PC to reflect the fact that it starts a new thread.

The method of assignment here is simply one of many possible ways of of assigning PCs that satisfy the requirements that each thread must have a unique PC and the value of a thread's PC must be different at different execution points in the thread. Each node is assigned with a PC and values for the PC at the entry and exit points of the node. A PC identifier is completely determined by its index: a strictly positive integer. A PC identifier is constructed by concatenating the string `"PC"` with the string representing the index. For example, the identifier for a PC with index 11 is `"PC11"`. For a node n:

- the PC index is denoted $PCidx(n)$,
- the PC entry value is denoted $PCval(n)$, and
- the PC exit value is denoted $xPC(n)$.

For convenience in describing the rules for assigning PCs and PC values, a next PC index is assigned for each node and is denoted $xidx(n)$ for node n. The rules for assigning the 4-tuple $\langle PCidx, PCval, xPC, xidx\rangle$ are as follows:

1. For r the root node of the BT:
   - $PCidx(r) \leftarrow 1$,
   - $PCval(r) \leftarrow 1$,
   - $xPC(r) \leftarrow 2$, and
   - $xidx(r) \leftarrow 2$.
2. If a non-leaf node n_0 that is not a branching node has been assigned a 4-tuple, then its child n_1 is assigned a 4-tuple as follows:
   - $PCidx(n_1) \leftarrow PCidx(n_0)$,
   - $PCval(n_1) \leftarrow xPC(n_0)$,

- $xPC(n_1) \leftarrow xPC(n_0) + 1$, and
- $xidx(n_1) \leftarrow xidx(n_0)$.

3. If an alternative branching node n_0 with i branches (to nodes $n_1...n_i$) has been assigned a 4-tuple, then n_1 is assigned a 4-tuple as per rule 2. For $j = 2, ..., i$, if n_{j-1} and its descendants have been assigned 4-tuples, then n_j is assigned assigned a 4-tuple as follows:

   - $PCidx(n_j) \leftarrow PCidx(n_0)$,
   - $PCval(n_j) \leftarrow xPC(n_0)$,
   - $xPC(n_j) \leftarrow xPC(m_1) + 1$, and
   - $xidx(n_j) \leftarrow xidx(m_2)$,

   where m_1 is the last descendant of n_{j-1} in the same thread to be assigned a 4-tuple and m_2 is the last descendant of n_{j-1} (not necessarily in the same thread) to be assigned a 4-tuple ($m_1 = m_2 = n_{j-1}$ if n_{j-1} does not have a descendant, $m_1 = n_{j-1}$ if n_{j-1} is a parallel branching node).

4. If a parallel branching node n_0 with i branches (to nodes $n_1...n_i$) has been assigned a 4-tuple, then n_1 is assigned a 4-tuple as follows:

   - $PCidx(n_1) \leftarrow xidx(n_0)$,
   - $PCval(n_1) \leftarrow 1$,
   - $xPC(n_1) \leftarrow 2$, and
   - $xidx(n_1) \leftarrow xidx(n_0) + i$.

   For $j = 2, ..., i$, if n_{j-1} and its descendants have been assigned 4-tuples, then n_j is assigned a 4-tuple as follows:

   - $PCidx(n_j) \leftarrow xidx(n_0) + j - 1$,
   - $PCval(n_j) \leftarrow 1$,
   - $xPC(n_j) \leftarrow 2$, and
   - $xidx(n_j) = xidx(m)$,

   where m is the last descendant of n_{j-1} to be assigned a 4-tuple ($m = n_{j-1}$ if n_{j-1} does not have a descendant).

Rules 2, 3 and 4 are repeated until all nodes have been assigned 4-tuples.

Once all BT nodes have been assigned 4-tuples, for each node n the PC identifier $PC(n)$ can be assigned the appropriate string based on $PCidx(n)$. Thereafter, only $PC(n)$, $PCval(n)$ and $xPC(n)$ are needed for each node n. Figure 4.2 shows PC values assigned to entry points of BT nodes.

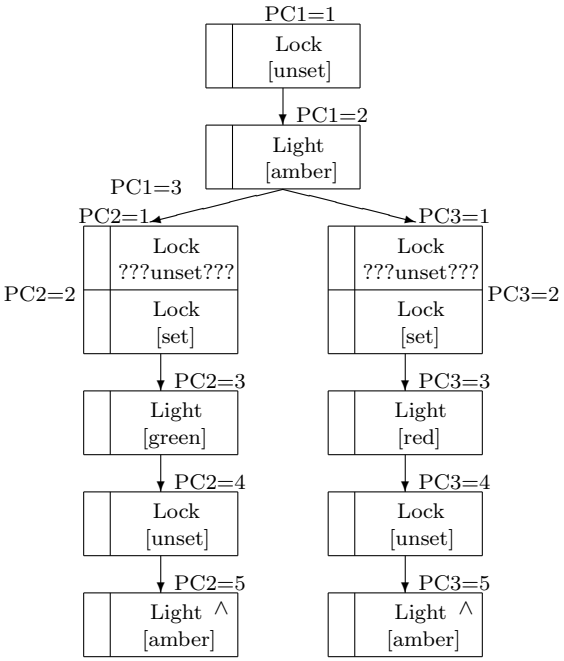

Figure 4.2: Assigned PC Values

4.4 BT Node Level Mapping

The third stage in the BT translation is BT node level mapping. The stage consists of three steps:

1. assigning default guarded updates to BT nodes,
2. processing the effects of branching, and
3. processing control flow directives.

4.4.1 Default Guarded Updates

The first step in the third stage of a BT translation is assigning default guarded updates to BT nodes. To help describe the construction of elementary blocks, a guarded update is broken into the following parts:

Class(n)	PCGuard	MGuard	PCUpdate	MUpdate
update	$PC(n) = PCval(n)$	$true$	$\{PC(n) \leftarrow xPC(n)\}$	$\{ca(n) \leftarrow val(n)\}$
guard	$PC(n) = PCval(n)$	$ex(n)$	$\{PC(n) \leftarrow xPC(n)\}$	{}
event	$PC(n) = PCval(n)$	$true$	$\{PC(n) \leftarrow xPC(n)\}$	{}

Table 4.1: Default Guarded Update Parts

- PCGuard: part of the guard about the relevant PC (usually of the form $PC(n) = PCval(n)$).
- MGuard: the "main" guard (usually not about the relevant PC).
- PCUpdate: update of PC variables.
- MUpdate: update of BT components and component attributes.

A guarded update is constructed from the parts as follows:

$$\big(\text{PCGuard} \wedge \text{MGuard}, \text{PCUpdate} \cup \text{MUpdate}\big).$$

The default values for the guarded update parts are based on the class of the node and are shown in Table 4.1. The three classes of BT nodes are:

- update: consisting of state realisation nodes;
- guard: consisting of selection nodes and guard nodes;
- event: consisting of event nodes, input nodes and output nodes.

Note that for the purpose of this translation, external input and output nodes can be treated as event nodes, thus input and output nodes here are internal I/O nodes.

For a node n in the update class, $ca(n)$ is the component or component attribute specified in node n and $val(n)$ is the component state or attribute value specified in n. For a node n in the guard class, $ex(n)$ is the boolean expression representing the guard. Note that $PC(n)$, $PCval(n)$, $xPC(n)$, $ca(n)$, $val(n)$ and $ex(n)$ are all placeholders.

The guarded update parts can be further modified at later stages of the translation. In addition, the translation of internal I/O causes single guarded updates to be partitioned into multiple guarded updates, causing the parts to be partitioned.

4.4.2 Branching

The second step in the third stage in a BT translation is translating the effects of branching. BTs have alternative branching and parallel branching. Nothing needs to be done

for nodes that start alternative branching. For a node that starts parallel branching, however, the effects of activating the subthreads need to be modified. If node n starts parallel branching to $n_1, ..., n_m$, then the PCUpdate for n needs to be modified as follows:

$$\text{PCUpdate}(n) \leftarrow \text{PCUpdate}(n) \cup \bigcup_{i=1}^{m} \{ (PC(n_i), PCval(n_i)) \}.$$

4.4.3 Control Flow Directives

The third step in the third stage in a BT translation is translating the effects of control flow directives *kill* ($--$), *revert* ($\wedge$) and, optionally, *reference* ($=>$).

A BT node with a kill control directive ($--$), when executed, causes the thread of the target BT node to be killed (deactivated). The "behavior" specified in the BT node must be cleared and replaced by the killing action. Thus if n is a BT node with a kill flag, then parts of n are modified as follows:

$$\begin{aligned} \text{MGuard}(n) &\leftarrow true, \\ \text{PCUpdate}(n) &\leftarrow \text{PCUpdate}(n) \oplus \text{Kill}(target), \\ \text{MUpdate}(n) &\leftarrow \{\}, \end{aligned}$$

where $\oplus$ is the override operator, *target* is the BT node that is the target of the kill, and Kill(*target*) represents killing the target thread:

$$\text{Kill}(target) = \bigcup_{PC \in \text{threadPCs}(target)} \{ PC \leftarrow 0 \},$$

where threadPCs(*target*) gives the PCs of the target thread and all its subthreads.

A BT node with a reversion control directive ($\wedge$), when executed, causes the target thread to be killed and restarted immediately after the target node. The PC update part of a BT node n with a reversion directive is modified as follows:

$$\text{PCUpdate}(n) \leftarrow \text{PCUpdate}(n) \oplus \text{Kill}(target) \oplus \text{PCUpdate}(target).$$

PCUpdate(*target*) will restart the target thread immediately after the target node. The BT node already has much of the "behavior" of the target node[1], thus only the "PC effects" need to be inherited from the target node.

If the goto semantics is used, a BT node with a reference control directive ($=>$), when executed, causes control to jump to the target BT node. This is restricted to cases where the referencing node and the target node are in the same thread. The handling of references when the copy semantics is used is already covered in Section 4.2. As with

[1]The "behavior" of the target node is executed at the node with the directive.

reversion, it is assumed that the "behavior" of the target node is executed at the node with the reference directive. Again, the "PC effects" are inherited from the target node. The PC update component of the referencing node is modified as follows:

$$\text{PCUpdate}(n) \leftarrow \text{PCUpdate}(n) \oplus \text{PCUpdate}(target).$$

The translation of a BT with control flow directives requires the following well-formedness conditions to be met:

- A BT node with a reversion or reference directive cannot have a descendant.
- The target of a kill/reversion/reference cannot be an atomic composition.
- To use the goto semantics, a reference must be to a non-leaf node in the same thread/subthread (the source and target nodes must have the same PC).
- Reversion must be to an ancestor node.

The target of a kill/reversion/reference must match the source in that the component and "behaviour" parts of the source node must be the same as the corresponding parts of the target node. Additionally, the source and target must either both be unlabelled or be labelled identically. There may be multiple BT nodes that qualify as a target for a source (i.e., the BT nodes match and satisfy the constraints above), thus there are "tie-breaking" rules for choosing the target in such a case. The tie-breaking rules are:

1. The target node with the nearest common ancestor with the source wins.
2. If there is still a tie after 1, the target in the leftmost path from the nearest common ancestor wins.
3. If there is still a tie after 2, then one is an ancestor of the others still in contention, and the ancestor wins.

Note that a node and an ancestor of the node have the ancestor as the nearest common ancestor. For a reversion, since the target must be an ancestor, the first rule will break a tie.

4.5 Creating Elementary Blocks

The fourth stage in a BT translation is creating elementary blocks. An elementary block is created for each BT node that is not in an atomic composition. BT nodes in an

atomic composition are merged into a single elementary block. An atomic composition is restricted to contain at most one event node (internal I/O or external event).

To create an elementary block b from the atomic composition of BT nodes $n_1, ..., n_m$, b's parts are set as follows:

$$\begin{aligned}
PC(b) &\leftarrow PC(n_1), \\
PCval(b) &\leftarrow PCval(n_1), \\
\text{PCGuard}(b) &\leftarrow (PC(n_1) = PCval(n_1)), \\
\text{MGuard}(b) &\leftarrow \bigwedge_{i=1}^{m} \text{MGuard}(n_i)[\theta_i], \\
\text{PCUpdate}(b) &\leftarrow \text{PCUpdate}(n_m), \\
\text{MUpdate}(b) &\leftarrow \text{MUpdate}(n_1) \oplus ... \oplus \text{MUpdate}(n_m),
\end{aligned}$$

where $[\theta_i]$ is a substitution operation based on updates:

$$[\theta_i] = \text{MUpdate}(n_1) \oplus ... \oplus \text{MUpdate}(n_{i-1}).$$

Because atomic composition can be combined with alternative branching, a BT node can be the head of more than one block. A BT node n with atomic alternative branching with m branches produces at least m blocks, each constructed as above with n as the first node of each block. If any of the branches in turn is atomically linked to another atomic alternative branching, then it produces more than m branches. In general, a BT node does not have to be one with atomic alternative branching to be the head of multiple blocks because branches are propagated up the tree along atomic compositions.

The combination of atomic composition with parallel branching is disallowed. In addition, an atomic composition is restricted to have at most one synchronised BT node and at most one BT node of the event class.

Each node n that is not in an atomic composition is turned into an elementary block b with

$$\begin{aligned}
PC(b) &\leftarrow PC(n), \\
PCval(b) &\leftarrow PCval(n), \\
\text{PCGuard}(b) &\leftarrow \text{PCGuard}(n), \\
\text{MGuard}(b) &\leftarrow \text{MGuard}(n), \\
\text{PCUpdate}(b) &\leftarrow \text{PCUpdate}(n), \\
\text{MUpdate}(b) &\leftarrow \text{MUpdate}(n).
\end{aligned}$$

Atomic compositions may cause some values of PCs to become unused. As an optimisation, PC values can be reassigned so that the range of values for a PC is contiguous.

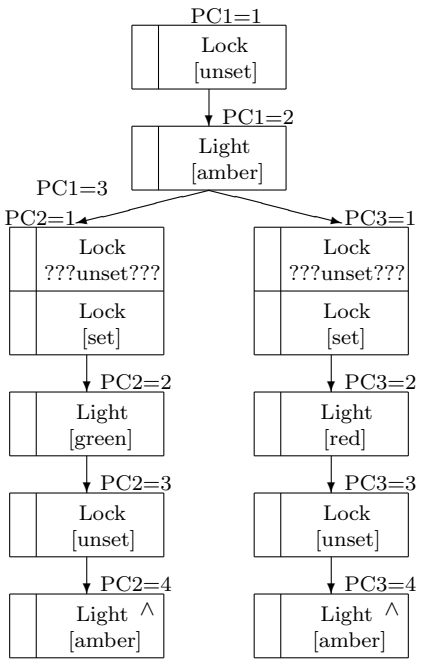

Figure 4.3: Reassigned PC Values

For a PC that has its values reassigned, all updates to the PC and all comparisons involving the PC's values must be adjusted accordingly. Figure 4.3 shows the PC values after reassignment.

4.6 Elementary Block Level Mapping

The fifth stage in a BT translation is elementary block level mapping. It consists of the following steps:

1. processing synchronisation,
2. processing internal input/output,
3. processing selection, and
4. processing prioritisation (optional).

4.6.1 Synchronisation

The first step in the fifth stage of a BT translation is translating the effects of the synchronise control directive (=). Since a synchronised BT node can be part of an atomic composition, the synchronisation is on elementary blocks. If S is a set of elementary

blocks that are to be synchronised, then the parts for S are formed as follows:

$$\begin{aligned} \mathrm{MGuard}(S) &\leftarrow \bigwedge_{b \in S} \mathrm{PCGuard}(b) \wedge \mathrm{MGuard}(b), \\ \mathrm{PCUpdate}(S) &\leftarrow \bigcup_{b \in S} \mathrm{PCUpdate}(b), \\ \mathrm{MUpdate}(S) &\leftarrow \bigcup_{b \in S} \mathrm{MUpdate}(b), \end{aligned}$$

where the parts are assigned to each $b \in S$. If there is a conflict in the update, i.e.,

$$\exists v, a, b : \{v \leftarrow a, v \leftarrow b\} \subseteq (\mathrm{PCUpdate}(S) \cup \mathrm{MUpdate}(S)) \wedge a \neq b,$$

then the BT is considered erroneous. For each $b \in S$, the following parts are set:

$$\begin{aligned} \mathrm{MGuard}(b) &\leftarrow \mathrm{MGuard}(S), \\ \mathrm{PCUpdate}(b) &\leftarrow \mathrm{PCUpdate}(S), \\ \mathrm{MUpdate}(b) &\leftarrow \mathrm{MUpdate}(S). \end{aligned}$$

The synchronised blocks are restricted to not only be in different threads (i.e., have different PCs), but the closest common ancestor of any two of the synchronised blocks must be a parallel branching node. The matching of nodes for full synchronisation is similar to the matching of source and target nodes for kill, reversion and reference (see Section 4.4.3). However, for synchronisation there is no source and target: all of the synchronised nodes must have the synchronisation flag. More than two nodes can participate in a synchronisation, thus no tie-breaking rules are necessary.

4.6.2 Internal I/O

The second step in the fifth stage of a BT translation is translating the effects of internal I/O. Internal output does not have to wait for input to be ready, but all ready inputs must receive the output. Since an atomic composition can have at most one event node, an elementary block is restricted to have at most one I/O node.

An output block can have any finite number of corresponding input blocks, including none. If b is an output block and I is the set of corresponding input blocks for b then the transition for b is partitioned into $2^{|I|}$ guarded updates. The partitions are indexed by $ri \in \mathcal{P}(I)$, representing the input blocks that are ready (an input block i is ready if the condition $\mathrm{PCGuard}(i) \wedge \mathrm{MGuard}(i)$ is satisfied). The partitioned parts for b are

constructed as follows (PCGuard(b) is not partitioned):

$$\begin{aligned}
\text{MGuard}(b)(ri) &\leftarrow \text{MGuard}(b) \wedge \Big(\bigwedge_{i \in ri} \text{PCGuard}(i) \wedge \text{MGuard}(i) \Big) \wedge \\
&\quad \Big(\bigwedge_{i \in I \setminus ri} \neg\big(\text{PCGuard}(i) \wedge \text{MGuard}(i)\big) \Big), \\
\text{PCUpdate}(b)(ri) &\leftarrow \text{PCUpdate}(b) \cup \Big(\bigcup_{i \in ri} \text{PCUpdate}(i) \Big), \\
\text{MUpdate}(b)(ri) &\leftarrow \text{MUpdate}(b) \cup \Big(\bigcup_{i \in ri} \text{MUpdate}(i) \Big).
\end{aligned}$$

An input block can only transition with a corresponding output block. All other matching input blocks that are ready must also transition. Let I be the set of other matching input blocks and O be the set of matching output blocks. The partitions for an input block use two indices: $o \in O$ and $ri \in \mathcal{P}(I)$. The partitioned parts for an input block b are constructed as follows:

$$\begin{aligned}
\text{MGuard}(b)(o)(ri) &\leftarrow \text{MGuard}(b) \wedge \text{PCGuard}(o) \wedge \text{MGuard}(o) \wedge \\
&\quad \Big(\bigwedge_{i \in ri} \text{PCGuard}(i) \wedge \text{MGuard}(i) \Big) \wedge \\
&\quad \Big(\bigwedge_{i \in I \setminus ri} \neg\big(\text{PCGuard}(i) \wedge \text{MGuard}(i)\big) \Big), \\
\text{PCUpdate}(b)(o)(ri) &\leftarrow \text{PCUpdate}(b) \cup \text{PCUpdate}(o) \cup \Big(\bigcup_{i \in ri} \text{PCUpdate}(i) \Big), \\
\text{MUpdate}(b)(o)(ri) &\leftarrow \text{MUpdate}(b) \cup \text{MUpdate}(o) \cup \Big(\bigcup_{i \in ri} \text{MUpdate}(i) \Big).
\end{aligned}$$

Note that the above allows for non-deterministic choice in the case where several matching output blocks are ready.

4.6.3 Selection

The third step in the fifth stage of a BT translation is translating the effects of selection. A selection node in a BT is similar to a guard node, except it must not cause its thread to be blocked, thus it does not wait until its guard becomes true. A selection node can be used to determine which alternative branch to take, in which case all branches must be guarded by selection. If a thread is at a selection point and none of the selections are ready[2] then the thread terminates. A selection node does not need to guard an alternative branch: its parent can be BT node without branching, in which case the thread of the selection node is killed when the node is reached and the selection is not ready (there are no other alternative selections).

[2]A selection is ready if its guard expression evaluates to *true*.

Selections are handled through blocks rather than individual BT nodes. A block is a selection block if it contains a selection node. At any point where there is a selection, an "else" block is synthesized which kills the thread when encountered and none of the selections are ready at that point. Note that parallel branches are different thread points (unlike alternative branches), thus selection at a parallel branch must be handled individually.

If A is the set of selection blocks at a selection point, an "else" block e is added at the same point. This means $PC(e) \leftarrow PC(b)$ and $PCval(e) \leftarrow PCval(b)$ for $b \in A$ (any b in A would do since all have the same $PC(b)$ and $PCval(b)$ values). Because an elementary block may have multiple guarded updates, we define CMGuard(b) as follows:

$$\text{CMGuard}(b) \triangleq \begin{cases} \bigvee_{ri \in \mathcal{P}(I)} \text{MGuard}(b)(ri) & \text{if b is an output block} \\ \bigvee_{o \in O} \bigvee_{ri \in \mathcal{P}(I)} \text{MGuard}(b)(o)(ri) & \text{if b is an input block} \\ \text{MGuard}(b) & \text{otherwise.} \end{cases}$$

The "else" block e is constructed with the following parts:

$$\begin{aligned} \text{PCGuard}(e) &\leftarrow (PC(e) = PCval(e)), \\ \text{MGuard}(e) &\leftarrow \bigwedge_{b \in A} \neg\text{CMGuard}(b), \\ \text{PCUpdate}(e) &\leftarrow \{PC(e) \leftarrow 0\}, \\ \text{MUpdate}(e) &\leftarrow \{\}. \end{aligned}$$

Note that the above construction also works in the case where the selection block does not start an alternative branch ($|A| = 1$).

4.6.4 Prioritisation

There is an option in the translation for prioritising system transitions over external events (including external input and output). With the prioritisation option, the guard of each elementary block e that contains an external event BT node is modified as follows:

$$\text{MGuard}(e) \leftarrow \text{MGuard}(e) \wedge \bigwedge_{b \in A} \neg(\text{PCGuard}(b) \wedge \text{CMGuard}(b)),$$

where A is the set of elementary blocks that do not contain external event BT nodes.

4.7 Collecting Guarded Update Parts

After all the stages of the translation have been completed, each guarded update is formed from its parts.

- If (g, u) represents the guarded command for an elementary block b that is neither an input block nor an output block, then it is formed as follows:

$$\begin{aligned} g &\leftarrow \text{PCGuard}(b) \wedge \text{MGuard}(b), \\ u &\leftarrow \text{PCUpdate}(b) \cup \text{MUpdate}(b). \end{aligned}$$

- An input block b has its transition partitioned into into multiple guarded updates. If (g, u) represents the guarded update for the partition indexed by $(o)(ri)$, then it is formed as follows:

$$\begin{aligned} g &\leftarrow \text{PCGuard}(b) \wedge \text{MGuard}(b)(o)(ri), \\ u &\leftarrow \text{PCUpdate}(b)(o)(ri) \cup \text{MUpdate}(b)(o)(ri). \end{aligned}$$

- An output block b has its transition partitioned into into multiple guarded updates. If (g, u) represents the guarded update for the partition indexed by (ri), then it is formed as follows:

$$\begin{aligned} g &\leftarrow \text{PCGuard}(b) \wedge \text{MGuard}(b)(ri), \\ u &\leftarrow \text{PCUpdate}(b)(ri) \cup \text{MUpdate}(b)(ri). \end{aligned}$$

4.8 Summary and Possible Future Work

A method for translating a substantial subset of the BT notation into objects in the proposed framework has been described in this chapter. Operators in expressions are omitted from the translation because they are problematic:

- integer arithmetic operates on an infinite domain, and
- both [Beh07] and [CH11] are unclear on whether set operations can be performed on arbitrary (untyped) sets.

Perhaps if BT components and attributes are typed (either using a typing mechanism or using assertions) such that they have finite domains, then the translation can be extended to handle operators in expressions.

Chapter 5

Reachability

This chapter discusses the computation of reachability in model checking. The main contribution in this chapter is an algorithm for computing reachability that in some cases is significantly more efficient than a naive reachability computation. Although computing reachability and then using the reachability information in further analysis is not always the best strategy, there are cases where it is a good strategy. The algorithm can be used for such cases. Other uses of reachability information include discovering deadlock situations and finding violations of safety properties.

The process of model checking involves the concept of reachable states. In checking that a temporal logic formula is satisfied by a model, typically we must show that particular states in the model are never reached or particular states in the model are reachable. As an example, to show that $\mathbf{G}p$ is satisfied by the model, we must show that no state is reachable (from an initial state), from which a (suffix) path where $\neg p$ holds can start.

Reachability is also important in other types of analysis. As an example, for a system that is intended to be non-terminating, deadlocked states can be characterised as states that are reachable but from which there are no transitions. In the example BT of Figure 3.1, the states characterised by

$$(PC1 = 3) \wedge (PC2 = 1) \wedge (PC3 = 1) \wedge (Lock = set) \wedge (Light = amber) \qquad (5.1)$$

resulting from thread 2 executing N7 while thread 3 is at N10 (and thus killing thread 3), or thread 3 executing N12 while thread 2 is at N5 (and thus killing thread 2), are reachable but there are no transitions from them, hence they represent deadlocked states.

A strategically important question in model checking is how and when do we decide whether or not a state is reachable? A bad strategy can result in unnecessary complexity in the model checking. Three basic strategies for dealing with reachability that are used in model checking are:

1. Precomputing reachable states.

2. Reachable by construction, e.g., in on-the-fly model checking.

3. Inferring the existence of reachable states.

The following sections discuss each of the basic strategies within the framework. An algorithm for computing reachable states that mimics Kildall's algorithm for flow analysis [Kil73] is proposed in Section 5.1, complete with its proof of correctness.

5.1 Precomputing Reachable States

Precomputing reachability can be prohibitively expensive. However, precomputing reachable states often simplifies the operations required to perform an analysis. Going back to the deadlock problem of the example BT of Figure 3.1, if $Reachable(N8)$ characterises the set of reachable states at the entrance of N8, then performing the following logical operation:

$$Reachable(N8) \wedge \neg\left(\bigvee_{b\in B}\left(\bigvee_{(g,u)\in GU(b)} g\right)\right)$$

immediately yields (5.1). Thus the result of precomputing reachable states can be used to simplify deadlock analysis. Perhaps more importantly, in the context of counterexample path generation, precomputing counterexample states (reachable "fair states") allows an analysis to focus on states that are guaranteed to be reachable, which may be desirable since the existence of a finite prefix to a cycle involving counterexample states is guaranteed.

A state is reachable if it is an initial state or there is a transition from a reachable state to it. A fixpoint characterisation of the set of reachable states for a system modelled by a set of elementary blocks B is

$$\mu Z.S_0 \vee f_B(Z) \tag{5.2}$$

where S_0 is a proposition that characterises the set of initial states and f_B is the overall forward transfer function for the system modelled by B. The fixpoint computation for reachable states can be performed using a single transition relation to compute $f_B(Z)$. However, it may require a large OBDD to represent a large monolithic transition relation. Keeping the forward transfer functions distributed mitigates this problem. One can go one step further by computing reachable states at elementary blocks using Algorithm 5.1 which is inspired by Kildall's algorithm [Kil73] (sometimes called *chaotic iteration*).

There are two global variables used by the algorithm:

- W, representing a working set of elementary blocks, and

- *Reachable*, a collection of reachable sets of states, indexed by elementary block.

Algorithm 5.1 computes the sets of reachable states at the entrances of elementary blocks, where choose(W) chooses an element of W. For each elementary block b, the resulting reachable states at the entrance of b is stored in $Reachable(b)$. The algorithm assumes that for each elementary block b, the image transfer function f_b is available.

Algorithm 5.1. Compute Reachable States

Step 1 - Initialisation:
$W \leftarrow \emptyset$;
for each $b \in B$ do
  Reachable(b) $\leftarrow S_0 \wedge (\mathrm{PC}(b) = \mathrm{PCval}(b))$;
  if Reachable(b) $\neq$ false then $W \leftarrow W \cup \{b\}$;

Step 2 - Iteration:
while $W \neq \emptyset$ do
  $b \leftarrow$ choose(W);
  $W \leftarrow W \setminus \{b\}$;
  $D \leftarrow f_b$(Reachable(b));
  for each $c \in B$ do
    $C \leftarrow D \wedge (\mathrm{PC}(c) = \mathrm{PCval}(c))$;
    if $\neg(C \Rightarrow$ Reachable(c)) then
      Reachable(c) $\leftarrow$ Reachable(c) $\vee C$;
      $W \leftarrow W \cup \{c\}$;

Since we deal with finite systems, it is not too difficult to prove the following theorem:

Theorem 5.1. *Algorithm 5.1 terminates and produces the following result:*

$$\forall b \in B : Reachable(b) \Leftrightarrow (\mu Z.S_0 \vee f_B(Z)) \wedge (PC(b) = PCval(b)).$$

Proof. Step 1 in the algorithm iterates over a finite set, so it terminates. Each iteration in step 2 deletes a block from the working set W and sometimes adds blocks to W. It only adds a block c to W if the set characterised by $Reachable(c)$ can be enlarged and enlarges the set in the process. Since the set characterised by $Reachable(c)$ has a maximum size that is finite, the addition of a block can only be performed a finite number of times, thus step 2 always terminates.

Let P denote $\mu Z.S_0 \vee f_B(Z)$. We now show that

$$\forall b \in B : Reachable(b) \Rightarrow (P \wedge PC(b) = PCval(b)) \tag{5.3}$$

is an invariant of step 2. Initialisation of $Reachable(b)$ in step 1 guarantees that the invariant holds on entry to step 2. The only place in step 2 where $Reachable$ is modified produces

$$newReachable(c) \Leftrightarrow Reachable(c) \vee (f_b(Reachable(b)) \wedge PC(c) = PCval(c)).$$

Since P is a solution for Z in $Z \Leftrightarrow (S_0 \vee \bigvee_{b \in B} f_b(Z))$,

$$P \Leftrightarrow (S_0 \vee \bigvee_{b \in B} f_b(P))$$

thus $f_b(P) \Rightarrow P$. Recall from Section 3.3 that all image and pre-image transfer functions are monotonic, thus f_b is monotonic, and since $Reachable(b) \Rightarrow P$, we have

$$f_b(Reachable(b)) \Rightarrow f_b(P)$$

and get

$$(f_b(Reachable(b)) \wedge PC(c) = PCval(c)) \Rightarrow (P \wedge PC(c) = PCval(c))$$

and since $Reachable(c) \Rightarrow (P \wedge PC(c) = PCval(c))$, we get

$$newReachable(c) \Rightarrow (P \wedge PC(c) = PCval(c)))$$

thus proving the invariant (5.3). A second invariant is

$$\forall b, c \in B : Reachable(b) \wedge PC(c) = PCval(c) \Rightarrow Reachable(c). \tag{5.4}$$

The invariant (5.4) holds after step 1 since

$$S_0 \wedge (PC(b) = PCval(b)) \Leftrightarrow Reachable(b) \text{ and}$$
$$S_0 \wedge (PC(b) = PCval(b)) \wedge (PC(c) = PCval(c)) \Rightarrow Reachable(c)$$

thus $Reachable(b) \wedge PC(c) = PCval(c) \Rightarrow Reachable(c)$. Similarly, in step 2 we have

$$Reachable(b) \wedge PC(c) = PCval(c) \Rightarrow Reachable(c),$$
$$Reachable(b) \vee (D \wedge (PC(b) = PCval(b))) \Leftrightarrow newReachable(b) \text{ and}$$
$$D \wedge (PC(b) = PCval(b)) \wedge (PC(c) = PCval(c)) \Rightarrow newReachable(c)$$

thus $newReachable(b) \wedge PC(c) = PCval(c) \Rightarrow newReachable(c)$, meaning the invariant (5.4) is preserved by each iteration of step 2. Upon termination, we have for all $b, c \in B$:

$$f_b(Reachable(b)) \wedge PC(c) = PCval(c) \Rightarrow Reachable(c),$$

whether $Reachable(b)$ was updated (to non-*false*) last in step 1 or step 2, since in either case b was put in W, and the processing of b in step 2 guarantees the condition (whether

or not $C \Rightarrow \text{Reachable}(c)$). In addition, step 1 ensures that $\forall b \in B : S_0 \wedge PC(b) = PCval(b) \Rightarrow Reachable(b)$ (step 2 never makes the set characterised by $Reachable(b)$ smaller, for any b). Thus we have for all $b \in B$:

$$(S_0 \vee \bigvee_{d \in B} f_d(Reachable(d))) \wedge PC(b) = PCval(b) \Rightarrow Reachable(b).$$

Invariant (5.4) together with $f_d(S) \equiv f_d(S \wedge PC(d) = PCval(d))$ guarantees that $f_d(Reachable(d)) \equiv f_d(\bigvee_{b \in B} Reachable(b))$. Thus for all $b \in B$:

$$(S_0 \vee f_B(\bigvee_{d \in B} Reachable(d))) \wedge PC(b) = PCval(b) \Rightarrow Reachable(b).$$

Taking the disjunction over all $b \in B$ gives us

$$(S_0 \vee f_B(\bigvee_{d \in B} Reachable(d))) \wedge \bigvee_{b \in B} PC(b) = PCval(b) \Rightarrow \bigvee_{b \in B} Reachable(b),$$

and since $(\bigvee_{b \in B} PC(b) = PCval(b)) \equiv true$ (we assume that a state is at the entrance of at least one block) we get

$$(S_0 \vee f_B(\bigvee_{d \in B} Reachable(d))) \Rightarrow \bigvee_{d \in B} Reachable(d).$$

Since P is the least solution for Z in $Z \equiv S_0 \vee f_B(Z)$, we get

$$P \Rightarrow \bigvee_{d \in B} Reachable(d),$$

thus for all $b \in B$:

$$P \wedge PC(b) = PCval(b) \Rightarrow Reachable(b). \tag{5.5}$$

The invariant (5.3) gives us for all $b \in B$:

$$Reachable(b)) \Rightarrow P \wedge PC(b) = PCval(b). \tag{5.6}$$

Combining (5.5) with (5.6) proves the algorithm produces the desired result. □

5.2 Reachable by Construction

Some search strategies start at initial states and search forward from the initial states. With such a search strategy, at each step, the next set of states is chosen that are reachable from the current set of states in exactly one transition. At all times, only states that are reachable from the initial states are considered. Thus reachability is maintained by construction.

There are two examples of a forward search in this thesis, where the states are reachable from the initial states by construction:

- on-the-fly symbolic LTL model checking (described in Section 7.2.3), and
- the final phase of directed counterexample path generation (described in Chapter 8).

The naive on-the-fly symbolic LTL model checking algorithm described in Section 7.2.3 is an example of a blind (unconstrained) search. It can find a counterexample path very quickly if one exists, since it typically only needs to go through a tiny fraction of the search space before a counterexample path is found. However, it performs badly if there is no counterexample because it needs to go through the entire search space. Its performance may be improved by constraining the search, e.g., using partial order reduction [PWW96], where certain paths are excluded from the search because other paths, which in some sense represent them, are already included in the search.

The final phase of directed counterexample path generation, described in Chapter 8, is different in that the search is guaranteed to be successful without any need for backtracking. At each step, the existence of a non-empty next set of states is guaranteed. This is because the search space has been constrained by earlier phases of the directed counterexample path generation, and the constrained search space has been determined to contain solutions. Thus the forward search in the final phase is very efficient.

For a forward search using symbolic states, some post-processing is required. Recall from Section 3.3 that in our framework, forward searches for a path operate on symbolic states. Although all symbolic states in a forward search are reachable from the initial symbolic state, once a tentative symbolic path is found, the symbolic states need to be narrowed using (3.19) so that (3.18) is satisfied. For both of the above examples, narrowing is performed after a tentative path is found so that (3.18) is satisfied.

In summary, reachable by construction generally entails a forward search, which can be effective if a depth-first search can quickly find a solution. In the case of on-the-fly symbolic LTL model checking, if there is a counterexample, then a depth-first search can quickly find a counterexample path. In the case of the final phase of directed counterexample path generation, a depth-first search is guaranteed to succeed quickly without any need for backtracking.

5.3 Inferring the Existence of Reachable States

The third strategy for reachability is to ignore it during the main computation. An example of the application of this strategy is in a fixpoint approach to symbolic model checking.

In symbolic LTL model checking using the fixpoint approach, the set of "fair states" is computed via a fixpoint computation. Ultimately, the goal is to find all fair states that are reachable from certain initial states (the initial states must satisfy additional properties, as will be explained in Section 7.2.2). An intermediate goal is to find the set of initial fair states that satisfy the additional properties. There are two basic strategies with respect to reachability in achieving the intermediate goal:

1. perform the fixpoint computation within the set of reachable states (precomputed, e.g., using Algorithm 5.1) producing the set of reachable fair states, or

2. ignore reachability during the fixpoint computation and simply produce the set of fair states.

To achieve the intermediate goal, it does not matter whether we compute the set of reachable fair states or simply the set of fair states. Once the set of (reachable) fair states has been computed, it is intersected with the set of initial states that satisfy additional properties, producing the set of initial counterexample states. A non-empty set of initial counterexample states guarantees the existence of a counterexample path (an infinite sequence of reachable counterexample states), without actually computing the entire set of counterexample states. This is a consequence of the definition of the set of fair states and having the initial states satisfy additional properties (the additional properties commit the initial states to start counterexample paths). Of course, if a counterexample path is desired, then it would need to be searched, but the search can take advantage of the set of initial counterexample states and the set of fair states.

Being able to obtain the set of initial counterexample states, whether the set is empty or not, can be useful. If the set is empty then there is no counterexample and there is nothing else to do. If the set is not empty, then it can be used to reduce the search space for counterexample paths. However, there are alternatives for symbolic model checking that do not require the computation of the set of initial counterexample states: for example, the on-the-fly symbolic LTL model checking described in Section 7.2.3.

5.4 Summary

Three basic strategies for reachability in model checking have been described: precomputing reachable states, obtaining reachability by construction, and inferring the existence of reachable states. A particular strategy may not be appropriate for all model checking problems. Some of the experiments described in Chapter 9 examine the appropriateness of the strategies for different problems and look at the trade-offs between strategies.

Chapter 6

Foundation for LTL Model Checking

This chapter identifies general concepts that are used in a wide variety of LTL model checking approaches. The concepts are applied, in Chapter 7, to the state-machine-level framework described in Chapter 3 to form a framework for symbolic LTL model checking and counterexample generation that accommodates various LTL encoding schemes: the schemes for encoding LTL model-checking problems in a states-and-transitions setting. The concepts are applied to the classic LTL encoding scheme developed by Lichtenstein and Pnueli [LP85], and the Transition-based Generalised Büchi Automaton (TGBA) encoding scheme developed by Rozier and Vardi [RV11].

The main contributions of the conceptual framework include:

- The use of *path commitments* and *transition constraints* as building blocks for LTL *tableaux*. This allows transition-oriented LTL encoding schemes as well as the classic LTL encoding scheme to be used. Because path commitments and transition constraints are propositional formulas, optimisations of LTL tableaux can be performed symbolically.

- Key theorems are stated using general concepts that are independent of the encoding scheme. To show that an LTL encoding scheme is sound, the key theorems must be proved within the encoding scheme.

- General proof plans for the key theorems are developed. The applications of the proof plans for the classic LTL encoding scheme and the TGBA encoding scheme are included in Appendix A.

To illustrate the concepts covered in this chapter, suppose the following LTL formula is to be checked against the model for the BT example in Figure 3.1:

$$\mathbf{GF}(Light = green) \wedge \mathbf{GF}(Light = red). \tag{6.1}$$

An LTL formula *spec* being model-checked is implicitly quantified over all paths in the model, and thus is taken to be the CTL* formula $\mathbf{A}spec$. The standard way of model checking $\mathbf{A}spec$ is to perform "proof by contradiction" by checking to see if $\mathbf{E}\neg spec$ holds in the model, i.e., $\neg spec$ is satisfied by a path in the model. In the rest of this thesis, φ is used to denote $\neg spec$, where *spec* is the LTL formula being model-checked. Thus, for our example *spec* of (6.1), we have $\varphi \equiv \neg spec$:

$$\varphi \equiv \mathbf{FG}(\neg(Light = green)) \vee \mathbf{FG}(\neg(Light = red)). \tag{6.2}$$

It is generally assumed that the model M being checked has a set of initial states $S_0 \subseteq S$. To show the satisfiability of φ in M, we must show that there exists a path π in M, with $\pi_0 \in S_0$, that satisfies φ, i.e., $M, \pi \models \varphi$. However, model checking usually does not operate directly on paths since each path is an infinite sequence of states and, in general, there are an infinite number of paths in the model. Instead, model checking usually operates on states and transitions.

To deal with path formulas in a states-and-transitions setting, all well-known approaches for checking the satisfiability of φ in a model $M = (S, R, L)$ essentially impose a *tableau* for φ on the model M, producing an augmented model M'. The tableau is a decision graph that determines the cases that need to be examined to check the satisfiability of φ. In an augmented model M', a state in M is augmented such that it is constrained to only start certain classes of paths. This is accomplished by encoding path commitments in augmented states and constraining transitions among augmented states based on the path commitments. As a result, one can keep track of classes of paths without explicitly constructing the paths, enabling the checking of paths that satisfy φ in a states-and-transitions setting.

Traditionally, M' has been described as the product of the Kripke structure M and the Kripke structure for the tableau. M' may also be viewed as a refinement of M where some behaviours (paths) in M that do not satisfy φ are thrown away, but behaviours in M that satisfy φ are preserved in M'. The idea is the additional structures in M' make it easier to identify paths that satisfy φ.

Section 6.1 describes the construction of M' from M. Paths that satisfy φ in M must necessarily map to the class of paths that are "committed" to φ in M'. However, if φ involves *eventual conditions* (subformulas of φ that need to eventually hold in suffix paths), then checking the class of paths is not sufficient. Section 6.2 describes how eventual conditions can be taken into account. To demonstrate the generality of the concepts described in this chapter, they are applied to two LTL encoding schemes: the classic LTL encoding scheme [LP85], and the TGBA encoding scheme [RV11].

6.1 The Augmented Model

All well-known approaches for checking the satisfiability of φ in a model M augment states and transitions in M with *path commitments* and *transition constraints*, although under different guises. An augmented model $M' = (S', R', L')$ is constructed as a result of augmenting states and transitions in M. The resulting M' depends on the LTL encoding scheme used as well as φ, but the concepts described in this section apply to all encoding schemes. Let us now delve into the concepts that are used in constructing M'.

Some encoding schemes rely on φ being in negation normal form (NNF). One advantage of having φ in NNF is that the satisfiability of φ does not rely on the satisfiability of a negation of a temporal operation as a subproblem.

Definition 6.1 (Negation normal form)**.** A path formula is in *negation normal form* (NNF) if negations apply only to atomic formulas.

The TGBA encoding scheme requires that φ be in NNF. An LTL formula can be normalised to NNF by applying the following rules:

$$\begin{aligned} \neg\neg p &\equiv p, \\ \neg(p \wedge q) &\equiv \neg p \vee \neg q, \\ \neg(p \vee q) &\equiv \neg p \wedge \neg q, \\ \neg\mathbf{X}p &\equiv \mathbf{X}\neg p, \\ \neg\mathbf{G}p &\equiv \mathbf{F}\neg p, \\ \neg\mathbf{F}p &\equiv \mathbf{G}\neg p, \\ \neg(p\,\mathbf{U}\,q) &\equiv \neg p\,\mathbf{R}\neg q, \text{ and} \\ \neg(p\,\mathbf{R}\,q) &\equiv \neg p\,\mathbf{U}\neg q. \end{aligned} \tag{6.3}$$

The φ in (6.2) is already in NNF.

The classic LTL encoding scheme does not require φ to be in NNF, but considers $\{\mathbf{X}, \mathbf{U}\}$ to be the set of primitive temporal operators, and requires the temporal operations in φ to be primitive. An LTL formula can be normalised to a form in which the temporal operators are restricted to be in $\{\mathbf{X}, \mathbf{U}\}$ by applying the following rules:

$$\begin{aligned} \mathbf{G}p &\equiv \neg(\mathit{true}\,\mathbf{U}\,\neg p), \\ \mathbf{F}p &\equiv \mathit{true}\,\mathbf{U}\,p, \text{ and} \\ p\,\mathbf{R}\,q &\equiv \neg(\neg p\,\mathbf{U}\,\neg q). \end{aligned} \tag{6.4}$$

Some LTL formulas cannot be normalised to be both in NNF and have the temporal operators restricted to be in $\{\mathbf{X}, \mathbf{U}\}$. For the classic LTL encoding scheme, the φ in (6.2) can be normalised to

$$\varphi \equiv (\mathit{true}\,\mathbf{U}\,\neg(\mathit{true}\,\mathbf{U}\,(\mathit{Light} = \mathit{green}))) \vee (\mathit{true}\,\mathbf{U}\,\neg(\mathit{true}\,\mathbf{U}\,(\mathit{Light} = \mathit{red}))).$$

The concept of path commitment is implicitly used in all LTL encoding schemes known to the author. Informally, if a commitment to satisfying a path formula p is "asserted" by an augmented state s', then any finite prefix path in M' that starts at s' is consistent with an infinite path that satisfies p (although the infinite path might not be in M'). If p is also a state formula (in LTL, a state formula cannot have any temporal operation), e.g., p is $(Light = red)$, then the commitment to satisfy p is asserted by an augmented state s' if and only if $M', s' \models p$. Such a path commitment is called a trivial path commitment since the commitment is trivially fulfilled by s'. In contrast, if p involves temporal operations, then the fulfillment of the path commitment may involve other augmented states.

Trivial path commitments are already "encoded" in unaugmented states in M. To enable non-trivial path commitments to be encoded in augmented states, some non-trivial path commitments need to be considered as atomic propositions in M'. This can be accomplished by representing such non-trivial path commitments as Boolean state variables in M'.

Definition 6.2 (Elementary formula)**.** A path formula p involved in checking the satisfiability of φ is called an *elementary formula* if and only if p is not a state formula and the commitment to satisfy p is represented as a Boolean state variable, denoted V_p, in M'. The set of elementary formulas, constructed for the purpose of checking the satisfiability of φ, is denoted $el(\varphi)$.

Note that the definition of elementary formulas here is different from [CGH94] where they include atomic propositions in $el(\varphi)$. Instead, it follows the definition in [RV11]. Because path commitments for atomic propositions in M are trivial path commitments, atomic propositions are not included in $el(\varphi)$.

Let us use $AP(\varphi)$ to denote the set of atomic propositions in φ; $AP(\varphi) \subseteq AP$. Exactly what are in $el(\varphi)$ depends on the LTL encoding scheme but the construction of $el(\varphi)$ must obey the following rules:

$$\begin{aligned} el(p) &= \{\}, \text{ if } p \in AP(\varphi), \\ el(\neg p) &= el(p), \\ el(p \wedge q) &= el(p) \cup el(q), \text{ and} \\ el(p \vee q) &= el(p) \cup el(q). \end{aligned}$$

For the classic LTL encoding scheme, two additional rules complete the definition of $el(\varphi)$:

$$\begin{aligned} el(\mathbf{X}p) &= \{\mathbf{X}p\} \cup el(p), \text{ and} \\ el(p\,\mathbf{U}\,q) &= \{\mathbf{X}(p\,\mathbf{U}\,q)\} \cup el(p) \cup el(q). \end{aligned}$$

The definition of $el(\varphi)$ for the TGBA encoding scheme includes the following additional rules:

$$\begin{aligned}
el(\mathbf{X}p) &= \{\}, \text{ if } p \text{ has no temporal operations,} \\
el(\mathbf{X}p) &= \{p\} \cup el(p), \text{ if } p \text{ has temporal operations,} \\
el(p\,\mathbf{U}\,q) &= \{p\,\mathbf{U}\,q\} \cup el(p) \cup el(q), \\
el(p\,\mathbf{R}\,q) &= \{p\,\mathbf{R}\,q\} \cup el(p) \cup el(q), \\
el(\mathbf{GF}p) &= \{\mathbf{GF}p\} \cup el(p), \\
el(\mathbf{G}p) &= \{\mathbf{G}p\} \cup el(p), \text{ if } p \neq \mathbf{F}q \text{ for any } q, \text{ and} \\
el(\mathbf{F}p) &= \{\mathbf{F}p\} \cup el(p).
\end{aligned}$$

In addition, for the TGBA encoding scheme, if there is a path formula p of the form $\mathbf{X}q$ for some q, and there is an occurrence of p in φ that is not inside another temporal operation, then φ is added to $el(\varphi)$. This ensures that the path commitment for φ can be encoded in an augmented state. There can be variations for the rules for $el(\mathbf{X}p)$, but the rules must enable the commitment to satisfy p to be encoded in an augmented state. For example, the second rule for $el(\mathbf{X}p)$ can be replaced by the following two rules:

$$\begin{aligned}
el(\mathbf{X}p) &= el(p), \text{ if there is no occurrence of } \mathbf{X}q \text{ in } p \text{ not inside another} \\
&\qquad \text{temporal operation, for any } q, \text{ and} \\
el(\mathbf{X}p) &= \{p\} \cup el(p), \text{ if there is an occurrence of } \mathbf{X}q \text{ in } p \text{ not inside another} \\
&\qquad \text{temporal operation, for some } q.
\end{aligned}$$

Only certain formulas may be involved in checking the satisfiability of φ.

Concept 6.1 (Closure)**.** In checking the satisfiability of φ, the *closure* of φ, denoted $cl(\varphi)$ is a set of formulas whose construction is dependent on the encoding scheme. The satisfiability of φ can depend on the satisfiability of p as a subproblem only if $p \in cl(\varphi)$. It is required in the framework that $\varphi \in cl(\varphi)$.

For the classic LTL encoding scheme, $cl(\varphi) \triangleq el(\varphi) \cup sub(\varphi)$ where $sub(\varphi)$ denotes the set of subformulas of φ. For the TGBA encoding scheme, $cl(\varphi)$ is inductively defined as follows:

- If $p \in el(\varphi) \cup AP(\varphi)$ then $p \in cl(\varphi)$.
- If $p \in cl(\varphi)$ and $(\neg p) \in sub(\varphi)$ then $(\neg p) \in cl(\varphi)$.
- If $p \in cl(\varphi)$, $q \in cl(\varphi)$ and $(p \wedge q) \in sub(\varphi)$ then $(p \wedge q) \in cl(\varphi)$.

- If $p \in cl(\varphi)$, $q \in cl(\varphi)$ and $(p \vee q) \in sub(\varphi)$ then $(p \vee q) \in cl(\varphi)$.

It is not difficult to show that for both the classic LTL encoding scheme and the TGBA encoding scheme, $\varphi \in cl(\varphi)$.

An LTL encoding scheme may add Boolean state variables called promise variables to the augmented model M'. Promise variables are used in checking eventual conditions, to be described in Section 6.2. For the TGBA encoding scheme, for each $p \in el(\varphi)$ of the form $q \,\mathbf{U}\, r$, $\mathbf{GF}q$ or $\mathbf{F}q$, a promise variable denoted P_p is added to M'. Let us denote the set of promise variables added $\boldsymbol{P_\varphi}$, i.e., for the TGBA encoding scheme,

$$\boldsymbol{P_\varphi} = \{P_p \mid p \in el(\varphi) \text{ and } p \text{ is of the form } q \,\mathbf{U}\, r, \mathbf{GF}q \text{ or } \mathbf{F}q\}.$$

For the classic LTL encoding scheme, $\boldsymbol{P_\varphi} = \{\}$. Let us also define $\boldsymbol{V_\varphi} = \{V_p \mid p \in el(\varphi)\}$ and $vars_\varphi = \boldsymbol{V_\varphi} \cup \boldsymbol{P_\varphi}$. A state $s \in S$ is refined into 2^n augmented states in S' where $n = |vars_\varphi|$. Each augmented state $s' \in S'$ is an assignment

$$\{v_1 \leftarrow d_1, ..., v_m \leftarrow d_m, v_{m+1} \leftarrow b_1, ..., v_{m+n} \leftarrow b_n\}$$

where $v_1, ..., v_m$ are the state variables of M, $v_{m+1}, ..., v_{m+n}$ are the Boolean state variables in $vars_\varphi$, for $1 \leq i \leq n : d_i \in type(v_i)$, and for $1 \leq i \leq m : b_i \in \{true, false\}$. S' is then defined as follows:

$$S' \triangleq type(v_1) \times type(v_2) \times ... \times type(v_m) \times \underbrace{bool \times ... \times bool}_{n \text{ times}}. \tag{6.5}$$

The labelling L' is defined such that for each $s' \in S'$, we have:

$$L'(s') = L(s) \cup \{v \mid v \in vars_\varphi \wedge (M', s' \models v)\}. \tag{6.6}$$

For each $s' \in S'$, the projection of s' denoted $proj(s')$ is defined to be the $s \in S$ such that $L(s) = L'(s') \setminus vars_\varphi$ (recall from Section 2.1 that a state is completely determined from its label). The projection of a path π' in M' is obtained by projecting each augmented state in π':

$$proj(\pi') = \langle proj(\pi'_0), proj(\pi'_1), proj(\pi'_2), ...\rangle.$$

Let us define $AP_T \triangleq AP(\varphi) \cup vars_\varphi$. We now have enough concepts to describe path commitments and transition constraints.

Concept 6.2 (Path commitment)**.** For each $p \in cl(\varphi)$, the *path commitment* to satisfy p, denoted $S_\varphi(p)$, is a state formula in the augmented model M' whose atomic propositions are in AP_T. The exact definition of $S_\varphi(p)$ is specific to the LTL encoding scheme used,

but must include the following rules:

$$\begin{aligned} S_\varphi(p) &\equiv p, \text{ if } p \in AP(\varphi), \\ S_\varphi(p) &\equiv V_p, \text{ if } p \in el(\varphi), \\ S_\varphi(\neg p) &\equiv \neg S_\varphi(p), \\ S_\varphi(p \wedge q) &\equiv S_\varphi(p) \wedge S_\varphi(q), \text{ and} \\ S_\varphi(p \vee q) &\equiv S_\varphi(p) \vee S_\varphi(q). \end{aligned}$$

For the classic LTL encoding scheme, the following additional rule completes the definition of S_φ:

$$S_\varphi(p \,\mathbf{U}\, q) \equiv S_\varphi(q) \vee (S_\varphi(p) \wedge S_\varphi(\mathbf{X}(p \,\mathbf{U}\, q))).$$

For the TGBA encoding scheme, no additional rule for S_φ is required.

A path commitment in an augmented state restricts the paths that can start from the augmented state. Part of the restriction is enforced using *transition constraints*. For $p \in cl(\varphi)$, if the path commitment $S_\varphi(p)$ holds in an augmented state s', then it gives rise to a transition constraint $T_\varphi(p)$ for any transition from s'.

Concept 6.3 (Transition constraint)**.** For $p \in cl(\varphi) \cup sub(\varphi)$, a *transition constraint* $T_\varphi(p)$ is a propositional formula whose atoms are in $AP_T \cup \{next(q) \mid q \in prop(\varphi)\}$, where $prop(\varphi)$ is the set of propositions whose atoms are in AP_T, and an atom $next(q)$ means the proposition q holds in the next state (the state that is the target of the transition). The exact definition of $T_\varphi(p)$ is specific to the encoding scheme, but must include the following rules:

$$\begin{aligned} T_\varphi(p) &\equiv p, \text{ if } p \in AP(\varphi), \\ T_\varphi(\neg p) &\equiv \neg T_\varphi(p), \\ T_\varphi(p \wedge q) &\equiv T_\varphi(p) \wedge T_\varphi(q), \\ T_\varphi(p \vee q) &\equiv T_\varphi(p) \vee T_\varphi(q), \text{ and} \\ T_\varphi(\mathbf{X}p) &\equiv next(S_\varphi(p)). \end{aligned}$$

For the classic LTL encoding scheme, the following additional rule completes the definition of T_φ:

$$T_\varphi(p \,\mathbf{U}\, q) \equiv T_\varphi(q) \vee (T_\varphi(p) \wedge T_\varphi(\mathbf{X}(p \,\mathbf{U}\, q))).$$

For the TGBA encoding scheme, the following additional rules complete the definition of T_φ (note that the domain of T_φ is $cl(\varphi) \cup sub(\varphi)$ which, for the TGBA encoding scheme,

is $sub(\varphi)$):

$$\begin{aligned}
T_\varphi(p\,\mathbf{U}\,q) &\equiv T_\varphi(q) \vee (T_\varphi(p) \wedge P_{p\,\mathbf{U}\,q} \wedge T_\varphi(\mathbf{X}(p\,\mathbf{U}\,q))),\\
T_\varphi(p\,\mathbf{R}\,q) &\equiv T_\varphi(q) \wedge (T_\varphi(p) \vee T_\varphi(\mathbf{X}(p\,\mathbf{R}\,q))),\\
T_\varphi(\mathbf{G}p) &\equiv T_\varphi(p) \wedge T_\varphi(\mathbf{X}(\mathbf{G}p)),\ \text{if } p \neq \mathbf{F}q \text{ for any } q,\\
T_\varphi(\mathbf{F}p) &\equiv T_\varphi(p) \vee (P_{\mathbf{F}p} \wedge T_\varphi(\mathbf{X}(\mathbf{F}p))),\ \text{and}\\
T_\varphi(\mathbf{GF}p) &\equiv T_\varphi(\mathbf{X}(\mathbf{GF}p)) \wedge (T_\varphi(p) \vee P_{\mathbf{GF}p})
\end{aligned}$$

Informally, a transition constraint $T_\varphi(p)$ ensures that an augmented state s' with a commitment $S_\varphi(p)$ can transition to an augmented state t' only if the sequence $\langle s', t'\rangle$ can be the prefix of a path that satisfies p, thus ensuring that the transition is consistent with the path commitment. As an example, with the TGBA encoding scheme, suppose the path commitment $S_\varphi(\mathbf{G}p)$ holds in an augmented state s', meaning s' is committed to start only paths that satisfy $\mathbf{G}p$. Suppose p is not of the form $\mathbf{F}q$ for any q. The semantic definition of $\mathbf{G}p$ from Section 2.1 is:

$$M, \pi \models \mathbf{G}p \Leftrightarrow \forall i \geq 0 : (M, \pi^i \models p), \tag{6.7}$$

which is equivalent to the definition

$$M, \pi \models \mathbf{G}p \Leftrightarrow (M, \pi \models p) \wedge (M, \pi^1 \models \mathbf{G}p) \tag{6.8}$$

which can be read as s' can start only paths that satisfy p and t' can start only paths that satisfy $\mathbf{G}p$. The transition constraint $T_\varphi(\mathbf{G}p)$ is

$$T_\varphi(\mathbf{G}p) \equiv T_\varphi(p) \wedge T_\varphi(\mathbf{X}(\mathbf{G}p))$$

where $T_\varphi(p)$ corresponds to $M, \pi \models p$ in (6.8) (s' can start only paths that satisfy p) and $T_\varphi(\mathbf{X}(\mathbf{G}p))$ corresponds to $M, \pi^1 \models \mathbf{G}p$ in (6.8) (t' can start only paths that satisfy $\mathbf{G}p$). Because the transition constraint corresponds exactly with the definition of the satisfiability of $\mathbf{G}p$, the transition constraint is sufficient to ensure that the commitment $S_\varphi(\mathbf{G}p)$ is fulfilled by the suffix paths (provided subsidiary commitments are also fulfilled). As we shall see in Section 6.2, transition constraints are not sufficient for temporal operations involving eventual conditions.

Definition 6.3 (Satisfaction of transition constraint)**.** For augmented states s'_1, s'_2, and a transition constraint T, the satisfaction of T by the transition (s'_1, s'_2), denoted $(s'_1, s'_2) \models T$, is evaluated using the following rules:

$$\begin{aligned}
(s'_1, s'_2) \models p &\equiv M', s'_1 \models p,\ \text{if } p \in AP_T,\\
(s'_1, s'_2) \models next(p) &\equiv M', s'_2 \models p,\\
(s'_1, s'_2) \models \neg p &\equiv (s'_1, s'_2) \not\models p,\\
(s'_1, s'_2) \models p \vee q &\equiv ((s'_1, s'_2) \models p) \vee ((s'_1, s'_2) \models q),\ \text{and}\\
(s'_1, s'_2) \models p \wedge q &\equiv ((s'_1, s'_2) \models p) \wedge ((s'_1, s'_2) \models q).
\end{aligned}$$

Augmenting states in M with Boolean state in $vars_\varphi$ and adding the necessary transition constraints, produces an augmented model $M' = (S', R', L')$, with S' defined by (6.5) and L' defined by (6.6).

Definition 6.4 (Transition Relation for Augmented Model)**.** The transition relation R' is defined as follows:

$$\begin{aligned}(s'_1, s'_2) \in R' \;\Leftrightarrow\; & (proj(s'_1), proj(s'_2)) \in R \wedge \\ & (\forall p \in el(\varphi) : V_p \in L'(s'_1) \Leftrightarrow (s'_1, s'_2) \models T_\varphi(p)).\end{aligned} \tag{6.9}$$

If φ is in NNF, the following alternative definition can be used:

$$\begin{aligned}(s'_1, s'_2) \in R' \;\Leftrightarrow\; & (proj(s'_1), proj(s'_2)) \in R \wedge \\ & (\forall p \in el(\varphi) : V_p \in L'(s'_1) \Rightarrow (s'_1, s'_2) \models T_\varphi(p)).\end{aligned} \tag{6.10}$$

If (6.9) is used as the definition of R', then the encoding scheme is said to be using a a *strict encoding*. If φ is in NNF and (6.10) is used as the definition of R', then the encoding scheme is said to be using a *loose encoding* (called *sloppy encoding* in [RV11]). The use of a loose encoding when φ is in NNF causes $M', \pi' \not\models p$ to be treated as a "do not care" for $p \in el(\varphi)$, which may lead to a more efficient encoding than a strict encoding.

The elementary formulas of φ and transition constraints associated with the elementary formulas form a tableau for checking the satisfiability of φ. The tableau may be thought of as a mechanism to ensure that all relevant cases (of behaviours) are considered. The cases are determined by $el(\varphi)$, and there are $2^{el(\varphi)}$ elements in $\mathcal{P}(el(\varphi))$ representing the combinations of elementary formulas that make up the cases.

Definition 6.5 (Symbolic Tableau)**.** If R' is defined using (6.9), then the symbolic tableau for φ is:

$$\bigvee_{c \in \mathcal{P}(el(\varphi))} \left(\bigwedge_{p \in el(\varphi)} \text{if } p \in c \text{ then } V_p \wedge T_\varphi(p) \text{ else } \neg V_p \wedge \neg T_\varphi(p) \right) \tag{6.11}$$

If R' is defined using (6.10) then the symbolic tableau for φ is:

$$\bigvee_{c \in \mathcal{P}(el(\varphi))} \left(\bigwedge_{p \in el(\varphi)} \text{if } p \in c \text{ then } V_p \wedge T_\varphi(p) \text{ else } \neg V_p \right) \tag{6.12}$$

A symbolic a tableau is a transition constraint. In Chapter 7, a symbolic tableau is further processed into a collection of simpler transition constraints.

The purpose of augmenting states is to keep track of modalities in the search for a path that satisfies φ. Augmenting states in a path introduces no "new truths" for the path. This is made formal in Theorem 6.1.

Theorem 6.1 (Conservative Extension)**.**

$$\forall i \geq 0, p \in cl(\varphi) \cup sub(\varphi) : (M', \pi'^i \models p) \Leftrightarrow (M, proj(\pi'^i) \models p).$$

Proof. The proof is by induction on the structure of p. For the base case, we have $p \in AP(\varphi)$ in which case $M', \pi'^i \models p$ and $M, proj(\pi'^i) \models p$ are identically determined by $L(proj(\pi'_i))$. Each of the inductive cases follows from the semantics of temporal logic described in Section 2.1 (the number of cases depends on the encoding scheme, but the proof for each case is independent of the encoding scheme). □

The purpose of constructing the augmented model M' is to help keep track of paths and rule out irrelevant paths. However, we do not want paths that satisfy φ to be ruled out. The following theorem ensures that paths that satisfy φ are not ruled out.

Theorem 6.2 (Preservation of Satisfying Paths)**.**

$$(M, \pi \models \varphi) \;\; \Rightarrow \;\; \exists \pi' : proj(\pi') = \pi \;\wedge (M', \pi'_0 \models S_\varphi(\varphi)).$$

Proof. The proof of Theorem 6.2 is dependent on the encoding scheme. The proofs for the classic LTL encoding scheme and the TGBA encoding scheme are provided in Appendix A. □

6.2 Eventual Conditions

Unfortunately, the converse of Theorem 6.2 (the $\Leftarrow$ direction) does not hold. The existence of π' as in Theorem 6.2, with $M', \pi'_0 \models S_\varphi(\varphi)$, does not guarantee that $M, \pi \models \varphi$. Eventual conditions cannot be guaranteed using only transition constraints. Figure 6.1

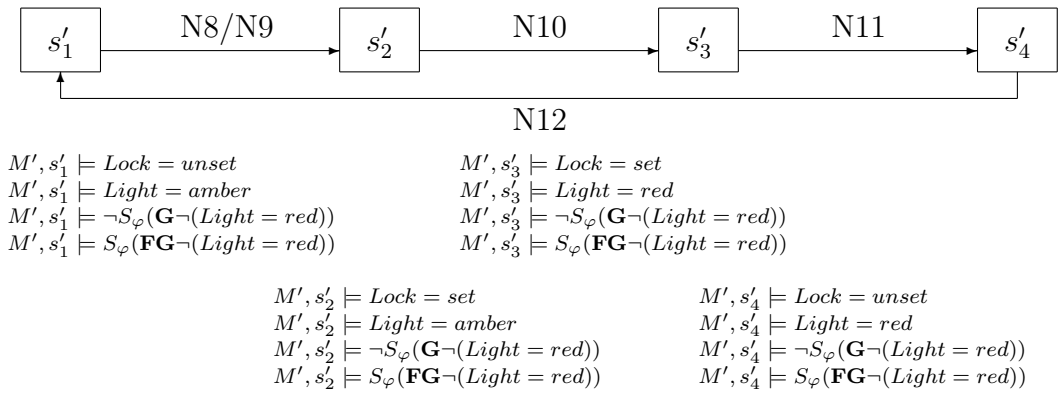

Figure 6.1: A cycle without eventual condition $\mathbf{G}\neg(Light = red)$ fulfilled.

illustrates a case where an eventual condition $\mathbf{G}\neg(Light = red)$ is never fulfilled although

the transitions in the cycle satisfy the transition constraints for $S_\varphi(\mathbf{FG}\neg(Light = red))$ (the example assumes the TGBA encoding scheme is used).

Let π' be the indefinite repetition of the cycle in Figure 6.1 starting at s'_1, i.e., $\pi' = \langle s'_1, s'_2, s'_3, s'_4, s'_1, ...\rangle$. Then $M', \pi'_0 \models S_\varphi(\mathbf{FG}\neg(Light = red))$ but because the eventual condition $\mathbf{G}\neg(Light = red)$ is never fulfilled in a suffix of π', i.e., $\forall i \geq 0 : M', \pi'_i \not\models S_\varphi(\mathbf{G}\neg(Light = red))$, we get $M', \pi' \not\models \mathbf{FG}\neg(Light = red)$, thus $M, proj(\pi') \not\models \mathbf{FG}\neg(Light = red)$ (from Theorem 6.1). Note that a cycle with the same transitions but with $\neg S_\varphi(\mathbf{G}\neg(Light = red))$ replaced by $S_\varphi(\mathbf{G}\neg(Light = red))$ is ruled out of M' because $M', s'_3 \models Light = red$.

Let us examine why transition constraints, although necessary, are insufficient to ensure that the commitment $S_\varphi(\mathbf{F}p)$ is fulfilled for an arbitrary path formula p. The semantic definition of $\mathbf{F}p$ is

$$M, \pi \models \mathbf{F}p \Leftrightarrow \exists i \geq 0 : (M, \pi^i \models p) \tag{6.13}$$

which is equivalent to the following definition:

$$M, \pi \models \mathbf{F}p \Leftrightarrow (M, \pi \models p) \vee (M, \pi^1 \models \mathbf{F}p) \wedge \exists i \geq 1 : (M, \pi^i \models p). \tag{6.14}$$

The transition constraint $T_\varphi(\mathbf{F}p)$ from a state with path commitment $S_\varphi(\mathbf{F}p)$ is

$$T_\varphi(\mathbf{F}p) \equiv T_\varphi(p) \vee (P_{\mathbf{F}p} \wedge T_\varphi(\mathbf{X}(\mathbf{F}p))).$$

$T_\varphi(p)$ corresponds to $M, \pi \models p$ in (6.14) and $T_\varphi(\mathbf{X}(\mathbf{F}p))$ corresponds to $M, \pi^1 \models \mathbf{F}p$, but $\exists i \geq 1 : (M, \pi^i \models p)$ in (6.14) is not covered by the transition constraint (a transition constraint can only "look ahead" one transition, thus can guarantee only that a prefix up to the next state is consistent with a satisfying path). The condition p in $\exists i \geq 1 : (M, \pi^i \models p)$ is called an *eventual condition*.

Concept 6.4 (Eventual condition)**.** The *eventual condition* for a path formula $p \in cl(\varphi)$, denoted $ec(p)$, is a path formula whose fulfillment is necessary for satisfying p in the sense

$$(M, \pi \models p) \Rightarrow \exists i \geq 0 : (M, \pi^i \models ec(p)),$$

and it is based on the form of p. The exact definition of ec is dependent on the encoding scheme, but must include the following rule:

$$ec(p\,\mathbf{U}\,q) \equiv q.$$

For the classic LTL encoding scheme, the following rule completes the definition of ec:

$$ec(p) \equiv true, \text{ if } p \text{ is not of the form } q\,\mathbf{U}\,r \text{ for any } q \text{ and } r.$$

For the TGBA encoding scheme, the following rules complete the definition of ec:

$$\begin{aligned} ec(\mathbf{F}p) &\equiv p, \\ ec(\mathbf{GF}p) &\equiv p, \text{ and} \\ ec(p) &\equiv true, \text{ if } p \text{ is not of the form } q\,\mathbf{U}\,r, \mathbf{F}q, \mathbf{GF}q, \text{ for any } q \text{ and } r. \end{aligned}$$

If φ is not in NNF and the encoding scheme is not the classic LTL encoding scheme, then we need to be able to handle the following:

$$(M, \pi \not\models p) \Rightarrow \exists i \geq 0 : (M, \pi^i \models ec(\neg p)),$$

which may require rules such as

$$ec(\neg\mathbf{G}p) \equiv \neg p \text{ and } ec(\neg(p\,\mathbf{R}\,q)) \equiv \neg q.$$

Although the equivalence

$$M, \pi \models \mathbf{F}p \Leftrightarrow (M, \pi \models p) \vee (M, \pi^1 \models \mathbf{F}p) \tag{6.15}$$

is a consequence of the definition (6.13), (6.15) as a co-inductive definition is not equivalent to (6.13). Had (6.15) been the definition, then transition constraints would have been sufficient, but $\pi' = \langle s'_1, s'_2, s'_3, s'_4, s'_1, ...\rangle$ would have incorrectly satisfied $\mathbf{FG}\neg(Light = red)$ (π' does not satisfy $\exists i \geq 0 : \pi'^i \models \mathbf{G}\neg(Light = red)$). Thus (6.15) is incorrect as a definition, and (6.14) must be used instead. Because transition constraints do not ensure that eventual conditions will be fulfilled, additional checking must be performed to ensure the fulfillment of eventual conditions.

Rather than directly tracking fulfillments of eventual conditions, LTL checking typically uses fairness constraints for checking eventual conditions. (Recall from Section 2.2.3, Lichtenstein and Pnueli showed that the check for eventual conditions can be performed in terms of fairness constraints.) The set of fairness constraints for the purpose of checking the satisfiability of φ is denoted C_φ. For the classic LTL encoding scheme, C_φ is defined as follows:

$$C_\varphi \triangleq \{\neg S_\varphi(p) \vee S_\varphi(ec(p)) \mid p \in cl(\varphi) \wedge (ec(p) \not\equiv true)\}. \tag{6.16}$$

For the TGBA encoding scheme, $S_\varphi(ec(p))$ is not necessarily defined for $p \in cl(\varphi)$. Instead, the TGBA encoding scheme uses promise variables to track fulfillment of eventual conditions. For the TGBA encoding scheme, C_φ is defined as follows:

$$C_\varphi \triangleq \{\neg v \mid v \in \boldsymbol{P_\varphi}\}. \tag{6.17}$$

The theorem that is the basis for all LTL model checking algorithms known to the author is as follows:

Theorem 6.3 (Main Theorem for LTL Checking)**.**

$$M, \pi \models \varphi \;\Leftrightarrow\; \exists \pi' : \quad proj(\pi') = \pi \wedge (M', \pi'_0 \models S_\varphi(\varphi)) \\ \wedge \;\; \forall c \in C_\varphi, i \geq 0 : \exists j \geq i : M', \pi'_j \models c.$$

Proof. The proof of Theorem 6.3 is dependent on the encoding scheme. The proofs for the classic LTL encoding scheme and the TGBA encoding scheme are provided in Appendix A. □

6.3 Proof Plans

6.3.1 Proof Plan for Theorem 6.2

Let us start with a proof plan for Theorem 6.2:

$$(M, \pi \models \varphi) \;\Rightarrow\; \exists \pi' : proj(\pi') = \pi \;\wedge (M', \pi'_0 \models S_\varphi(\varphi)).$$

Given that $M, \pi \models \varphi$, how do we construct a path π' in M' such that $proj(\pi') = \pi \;\wedge (M', \pi'_0 \models S_\varphi(\varphi))$? The safest way is to make π' a "strict mapping" of π in that

$$\forall i \geq 0, p \in el(\varphi) : (M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models V_p). \tag{6.18}$$

Additionally, for an encoding scheme that uses promise variables, the values of the promise variables at each augmented state in the path are determined as follows:

$$\forall i \geq 0, P_p \in P_\varphi : (M, \pi^i \models p \wedge \neg ec(p)) \Leftrightarrow (M', \pi'_i \models P_p). \tag{6.19}$$

Thus for an encoding scheme that does not use promise variables, each augmented state π'_i is constructed as follows:

$$L'(\pi'_i) = L(\pi_i) \cup \{V_p \mid p \in el(\varphi) \wedge (M, \pi^i \models p)\}, \tag{6.20}$$

and for an encoding scheme that uses promise variables, each augmented state π'_i is constructed as follows:

$$L'(\pi'_i) = L(\pi_i) \cup \{V_p \mid p \in el(\varphi) \wedge (M, \pi^i \models p)\} \cup \{P_p \mid P_p \in P_\varphi \wedge (M, \pi^i \models p \wedge \neg ec(p))\} \tag{6.21}$$

(recall from Section 2.1 that a state is completely determined by its label).

To show that π' constructed as above is in fact a path in M', we must show that

$$\forall i \geq 0 : (\pi'_i, \pi'_{i+1}) \in R'. \tag{6.22}$$

The proof can be made easier if Theorem 6.2 is strengthened to state the existence of a "strict" mapping of a path that satisfies φ.

For an encoding scheme that does not use promise variables, the strengthening produces Lemma 6.1.

Lemma 6.1.

$$\begin{aligned} & M, \pi \models \varphi \\ \Rightarrow \quad & \exists \pi' : proj(\pi') = \pi \wedge \forall i \geq 0, p \in cl(\varphi) : (M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models S_\varphi(p)). \end{aligned}$$

The proof of Lemma 6.1 is specific to the encoding scheme but can use the following plan:

1. show that π' as a sequence of states constructed according to (6.20) satisfies

$$\forall i \geq 0, p \in cl(\varphi) : (M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models S_\varphi(p)), \tag{6.23}$$

   and

2. given that π' satisfies (6.23) show that π' satisfies (6.22).

The application of the proof plan to the classic LTL encoding scheme, proving Lemma 6.1 in the scheme, is described in Section A.1.

For an encoding scheme that uses promise variables and requires φ to be in NNF, the strengthening produces Lemma 6.2.

Lemma 6.2.

$$\begin{aligned} & M, \pi \models \varphi \\ \Rightarrow \quad & \exists \pi' : proj(\pi') = \pi \wedge \forall i \geq 0, p \in cl(\varphi) : (M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models S_\varphi(p)) \\ & \quad \wedge \forall i \geq 0, p \in el(\varphi) \wedge (ec(p) \not\equiv true) : (M, \pi^i \models p \wedge \neg ec(p)) \Leftrightarrow (M', \pi'_i \models P_p). \end{aligned}$$

The proof of Lemma 6.2 is specific to the encoding scheme but can use the following plan:

1. show that π' as a sequence of states constructed according to (6.21) satisfies

$$\begin{aligned} & \forall i \geq 0, p \in cl(\varphi) : (M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models S_\varphi(p)) \\ \wedge \quad & \forall i \geq 0, p \in el(\varphi) \wedge (ec(p) \not\equiv true) : (M, \pi^i \models p \wedge \neg ec(p)) \Leftrightarrow (M', \pi'_i \models P_p), \end{aligned} \tag{6.24}$$

   and

2. given that π' satisfies (6.24) show that π' satisfies (6.22).

The application of the proof plan to the TGBA encoding scheme, proving Lemma 6.2 in the scheme, is described in Section A.2.

For an encoding scheme that does not use promise variables, Theorem 6.2 follows from Lemma 6.1 (since $\varphi \in cl(\varphi)$). For an encoding scheme that uses promise variables (and requires φ to be in NNF), Theorem 6.2 follows from Lemma 6.2 (since $\varphi \in cl(\varphi)$).

6.3.2 Proof Plan for Theorem 6.3

Next, let us develop a proof plan for Theorem 6.3:

$$\begin{aligned} M, \pi \models \varphi \quad \Leftrightarrow \quad \exists \pi' : \quad & proj(\pi') = \pi \wedge (M', \pi'_0 \models S_\varphi(\varphi)) \\ \wedge \quad & \forall c \in C_\varphi, i \geq 0 : \exists j \geq i : M', \pi'_j \models c. \end{aligned}$$

Let us start with the $\Rightarrow$ direction.

For an encoding scheme that does not use promise variables, using Lemma 6.1, what remains to prove the $\Rightarrow$ direction is to prove Lemma 6.3.

Lemma 6.3.

$$\begin{aligned} & (M, \pi \models \varphi) \wedge proj(\pi') = \pi \\ \wedge \quad & \forall i \geq 0, p \in cl(\varphi) : (M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models S_\varphi(p)) \\ \Rightarrow \quad & \forall c \in C_\varphi, i \geq 0 : \exists j \geq i : M', \pi'_j \models c, \end{aligned}$$

The proof of Lemma 6.3 is specific to the encoding scheme but can use the following plan:

1. expand C_φ according to its definition,
2. transform the quantification over elements of C_φ into quantification over formulas in the closure of φ that have non-trivial eventual conditions, and
3. use the last antecedent in the lemma, from which the conclusion of the lemma follows, to complete the proof.

The application of the proof plan to the classic LTL encoding scheme, proving Lemma 6.3 in the scheme, is described in Section A.1.

For an encoding scheme that uses promise variables and requires φ to be in NNF, using Lemma 6.2, what remains to prove the $\Rightarrow$ direction is to prove Lemma 6.4.

Lemma 6.4.

$$\begin{aligned} & (M, \pi \models \varphi) \wedge proj(\pi') = \pi \\ \wedge \quad & \forall i \geq 0, p \in cl(\varphi) : (M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models S_\varphi(p)) \\ \wedge \quad & \forall i \geq 0, p \in el(\varphi) \wedge (ec(p) \not\equiv true) : (M, \pi^i \models p \wedge \neg ec(p)) \Leftrightarrow (M', \pi'_i \models P_p) \\ \Rightarrow \quad & \forall c \in C_\varphi, i \geq 0 : \exists j \geq i : M', \pi'_j \models c. \end{aligned}$$

The proof of Lemma 6.4 is specific to the encoding scheme but can use the following plan:

1. expand C_φ according to its definition,

2. transform the quantification over elements of C_φ into quantification over elementary formulas of φ that have non-trivial eventual conditions, and

3. use the last antecedent in the lemma, from which the conclusion of the lemma follows, to complete the proof.

The application of the proof plan to the TGBA encoding scheme, proving Lemma 6.4 in the scheme, is described in Section A.2.

For the $\Leftarrow$ direction, we must prove

$$\begin{aligned} & proj(\pi') = \pi \wedge (M', \pi'_0 \models S_\varphi(\varphi)) \wedge \forall c \in C_\varphi, i \geq 0 : \exists j \geq i : M', \pi'_j \models c \\ \Rightarrow \quad & (M, \pi \models \varphi) \end{aligned} \tag{6.25}$$

which can be done with the help of Lemma 6.5.

Lemma 6.5.

$$\begin{aligned} & \forall c \in C_\varphi, i \geq 0 : \exists j \geq i : M', \pi'_j \models c \\ \Rightarrow \quad & \forall i \geq 0, p \in cl(\varphi) : (M', \pi'_i \models S_\varphi(p)) \Rightarrow (M', \pi'^i \models p). \end{aligned}$$

The proof of Lemma 6.5 is specific to the encoding scheme but can use the following plan:

1. expand C_φ according to its definition,

2. transform the quantification over elements of C_φ into quantification over formulas in the closure of φ or elementary formulas of φ, depending on whether the encoding scheme uses promise variables, and

3. complete the proof by induction on the structure of the variable quantified over formulas in the closure of φ.

The application of the proof plan to the classic LTL encoding scheme, proving Lemma 6.5 in the scheme, is described in Section A.1. The application of the proof plan to the TGBA encoding scheme, proving Lemma 6.5 in the scheme, is described in Section A.2.

(6.25) follows from Lemma 6.5 (with the second i instantiated to 0 and p instantiated to φ) and Theorem 6.1.

6.3.3 Summary

In summary, for an encoding scheme that does not use promise variables, the proof plans developed have reduced the proof of Theorem 6.2 to a proof of Lemma 6.1, and the proof of Theorem 6.3 to proofs of Lemmas 6.3 and 6.5. For an encoding scheme that uses promise variables (and requires φ to be in NNF), the proof plans developed have reduced the proof of Theorem 6.2 to a proof of Lemma 6.2 and the proof of Theorem 6.3 to proofs of Lemmas 6.4 and Lemma 6.5.

6.4 Summary and Discussion

Essential concepts in LTL model checking have been presented in a fashion that is independent of the LTL encoding scheme and the model checking approach and strategy. This results in a framework for LTL model checking that supports multiple approaches and strategies. A tool based on the framework can be developed that accommodates a multitude of approaches and strategies. This is important because no single approach or strategy has been found to be best for all model checking problems. It also allows the possibility of running different approaches and strategies in parallel and using the result of whichever succeeds first.

All LTL model checking approaches known to the author are based on a version of Theorem 6.3, thus our framework covers all of those approaches. The soundness of an LTL encoding scheme in model checking can be assured by proving Theorem 6.3 within the encoding scheme. Proof plans for proving theorems that represent the soundness of an LTL encoding scheme have been developed. The applications of the proof plans to show the soundness of the classic LTL encoding scheme and the TGBA encoding scheme are included in Appendix A.

Chapter 7

LTL Model Checking within the Framework

Chapter 6 identified the essential concepts used in LTL model checking that are independent of the encoding scheme and the model checking approach. The concepts identified, together with the symbolic framework for state machines described in Chapter 3, serve as a foundation for symbolic LTL model checking and counterexample generation. This chapter describes how the concepts are applied and how model checking might be performed within the resulting framework.

The novel contributions of this chapter include:

- The use of *global constraints* to optimise analysis. For some forms of LTL formulas, the checking of the formula can be optimised by treating part of the formula as a global constraint and checking a smaller formula under the global constraint.

- The idea of normalising transition constraints for an LTL tableau into disjunctive normal form, where each disjunct is a simple transition constraint and represents a case in the tableau. A simple transition constraint may be viewed as consisting of a *guard* and a *post condition*, and is used in the framework to construct transfer functions in the augmented model.

- An algorithm for on-the-fly symbolic LTL model checking. The algorithm is a novel adaptation of the double depth-first search (double-DFS) algorithm of [CVWY92] in a symbolic setting.

Section 7.1 shows how the augmented model M' described in Section 6.1 can be constructed within the framework described in Chapter 3, and defines the concept of symbolic counterexample paths. The result provides a framework for LTL model checking and counterexample generation. Section 7.2 shows how different techniques and strategies

for LTL model checking and counterexample generation can be performed within the resulting framework.

7.1 The LTL Model Checking Framework

7.1.1 Normalisation of φ

Recall from Section 6.1 that an LTL encoding scheme typically requires φ to be in some normal form. Normalisation involves the application of transformation rules. The classic LTL encoding scheme requires that φ be normalised to contain temporal operators only from the set $\{\mathbf{X}, \mathbf{U}\}$. The normalisation for the classic LTL encoding scheme can be performed using the transformation rules in (6.4). The TGBA encoding scheme requires that φ be normalised into NNF. Normalisation into NNF can be performed using the transformation rules in (6.3).

Although not necessary, additional transformations may be performed to obtain an efficient encoding. As an example, the following transformation rules may be applied to obtain fewer temporal operations:

$$\begin{aligned} \mathbf{G}p \wedge \mathbf{G}q &\equiv \mathbf{G}(p \wedge q), \text{ and} \\ \mathbf{F}p \vee \mathbf{F}q &\equiv \mathbf{F}(p \vee q). \end{aligned}$$

For the classic LTL encoding scheme, the above optimising transformation rules need to be applied before the normalising transformation rules.

Another type of optimisation deals with *global constraints*. Consider the following LTL specification

$$\neg\mathbf{G}p \vee q \tag{7.1}$$

with p being a state formula (thus p contains no temporal operation). The negation of (7.1) is equivalent to

$$\mathbf{G}p \wedge \neg q. \tag{7.2}$$

Checking the satisfiability of (7.2) can be made more efficient if p is made into a global constraint (i.e., all states must satisfy p). The satisfiability checking of (7.2) would then be performed with $\varphi \equiv \neg q$ under the global constraint p.

Finally, some LTL model checking problems require no tableau construction. Consider the following LTL specification

$$\mathbf{G}p \tag{7.3}$$

with p being a state formula (thus p contains no temporal operation). The negation of (7.3) is equivalent to

$$\mathbf{F}\neg p. \tag{7.4}$$

If the system is non-terminating, then checking the satisfiability of (7.4) amounts to checking for reachable states that satisfy $\neg p$, since a finite path to a state that satisfies $\neg p$ can be extended into an infinite path. For a system that may terminate, although strictly speaking such a system is not covered by standard temporal logic, the reachability of a state that satisfies $\neg p$ may be viewed as a violation of (7.3) as a safety property, even if a finite path to p cannot be extended into an infinite path.

7.1.2 Tableau Generation

Recall from Section 6.1 that the tableau for φ is characterised by the transition constraint

$$\bigvee_{c \in \mathcal{P}(el(\varphi))} \left(\bigwedge_{p \in el(\varphi)} \text{if } p \in c \text{ then } V_p \wedge T_\varphi(p) \text{ else } \neg V_p \wedge \neg T_\varphi(p) \right)$$

if R' is defined by (6.9), or

$$\bigvee_{c \in \mathcal{P}(el(\varphi))} \left(\bigwedge_{p \in el(\varphi)} \text{if } p \in c \text{ then } V_p \wedge T_\varphi(p) \text{ else } \neg V_p \right)$$

if R' is defined by (6.10). For simplicity, let us assume that the Boolean state variables added when constructing M' (i.e., elements of $V_\varphi \cup P_\varphi$) are also OBDD variables. The sequence of these variables ordered according to the OBDD ordering is denoted $xvars(\varphi)$ (the sequence represents a totally ordered set).

Since a transition constraint is a propositional formula, it can be normalised into disjunctive normal form. Moreover, each disjunct can be further normalised such that it contains at most one literal involving *next* and the literal occurs positively. Such a normalised disjunct is called a *simple transition constraint*. The normalisation of a disjunct can use the following transformation rules:

$$\begin{aligned} \neg next(p) &\equiv next(\neg p), \text{ and} \\ next(p) \wedge next(q) &\equiv next(p \wedge q). \end{aligned}$$

A transition constraint is in *normal form* if and only if it is in disjunctive normal form and each disjunct is a simple transition constraint. A *normalised transition constraint* is a transition constraint in normal form.

For a simple transition constraint ctr, the conjunction of the non-*next* literals in ctr is called the *guard* of ctr, denoted $guard(ctr)$, and if there is a literal of the form $next(p)$ in ctr, then p is called the *post condition* of ctr, denoted $post(ctr)$. If ctr does not have a *next* literal, then $post(ctr) \equiv true$. A transition (s', t') satisfies a simple transition constraint ctr if and only if $M', s' \models guard(ctr)$ and $M', t' \models post(ctr)$, i.e.,

$$((s', t') \models c) \Leftrightarrow (M', s' \models guard(c)) \wedge (M', t' \models post(c)).$$

An augmented state s' can transition to an augmented state t' if and only if there is a disjunct ctr of the normalised transition constraint characterising the tableau for φ such that $(s', t') \models ctr$. A disjunct that contains $next(false)$ as a literal can be removed from a normalised transition constraint since no transition can satisfy the post condition $false$.

Let TC_φ represent the set of disjuncts of a normalised transition constraint characterising the tableau for φ. Then the symbolic image and pre-image transfer functions in M' for each elementary block b are defined as follows:

$$f'_b(I) \triangleq \bigvee_{ctr \in TC_\varphi} f_b(ignore(xvars(\varphi), I \wedge guard(ctr))) \wedge post(ctr), \tag{7.5}$$

$$r'_b(O) \triangleq \bigvee_{ctr \in TC_\varphi} r_b(ignore(xvars(\varphi), O \wedge post(ctr))) \wedge guard(ctr). \tag{7.6}$$

Note that the normalisation procedure for the overall transition constraint TC_φ can perform propositional simplification. In fact, with the use of OBDDs, some impossible transitions are automatically removed. The Minato-Morreale algorithm developed by Minato [Min93] based on the recursive operators of Morreale [Mor70] can be used to produce disjunctive normal forms from OBDDs.

To illustrate the construction of a tableau on a concrete example, let us use the φ in (6.2) as an example, using the TGBA encoding scheme:

$$\varphi \equiv \mathbf{FG}(\neg(Light = green)) \vee \mathbf{FG}(\neg(Light = red)).$$

The set of elementary formulas, $el(\varphi)$ would be

$$\begin{aligned} el(\varphi) \;=\; & \{\mathbf{FG}\neg(Light = green), \mathbf{G}\neg(Light = green), \\ & \mathbf{FG}\neg(Light = red), \mathbf{G}\neg(Light = red)\}. \end{aligned}$$

Let

$$\begin{aligned} v_1 &\text{ denote } V_{\mathbf{FG}\neg(Light=green)}, \\ v_2 &\text{ denote } V_{\mathbf{G}\neg(Light=green)}, \\ v_3 &\text{ denote } V_{\mathbf{FG}\neg(Light=red)}, \\ v_4 &\text{ denote } V_{\mathbf{G}\neg(Light=red)}, \\ p_1 &\text{ denote } P_{\mathbf{FG}\neg(Light=green)}, \text{ and} \\ p_2 &\text{ denote } P_{\mathbf{FG}\neg(Light=red)}. \end{aligned}$$

Suppose R' is defined by (6.10) (a loose encoding is used). For simplicity, let us further suppose that the normalisation into disjunctive normal form is naive and considers each $c \in \mathcal{P}(el(\varphi))$ as a separate case independent of the other elements of $\mathcal{P}(el(\varphi))$. The case for $c = \{v_2, v_3\}$ corresponds to the following transition constraint:

$$\neg v_1 \wedge v_2 \wedge v_3 \wedge \neg v_4 \wedge T_\varphi(\mathbf{G}(\neg(Light = green))) \wedge T_\varphi(\mathbf{FG}(\neg(Light = red)))$$

(since v_2 denotes $V_{\mathbf{G}\neg(Light=green)}$ and v_3 denotes $V_{\mathbf{FG}\neg(Light=red)}$), which, when evaluated using the rules for T_φ and S_φ, and simplified using propositional logic and the rule for *next*, produces the normalised transition constraint

$$\begin{aligned}\neg v_1 \wedge v_2 \wedge v_3 \wedge \neg v_4 \wedge \neg(Light = green) \wedge \neg(Light = red) \wedge next(v_2 \wedge v_4) \quad \vee \\ \neg v_1 \wedge v_2 \wedge v_3 \wedge \neg v_4 \wedge p_2 \wedge \neg(Light = green) \wedge (Light = red) \wedge next(v_2 \wedge v_3),\end{aligned}$$

which has two disjuncts. Each disjunct is a simple transition constraint with a guard and a post condition. Let ctr_1 be the first disjunct and ctr_2 be the second disjunct. Then we have:

$$\begin{aligned} guard(ctr_1) &\equiv \neg v_1 \wedge v_2 \wedge v_3 \wedge \neg v_4 \wedge \neg(Light = green) \wedge \neg(Light = red), \\ post(ctr_1) &\equiv v_2 \wedge v_4, \\ guard(ctr_2) &\equiv \neg v_1 \wedge v_2 \wedge v_3 \wedge \neg v_4 \wedge p_2 \wedge \neg(Light = green) \wedge (Light = red), \\ post(ctr_2) &\equiv v_2 \wedge v_3. \end{aligned}$$

Both ctr_1 and ctr_2 would become elements of TC_φ. The guards and post conditions would be used to modify the image and pre-image transfer functions as specified by (7.5) and (7.6).

7.1.3 Symbolic Counterexample Paths

Recall from Section 2.2.3, Lichtenstein and Pnueli [LP85] showed that φ is satisfiable in M' if and only if there is a prefix to a self-fulfilling strongly connected component (SCC) in the graph of states and transitions in M'. A self-fulfilling cycle within the self-fulfilling SCC can always be constructed, thus if φ is satisfiable in M', then there is a path π' in M' that satisfies φ of the form $p \cdot s^\omega$ where p is a possibly empty finite sequence of augmented states and s is a non-empty finite sequence of augmented states that forms a self-fulfilling cycle that is repeated forever (π' is of the so-called *lasso* form). π' can be projected to M to become π with $M, \pi \models \varphi$ (π retains the lasso form of π'). Although there may also be paths that satisfy φ that are not of the lasso form, such paths are difficult to characterise and are of no practical interest.

A counterexample path of the form $p \cdot s^\omega$ is difficult to read if the model M is large because the value of each state variable must be specified at each state in the path. An advantage of a symbolic setting is that a symbolic state (which represents a set of states) often has a more concise characterisation than an explicit state. Also, with a counterexample path of the form $p \cdot s^\omega$, there is no indication of how a state transitions to the next state in the path. A counterexample path can be easier to comprehend if the transitions are included at the level of the modelling notation used. Many model checkers

present counterexample paths with these high-level transitions. In our framework, these high-level transitions are represented by elementary blocks.

Definition 7.1 (Symbolic Counterexample Path)**.** A symbolic counterexample path is a triple:

$$(I_\pi, p_\pi, s_\pi)$$

where I_π characterises a set of initial states, prefix p_π is a possibly empty finite sequence of elementary blocks, and cycle s_π is a non-empty finite sequences of elementary blocks that is repeated forever. The elementary blocks represent high-level transitions.

Symbolic states after I_π can be computed successively using the image transfer functions for the corresponding elementary blocks in p_π and s_π. For a symbolic counterexample path in M', a symbolic state represents a set of augmented states and the image transfer function used for an elementary block b is f'_b. A symbolic counterexample path in M' can be projected to a symbolic counterexample path in M by projecting each symbolic augmented state in the path.

If $\langle SS_0, SS_1, SS_2, ...\rangle$ represents the sequence of symbolic augmented states in a symbolic counterexample path, and $\langle b_0, b_1, b_2, ...\rangle$ represents the sequence of transitions, then the following must hold:

$$\forall i \geq 0 : (SS_{i+1} \equiv f'_{b_i}(SS_i)) \wedge (SS_i \equiv r'_{b_i}(SS_{i+1})). \tag{7.7}$$

As a consequence, if $p_\pi = \langle b_{p_1}, b_{p_2}, ..., b_{p_m}\rangle$ and $s_\pi = \langle b_{s_1}, b_{s_2}, ..., b_{s_n}\rangle$ (where $m \geq 0$ is the length of the prefix and $n > 0$ is the length of the cycle), then the following must hold:

$$\begin{aligned} & (SS_0 \equiv I_\pi) \\ \wedge \quad & \forall i : 1 \leq i \leq m \Rightarrow (SS_i \equiv f'_{b_{p_i}}(SS_{i-1})) \wedge (SS_{i-1} \equiv r'_{b_{p_i}}(SS_i)) \\ \wedge \quad & \forall i : 1 \leq i \leq n \Rightarrow (SS_{m+i} \equiv f'_{b_{s_i}}(SS_{m+i-1})) \wedge (SS_{m+i-1} \equiv r'_{b_{s_i}}(SS_i)) \\ \wedge \quad & (SS_{m+n} \equiv SS_m). \end{aligned} \tag{7.8}$$

Note that m can be 0 (p_π can be an empty sequence). Since each elementary block b is deterministic, we have $|f'_b(SS)| \leq |SS|$ (the cardinality of the the image of SS under R'_b is not greater than the cardinality of SS). Thus all symbolic states in a cycle must have the same cardinality.

During a search for a symbolic counterexample path, (7.8) may be temporarily violated. This is because a forward search involves making choices of elementary blocks for transitions. As an example, suppose from a symbolic state SS_i the prefix search chooses elementary block $b_{p_{i+1}}$ as the transition. Some of the states in SS_i might not be able to transition through $b_{p_{i+1}}$, thus the equivalence $r'_{b_{p_{i+1}}}(f'_{b_{p_{i+1}}}(SS_i)) \equiv SS_i$ might not be satisfied, which means (7.8) might not be satisfied. Hence, once p_π, s_π and SS_{m+n} have been

determined, the symbolic states in the path must be narrowed so that (7.8) is satisfied. The symbolic states that are affected by s_π are narrowed first as follows:

$$\text{for } i \text{ from } n \text{ down to } 1 \text{ do } SS_{m+i-1} \leftarrow SS_{m+i-1} \wedge r'_{b_{s_i}}(SS_{m+i}). \tag{7.9}$$

Note that we must have $SS_{m+n} \not\equiv false$. If the result of narrowing SS_m results in $SS_{m+n} \not\equiv SS_m$, then we do not have a cycle and some recovery action is needed (see Section 8.4 for an example recovery action).. Otherwise the symbolic states affected by p_π are narrowed as follows:

$$\begin{array}{l} \text{for } i \text{ from } m \text{ down to } 1 \text{ do } SS_{i-1} \leftarrow SS_{i-1} \wedge r'_{b_{p_i}}(SS_i); \\ I_\pi \leftarrow SS_0. \end{array} \tag{7.10}$$

7.2 Analyses within the Framework

7.2.1 Strategies for Model Checking

The LTL model checking framework described in Section 7.1 does not specify how model checking and counterexample generation are to be performed. A range of approaches and strategies can be applied within the framework. The complete analysis comprising model checking and counterexample generation can be incremental, e.g., perform the following tasks:

1. compute the set of reachable states,
2. compute the set of reachable "fair" states using the set of reachable states,
3. search for a counterexample path using the set of reachable fair states,

or it can be a single task as in on-the-fly LTL model checking, which directly searches for a counterexample path.

In the incremental approach, the set of fair states is computed using fixpoint operations before the search for a counterexample path is performed. Strategically, there is choice between determining reachability eagerly or lazily. Reachability determination can be postponed until the search for a counterexample path, skipping the first task in the above sequence and computing the set of fair states instead of the set of reachable fair states. Regardless of the strategy for reachability, subsequent searches for counterexample paths need not repeat the computation of fair states.

Two standard approaches for LTL model checking are:

- the fixpoint approach, and

- the on-the-fly approach.

The following sections describes how the two approaches can be applied within our framework.

7.2.2 The Fixpoint Approach

Symbolic model checking as originally proposed by McMillan [McM92], follows a fixpoint approach first proposed by Emerson and Lei [EL86], using the fixpoint operations of μ-calculus [Koz83]. Although the original symbolic model checker was for CTL with fairness constraints, Clarke et al [CGMZ95] showed how LTL model checking can be encoded as a model checking problem for CTL with fairness constraints. The fixpoint approach for LTL model checking here is essentially that of Clarke et al, but is described directly in terms of fixpoint and propositional operations. Fixpoint operations are explained in Section 3.3, and Algorithm 5.1 for computing reachability in Chapter 5 provides an example of an implementation of a fixpoint operation.

Recall from Section 6.2 that as part of checking the satisfiability of φ, eventual conditions are checked using a set C_φ of fairness constraints.

Definition 7.2 (Fair States)**.** An augmented state is said to be fair with respect to C_φ if it can start a path in which each of the fairness constraints in C_φ is satisfied infinitely often.

The set of fair states is denoted F_φ, and has the following fixpoint characterisation (see [McM92] where transition relations are used rather than pre-image functions under transition relations):

$$F_\varphi \triangleq \nu Z. \bigwedge_{c \in C_\varphi} r'_B(\mu Y.(Z \wedge c) \vee r'_B(Y)). \tag{7.11}$$

The actual computation of $r'_B(p)$ can be distributed among the elementary blocks using r'_b for each elementary block b. Equation (7.11) applies to all encoding schemes.

Theorem 7.1. *An LTL formula φ is satisfiable in the underlying unaugmented model M if and only if $F_\varphi \wedge S_\varphi(\varphi) \wedge S_0$ is not false in the augmented model M'.*

Proof. The theorem is a symbolic formulation of Theorem 6.3 with paths constrained to start at initial states characterised by S_0. Theorem 6.3 states that a path π satisfies φ if and only if there exists a refinement π' of π in M' which starts at an augmented state that satisfies $S_\varphi(\varphi)$ and is fair with respect to C_φ. The existence of such π' that starts at an augmented state in S_0 can be determined from $F_\varphi \wedge S_\varphi(\varphi) \wedge S_0$ since F_φ characterises all augmented states that can start fair (with respect to C_φ) paths. □

Since φ is the negation of the LTL formula model checked, $F_\varphi \wedge S_\varphi(\varphi) \wedge S_0$ characterises the set of augmented states that can start counterexample paths, hence the following definition

Definition 7.3 (Initial counterexample states)**.** I_φ defined as

$$I_\varphi \triangleq F_\varphi \wedge S_\varphi(\varphi) \wedge S_0$$

characterises the set of initial counterexample states.

Equation (7.11) and Theorem 7.1 form the basis for symbolic LTL model checking using the fixpoint approach. Both are independent of the encoding scheme. If $I_\varphi \not\equiv \mathit{false}$ then the existence of a counterexample path is guaranteed. States in a counterexample path are necessarily reachable fair states. On the other hand, if $I_\varphi \equiv \mathit{false}$, then there is no counterexample, and the original LTL specification is satisfied by M. However, I_φ does not immediately give us a counterexample path. We still need to search for a counterexample path.

The first method for generating a counterexample path given C_φ, F_φ and I_φ was developed by Clarke et al [CGMZ95] in the context of symbolic CTL model checking with fairness constraints. The search for a counterexample path is guided by fair states that satisfy fairness constraints. In our framework, the set of fair states that satisfy a fairness constraint $c \in C_\varphi$ is characterised by $c \wedge F_\varphi$. If Q_i^c denotes the set of fair states that can reach $c \wedge F_\varphi$ in exactly i steps, we have

$$\begin{aligned} Q_0^c &\leftarrow c \wedge F_\varphi, \text{ and} \\ Q_i^c &\leftarrow r'_B(Q_{i-1}^c) \text{ for } i \geq 1, \end{aligned}$$

In [CGMZ95], Q_i^c denotes the set of fair states that can reach $c \wedge F_\varphi$ in i or fewer steps instead of exactly i steps. We could do the same thing by computing

$$\begin{aligned} Q_0^c &\leftarrow c \wedge F_\varphi, \text{ and} \\ Q_i^c &\leftarrow Q_{i-1}^c \vee r'_B(Q_{i-1}^c) \text{ for } i \geq 1, \end{aligned}$$

but it is not clear which approach is better. Perhaps the approach in [CGMZ95] takes advantage of the fixpoint computation already performed. The rest of the search proceeds as in [CGMZ95]. Narrowing using (7.9) and (7.10) needs to be performed on a potential symbolic counterexample path.

The method of Clarke et al is not the only way to generate counterexample paths once I_φ is computed. A method for directed counterexample path generation that can be performed after F_φ is computed is proposed in Chapter 8.

Recall from Section 7.2.1 that different reachability strategies can be used in a fixpoint approach to model checking. For general reachability, we have two basic strategies:

1. Precompute the set of reachable states (reachable from S_0) and use the information for subsequent analysis. We call this strategy *eager reachability strategy* (ERS).

2. Do not precompute general reachability and compute fair states without any concern for reachability. Reachability of fair states can be determined in subsequent analysis. We call this strategy *lazy reachability strategy* (LRS).

There are also two basic strategies for reachability with respect to counterexample path generation:

1. Precompute the set of reachable counterexample states (the set $F_{C\varphi}$ of fair states reachable from $S_0 \wedge S_\varphi(\varphi)$), and use $F_{C\varphi}$ as the search space for counterexample path generation. We call this strategy *eager counterexample strategy* (ECS).

2. Do not precompute the set of reachable counterexample states, and use F_φ as the search space for counterexample path generation. We call this strategy *lazy counterexample strategy* (LCS).

7.2.3 On-the-fly Symbolic LTL Model Checking

The proposed on-the-fly symbolic model checking algorithm is an adaptation of the double depth-first-search (double-DFS) algorithm [CVWY92] that is commonly associated with an explicit automata approach in LTL model checking. No fixpoint computation of fair states is needed. The algorithm searches for a counterexample path directly from the set of initial states. Fairness is discovered on-the-fly in a depth-first search for a symbolic counterexample path.

Algorithm 7.1. Standard Double-DFS Algorithm

dfs1(s) :
push($s, stk1$);
for each successor s' of s do
  if s' not in $stk1$ then dfs1(s');
if s is an accepting state then dfs2(s);
pop($stk1$);

dfs2(s) :
push($s, stk2$);
for each successor s' of s do
  if s' in $stk1$ then **terminate**($true$);
  else if s' not in $stk2$ then dfs2(s')
pop($stk2$);

The basic double-DFS algorithm in [CVWY92] is shown as Algorithm 7.1 (but without hashing). The procedure dfs1 searches for a reachable accepting state and the procedure dfs2 searches for a cycle involving the reachable accepting state found. To use the algorithm, dfs1 is invoked with each initial state until either success (when **terminate**(*true*) is executed) or there is no more initial state to process. The algorithm implements an emptiness check for a language of infinite paths. The language may be viewed as being accepted by a Büchi automaton (BA) [Büc60]. When the algorithm detects non-emptiness, a witness for the non-emptiness can be produced from the contents of $stk1$ and $stk2$. In our application, the witness corresponds to a counterexample path (a path satisfying φ).

The proposed on-the-fly symbolic LTL checking algorithm is an adaptation of the standard double-DFS algorithm with the following important differences:

- The proposed algorithm works with symbolic transitions (using elementary blocks) and symbolic states.
- The proposed algorithm constructs the Büchi automaton implicitly and on-the-fly.

The proposed algorithm is first and foremost a search for a counterexample path rather than simply an emptiness check for a language accepted by a BA. A search for a counterexample path needs to keep track of fairness constraints fulfilled. As the number of fairness constraints becomes large, precise tracking of fairness constraints fulfilled becomes infeasible. Here we follow an approach that is commonly used in moving from a generalised Büchi automaton (GBA) to a BA[1]: serialising the fulfillment of fairness constraints in a specific order. The fairness constraints are placed in some order, say $c_1, c_2, ..., c_n$, and their fulfillment is strictly one by one in that order. A counter ctr is used to keep track of fairness constraints fulfilled in the path considered so far. Initially, no fairness constraint is fulfilled and $ctr = 0$. Thereafter, $ctr = i$ means for all j such that $1 \leq j \leq i$, c_j has been fulfilled, but for all j such that $i < j \leq n$, c_j has not been recorded as being fulfilled (although it might have been fulfilled). Using this method of imprecise but safe tracking (if a fairness constraint has been recorded as being fulfilled, then it has been fulfilled), the counter ctr is reset to 0 upon transition from a state in which $ctr = n$. For our application, a state in a BA is an augmented state further augmented with the counter ctr. An accepting state in the Büchi automaton is a state with $ctr = n$.

Because the tracking of fairness constraints is imprecise, some counterexample paths may not be recognised by the algorithm. However, if there is a counterexample path, then there is one that is recognised by the algorithm because a counterexample path

[1]The GBA and the BA do not necessarily accept the same language. The GBA allows precise tracking of fairness constraint and the BA corresponds to imprecise tracking.

can always be "serialised" by transforming its cycle part (whose states belong to a "self-fulfilling" strongly connected component) to one that fulfills the fairness constraints in order. In general, the serialisation may result in a longer cycle.

Algorithm 7.2. Naive Symbolic On-the-fly LTL Counterexample Path Generation

dfs1(S, ctr) :
push$((S, ctr), stk1)$;
for each $b \in B$ do
  push$(b, bstk1)$;
  if $ctr = n$ then
    $S_1 \leftarrow f'_b(S)$;
    if $(S_1 \not\equiv false) \wedge \neg$in$((S_1, 0), stk1)$ then dfs1$(S_1, 0)$;
  else
    $S_1 \leftarrow f'_b(S) \wedge C(ctr + 1)$;
    if $(S_1 \not\equiv false) \wedge \neg$in$((S_1, ctr + 1), stk1)$ then dfs1$(S_1, ctr + 1)$;
    $S_1 \leftarrow f'_b(S) \wedge \neg C(ctr + 1)$;
    if $(S_1 \not\equiv false) \wedge \neg$in$((S_1, ctr), stk1)$ then dfs1(S_1, ctr);
  pop$(bstk1)$;
if $ctr = n$ then dfs2(S, n);
pop$(stk1)$;

dfs2(S, ctr) :
push$((S, ctr), stk2)$;
for each $b \in B$ do
  push$(b, bstk2)$;
  if $ctr = n$ then
    $S_1 \leftarrow f'_b(S)$;
    if in$((S_1, 0), stk1)$ then **terminate**$(S_1, 0)$;
    if $(S_1 \not\equiv false) \wedge \neg$in$((S_1, 0), stk2)$ then dfs2$(S_1, 0)$;
  else
    $S_1 \leftarrow f'_b(S) \wedge C(ctr + 1)$;
    if in$((S_1, ctr + 1), stk1)$ then **terminate**$(S_1, ctr + 1)$;
    if $(S_1 \not\equiv false) \wedge \neg$in$((S_1, ctr + 1), stk2)$ then dfs2$(S_1, ctr + 1)$;
    $S_1 \leftarrow f'_b(S) \wedge \neg C(ctr + 1)$;
    if in$((S_1, ctr), stk1)$ then **terminate**(S_1, ctr);
    if $(S_1 \not\equiv false) \wedge \neg$in$((S_1, ctr), stk2)$ then dfs2(S_1, ctr);
  pop$(bstk2)$;
pop$(stk2)$;

Algorithm 7.2 is to be invoked with dfs1$(S_0 \wedge S_\varphi(\varphi), 0)$, and C is assumed to be an array of fairness constraints with $C(i) \equiv c_i$ for $1 \leq i \leq n$. The stacks $stk1$, $bstk1$, $stk2$, $bstk2$ are initially empty. Termination via **terminate** means a counterexample path is found. Further processing needs to be performed to obtain (I_π, p_π, s_π) from $S_0 \wedge S_\varphi(\varphi)$, $stk1$, $bstk1$, $stk2$, $bstk2$, and the symbolic state and index arguments of the **terminate**. The stacks $bstk1$ and $bstk2$ contain the symbolic transitions, while $stk1$ and $stk2$ contain the provisional symbolic states between symbolic transitions. Note that upon termination via **terminate**, $length(bstk1) = length(stk1)$ and $length(bstk2) = length(stk2)$. The algorithm terminates (via **terminate** or otherwise) since the model is finite and n is finite.

If Algorithm 7.2 terminates via **terminate** with symbolic state SS and index ii, the cycle part of a symbolic counterexample path s_π can be computed as follows:

- $j \leftarrow stackIndex((SS, ii), stk1)$;
- $s_\pi \leftarrow append(substack(bstk1, j), bstk2)$;

where $stackIndex((SS, ii), stk1)$ gives the length of the prefix of $stk1$ up to and including (SS, ii), $substack(bstk1, j)$ gives the prefix of $bstk1$ of length j, and *append* is the sequence append operation. Note that we assume that when a stack is treated as a sequence, a push operation pushes an element to the tail end of the sequence. Let $stk3$ and $bstk3$ be the results of popping j times the stacks $stk1$ and $bstk1$ respectively. Then $p_\pi \leftarrow bstk3$. Let $i = length(stk3)$, $stk3 = \langle (SS_0, ii_0), ..., (SS_{i-1}, ii_{i-1}) \rangle$, $p_\pi = \langle b_{p_1}, ..., b_{p_m} \rangle$ and $SS_i \equiv SS$. Then after narrowing symbolic states using (7.10), the narrowed SS_0 becomes I_π.

In Algorithm 7.2, the operation in$((SS, ii), stk)$ simply checks to see if (SS, ii) is an entry in the stack stk. A less naive algorithm can use a subsumption test instead of equality when checking if (SS, ii) is a stack entry so that in$((SS, ii), stk)$ is equivalent to there exists an entry (SSS, ii) in stk such that $SSS \wedge SS \equiv SS$. However, the resulting cycle needs to have the symbolic states narrowed and the narrowed initial and final symbolic states in the cycle need to be identical (see Section 7.1.3). Otherwise, s_π does not necessarily produce a valid cycle.

Theorem 7.2. *φ is satisfiable in M' if and only if Algorithm 7.2 terminates via* **terminate** *when invoked with dfs1$(S_0 \wedge S_\varphi(\varphi), 0)$.*

Proof. The only differences between Algorithm 7.2 and Algorithm 7.1 (whose correctness has been shown in [CVWY92]) are:

- Algorithm 7.2 uses symbolic transitions and symbolic states, and
- Algorithm 7.2 constructs the Büchi automaton on-the-fly.

A set of Büchi automaton states is represented by a pair consisting of a symbolic state S and a counter ctr. Whereas a successor s' of a BA state is chosen directly from the BA, Algorithm 7.2 implicitly constructs the BA on-the-fly, and breaks the handling of successors into three cases:

1. The case where $ctr = n$ (which means the symbolic state is an accepting state), the successor must have $ctr = 0$ regardless of fairness constraints satisfied in the successor.

2. The case where $ctr = i$, $0 \leq i < n$, and the successor symbolic state satisfies $C(i+1)$, ctr is incremented to $i+1$.

3. The case where $ctr = i$, $0 \leq i < n$, and the successor symbolic state does not satisfy $C(i+1)$, ctr remains at i.

The three cases represent the serialisation of fulfillment of fairness constraints. The use of symbolic states simply means the enumeration of explicit states is avoided or postponed. A symbolic state in the search (which in the algorithm is checked to be not equivalent to $false$) represents a non-empty set of states. The explicit states in the symbolic state are exactly those that can be reached from any state satisfying $S_\varphi(\varphi) \wedge S_0$ through the transitions (elementary blocks) chosen up to that point. □

Partial order reduction [PWW96] can be used to reduce the search space in on-the-fly LTL checking. [HPY96] shows how the double-DFS algorithm can be modified to accommodate partial order reduction in explicit on-the-fly model checking. To accommodate partial order reduction in a symbolic setting, Algorithm 7.2 needs to be similarly modified.

A variation of the on-the-fly approach might be an "iterative deepening" search for counterexamples, achieved by placing a depth limit in the depth-first search, and iteratively increasing the depth limit.

7.3 Summary

This chapter described how the conceptual foundation for LTL model checking from Chapter 6 can be applied using the state machine framework of Chapter 3. The result is a symbolic framework for LTL model checking and counterexample generation that accommodates various approaches and strategies, including on-the-fly LTL model checking as well as the traditional fixpoint approach. A novel idea proposed in this chapter is that of on-the-fly symbolic LTL model checking.

Chapter 8

Directed Counterexample Path Generation

With on-the-fly LTL model checking, the search for a counterexample path is a blind search, while the method of Clarke et al [CGMZ95] performs a search that is guided by states that satisfy fairness constraints. Without modification, such searches correspond to blunt instruments that cannot be directed towards or away from specific cases of interest. In this chapter, a method for symbolic counterexample path generation that involves a goal-directed search is proposed.

Section 8.1 gives a motivating example for directed counterexample path generation. Section 8.2 outlines the proposed method for directed counterexample path generation. Sections 8.3 and 8.4 provide details of the proposed method. Section 8.5 discusses possible uses of directed counterexample path generation.

8.1 A Motivating Example

Consider the BT example in Figure 8.1 representing a simple hypothetical system, chosen to illustrate the capabilities required for directed search. The system has three subsystems — SubA, SubB and SubC — and two critical components: CompA and CompB. After CompA and CompB are initialised, five threads are spawned. Two of the threads allow components CompA and CompB to fail, and the other three control the operations of subsystems SubA, SubB and SubC. The operation of subsystem SubA, represented by BT nodes 7 through 11, requires both CompA and CompB to be operational. The operation of SubB, represented by BT nodes 12 through 15, requires CompA to be operational. The operation of SubC, represented by BT nodes 16 through 18, does not require either component to be operational. External events (indicated by $\rangle\rangle$event$\langle\langle$) and prioritisation of system transitions over external events (explained in Section 4.6.4) are used in the

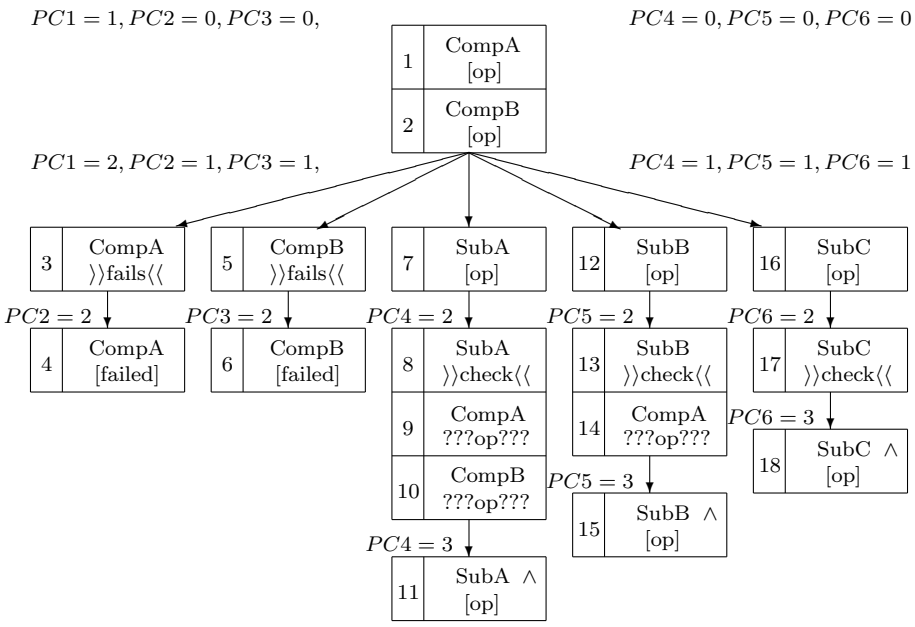

Figure 8.1: A Motivating Example

model to prevent race conditions.

When the system is operating failure-free, CompA and CompB would be operational, indicated in the BT model by both CompA and CompB having the value *op* after their initialisation in BT nodes 1 and 2. Failure-free operation of the system can be formalised as the LTL formula:

$$\mathbf{G}((PC1 = 1) \vee (CompA = op) \wedge (CompB = op)). \tag{8.1}$$

The state formula $(PC1 = 1)$ characterises the set of starting states before CompA and CompB are initialised (the BT translation described in Chapter 4 does not assume that state variables are initialised, thus there may be many possible starting states).

Suppose we want to know if subsystem SubC can still operate after a component fails. The continued operation of SubC is represented in the BT model by a cycle that goes through BT node 18 (which is a reversion back to the BT node 16, meaning the effect of executing BT node 18 is the same as the effect of executing BT node 16). Thus, the continued operation of SubC after a component failure is indicated by a counterexample to the LTL formula (8.1) whose cycle goes through BT node 18. This motivates the idea of a cycle constraint in a counterexample path.

Definition 8.1 (Cycle Constraint)**.** We define a cycle constraint *cc* in a symbolic counterexample path to be a state formula that must be satisfied by at least one symbolic state in the cycle part of the counterexample path.

For continued operation of SubC, $cc \equiv (PC6 = 3)$ would be sufficient. This is because

- system transitions are prioritised over external events in the model, and
- each iteration of a thread in the model goes through at least one external event,

thus when the thread for SubC is at $(PC6 = 3)$ (at the entrance to BT node 18), the other threads would all be waiting for external events, and since the execution of BT node 18 is considered a system transition, the other threads would all be blocked leaving the execution of BT node 18 as the only possible transition.

An example symbolic counterexample path for (8.1) that satisfies the cycle constraint $(PC6 = 3)$ is the following:

$$\begin{aligned} I_\pi &\equiv (PC1 = 1) \wedge (PC2 = 0) \wedge (PC3 = 0) \wedge (PC4 = 0) \\ &\quad \wedge (PC5 = 0) \wedge (PC6 = 0), \\ p_\pi &= \langle (1,2), 7, 12, 16, 3, 4, 17 \rangle, \\ s_\pi &= \langle 18, 17 \rangle. \end{aligned}$$

The counterexample path was in fact produced by a prototype described in Section 9.1. The counterexample path can be read as: after initialisations of CompA and CompB, each of the subsystems advances to its wait-for-event state, CompA then fails and SubC proceeds indefinitely in a cycle. The counterexample path corresponds to CompA failing and SubC keeps on operating, and shows how it can happen.

The above counterexample involves CompA failing. Suppose now we would like to know if there are counterexamples to (8.1) in which SubC keeps on operating but CompA does not fail. A global constraint can be used here.

Definition 8.2 (Global Constraint)**.** We define a global constraint gc in a counterexample path to be a state formula that must be satisfied by all states in the counterexample path.

By specifying $gc \equiv \neg(CompA = failed)$ as well as $cc \equiv (PC6 = 3)$, we are specifying constraints for a counterexample to (8.1) in which CompA does not fail and SubC keeps on operating.

When asked to find a symbolic counterexample path to (8.1) that satisfies the cycle constraint $(PC6 = 3)$ and the global constraint $\neg(CompA = failed)$, the prototype produced a symbolic counterexample path with I_π and s_π as above, and

$$p_\pi = \langle (1,2), 7, 12, 16, 5, 6, 17 \rangle.$$

This corresponds to CompB failing instead of CompA. The global constraint essentially directs the search for a counterexample away from cases where CompA fails.

Cycle constraints can also be used to direct the search for a counterexample path towards a specific set of components failing. As an example, we can specify $cc \equiv (PC6 = 3) \wedge (CompA = \mathit{failed}) \wedge (CompB = \mathit{failed})$ to direct the search towards a counterexample path where SubC still operates after CompA and CompB both failed. When asked to find a symbolic counterexample path that satisfies $cc \equiv (PC6 = 3) \wedge (CompA = failed) \wedge (CompB = failed)$, the prototype produced a symbolic counterexample path with I_π and s_π as above and

$$p_\pi = \langle (1,2), 7, 12, 16, 3, 4, 5, 6, 17 \rangle.$$

The prefix is longer than in the previous two counterexample paths since now both components need to fail before the cycle is reached.

Suppose now we direct the search for a counterexample path for (8.1) with the cycle constraint $(PC5 = 3)$. The prototype produced the following symbolic counterexample path:

$$\begin{aligned} I_\pi &\equiv (PC1 = 1) \wedge (PC2 = 0) \wedge (PC3 = 0) \wedge (PC4 = 0) \\ &\quad \wedge (PC5 = 0) \wedge (PC6 = 0), \\ p_\pi &= \langle (1,2), 7, 12, 16, 5, 6, (13,14) \rangle, \\ s_\pi &= \langle 15, (13,14) \rangle. \end{aligned}$$

The counterexample path corresponds to CompB failing and SubB keeps on operating.

If we direct the search for a counterexample path for (8.1) with $cc \equiv (PC5 = 3)$ and $gc \equiv \neg(CompB = failed)$ then no counterexample path would be found. Similarly, for the case where SubA keeps on operating, there is no counterexample to (8.1).

8.2 Method Outline

The proposed method for directed symbolic counterexample path generation is invoked with two parameters:

- *cc* - a state formula (a proposition without temporal operations) specifying a constraint that must be satisfied by a symbolic state in the cycle part of the symbolic counterexample path (all states in the symbolic state must satisfy *cc*). A *cc* value of *true* means there is no cycle constraint.

- *gc* - a state formula specifying a global constraint (all states in the symbolic counterexample path must satisfy *gc*). A *gc* value of *true* means there is no global constraint.

The two parameters direct the search for a counterexample path. Neither of the two parameters can be *false*.

The basic strategy in the search for a counterexample path is as follows:

- Find a cycle in which all fairness constraints are fulfilled, which visits a symbolic state that satisfies $gc \wedge cc$ (i.e., all states in the symbolic state satisfy $gc \wedge cc$), and in which all the symbolic states satisfy gc. The cycle found produces s_π.
- Find a prefix p_π from a symbolic state that satisfies $S_\varphi(\varphi) \wedge S_0 \wedge gc$ to the cycle. If found, the prefix determines I_π and completes the generation of a symbolic counterexample path.

The symbolic state that functions as a starting point for the search is SB_0. If the method is invoked after F_φ has been computed, then SB_0 is initialised as follows:

$$SB_0 \leftarrow gc \wedge cc \wedge F_\varphi.$$

Otherwise SB_0 is initialised as follows:

$$SB_0 \leftarrow gc \wedge cc.$$

SB_0 in effect becomes the cycle constraint.

8.3 Cycle Search

The method searches for the cycle part of a symbolic counterexample path first. The cycle search consists of the following stages:

- Search backward from SB_0 to find a starting point for "choosing" a cycle, while partitioning the possible intermediate symbolic states in the cycle based on the fairness constraints to be fulfilled.
- From the starting point found, choose transitions (elementary blocks) to form a candidate cycle, using the partitioned possible intermediate symbolic states.
- Narrow and validate the the symbolic states in the candidate cycle (see Section 7.1.3 for why this is needed).

The cycle search described here uses precise tracking of fairness constraints. If the number of fairness constraints is large, imprecise tracking, e.g., via the serialisation technique as in Section 7.2.3, can be used instead to avoid the exponential complexity of precise tracking.

Possible intermediate states are partitioned. At each step in the backward search stage of the cycle, the symbolic state is partitioned based on the fairness constraints fulfilled in paths from the symbolic state to SB_0. There are 2^k partitions at each step i, denoted $P_i(cmb)$ where each index cmb is a subset of C_φ, and $k = |C_\varphi|$. The partition $P_i(cmb)$ represents states that can transition to states in SB_0 in exactly i steps with exactly the fairness constraints in cmb fulfilled along the path (inclusive of the end points). P_0 is initialised as follows:

for $cmb \subseteq C_\varphi$ do
  $P_0(cmb) \leftarrow SB_0 \wedge \bigwedge_{c \in C_\varphi}$ if $c \in cmb$ then c else $\neg c$.

Procedure 8.1, when successful (via **terminate**(success)), produces a positive integer n and partitions $P_i(cmb)$ for $0 \leq i \leq n$ and $cmb \subseteq C_\varphi$. It is invoked after initialisation of P_0, using cycle-stage1(1). Termination other than through **terminate**(success) is considered a failure, and **restart**(cycle-stage-1(1)) abandons the current search and restarts a new search (with revised SB_0 and P_0). The variables SB_0, n, and $P_i(cmb)$ for $0 \leq i \leq n, cmb \subseteq C_\varphi$, are global variables.

Procedure 8.1. Cycle Search Stage 1

cycle-stage1(i) :
$SB_i \leftarrow r'_B(SB_{i-1}) \wedge gc$;
for each $cmb \subseteq C_\varphi$ do
  $P_i(cmb) \leftarrow SB_i \wedge \left(\bigwedge_{c \in C_\varphi \setminus cmb} \neg c\right) \wedge \bigvee_{t \subseteq cmb} \left(r'_B(P_{i-1}(t)) \wedge \bigwedge_{c \in cmb \setminus t} c\right)$;
if $(P_i(C_\varphi) \wedge SB_0) \equiv SB_0$ then
  $n \leftarrow i$;
  terminate(success);
if $(P_i(C_\varphi) \wedge SB_0) \not\equiv false$ then
  $SB_0 \leftarrow P_i(C_\varphi) \wedge SB_0$;
  for each $cmb \subseteq C_\varphi$ do
    $P_0(cmb) \leftarrow SB_0 \wedge \bigwedge_{c \in C_\varphi}$ if $c \in cmb$ then c else $\neg c$;
  restart(cycle-stage-1(1));
if $\neg\exists j : i > j \geq 0 \wedge \forall cmb \subseteq C_\varphi : P_i(cmb) \equiv P_j(cmb)$ then
  cycle-stage-1($i + 1$);

In the case where $(P_i(C_\varphi) \wedge SB_0) \not\equiv SB_0$ but $(P_i(C_\varphi) \wedge SB_0) \not\equiv false$, SB_0 is narrowed and the backward search is restarted. This step solves the problem of SB_0 possibly containing states that cannot be in cycles (and thus cannot be the starting point for choosing a cycle). This step also tends to favour short cycles.

Theorem 8.1. *If* Procedure 8.1 *terminates successfully via* **terminate**(success)*, then the resulting SB_0 characterises a non-empty subset of the set characterised by the original SB_0, and from any state in the resulting SB_0, there is a path with exactly n transitions to a state in SB_0, where all fairness constraints in C_φ are fulfilled in the path. Otherwise* Procedure 8.1 *terminates unsuccessfully. Note that only states that satisfy the global constraint gc are considered.*

Proof. For the first part of Theorem 8.1 (where Procedure 8.1 terminates successfully), we first prove (8.2).

$$\begin{aligned}&\forall 0 \leq i \leq n, cmb \subseteq C_\varphi : \forall s \in P_i(cmb) : \\ &\quad \exists p : p \text{ is a path from } s, \text{ of exactly } i \text{ transitions, to a state in } SB_0 \\ &\qquad \text{and for each } c \in C_\varphi : c \text{ is fulfilled in } p \text{ iff } c \in cmb.\end{aligned} \tag{8.2}$$

The proof of (8.2) is by induction on i.

- Case $i = 0$ (base case). This follows from the initialisation of P_0 if Procedure 8.1 is never restarted or from the last assignment of P_0 if there is a restart.
- Case $i > 0$. This follows from the assignment

$$P_i(cmb) \leftarrow SB_i \wedge \left(\bigwedge_{c \in C_\varphi \setminus cmb} \neg c \right) \wedge \bigvee_{t \subseteq cmb} \left(r'_B(P_{i-1}(t)) \wedge \bigwedge_{c \in cmb \setminus t} c \right);$$

  in Procedure 8.1. The induction hypothesis assures us that $P_{i-1}(t)$ contains exactly the states that can transition to SB_0 in exactly $i-1$ transitions with exactly the fairness constraints in t fulfilled in the path to SB_0. A fairness constraint c not in cmb cannot be fulfilled in a state in $P_i(cmb)$ or in a path of $i-1$ transitions from a state in $P_{i-1}(t)$ to a state in SB_0 for all $t \subseteq cmb$, thus cannot be fulfilled in a path of i transitions from a state in $P_i(cmb)$ to a state in SB_0. The above assignment also guarantees that a state s in $P_i(cmb)$ can transition to a state in $P_{i-1}(t)$ where $t \subseteq cmb$ and any $c \in cmb$ that is not fulfilled by s is in t. Thus $P_i(cmb)$ contains exactly those state from which there are paths of exactly i transitions to SB_0 where all fairness constraints in cmb are fulfilled.

If Procedure 8.1 terminates successfully, then SB_0 is a subset of $P_n(C_\varphi)$, and from (8.2) we can conclude that from any state in SB_0 there is a path of n transitions to a state in SB_0 with all the fairness constraints in C_φ are fulfilled in the path.

The second part of Theorem 8.1 is guaranteed by the "visited check":

$$\text{if } \neg\exists j : i > j \geq 0 \wedge \forall cmb \subseteq C_\varphi : P_i(cmb) \equiv P_j(cmb)$$

and the finiteness of the model. □

If Procedure 8.1 terminates successfully, then the second stage — represented by Procedure 8.2 — can start. Procedure 8.2 produces a provisional cycle represented by transitions $b_1, ..., b_n$ and intermediate symbolic states $SS_0, ..., SS_n$. It is invoked using cycle-stage2(1) after SS_0 and cmb_0 are initialised as follows:

$$SS_0 \leftarrow SB_0;$$
$$cmb_0 \leftarrow C_\varphi.$$

The variables $b_1, ..., b_n$, $SS_0, ..., SS_n$ and $cmb_0, ..., cmb_n$ are global variables.

Procedure 8.2. Cycle Search Stage 2

cycle-stage2(i) :
for each $b \in B, t \subseteq cmb_{i-1}$ do
  $SS_i \leftarrow f'_b(SS_{i-1} \wedge \bigwedge_{c \in cmb_{i-1} \setminus t} c) \wedge P_{n-i}(t)$;
  if $SS_i \not\equiv false$ then
    $b_i \leftarrow b$;
    $cmb_i \leftarrow t$;
    if $i = n$ then **terminate**(success);
    else cycle-stage2($i + 1$);

Theorem 8.2. *If invoked after a successful* Procedure 8.1, Procedure 8.2 *terminates successfully.*

Proof. At every iteration (recursive call) of Procedure 8.2, the choice of elementary block b_i and combination cmb_i (which means partition $P_{n-i}(cmb_i)$ is chosen) together with (8.2) ensure success without any need to backtrack (cmb_i represents fairness constraints to be fulfilled starting from the next iteration). □

After Procedure 8.2 terminates via **terminate**(success), $SS_0, ..., SS_n$ and $b_1, ..., b_n$ represent a provisional cycle. The provisional symbolic states must be narrowed as follows:

$$\text{for } i \text{ from } n \text{ down to } 1 \text{ do } SS_{i-1} \leftarrow SS_{i-1} \wedge r'_{b_i}(SS_i).$$

After narrowing the symbolic states, the equivalence $SS_0 \equiv SS_n$ is checked. If the equivalence holds, then the cycle search is successful, with $s_\pi \leftarrow \langle b_1, ..., b_n \rangle$, and the prefix search can begin.

8.4 Prefix Search

If the SS_0 from the cycle search gives us $(SS_0 \wedge S_0) \not\equiv false$, then an empty prefix has been found and we assign $p_\pi \leftarrow \langle\rangle$ and $I_\pi \leftarrow SS_0 \wedge S_0$. Since the starting symbolic state

for the cycle is now I_π, SS_0 must be adjusted: $SS_0 \leftarrow I_\pi$, and the symbolic states in the cycle further narrowed. It is possible, although unlikely in practice, that the resulting SS_0 is not equivalent to the resulting SS_n. In such a case, (I_π, p_π, s_π) is still a symbolic counterexample path (since all states in the original unnarrowed cycle can participate in the cycle and remain in the original cycle), but the cycle is of length a multiple of n. The intermediate symbolic states in the cycle can be computed as follows:

$SS_0 \leftarrow I_\pi$;
$j \leftarrow 0$;
repeat
  for i from 1 to n do
    $SS_{jn+i} \leftarrow f'_{b_i}(SS_{jn+i-1})$;
  $j \leftarrow j + 1$;
until $SS_{jn} \equiv SS_0$

The real length of the cycle is jn.

If $(SS_0 \wedge S_0) \equiv \mathit{false}$, then a search for a prefix to the cycle must be performed. The search for a prefix to the cycle also consists of a backward search stage followed by a forward "choosing" stage, but the stages are simpler than the corresponding stages for the cycle search because there is no need to track fairness constraints, thus the restricted possible intermediate symbolic states in the prefix need not be partitioned. The symbolic state that functions as the starting point for the backward search is SP_0, initialised as follows:

$$SP_0 \leftarrow SS_0.$$

The backward search stage of finding a prefix is represented by Procedure 8.3. Procedure 8.3, when successful (via **terminate**(success)), produces a positive integer m representing the length of the prefix, and $SP_0, ..., SP_m$ representing the restricted possible intermediate symbolic states. After the initialisation of SP_0, it is invoked using prefix-stage1(1).

Procedure 8.3. Prefix Search Stage 1

prefix-stage1(i) :
$SP_i \leftarrow r'_B(SP_{i-1}) \wedge gc$;
$SP_i \wedge S_0 \not\equiv \mathit{false}$ then
  $m \leftarrow i$;
  terminate(success)
if $\neg\exists j : i > j \geq 0 \wedge (SP_i \equiv SP_j)$ then
  prefix-stage-1($i + 1$);

Theorem 8.3. *If* Procedure 8.3 *terminates successfully, then from any state in the resulting SP_m, there is a path with exactly m transitions to a state in SP_0. Otherwise* Procedure 8.3 *terminates with failure.*

Proof. For the first part of Theorem 8.3 (where Procedure 8.3 terminates successfully), we first prove (8.3).

$$\begin{aligned} &\forall 0 \leq i \leq m : \forall s \in SP_i : \\ &\quad \exists p : p \text{ is a path from } s, \text{ of exactly } i \text{ transitions, to a state in } SP_0. \end{aligned} \tag{8.3}$$

The proof of (8.3) is by induction on i.

- Case $i = 0$ (base case). This is trivial.
- Case $i > 0$. This follows from the assignment

  $$SP_i \leftarrow r'_B(SP_{i-1}) \wedge gc;$$

  in Procedure 8.1. The induction hypothesis assures us that SP_{i-1} contains exactly the states that can transition to SP_0 in exactly $i - 1$ transitions.

The first part of Theorem 8.3 follows directly from (8.3).

The second part of Theorem 8.3 is guaranteed by the "visited check":

$$\text{if } \neg\exists j : i > j \geq 0 \wedge (SP_i \equiv SP_j)$$

and the finiteness of the model. □

If Procedure 8.3 terminates via **terminate**(success), then a prefix can be found and the second stage, represented by Procedure 8.4, can start. The starting point for the choosing stage is the symbolic state SSS_0, initialised as follows:

$$SSS_0 \leftarrow SP_m \wedge S_0.$$

After initialisation, Procedure 8.4 is invoked using prefix-stage2(1).

Procedure 8.4. Prefix Search Stage 2

```
prefix-stage2(i) :
for each b ∈ B do
  SSS_i ← f'_b(SSS_{i-1}) ∧ SP_{m-i};
  if SSS_i ≢ false then
    b_i ← b;
    if i = m then
      terminate(success);
    else
      prefix-stage2(i + 1);
```

Theorem 8.4. *If invoked after a successful* Procedure 8.3*,* Procedure 8.4 *terminates successfully.*

Proof. At every iteration (recursive call) of Procedure 8.4, the choice of elementary block b_i together with (8.3) ensure success without any need to backtrack. □

After Procedure 8.3 is successful, invocation of Procedure 8.4 will terminate successfully, with $SSS_0, ..., SSS_m$ and $b_1, ..., b_m$ representing a provisional prefix. The provisional symbolic states must be narrowed as follows:

$$\text{for } i \text{ from } m \text{ down to } 1 \text{ do } SSS_{i-1} \leftarrow SSS_{i-1} \wedge r'_{b_i}(SSS_i).$$

After narrowing the symbolic states for the prefix, we have $I_\pi \leftarrow SSS_0$ and $p_\pi \leftarrow \langle b_1, ..., b_m \rangle$, and SS_0 must be adjusted: $SS_0 \leftarrow SSS_m$, and the symbolic states in the cycle further narrowed. It is possible, although unlikely in practice, that the resulting SS_0 is not equivalent to the resulting SS_n, just like in the null prefix case. The treatment for when $SS_0 \not\equiv SS_n$ is similar to the null prefix case, but with $SS_0 \leftarrow SSS_m$ instead of $SS_0 \leftarrow I_\pi$.

Although it is possible to get variations of directed counterexample search that guarantee the production of a counterexample path satisfying the constraints if one exists, it may be preferable to not provide the guarantee and use a fixpoint approach to verify the absence of counterexample paths. For example, to ensure that there are no counterexample paths satisfying the global constraint gc, then the fixpoint computation

$$\nu Z. \bigwedge_{c \in C_\varphi} gc \wedge r'_B(\mu Y.(gc \wedge Z \wedge c) \vee gc \wedge r'_B(Y))$$

can be performed with Z initialised to F_φ. The intersection of the result with S_0 and $S_\varphi(\varphi)$ can be checked for emptiness. Note that it is not sufficient to check only $S_0 \wedge S_\varphi(\varphi) \wedge F_\varphi \wedge gc$.

8.5 Using Directed Counterexample Generation

The example in Section 8.1 illustrates a case where directed counterexample path generation can be useful. Although the model in the example is a simplified model, it illustrates concepts that can be used in larger and more realistic models.

The global constraint gc in directed counterexample path generation provides a simple and efficient way of ruling out certain parts of the model from counterexamples. One systematic way of finding multiple counterexamples of interest is as follows:

1. Let the model checker find the first counterexample.

2. Based on the counterexample found, determine certain parts of the model to rule out from subsequent counterexamples. Characterise those parts as a global constraint gc_1.

3. Set $gc \leftarrow \neg gc_1$, $i \leftarrow 2$.

4. Let directed counterexample path generation find a counterexample with the global constraint gc.

5. If a counterexample path is found, determine more parts of the model to rule out from subsequent counterexamples. Characterise the parts as a global constraint gc_i, set $gc \leftarrow gc \wedge \neg gc_i$, $i \leftarrow i + 1$ and repeat from step 4.

6. Otherwise use the fixpoint computation as in Section 8.4 with global constraint gc to verify that there are no more counterexamples.

If step 2 can be automated, then the generation of multiple counterexamples can be automated. Whether step 2 can be automated depends on the problem domain. An example application where step 2 can be automated is described in Section 9.5

In Section 8.1 we saw how cycle constraints can be used to rule in certain parts of the model in the cycle parts of counterexample paths. Setting cc to some value other than $true$ may be useful even if we are not interested in the cycle part of a counterexample path. For example, if we know that the system has a main loop, then specifying cc to correspond to a point in the main loop may speed up the search for a counterexample path. For BT models, an obvious candidate for the point is a reversion BT node. Thus, suppose elementary block b corresponds to the reversion BT node, $PC(b) = PC_i$ and $PCval(b) = val_i$; then we would specify cc to be $PC_i = val_i$.

8.6 Summary

In this chapter, a novel technique for directed counterexample path generation has been presented. The mechanism allows a search for a counterexample path to be directed away from certain parts of the model and/or towards certain parts of the model. The mechanism is intended to support a controlled and incremental generation of multiple counterexample paths.

Chapter 9

Experiments with a Prototype

As a proof of concept, a prototype LTL model checker that incorporates the ideas proposed in this thesis has been developed. A brief description of the prototype appears in Section 9.1. Experiments were performed with the prototype for the following purposes:

- To demonstrate the utility of directed counterexample path generation in ruling in or ruling out certain classes of counterexamples.
- To confirm that using directed counterexample path generation in an incremental approach to generating multiple counterexamples is more efficient than the approach whereby the LTL specification is modified and the model checker is rerun.
- To examine the effect of choice of reachability strategy on the performance (execution time) of directed counterexample path generation.
- To assess the viability of mixing and matching directed counterexample path generation with other techniques, in particular with the symbolic on-the-fly LTL model checking technique described in Section 7.2.3.

The utility of directed counterexample path generation is partially demonstrated by the example in Section 8.1. However, the example is small. Larger examples were tried to assess how directed counterexample path generation scales up. We hypothesise that directed counterexample path generation scales up as well as model checking using the fixpoint approach. These examples are described in Sections 9.2 and 9.3.

Sections 9.2 and 9.3 also provide timing comparisons between the incremental approach using directed counterexample path generation versus the approach whereby the LTL specification is modified and the model checker is rerun. A favourable timing comparison strengthens the case for the directed counterexample path generation approach. This is in addition to other advantages of the directed counterexample path generation

approach, including being less prone to error because of its simplicity when compared to modifying the LTL specification.

The effects of different reachability strategies on directed counterexample path generation are examined in Sections 9.2 and 9.3. The main decisions with respect to reachability strategy in LTL model checking using the fixpoint approach and the subsequent counterexample path generation are:

- Whether to compute the set of reachable states and use the set to directly compute the set of reachable fair states or simply compute the set of fair states F_φ. (Recall from Section 7.2.2 that the strategy whereby the set of reachable states is computed and used to compute the set of reachable fair states is called the *eager reachability strategy*, and the strategy whereby the set F_φ of fair states is computed without computing the set of reachable states first is called the *lazy reachability strategy*.)

- Whether or not to compute the set $F_{C\varphi}$ of fair states reachable from $F_\varphi \wedge S_\varphi(\varphi) \wedge S_0$. If $F_{C\varphi}$ is computed, then directed counterexample path generation initialises SB_0 as follows: $SB_0 \leftarrow gc \wedge cc \wedge F_{C\varphi}$, where gc represents a global constraint and cc represents a cycle constraint (see Section 8.2 for the roles of SB_0, gc and cc in directed counterexample path generation). (Recall from Section 7.2.2 that the strategy whereby $F_{C\varphi}$ is computed is called the *eager counterexample strategy*, and the strategy whereby $F_{C\varphi}$ is not computed is called the *lazy counterexample strategy*.)

The symbolic on-the-fly LTL model checking technique described in Section 7.2.3 can sometimes find a counterexample path much faster than the fixpoint approach, but the counterexample path found tends to have a much longer prefix and cycle than necessary. Experiments were conducted to assess the viability of using directed counterexample path generation as a post-processor to improve the counterexample path found by the on-the-fly technique. These experiments are described in Section 9.4.

9.1 The Prototype

The prototype is written in ANSI Common Lisp and includes its own ordered binary decision diagram (OBDD) package with the capability to output formulas directly from OBDDs in disjunctive normal form or conjunctive normal form using the Minato-Morreale algorithm [Min93, Mor70].

The prototype includes a translator from a substantial subset of BT to elementary blocks as described in Chapter 4. The prototype allows the BT translation to either prioritise system transitions over external events or not prioritise at all.

Directed counterexample path generation has been implemented in the prototype with an interface that allows the analyst to specify the entrance of a BT node to become part of the cycle constraint cc. For the example in Figure 8.1, the analyst can simply specify BT node 18, and the interface automatically adds $PC6 = 3$ as a conjunction to cc.

The symbolic on-the-fly LTL model checking technique described in Section 7.2.3 has been implemented in the prototype. The implementation allows a global constraint gc to be specified, in which case the search space is restricted to states satisfying gc.

All of the experiments were performed on a notebook computer with an Intel i7-3740QM processor and 16GB of RAM running 64-bit Ubuntu 14.04. The Common Lisp used was Steel Bank Common Lisp version 1.1.9.

9.2 An Example from a Case Study

Experiments were conducted using an example from a case study on model-based safety risk assessment [LWK12] for the following purposes:

- To assess how directed counterexample path generation scales up. We hypothesise that directed counterexample path generation scales up as well as model checking using the fixpoint approach.
- To confirm that an incremental approach using directed counterexample path generation is more efficient than modifying the LTL specification and rerunning the model checker.
- To examine the effect of reachability strategy on directed counterexample path generation.

The system in the case study is part of a command and control system for drop missions of a firefighting aircraft. A drop solution for a mission is calculated based on meteorological and navigational data. Meteorological data is sent from a command centre through a data link with backup through voice communication. The entire case study is available on the web at `http://itee.uq.edu.au/sse/afms/`.

The model used as an example here is the model at the second stage of the risk assessment process (the model is called Model 2 on the web). The BT model has 79 BT nodes, 10 threads, and 2^{60} states. The model has system transitions prioritised over external events. At this stage of the process, functional failures such as the data link going down are included in the model.

Several hazardous situations were formulated in the original case study. For the example here, we choose the hazard where a drop solution is calculated with out-of-date

meteorological data.

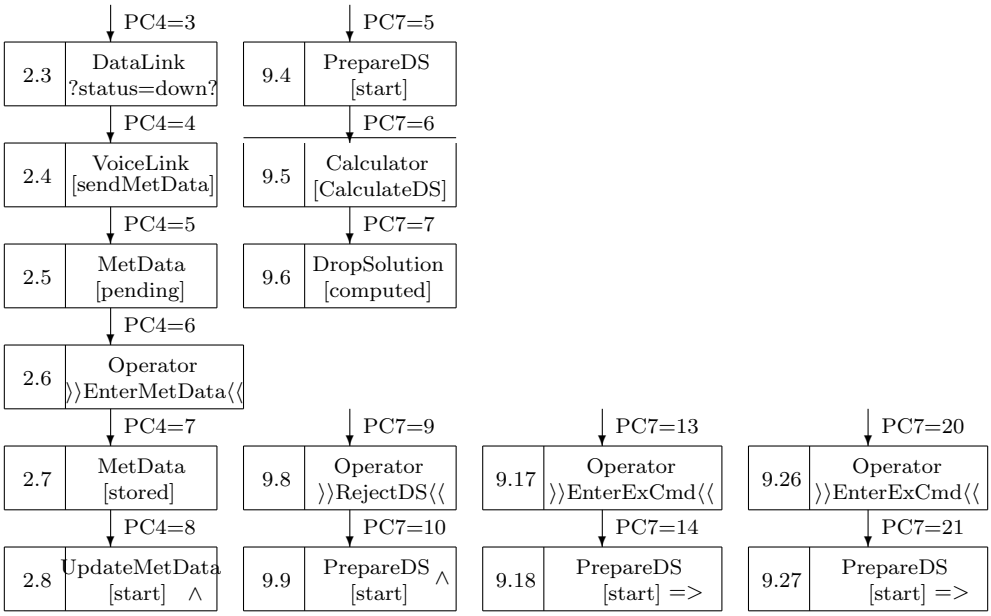

Figure 9.1: Fragments of the BT Model

The fragment at the top of the second column in Figure 9.1 is where the drop solution is calculated in the BT model (it is calculated with the execution of BT node 9.5). Out-of-date meteorological data is represented by $MetData = Pending$, thus the hazard H where a drop solution is calculated with out-of-date meteorological data can be formulated as follows:

$$\begin{aligned} H \quad \equiv \quad & \neg(Calculator = CalculateDS) \\ \wedge \quad & MetData = Pending \\ \wedge \quad & \mathbf{X}(Calculator = CalculateDS). \end{aligned}$$

The formula $\mathbf{G}(\neg H)$ states that the hazard H does not occur in any behaviour:

$$\begin{aligned} \mathbf{G}(\neg(\neg(&Calculator = CalculateDS) \\ &\wedge MetData = Pending \\ &\wedge \mathbf{X}(Calculator = CalculateDS))). \end{aligned} \tag{9.1}$$

A counterexample found in model checking (9.1) represents a behaviour in which the hazard H occurs. Different situations that lead to the hazard are represented by different counterexamples.

9.2.1 The Original Approach

In the original case study, SAL [dMOS03] was used as the model checker. The BT-to-SAL translator mentioned in Section 4.1 was used to translate the BT model into SAL. The

method for finding multiple counterexamples for $\mathbf{G}(\neg H)$ in the original case study was as follows:

1. Model check $\mathbf{G}(\neg H)$.
2. If there is a counterexample then analyse the counterexample to determine the situation that leads to the hazard.
3. Modify the LTL specification so that the situations covered by counterexamples already found do not appear as subsequent counterexamples.
4. Rerun the model checker with the modified LTL specification and repeat from step 2.

Step 2 was a manual process that required some problem domain expertise. The modification of the LTL specification in step 3 was rather ad hoc and could have been made more systematic.

The counterexample path that SAL produced when model checking (9.1) executed BT nodes 9.17 and 9.18 with $MetData = Pending$ just before BT node 9.5 is executed (execution of the counterexample path prefix to BT node 9.17, involving BT node 2.5, causes $MetData$ to be set to $Pending$). Manual examination of the counterexample path determined that this corresponds to a situation under which the hazard H occurs after the operator enters the "execute" command when meteorological data is pending (the "execute" command is indicated by an event that occurs in BT node 9.17 and BT node 9.26).

Recall from Section 4.1 that the SAL translation of a BT model assigns a Boolean SAL input variable to each BT external input event. A BT node in which an external input event occurs can only be executed if the corresponding SAL input variable has the value *true*. Let *xcmd* represent the SAL input variable for the "execute command" event. Then the LTL specification is modified to

$$(\neg\mathbf{G}(\neg((MetData = Pending) \wedge xcmd))) \vee (\mathbf{G}\neg H). \tag{9.2}$$

When given (9.2) to model check, SAL produced a counterexample path that executed BT nodes 9.3 and 9.4 with $MetData = Pending$ just before BT node 9.5 is executed. Manual examination determined that the counterexample corresponds to a situation under which the hazard H occurs when the mission is of a scheduled type (one of three types of drop missions). To prevent the situation from appearing in subsequent counterexamples, the LTL specification was modified to

$$(DMP.Type = Scheduled) \vee (\neg\mathbf{G}(\neg((MetData = Pending) \wedge xcmd))) \vee (\mathbf{G}\neg H). \tag{9.3}$$

Table 9.1: Results for SAL on Case Study Example

	run1	run2	run3	run4
time	3.8s	15.2s	25.4s	15.1s
prefix length	20	20	25	N/A
cycle length	N/A	5	5	N/A

The clause ($DMP.Type = Scheduled$) appears in (9.3) without the **G** operation because $DMP.Type$ never changes in the BT model. When given (9.3) to model check, SAL produced a counterexample path that executed BT nodes 9.8 and 9.9 with $MetData = Pending$ just before BT node 9.5 is executed. Manual examination determined that the counterexample corresponds to a situation under which the operator rejected a drop solution with meteorological data pending. Let $rdrp$ represent the SAL input variable for the event "operator rejects drop solution". The LTL specification was then modified to

$$(DMP.Type = Scheduled) \vee (\neg \mathbf{G}(\neg((MetData = Pending) \wedge (xcmd \vee rdrp)))) \vee (\mathbf{G} \neg H). \tag{9.4}$$

When given (9.4) to model check, SAL determined that there is no counterexample. Assuming SAL and the manual examinations are correct, this means all situations that can lead to hazard H have been covered.

Table 9.1 shows statistics from the SAL runs when rerun on the machine described in Section 9.1. SAL version 3.3 was used. The runs run1, run2, run3 and run4 correspond to model checking (9.1), (9.2), (9.3) and (9.4) respectively. Note that for run1, SAL produced a prefix only. This suggests that SAL recognised (9.1) as a safety property and optimised its model checking (it did not perform a "full" LTL model checking). The timing results show that modifying the LTL specification can increase the SAL running time considerably.

9.2.2 Incremental Approach using Directed Counterexample Path Generation

In the incremental approach using directed counterexample path generation, computation of the set F_φ of fair states is performed just once, and directed counterexample path generation is used multiple times to produce multiple counterexamples. Global constraints are used to direct the search for counterexample paths away from certain parts of the model.

The methodology used in the incremental approach is slightly more systematic than

the methodology used in the original case study. For each ith counterexample found, the situation under which the hazard H occurs is characterised by a state formula c_i (the situation corresponds to a class of counterexamples, and the counterexample found is a representative of the class). Note that in general a class of counterexamples cannot be characterised using a state formula. Without directed counterexample path generation, if the LTL specification had to be modified, then the modified LTL formula for the ith model checking run (where $i > 1$) can be as follows:

$$(\neg\mathbf{G}(\neg c_1 \wedge ... \wedge \neg c_{i-1})) \vee (\mathbf{G}\neg H). \tag{9.5}$$

With directed counterexample path generation, $\neg c_1 \wedge ... \wedge \neg c_{i-1}$ becomes the global constraint gc, without the LTL specification $\mathbf{G}(\neg H)$ modified. The steps in the methodology are as follows:

1. Compute the set of fair states F_φ where $\varphi \equiv \mathbf{F}(H)$.
2. Set i to 1.
3. Use directed counterexample path generation with global constraint $\neg c_1 \wedge \neg c_2 \wedge ... \wedge \neg c_{i-1}$.
4. If a counterexample path is produced, characterise the new situation under which the hazard H occurs as a state formula c_i. Repeat from step 3 after incrementing i by 1.

Let ctr$i$ denote the generation of the ith counterexample path according to the above steps. For ctr1, directed counterexample path generation produced a counterexample path that executed BT nodes 9.3 and 9.4 with $MetData = Pending$ just before BT node 9.5 is executed. This corresponds to the second counterexample found by SAL (where $DMP.Type = Scheduled$). Since $PC7 = 5$ at the entrance to BT node 9.4, we set c_1 as follows:

$$c_1 \leftarrow (MetData = pending) \wedge (PC7 = 5).$$

For ctr2, directed counterexample path generation produced a counterexample path that executed BT nodes 9.17 and 9.18 with $MetData = Pending$ just before BT node 9.5 is executed. This corresponds to the first counterexample found by SAL in which the operator enters the "execute" command while meteorological data is pending. Since the event "operator enters the execute command" only occurs in BT node 9.17 and BT node 9.26, $PC7 = 13$ at the entrance to BT node 9.17 and $PC7 = 20$ at the entrance to BT node 9.26, we set c_2 as follows:

$$c_2 \leftarrow (MetData = pending) \wedge ((PC7 = 13) \vee (PC7 = 20)).$$

For ctr3, directed counterexample path generation produced a counterexample path that executed BT nodes 9.8 and 9.9 with $MetData = Pending$ just before BT node 9.5 is executed. This corresponds to the third counterexample found by SAL in which the operator rejects a drop solution while meteorological data is pending. Since $PC7 = 10$ at the entrance to BT node 9.9, we set c_3 as follows:

$$c_3 \leftarrow (MetData = pending) \wedge (PC7 = 10).$$

No counterexample path was found by directed counterexample path generation for ctr4. A check using the fixpoint approach described in Section 8.4 was performed for ctr4 to ensure there are no more counterexamples.

Experiments were performed with different reachability strategies. Recall from Section 7.2.2 that there are two basic strategies for general reachability:

1. eager reachability strategy (ERS) whereby the set *reachable* of reachable states is computed and *reachable* is used in subsequent analysis (computing $reachable \wedge F_\varphi$ instead of F_φ, for example), and

2. lazy reachability strategy (LRS) whereby *reachable* is not computed and F_φ is directly computed.

There are also two basic strategies for reachability with respect to counterexample generation:

1. eager counterexample strategy (ECS) whereby the set $F_{C\varphi}$ of fair states reachable from $S_0 \wedge S_\varphi(\varphi)$ is computed and used as the search space for counterexample generation, and

2. lazy counterexample strategy (LCS) whereby $F_{C\varphi}$ is not computed and F_φ or $reachable \wedge F_\varphi$ is used as the search space for counterexample generation.

Four different overall reachability strategies were tried for the experiments:

1. ERS-LCS: ERS followed by LCS,

2. ERS-ECS: ERS followed by ECS,

3. LRS-LCS: LRS followed by LCS, and

4. LRS-ECS: LRS followed by ECS.

Table 9.2 shows the timing results for the four reachability strategies above. The column labelled "reach" is for computing *reachable*. The column labelled "check" is for

Table 9.2: Timing Results for the Incremental Approach

strategy	reach	F_φ	$F_{C\varphi}$	ctr1	ctr2	ctr3	ctr4	check
ERS-LCS	9.72s	1.15s	N/A	3.62s	2.82s	2.11s	1.81s	0.56s
ERS-ECS	9.73s	1.15s	7.07s	1.43s	1.05s	0.62s	0.41s	0.70s
LRS-LCS	N/A	275s	N/A	18.9s	20.5s	24.9s	14.4s	219s
LRS-ECS	N/A	275s	9.2s	1.6s	1.1s	0.6s	0.4s	220s

checking that there are no more counterexamples, using the fixpoint approach described in Section 8.4. For i from 1 to 4, the column ctr$i$ is for directed counterexample generation with global constraint $\neg c_1 \wedge \neg c_2 \wedge ... \wedge \neg c_{i-1}$. Note that for an overall strategy that uses ERS, the column F_φ is for computing $reachable \wedge F_\varphi$. Regardless of the strategy, the same set of counterexamples was found. The paths ctr1 and ctr2 each has a prefix length of 24 and a cycle length of 7, and ctr3 has a prefix length of 29 and cycle length of 7. However, in each case, the counterexample path could have been shortened by rolling part of the prefix into the cycle.

The following are some observations from Table 9.2.

- For the example, ERS is a better general reachability strategy than LRS.
- For the example, ECS improves (lowers) the running times of directed counterexample generation compared to LCS, but incurs a one-time overhead of computing $F_{C\varphi}$.
- For the example, a good reachability strategy with respect to counterexample generation (ECS) can salvage a bad basic reachability strategy (LRS).
- In each case, the run time for directed counterexample generation is lower than the run time for the fixpoint computations (*reachable* plus F_φ plus $F_{C\varphi}$).

To demonstrate the advantages of an incremental approach, experiments were also performed with the traditional approach whereby the model checker is rerun with each modified LTL specification, allowing comparisons between the approaches. The modifications to the LTL specification (9.1) were in accordance with (9.5). The counterexample paths produced using the traditional approach are the same as the corresponding counterexample paths produced using the incremental approach.

Table 9.3 shows the timing results for individual stages as well as the total for model checking and counterexample generation. The following are some observations from table 9.3.

Table 9.3: Timing Results for the Traditional Approach

strategy		ctr1	ctr2	ctr3	ctr4
ERS-LCS	*reachable*	9.78s	7.72s	6.79s	6.82s
ERS-LCS	$reachable \wedge F_{\varphi}$	1.14s	2.16s	3.77s	1.38s
ERS-LCS	ctr	3.61s	3.31s	4.4s	0s
ERS-LCS	total	14.53s	13.19s	14.96s	8.2s
ERS-ECS	*reachable*	9.75s	7.64s	6.81s	6.79s
ERS-ECS	$reachable \wedge F_{\varphi}$	1.14s	2.14s	3.77s	1.38s
ERS-ECS	$F_{C\varphi}$	7.07s	6.23s	5.63s	0s
ERS-ECS	ctr	1.42s	1.62s	1.11s	0s
ERS-ECS	total	19.38s	17.63s	17.32s	8.17s
LRS-LCS	F_{φ}	275s	416s	1037s	1127s
LRS-LCS	ctr	19s	23s	122s	0s
LRS-LCS	total	294s	439s	1159s	1127s
LRS-ECS	F_{φ}	275s	348s	1035s	1126s
LRS-ECS	$F_{C\varphi}$	9.3s	7.6s	7.4s	0s
LRS-ECS	ctr	1.6s	1.3s	1.4s	0s
LRS-ECS	total	285.9s	356.9s	1043.8s	1126s

Table 9.4: Incremental Approach vs Traditional Approach

approach	strategy	ovhd	ctr1	ctr2	ctr3	ctr4	total
incremental	ERS-LCS	10.87s	3.62s	2.82s	2.11s	12.37s	31.79s
traditional	ERS-LCS	N/A	14.53s	13.19s	14.96s	8.2s	50.88s
incremental	ERS-ECS	17.95s	1.43s	1.05s	0.62s	1.11s	22.16s
traditional	ERS-ECS	N/A	19.38s	17.63s	17.32s	8.17s	62.5s
incremental	LRS-LCS	275s	18.9s	20.5s	24.9s	234s	573.3s
traditional	LRS-LCS	N/A	294s	439s	1159s	1127s	3019s
incremental	LRS-ECS	284.2s	1.6s	1.1s	0.6s	220.4s	507.9s
traditional	LRS-ECS	N/A	285.9s	356.9s	1043.8	1126s	2812.6s

- The worst reachability strategy (LRS-LCS) applies equally bad to the original LTL formula and all modified LTL formulas.
- For the example, a good reachability strategy with respect to counterexample generation (ECS) can salvage a bad basic reachability strategy (LRS).
- The effect of a bad reachability strategy is magnified as the problem gets bigger (comparing LRS-LCS versus ERS-LCS).

A side-by-side comparison of the incremental approach versus the traditional approach is shown in Table 9.4. The column labelled "ovhd" is for the initial overhead for fixpoint computations in the incremental approach. Irrespective of the reachability strategy, the total time for the incremental approach is less than the total time for the traditional approach.

The following conclusions are reached based on the results of the experiments:

- The results of the experiments are consistent with the hypothesis that directed counterexample path generation scales up as well as model checking using the fixpoint approach. For all of the examples, the inital overhead of computing reachability and F_φ is greater than the computation time for an individual directed counterexample path generation.
- The results in Table 9.4 are consistent with the hypothesis that the incremental approach using directed counterexample path generation is more efficient than the traditional approach whereby the LTL specification is modified.
- For the example, it is better to use ERS as the strategy for general reachability rather than LRS. If LRS is used, ECS can be used to salvage the situation.

Table 9.5: Timing Results for Non-prioritised Model

strategy	reach	F_φ	$F_{C\varphi}$	ctr1	ctr2	ctr3	ctr4
ERS-ECS	756s	1999s	1282s	149s	161s	151s	152s
LRS-LCS	N/A	21.2s	N/A	3.2s	3.9s	4.4s	4.1s

Table 9.6: Results for SAL on Non-prioritised Model

	run1	run2	run3	run4
time	8.2s	124.8s	107.9s	200.3s
prefix length	17	17	17	17
cycle length	N/A	4	4	4

9.3 Example Without Prioritisation

In the example here, we modify the model from Section 9.2 by not prioritising system transitions. The directed counterexample path generation produced paths with prefix length of 21 and cycle length of 5 for ctr1, ctr2 and ctr3, and produced a path with prefix length 23 and cycle length 5 for ctr4. As with the experiments with the prioritised model, each counterexample path could have been shortened by rolling part of the prefix into the cycle.

Two reachability strategies were tried for the example: ERS-ECS and LRS-LCS. In contrast to the prioritised model, here LRS-LCS is a much better strategy than ERS-ECS. The timing results are shown in Table 9.5. An implementation of Algorithm 5.1 was used for computing *reachable* and $F_{C\varphi}$. When an implementation of a naive fixpoint computation algorithm was used instead, the computation of *reachable* took 1806s (versus 756s) and the computation of $F_{C\varphi}$ took 2902s (versus 1282s).

Results for SAL are included in Table 9.6 for comparison. Side by side comparison of the incremental approach using directed counterexample path generation versus the traditional approach of model checking modified LTL specifications as per (9.5) appears in Table 9.7, with the LRS-LCS reachability strategy. The traditional approach with the modified LTL specifications produced the same set of counterexample paths as the incremental approach using directed counterexample path generation.

Table 9.7: Incremental Approach vs Traditional Approach for Non-prioritised Model

approach	strategy	F_φ	ctr1	ctr2	ctr3	ctr4	total
incremental	LRS-LCS	21.2s	3.2s	3.9s	4.4s	4.1s	36.8s
traditional	LRS-LCS	N/A	24.2s	19.8s	126.1s	243.2s	413.3s

Table 9.8: Results for Symbolic On-the-fly LTL Model Checking

	ctr1	ctr2	ctr3	ctr4
time	0.200s	0.223s	0.109s	∞
prefix	144	144	144	N/A
cycle	14	14	14	N/A

Based on the results of the experiments, the following conclusions are reached:

- The results in Table 9.5 are consistent with the hypothesis that directed counterexample path generation scales up as well as model checking using the fixpoint approach. In each case, the time required for directed counterexample path generation is much less than the time required to compute F_φ.
- The side by side comparison in Table 9.7 supports the hypothesis that the incremental approach using directed counterexample path generation is more efficient than the approach where the LTL specification is modified.
- In contrast to the prioritised model, ERS is a bad strategy for the non-prioritised model.
- Algorithm 5.1 scales up better than a naive fixpoint computation algorithm.

9.4 Combination with On-the-fly Approach

Experiments were conducted using the model from Section 9.2 to assess the viability of using directed counterexample path generation as a post-processor to improve counterexample paths found by symbolic on-the-fly LTL model checking described in Section 7.2.3.

First, an experiment was performed using symbolic on-the-fly LTL model checking with global constraints to find multiple counterexamples, emulating the methodology in Section 9.2. The purpose of the experiment is to assess symbolic on-the-fly LTL model checking in terms of computation time and quality of counterexample paths found.

The second experiment was to use directed counterexample path generation as a post processor for improving a counterexample path found by the symbolic on-the-fly LTL model checker. For each counterexample path found by the on-the-fly LTL model checker, the starting symbolic state for the cycle part of the counterexample path is used as SB_0 in directed counterexample path generation (see Section 8.2).

Table 9.8 shows the results from symbolic on-the-fly LTL model checking with global constraints. The timing results show that symbolic on-the-fly LTL model checking can

Table 9.9: Combined On-the-fly and Directed Counterexample Generation

	reach	ctr1	ctr2	ctr3	ctr4
time	11.43s	2.48s	2.14s	1.56s	0.72s
prefix	N/A	33	33	38	N/A
cycle	N/A	7	7	7	N/A

find a counterexample path quickly if there is one. However, the quality of the counterexample paths are not very good as both the prefixes and the cycles are too long. This is caused by the depth-first nature of the search plus the imprecise tracking of fairness constraints. The on-the-fly LTL checker also had trouble with ctr4, which has no counterexample. The checking was aborted after 24 hours. Perhaps a technique such as partial order reduction [PWW96] can help, but it would need to be adjusted to handle the **X** operator. Partial order reduction would also help for a non-prioritised model.

Table 9.9 shows the result of experiments with using directed counterexample path generation as a post-processor to symbolic on-the-fly LTL model checking (the ctr4 timing is for verifying that there are no more counterexamples). The counterexample paths were improved by the post-processing (the cycle lengths were reduced to 7 from 14, and the prefix lengths were reduced to 33 and 38 from 144). However, the quality of the counterexample paths are not as good as using directed counterexample path generation directly using F_φ as the search space. Perhaps the SB_0 determined using on-the-fly LTL checking is less than optimal.

9.5 Automated Multiple Counterexample Generation

To demonstrate the possibility of automating the generation of multiple counterexamples for specific problem domains, experiments were performed based on a case study in generating minimal cut sets in safety analysis [LWY10, LYW12]. The case study involves the hydraulic system in an Airbus A320. There are three redundant hydraulic subsystems involving eleven components that can fail (and assumed to fail independently of each other). The three hydraulic subsystems are called yellow, green and blue. The eleven components are:

1. the yellow distribution line (disty),
2. the green distribution line (distg),
3. the blue distribution line (distb),
4. engine 1 (E1),

5. engine 2 (E2),
6. power transfer unit (PTU)
7. engine-driven pump for the yellow subsystem (EDPy),
8. engine-driven pump for the green subsystem (EDPg),
9. electric motor pump for the blue subsystem (EMPb),
10. electric motor pump for the yellow subsystem (EMPy), and
11. a ram air turbine pump (RAT).

For a complete description of the model, see [LWY10] or [LYW12].

The model for the hydraulic system includes some behaviour so that, for example, RAT does not become active unless both engines are off and the aircraft is flying at a speed of greater then 100 knots. The failure status of a component is effectively represented by a Boolean state variable, for example the Boolean state variable $distyFailed$ represents the failure status of disty. (Technically, $distyFailed$ is of an enumerated type $(true, false)$, and the possible values of the failure status are represented by the equalities $distyFailed = true$ and $distyFailed = false$.)

For our purposes, a minimal cut set is a subset of

$$\{distyFailed, distgFailed, distbFailed, E1Failed, E2Failed, PTUFailed,\\ EDPyFailed, EDPgFailed, EMPbFailed, EMPyFailed, RATFailed\}$$

which "violates" some specification and a removal of any element from the subset will result in a non-violation. For example, $\{distyFailed, distgFailed\}$ would be a minimal cut set if

- there is a behaviour involving a state in which disty failed and distg failed that is a counterexample to the specification, and
- there is no counterexample where all the other components do not fail and at least one of disty or distg does not fail.

The experiments used the following specification:

$$\mathbf{G}((\text{initialised and all three active}) \Rightarrow \mathbf{G}(\text{at least two active})) \tag{9.6}$$

where "at least two active" is equivalent to

- yellow subsystem is active and green subsystem is active, or

- yellow subsystem is active and blue subsystem is active, or
- green subsystem is active and blue subsystem is active.

In the model, once a components fails, it remains in a failed state.

To arrive at cut sets (which are potential minimal cut sets), each symbolic counterexample path produced for (9.6) is examined. Specifically, we can zoom into the symbolic state that starts the cycle part of the counterexample path and normalise the symbolic state into conjunctive normal form (irredundant product of sums). As an example, the symbolic state that starts the cycle part of the first counterexample path found by the prototype for (9.6), in conjunctive normal form, is

$$\begin{aligned}
&(PC1 = 3) \wedge (PC2 = 3) \wedge (PC3 = 2) \wedge (PC4 = 1) \wedge (PC5 = 1) \wedge (PC6 = 2) \\
&\wedge (PC16 = 0) \wedge (PC17 = 2) \wedge (PC7 = 4) \wedge (PC8 = 2) \wedge (PC18 = 4) \wedge (PC19 = 1) \\
&\wedge (PC9 = 2) \wedge (PC20 = 2) \wedge (PC21 = 1) \wedge (PC10 = 2) \wedge (PC22 = 2) \wedge (PC23 = 1) \\
&\wedge (PC11 = 1) \wedge (PC12 = 2) \wedge (PC24 = 3) \wedge (PC25 = 1) \wedge (PC13 = 2) \wedge (PC26 = 1) \\
&\wedge (PC27 = 1) \wedge (PC14 = 2) \wedge (PC28 = 0) \wedge (PC30 = 0) \wedge (PC31 = 0) \wedge (PC29 = 2) \\
&\wedge (PC15 = 2) \wedge (PC32 = 3) \wedge (PC33 = 1) \wedge (Pilot = ready) \wedge (Engine1 = active) \\
&\wedge (Engine2 = active) \wedge (yellow = off) \wedge (PTUy = active) \wedge (PTU = active) \\
&\wedge (EDPy = active) \wedge (EDPg = active) \wedge (EMPy = active) \wedge (EMPb = active) \\
&\wedge (RAT = off) \wedge distyFailed \wedge \neg distgFailed \wedge distbFailed \wedge \neg E1Failed \\
&\wedge \neg E2Failed \wedge \neg PTUFailed \wedge \neg EDPyFailed \wedge \neg EDPgFailed \wedge \neg EMPbFailed \\
&\wedge \neg EMPyFailed \wedge \neg RATFailed \wedge (green = active) \wedge (blue = off) \\
&\wedge (Aircraft = flyingSlow) \wedge (System = operating).
\end{aligned}$$

(In the prototype, the symbolic state is represented by a Lisp s-expression which may be viewed as an abstract syntax representation.)

We can collect conjuncts that represent failed components from the conjunctive normal form. In the above example, we get $\{distyFailed, distbFailed\}$, and this becomes a cut set and therefore a potential minimal cut set. To verify that $\{distyFailed, distbFailed\}$ is in fact a minimal cut set, we must show that there is no counterexample for (9.6) in which all the other components do not fail and at least one of disty or distb does not fail. This can be performed by model checking (9.6) with the global constraint

$$\begin{aligned}
&\neg distgFailed \wedge \neg E1Failed \wedge \neg E2Failed \wedge \neg PTUFailed \wedge \neg EDPyFailed \\
&\wedge \neg EDPgFailed \wedge \neg EMPbFailed \wedge \neg EMPyFailed \wedge \neg RATFailed \\
&\wedge (\neg distyFailed \vee \neg distbFailed).
\end{aligned} \tag{9.7}$$

The prototype was able to verify that (9.6) with global constraint (9.7) has no counterexample. Had (9.6) with global constraint (9.7) produce a counterexample, then the

counterexample would have at least one less failing component (because of the global constraint) and we get a smaller cut set. The process can be repeated until a minimal cut set is found.

Once a minimal cut set is found, it can be eliminated from further consideration using the technique in Section 9.2.2. This can be repeated until we can verify that there are no more cut sets. The entire process has been automated in a script written in Lisp. Using the script, there were five 2-element minimal cut sets found:

$\{distyFailed, distbFailed\}, \{distyFailed, distgFailed\}, \{distgFailed, distbFailed\},$
$\{distyFailed, EMPbFailed\}, \{distgFailed, EMPbFailed\}.$

There were ten 3-element minimal cut sets found:

$\{distyFailed, PTUFailed, EDPgFailed\}, \{distbFailed, PTUFailed, EDPgFailed\},$
$\{distyFailed, E1Failed, PTUFailed\}, \{distbFailed, E1Failed, PTUFailed\},$
$\{PTUFailed, EDPgFailed, EMPbFailed\}, \{E1Failed, PTUFailed, EMPbFailed\},$
$\{EDPyFailed, EDPgFailed, EMPyFailed\}, \{E1Failed, EDPyFailed, EMPyFailed\},$
$\{E2Failed, EDPgFailed, EMPyFailed\}, \{E1Failed, E2Failed, EMPyFailed\}.$

There were six 4-element minimal cut sets found:

$\{distbFailed, PTUFailed, EDPyFailed, EMPyFailed\},$
$\{distgFailed, PTUFailed, EDPyFailed, EMPyFailed\},$
$\{distbFailed, E2Failed, PTUFailed, EMPyFailed\},$
$\{distgFailed, E2Failed, PTUFailed, EMPyFailed\},$
$\{PTUFailed, EDPyFailed, EMPbFailed, EMPyFailed\},$
$\{E2Failed, PTUFailed, EMPbFailed, EMPyFailed\}.$

The script was able to verify that there are no more cut sets. During the run, it produced 3 cut sets that are non-minimal before all the minimal cut sets were found. The running time for the script was around 65 minutes.

When the script was modified to use the combination approach of Section 9.4 instead of directed counterexample path generation, it performed worse with a running time of almost 4 hours. Although counterexamples were found faster, many were of poor quality in that they were unnecessarily long involving too many component failures (caused by the depth-first nature of the on-the-fly LTL model checking). This resulted in the script producing 58 cut sets that were non-minimal before all minimal cut sets were found. Thus, although the individual cut set generation was faster, the total run time was longer.

9.6 Summary and Discussion

Experiments have been performed with a prototype to demonstrate the utility of the directed counterexample path generation. Results from the experiments support the hypothesis that directed counterexample path generation scales up as well as the fixpoint approach of model checking, i.e., if the computation of F_φ is feasible, then directed counterexample path generation is feasible.

Results from the experiments also support the hypothesis that the incremental approach of multiple counterexample generation using the directed search is more efficient than the traditional approach of modifying the LTL specification and rerunning the model checker.

Various reachability strategies were tried in the experiments. The experiments showed that the choice of strategy for reachability can have a dramatic effect on the model checking and counterexample path generation times. In general, if the BT model uses prioritisation, it is better to use an eager reachability strategy (ERS), and if the BT model does not use prioritisation, it is better to use a lazy reachability strategy (LRS).

With respect to OBDD ordering, the prototype simply uses the *dynamic weight assignment* method of Minato et al [MIY90] with guards and post conditions viewed as forming a circuit. Although OBDD ordering is important and can have a dramatic effect on model checking time, it is beyond the scope of this thesis.

The OBDD package is not a state-of-the-art OBDD package (it only has about half the features of a state-of-the-art OBDD package). The OBDD package is used because of its ease of integration into the prototype. The purpose of the prototype is not to build a faster model checker but to demonstrate potential advantages of directed counterexample path generation.

Experiments on combining symbolic on-the-fly LTL model checking with directed counterexample path generation produced some mixed results. Although the simple combination improved the quality of counterexample paths from those produced by the on-the-fly checking, the counterexample paths produced are not as good as those produced with directed counterexample path generation directly using F_φ as the search space. Much work is required to produce an effective combination of on-the-fly checking and directed counterexample path generation.

Experiments with a case study in cut set analysis demonstrated that the techniques developed in this thesis can be used to automatically and efficiently generate "meaningful" multiple counterexamples in a specific domain. The experiments also demonstrated the poor quality of counterexamples found using a depth-first search.

Chapter 10

Conclusion

10.1 Answers to Research Questions

This thesis has answered the research questions posed in Section 1.2 as follows.

- Research Question 1.
  - Question: Can the search for an LTL counterexample path be directed away from certain parts of a model or towards certain parts of a model?
  - Answer: Affirmative. *Global constraints* can be used to direct the search away from certain parts of the model, while *cycle constraints* can be used to direct the search towards certain parts of the model.
- Research Question 2.
  - Question: Can a method for such a directed counterexample path generation be made independent of the modelling notation?
  - Answer: Affirmative. This is accomplished by having the directed counterexample path generation operate in a general framework that is independent of the modelling notation.
- Research Question 3.
  - Question: Can a method for directed counterexample path generation be incorporated into a symbolic framework?
  - Answer: Affirmative. The general framework in which the directed counterexample path generation operates is a symbolic framework.
- Research Question 4.

  - Question: How do model checking strategies, such as when reachability is determined, affect directed counterexample path generation?
  - Answer: For reachability strategies, a model with a "sparse" transition relation (where the ratio between the number of transitions and the number of states is relatively small) seems to be better handled using an eager strategy such as ERS, while a model with a "dense" transition relation (where the ratio between the number of transitions and the number of states is relatively large) seems to be better handled using a lazy strategy such as LRS.

- Research Question 5.
  - Question: Can directed counterexample path generation be mixed and matched with existing techniques?
  - Answer: Affirmative. This was demonstrated by combining symbolic on-the-fly LTL model checking with directed counterexample path generation.

10.2 Contribution

The main contribution of this thesis is a technique for directed counterexample path generation in a symbolic framework for LTL model checking. Directed counterexample path generation can be used to control the generation of multiple counterexample paths, which is important in many applications of model checking, including safety analysis and test-case generation. It provides an alternative to a blind search for multiple counterexamples and to an approach in which either the LTL specification or the model is modified to rule out certain classes of counterexamples. With directed counterexample path generation, cycle and global constraints are used to rule in or rule out certain classes of counterexamples.

Directed counterexample path generation provides a mechanism to explore the counterexample space. Symbolic model checking using the fixpoint approach need only be performed once. Thereafter, either the set of counterexample states (states that can occur in counterexamples) or the set of fair states can be used as a counterexample space which can be explored using directed counterexample path generation. Multiple counterexamples can be generated without the need to perform a fixpoint computation each time a counterexample path is to be generated. Results of experiments with a prototype show that this incremental approach to generating multiple counterexamples can be more efficient than modifying the LTL specification and rerunning the model checker for each subsequent counterexample after the first counterexample.

Another contribution of this thesis is the symbolic framework for LTL model checking, consisting of a state-machine level and an LTL-encoding level. The choice of a symbolic setting reflects the author's view that symbolic techniques scale up better than techniques that operate on individual states and transitions. Also, transformations on symbolic expressions are easier to perform (e.g., using ordered binary decision diagrams) and prove correct than transformations on graph structures.

The state-machine level of the framework is essentially a Kripke structure with additional structures representing objects in the modelling notation in the form of elementary blocks. Elementary blocks allow analysis to take advantage of structural information from the modelling notation and simplifies the mapping of the result of analysis back to the modelling notation. The use of guarded updates in describing an elementary block lends to simple implementations of image and pre-image functions — under the transition relation for the block — without the need for relational products.

Essential concepts in LTL model checking are incorporated in the LTL-encoding level of the framework. Using the concepts of *path commitments* and *transition constraints*, the LTL-encoding level of the framework accommodates various LTL encoding schemes including the classic LTL encoding scheme and the Transition-based Generalised Büchi Automaton (TGBA) encoding scheme. Proof plans for showing the soundness of an encoding scheme within the framework are included. The use of the proof plans were demonstrated on the classic LTL encoding scheme and the TGBA encoding scheme.

As well as accommodating various LTL encoding schemes, the symbolic framework is neutral with respect to the model checking approach and strategy. In addition to the traditional fixpoint approach to symbolic model checking, an on-the-fly approach to LTL model checking, usually associated with an explicit automata-based approach, can be used within the symbolic framework, as evidenced by the symbolic on-the-fly LTL model checking algorithm proposed in the thesis. By accommodating different approaches and strategies within a single framework, the approaches and strategies can be mixed and matched. An example of mixing and matching was the experiment on combining symbolic on-the-fly model checking with directed counterexample path generation.

Being able to use different LTL encoding schemes, different model checking approaches and different model checking strategies within a single framework is important. This is because a good choice of LTL encoding scheme, model checking approach or model checking strategy for one problem can be a bad choice for a different problem. Experiments with the prototype demonstrated this with respect to the strategy on reachability.

10.3 Possible Future Work

Although the utility of directed counterexample path generation has been demonstrated, much work remains to make it even more effective. In particular, we have not investigated the effect of OBDD ordering on directed counterexample path generation. Perhaps using some ordering criteria can improve the performance of directed counterexample path generation.

So far, directed counterexample path generation has been used with symbolic model checking using the fixpoint approach. However, the algorithm does not assume that the search space is a subset of the set of fair states. More investigation is needed to determine whether directed counterexample path generation can be effective without restricting the search space to be a subset of the set of fair states.

Another possibility is to combine directed counterexample path generation with the counterexample generation technique of Clarke et al [CGMZ95]. In principle, global constraints can be used with the technique of Clarke et al. It remains to be seen whether the cycle constraint aspect of directed counterexample generation can be combined with the search strategy used in the technique of Clarke et al to produce a more efficient directed search.

The on-the-fly symbolic LTL model checking algorithm proposed in the thesis is a naive algorithm that does not perform well if there is no counterexample path. Incorporating a technique such as partial order reduction [PWW96] into the algorithm to improve its performance is a possibility. To make partial order reduction have wider applicability, it may need to be adjusted to be able to handle the **X** operator.

Finally, an approach similar to bounded model checking [BCCZ99] is a possibility with on-the-fly symbolic LTL model checking by placing a depth limit on the depth-first search. A variation on this would be to iteratively increase the depth limit (sometimes called *iterative deepening*).

Appendix A

Soundness Proofs

A.1 Soundness of the Classic Encoding Scheme

We now apply the proof plans from Section 6.3 to the classic LTL encoding scheme.

A.1.1 Proof of Theorem 6.2

To complete the proof of Theorem 6.2, we must prove Lemma 6.1:

$$\begin{aligned} & M, \pi \models \varphi \\ \Rightarrow \quad & \exists \pi' : proj(\pi') = \pi \wedge \forall i \geq 0, p \in cl(\varphi) : (M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models S_\varphi(p)). \end{aligned}$$

Proof. The first step according to the plan from Section 6.3.1 is to show that π' as a sequence of states constructed according to (6.20) satisfies (6.23):

$$\forall i \geq 0, p \in cl(\varphi) : (M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models S_\varphi(p)).$$

This corresponds to proving (A.1).

$$\begin{aligned} & (M, \pi \models \varphi) \wedge proj(\pi') = \pi \\ \wedge \quad & \forall i \geq 0, p \in el(\varphi) : (M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models V_p) \\ \Rightarrow \quad & \forall i \geq 0, p \in cl(\varphi) : (M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models S_\varphi(p)). \end{aligned} \tag{A.1}$$

Note that at this point, π' is not necessarily a path in M'. The proof is by induction on the structure of $p \in cl(\varphi)$. The cases are as follows:

- Case $p \in AP(\varphi)$:

$$(M, \pi^i \models p) \Leftrightarrow (M, \pi_i \models p) \Leftrightarrow (M', \pi'_i \models p) \Leftrightarrow (M', \pi'_i \models S_\varphi(p)).$$

- Case $p \equiv \neg q$, using the induction hypothesis, we have

$$(M, \pi^i \models \neg q) \Leftrightarrow (M, \pi^i \not\models q) \Leftrightarrow (M', \pi'_i \not\models S_\varphi(q)) \Leftrightarrow (M', \pi'_i \models S_\varphi(\neg q)).$$

- Case $p \equiv q \wedge r$, using the induction hypothesis, we have

$$\begin{aligned} & (M, \pi^i \models q \wedge r) \\ \Leftrightarrow \quad & (M, \pi^i \models q) \wedge (M, \pi^i \models r) \\ \Leftrightarrow \quad & (M', \pi'_i \models S_\varphi(q)) \wedge (M', \pi'_i \models S_\varphi(r)) \\ \Leftrightarrow \quad & (M', \pi'_i \models S_\varphi(q \wedge r)). \end{aligned}$$

- Case $p \equiv q \vee r$, using the induction hypothesis, we have

$$\begin{aligned} & (M, \pi^i \models q \vee r) \\ \Leftrightarrow \quad & (M, \pi^i \models q) \vee (M, \pi^i \models r) \\ \Leftrightarrow \quad & (M', \pi'_i \models S_\varphi(q)) \vee (M', \pi'_i \models S_\varphi(r)) \\ \Leftrightarrow \quad & (M', \pi'_i \models S_\varphi(q \vee r)). \end{aligned}$$

- Case $p \equiv \mathbf{X}q$ $(p \in el(\varphi))$:

$$(M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models V_p) \Leftrightarrow (M, \pi'_i \models S_\varphi(p)).$$

- Case $p \equiv q \,\mathbf{U}\, r$, using the induction hypothesis (note that $\mathbf{X}(q \,\mathbf{U}\, r) \in el(\varphi)$):

$$\begin{aligned} & (M, \pi^i \models q \,\mathbf{U}\, r) \\ \Leftrightarrow \quad & (M, \pi^i \models r) \vee (M, \pi^i \models q) \wedge (M, \pi^i \models \mathbf{X}(q \,\mathbf{U}\, r)) \\ \Leftrightarrow \quad & (M', \pi'_i \models S_\varphi(r)) \vee (M', \pi'_i \models S_\varphi(q)) \wedge (M', \pi'_i \models S_\varphi(\mathbf{X}(q \,\mathbf{U}\, r))) \\ \Leftrightarrow \quad & (M', \pi'_i \models S_\varphi(q \,\mathbf{U}\, r)). \end{aligned}$$

The second step in the proof plan is, given that π' satisfies (6.23), show that π' satisfies (6.22). This corresponds to proving (A.2).

$$\begin{aligned} & (M, \pi \models \varphi) \wedge proj(\pi') = \pi \\ \wedge \quad & \forall i \geq 0, p \in el(\varphi) : (M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models V_p) \\ \Rightarrow \quad & \forall i \geq 0 : (\pi'_i, \pi'_{i+1}) \in R'. \end{aligned} \tag{A.2}$$

The mapping from π to π' where each π'_i satisfies (6.20) is a "strict mapping". As a result, the transitions in π' ought to satisfy R' as defined by (6.9) (which corresponds to a strict encoding), regardless of whether R' is defined by (6.9) or (6.10). Thus it is sufficient to prove that the transitions satisfy R' as defined by (6.9), even if R' is defined by (6.10).

Because π is a path in M, we have $\forall i \geq 0 : (proj(\pi'_i), proj(\pi'_{i+1})) \in R$. Since (A.1) has been proved, what remains is to prove

$$\begin{aligned} & (M, \pi \models \varphi) \wedge proj(\pi') = \pi \\ \wedge \quad & \forall i \geq 0, p \in el(\varphi) : (M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models V_p) \\ \wedge \quad & \forall i \geq 0, p \in cl(\varphi) : (M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models S_\varphi(p)) \\ \Rightarrow \quad & \forall i \geq 0, p \in el(\varphi) : (M', \pi'_i \models S_\varphi(p)) \Leftrightarrow ((\pi'_i, \pi'_{i+1}) \models T_\varphi(p)). \end{aligned} \tag{A.3}$$

For the classic LTL encoding scheme, an elementary formula p is of the form $\mathbf{X}q$ for some q, and we have

$$\begin{aligned} & (M', \pi'_i \models S_\varphi(\mathbf{X}q)) \\ \Leftrightarrow\ & (M, \pi^i \models \mathbf{X}q) \\ \Leftrightarrow\ & (M, \pi^{i+1} \models q) \\ \Leftrightarrow\ & (M', \pi'_{i+1} \models S_\varphi(q)) \\ \Leftrightarrow\ & ((\pi'_i, \pi'_{i+1}) \models next(S_\varphi(q))) \\ \Leftrightarrow\ & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(\mathbf{X}q)), \end{aligned}$$

thus $(M', \pi'_i \models S_\varphi(p)) \Leftrightarrow ((\pi'_i, \pi'_{i+1}) \models T_\varphi(p))$, which proves (A.3), and thus (A.2) irrespective of whether (6.9) or (6.10) is used for the definition of R'. Combining (A.1) and (A.2) gives us a proof of Lemma 6.1. □

The proof of Lemma 6.1 completes the proof of Theorem 6.2 (see Section 6.3.1).

A.1.2 Proof of Theorem 6.3

To complete the proof of Theorem 6.3, we need to prove Lemmas 6.3 and 6.5. Let us first prove Lemma 6.3:

$$\begin{aligned} & (M, \pi \models \varphi) \wedge proj(\pi') = \pi \\ \wedge\ & \forall i \geq 0, p \in cl(\varphi) : (M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models S_\varphi(p)) \\ \Rightarrow\ & \forall c \in C_\varphi, i \geq 0 : \exists j \geq i : M', \pi'_j \models c. \end{aligned}$$

Proof. The first two steps in the proof plan for Lemma 6.3 from Section 6.3.2 is to expand C_φ in $\forall c \in C_\varphi, i \geq 0 : \exists j \geq i : M', \pi'_j \models c$, and transform the quantification:

$$\begin{aligned} & \forall c \in C_\varphi, i \geq 0 : \exists j \geq i : M', \pi'_j \models c \\ \Leftrightarrow\ & \forall c \in \{\neg S_\varphi(p) \vee S_\varphi(ec(p)) \mid p \in cl(\varphi) \wedge (ec(p) \not\equiv true)\}, i \geq 0 : \exists j \geq i : M', \pi'_j \models c \\ \Leftrightarrow\ & \forall p \in cl(\varphi), i \geq 0 : (ec(p) \not\equiv true) \Rightarrow \exists j \geq i : M', \pi'_j \models \neg S_\varphi(p) \vee S_\varphi(ec(p)). \end{aligned}$$

Since only p of the form $q\,\mathbf{U}\,r$ has $ec(p) \not\equiv true$, all we need to prove is

$$\begin{aligned} & (M, \pi \models \varphi) \wedge proj(\pi') = \pi \\ \wedge\ & \forall i \geq 0, p \in cl(\varphi) : (M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models S_\varphi(p)) \\ \Rightarrow\ & \forall i \geq 0 : \exists j \geq i : M', \pi'_j \models \neg S_\varphi(q\,\mathbf{U}\,r) \vee S_\varphi(r). \end{aligned} \tag{A.4}$$

Zooming in on $\forall i \geq 0 : \exists j \geq i : M', \pi'_j \models \neg S_\varphi(q\,\mathbf{U}\,r) \vee S_\varphi(r)$, suppose (A.4) does not hold, i.e., there exists $i_0 \geq 0$ such that $\forall j \geq i_0 : M', \pi'_j \not\models \neg S_\varphi(q\,\mathbf{U}\,r) \vee S_\varphi(r)$, thus we have $\forall j \geq i_0 : M', \pi'_j \models S_\varphi(q\,\mathbf{U}\,r) \wedge \neg S_\varphi(r)$. But from the last antecedent in (A.4) we have $(M', \pi'_j \models S_\varphi(q\,\mathbf{U}\,r)) \Leftrightarrow (M, \pi^j \models q\,\mathbf{U}\,r)$ and $(M', \pi'_j \models S_\varphi(r)) \Leftrightarrow (M, \pi^j \models r)$ for all $j \geq i_0$. Thus we have $M, \pi^{i_0} \models q\,\mathbf{U}\,r$ and $\forall j \geq i_0 : M, \pi^j \not\models r$, which is a contradiction, therefore (A.4) holds. The proof of Lemma 6.3 is complete. □

Next, we prove Lemma 6.5:

$$\begin{aligned} & \forall c \in C_\varphi, i \geq 0 : \exists j \geq i : M', \pi'_j \models c \\ \Rightarrow \quad & \forall i \geq 0, p \in cl(\varphi) : (M', \pi'_i \models S_\varphi(p)) \Rightarrow (M', \pi'^i \models p). \end{aligned}$$

Proof. Following the proof plan in Section 6.3.2, after expanding the definition of C_φ and transforming the quantification, we get

$$\begin{aligned} & \forall p \in cl(\varphi) : (ec(p) \not\equiv true) \Rightarrow \forall i \geq 0 : \exists j \geq i : M', \pi'_j \models \neg S_\varphi(p) \vee S_\varphi(ec(p)) \\ \Rightarrow \quad & \forall p \in cl(\varphi), i \geq 0 : (M', \pi'_i \models S_\varphi(p)) \Rightarrow (M', \pi'^i \models p). \end{aligned}$$

The final step according to the proof plan is the induction on the structure of p. For the classic LTL encoding scheme, either a strict encoding is used (R' defined by (6.9)) or φ is in NNF and a loose encoding may be used (R' defined by (6.10)). Zooming in on $(M', \pi'_i \models S_\varphi(p)) \Rightarrow (M', \pi'^i \models p)$, the cases are as follows:

- Case $p \in AP(\varphi)$: $(M', \pi'_i \models S_\varphi(p)) \Leftrightarrow (M', \pi'_i \models p) \Leftrightarrow (M', \pi'^i \models p)$.

- Case $p \equiv \neg q$, for a strict encoding, using the induction hypothesis, we have:

  $$(M', \pi'_i \models S_\varphi(\neg q)) \Leftrightarrow (M', \pi'_i \not\models S_\varphi(q)) \Rightarrow (M', \pi'^i \not\models q) \Leftrightarrow (M', \pi'^i \models \neg q),$$

  and for a loose encoding (where $q \in AP(\varphi)$), we have:

  $$(M', \pi'_i \models S_\varphi(\neg q)) \Leftrightarrow (M', \pi'_i \not\models S_\varphi(q)) \Leftrightarrow (M', \pi'^i \not\models q) \Leftrightarrow (M', \pi'^i \models \neg q).$$

- Case $p \equiv q \wedge r$, using the induction hypothesis:

  $$\begin{aligned} & (M', \pi'_i \models S_\varphi(q \wedge r)) \\ \Leftrightarrow \quad & (M', \pi'_i \models S_\varphi(q)) \wedge (M', \pi'_i \models S_\varphi(r)) \\ \Rightarrow \quad & (M', \pi'^i \models q) \wedge (M', \pi'^i \models r) \\ \Leftrightarrow \quad & (M', \pi'^i \models q \wedge r). \end{aligned}$$

- Case $p \equiv q \vee r$, using the induction hypothesis:

  $$\begin{aligned} & (M', \pi'_i \models S_\varphi(q \vee r)) \\ \Leftrightarrow \quad & (M', \pi'_i \models S_\varphi(q)) \vee (M', \pi'_i \models S_\varphi(r)) \\ \Rightarrow \quad & (M', \pi'^i \models q) \vee (M', \pi'^i \models r) \\ \Leftrightarrow \quad & (M', \pi'^i \models q \vee r). \end{aligned}$$

- Case $p \equiv \mathbf{X}q$, using the induction hypothesis and R', for a strict encoding we have:

  $$(M', \pi'_i \models S_\varphi(\mathbf{X}q)) \Leftrightarrow (M', \pi'_{i+1} \models S_\varphi(q)) \Rightarrow (M', \pi'^{i+1} \models q) \Leftrightarrow (M', \pi'^i \models \mathbf{X}q),$$

  and for a loose encoding we have:

  $$(M', \pi'_i \models S_\varphi(\mathbf{X}q)) \Rightarrow (M', \pi'_{i+1} \models S_\varphi(q)) \Rightarrow (M', \pi'^{i+1} \models q) \Leftrightarrow (M', \pi'^i \models \mathbf{X}q).$$

- Case $p \equiv q\,\mathbf{U}\,r$, the conjecture's antecedent gives us $\exists j \geq i : (M', \pi'_j \models \neg S_\varphi(q\,\mathbf{U}\,r) \vee S_\varphi(r))$. Let j_0 be the smallest j such that $j \geq i \wedge (M', \pi'_j \models \neg S_\varphi(q\,\mathbf{U}\,r) \vee S_\varphi(r))$, i.e.,

$$\begin{array}{rl} & j_0 \geq i \wedge (M', \pi'_{j_0} \models \neg S_\varphi(q\,\mathbf{U}\,r) \vee S_\varphi(r)) \\ \wedge & \forall j : i \leq j < j_0 \Rightarrow (M', \pi'_j \models S_\varphi(q\,\mathbf{U}\,r) \wedge \neg S_\varphi(r)). \end{array} \tag{A.5}$$

  If $j_0 = i$ then, using the induction hypothesis and $M', \pi'_{j_0} \models \neg S_\varphi(q\,\mathbf{U}\,r) \vee S_\varphi(r)$, we have

$$(M', \pi'_i \models S_\varphi(q\,\mathbf{U}\,r)) \Leftrightarrow (M', \pi'_i \models S_\varphi(r)) \Rightarrow (M', \pi'^i \models r) \Rightarrow (M', \pi'^i \models q\,\mathbf{U}\,r)$$

  and we are done. Otherwise, $j_0 > i$ and from (A.13) and the definition of $S_\varphi(q\,\mathbf{U}\,r)$ we get

$$(M', \pi'_i \models S_\varphi(q\,\mathbf{U}\,r)) \Rightarrow \forall j : i \leq j < j_0 \Rightarrow (M', \pi'_j \models S_\varphi(q)).$$

  Using the definitions of $S_\varphi(q\,\mathbf{U}\,r)$ and R', we have

$$\begin{array}{rl} & (M', \pi'_{j_0-1} \models S_\varphi(q\,\mathbf{U}\,r) \wedge \neg S_\varphi(r)) \\ \Rightarrow & (M', \pi'_{j_0-1} \models S_\varphi(q) \wedge S_\varphi(\mathbf{X}(q\,\mathbf{U}\,r))) \\ \Rightarrow & (M', \pi'_{j_0} \models S_\varphi(q\,\mathbf{U}\,r)), \end{array}$$

  and combined with $(M', \pi'_{j_0} \models \neg S_\varphi(q\,\mathbf{U}\,r) \vee S_\varphi(r))$, we get $(M', \pi'_{j_0} \models S_\varphi(r))$. Therefore, using the induction hypothesis, we have

$$\begin{array}{rl} & (M', \pi'_i \models S_\varphi(q\,\mathbf{U}\,r)) \\ \Rightarrow & (M', \pi'_{j_0} \models S_\varphi(r)) \wedge (\forall j : i \leq j < j_0 \Rightarrow (M', \pi'_j \models S_\varphi(q))) \\ \Rightarrow & (M', \pi'^{j_0} \models r) \wedge (\forall j : i \leq j < j_0 \Rightarrow (M', \pi'^j \models q)) \\ \Rightarrow & (M', \pi'^i \models q\,\mathbf{U}\,r). \end{array}$$

□

We have proved Theorem 6.2 and Theorem 6.3 within the classic LTL encoding scheme. The proofs handle both the case where the encoding is strict (i.e., R' is defined by (6.9)) and the case where the encoding is loose (i.e., R' is defined by (6.10) and φ is in NNF).

A.2 Soundness of the TGBA Encoding Scheme

We now apply the proof plans from Section 6.3 to the TGBA encoding scheme.

A.2.1 Proof of Theorem 6.2

To complete the proof of Theorem 6.2, we must prove Lemma 6.2:

$$\begin{aligned} & M, \pi \models \varphi \\ \Rightarrow \quad & \exists \pi' : proj(\pi') = \pi \wedge \forall i \geq 0, p \in cl(\varphi) : (M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models S_\varphi(p)) \\ & \wedge \forall i \geq 0, p \in el(\varphi) \wedge (ec(p) \not\equiv true) : (M, \pi^i \models p \wedge \neg ec(p)) \Leftrightarrow (M', \pi'_i \models P_p). \end{aligned}$$

Proof. The first step according to the plan from Section 6.3.1 is to show that π' as a sequence of states constructed according to (6.21) satisfies (6.23):

$$\forall i \geq 0, p \in cl(\varphi) : (M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models S_\varphi(p)).$$

This corresponds to proving (A.6).

$$\begin{aligned} & (M, \pi \models \varphi) \wedge proj(\pi') = \pi \\ \wedge \quad & \forall i \geq 0, p \in el(\varphi) : (M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models V_p) \\ \wedge \quad & \forall i \geq 0, p \in el(\varphi) \wedge (ec(p) \not\equiv true) : (M, \pi^i \models p \wedge \neg ec(p)) \Leftrightarrow (M', \pi'_i \models P_p) \\ \Rightarrow \quad & \forall i \geq 0, p \in cl(\varphi) : (M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models S_\varphi(p)). \end{aligned} \tag{A.6}$$

Note that at this point, π' is not necessarily a path in M'. The proof is by induction on the structure of $p \in cl(\varphi)$. The cases are as follows:

- Case $p \in AP(\varphi)$:

$$(M, \pi^i \models p) \Leftrightarrow (M, \pi_i \models p) \Leftrightarrow (M', \pi'_i \models p) \Leftrightarrow (M', \pi'_i \models S_\varphi(p)).$$

- Case $p \in el(\varphi)$: $(M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models V_p) \Leftrightarrow (M, \pi'_i \models S_\varphi(p))$.

- Case $p \equiv \neg q$, since φ is in NNF, $q \in AP(\varphi)$, and we have

$$(M, \pi^i \models \neg q) \Leftrightarrow (M, \pi^i \not\models q) \Leftrightarrow (M', \pi'_i \not\models S_\varphi(q)) \Leftrightarrow (M', \pi'_i \models S_\varphi(\neg q)).$$

- Case $p \equiv q \wedge r$, using the induction hypothesis, we have

$$\begin{aligned} & (M, \pi^i \models q \wedge r) \\ \Leftrightarrow \quad & (M, \pi^i \models q) \wedge (M, \pi^i \models r) \\ \Leftrightarrow \quad & (M', \pi'_i \models S_\varphi(q)) \wedge (M', \pi'_i \models S_\varphi(r)) \\ \Leftrightarrow \quad & (M', \pi'_i \models S_\varphi(q \wedge r)). \end{aligned}$$

- Case $p \equiv q \vee r$, using the induction hypothesis, we have

$$\begin{aligned} & (M, \pi^i \models q \vee r) \\ \Leftrightarrow \quad & (M, \pi^i \models q) \vee (M, \pi^i \models r) \\ \Leftrightarrow \quad & (M', \pi'_i \models S_\varphi(q)) \vee (M', \pi'_i \models S_\varphi(r)) \\ \Leftrightarrow \quad & (M', \pi'_i \models S_\varphi(q \vee r)). \end{aligned}$$

The second step in the proof plan is, given that π' satisfies (6.23), show that π' satisfies (6.22). This corresponds to proving (A.7).

$$\begin{array}{ll} & (M, \pi \models \varphi) \wedge proj(\pi') = \pi \\ \wedge & \forall i \geq 0, p \in el(\varphi) : (M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models V_p) \\ \wedge & \forall i \geq 0, p \in el(\varphi) \wedge (ec(p) \not\equiv true) : (M, \pi^i \models p \wedge \neg ec(p)) \Leftrightarrow (M', \pi'_i \models P_p) \\ \Rightarrow & \forall i \geq 0 : (\pi'_i, \pi'_{i+1}) \in R'. \end{array} \tag{A.7}$$

The mapping from π to π' where each π'_i satisfies (6.21) is a "strict mapping". As a result, the transitions in π' ought to satisfy R' as defined by (6.9) (which corresponds to a strict encoding), regardless of whether R' is defined by (6.9) or (6.10). Thus it is sufficient to prove that the transitions satisfy R' as defined by (6.9).

Unlike the classic LTL encoding scheme whose elementary formulas are of the form $\mathbf{X}p$, elementary formulas in the TGBA encoding scheme has various forms. In addition, a transition constraint for an elementary formula in the TGBA encoding scheme may be defined in terms of subsidiary transition constraints on formulas that may not be elementary but are in $sub(\varphi)$. To help prove (A.7), the following is proved first:

$$\begin{array}{ll} & (M, \pi \models \varphi) \wedge proj(\pi') = \pi \\ \wedge & \forall i \geq 0, p \in el(\varphi) : (M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models V_p) \\ \wedge & \forall i \geq 0, p \in el(\varphi) \wedge (ec(p) \not\equiv true) : (M, \pi^i \models p \wedge \neg ec(p)) \Leftrightarrow (M', \pi'_i \models P_p) \\ \wedge & \forall i \geq 0, p \in cl(\varphi) : (M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models S_\varphi(p)) \\ \Rightarrow & \forall i \geq 0, p \in sub(\varphi) : (M, \pi^i \models p) \Leftrightarrow ((\pi'_i, \pi'_{i+1}) \models T_\varphi(p)). \end{array} \tag{A.8}$$

The proof is by induction on the structure of $p \in sub(\varphi)$. Zooming in on $(M, \pi^i \models p) \Leftrightarrow ((\pi'_i, \pi'_{i+1}) \models T_\varphi(p))$, the cases are as follows:

- Case $p \in AP(\varphi)$: $(M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models p) \Leftrightarrow ((\pi'_i, \pi'_{i+1}) \models T_\varphi(p))$.

- Case $p \equiv \neg q$: since φ is in NNF, we have $q \in AP(\varphi)$, and we get

$$(M, \pi^i \models \neg q) \Leftrightarrow (M, \pi^i \not\models q) \Leftrightarrow ((\pi'_i, \pi'_{i+1}) \not\models T_\varphi(q)) \Leftrightarrow ((\pi'_i, \pi'_{i+1}) \models T_\varphi(\neg q)).$$

- Case $p \equiv q \wedge r$: using the induction hypothesis, we get

$$\begin{array}{ll} & (M, \pi^i \models q \wedge r) \\ \Leftrightarrow & (M, \pi^i \models q) \wedge (M, \pi^i \models r) \\ \Leftrightarrow & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(q)) \wedge ((\pi'_i, \pi'_{i+1}) \models T_\varphi(r)) \\ \Leftrightarrow & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(q \wedge r)). \end{array}$$

- Case $p \equiv q \vee r$: using the induction hypothesis, we get

$$\begin{aligned} & (M, \pi^i \models q \vee r) \\ \Leftrightarrow\ & (M, \pi^i \models q) \vee (M, \pi^i \models r) \\ \Leftrightarrow\ & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(q)) \vee ((\pi'_i, \pi'_{i+1}) \models T_\varphi(r)) \\ \Leftrightarrow\ & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(q \vee r)). \end{aligned}$$

- Case $p \equiv \mathbf{X}q$: $q \in cl(\varphi)$, we get

$$\begin{aligned} & (M, \pi^i \models \mathbf{X}q) \\ \Leftrightarrow\ & (M, \pi^{i+1} \models q) \\ \Leftrightarrow\ & (M', \pi'_{i+1} \models S_\varphi(q)) \\ \Leftrightarrow\ & ((\pi'_i, \pi'_{i+1}) \models next(S_\varphi(q))) \\ \Leftrightarrow\ & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(\mathbf{X}q)). \end{aligned}$$

- Case $p \equiv q\,\mathbf{U}\,r$: using the induction hypothesis, we get

$$\begin{aligned} & (M, \pi^i \models q\,\mathbf{U}\,r) \\ \Leftrightarrow\ & (M, \pi^i \models r \vee (q\,\mathbf{U}\,r \wedge \neg r) \wedge q \wedge \mathbf{X}(q\,\mathbf{U}\,r)) \\ \Leftrightarrow\ & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(r) \vee T_\varphi(q) \wedge P_{q\,\mathbf{U}\,r} \wedge T_\varphi(\mathbf{X}(q\,\mathbf{U}\,r))) \\ \Leftrightarrow\ & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(q\,\mathbf{U}\,r)). \end{aligned}$$

- Case $p \equiv q\,\mathbf{R}\,r$: using the induction hypothesis, we get

$$\begin{aligned} & (M, \pi^i \models q\,\mathbf{R}\,r) \\ \Leftrightarrow\ & (M, \pi^i \models r \wedge (q \vee \mathbf{X}(q\,\mathbf{R}\,r))) \\ \Leftrightarrow\ & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(r) \wedge (T_\varphi(q) \vee T_\varphi(\mathbf{X}(q\,\mathbf{R}\,r)))) \\ \Leftrightarrow\ & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(q\,\mathbf{R}\,r)). \end{aligned}$$

- Case $p \equiv \mathbf{G}q$, $q \not\equiv \mathbf{F}r$ for any r: using the induction hypothesis, we get

$$\begin{aligned} & (M, \pi^i \models \mathbf{G}q) \\ \Leftrightarrow\ & (M, \pi^i \models q \wedge \mathbf{X}(\mathbf{G}q)) \\ \Leftrightarrow\ & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(q) \wedge T_\varphi(\mathbf{X}(\mathbf{G}q))) \\ \Leftrightarrow\ & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(\mathbf{G}q)). \end{aligned}$$

- Case $p \equiv \mathbf{F}q$: using the induction hypothesis, we get

$$\begin{aligned} & (M, \pi^i \models \mathbf{F}q) \\ \Leftrightarrow\ & (M, \pi^i \models q \vee (\mathbf{F}q \wedge \neg q) \wedge \mathbf{X}(\mathbf{F}q)) \\ \Leftrightarrow\ & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(q) \vee P_{\mathbf{F}q} \wedge T_\varphi(\mathbf{X}(\mathbf{F}q))) \\ \Leftrightarrow\ & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(\mathbf{F}q)). \end{aligned}$$

- Case $p \equiv \mathbf{GF}q$: using the induction hypothesis, we get

$$\begin{array}{ll} & (M, \pi^i \models \mathbf{GF}q) \\ \Leftrightarrow & (M, \pi^i \models (q \vee (\mathbf{GF}q \wedge \neg q)) \wedge \mathbf{X}(\mathbf{GF}q)) \\ \Leftrightarrow & ((\pi'_i, \pi'_{i+1}) \models (T_\varphi(q) \vee P_{\mathbf{GF}q}) \wedge T_\varphi(\mathbf{X}(\mathbf{GF}q))) \\ \Leftrightarrow & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(\mathbf{GF}q)). \end{array}$$

The proof of (A.8) is complete. From (A.6) and (A.8), we get

$$\begin{array}{ll} & (M, \pi \models \varphi) \wedge proj(\pi') = \pi \\ \wedge & \forall i \geq 0, p \in el(\varphi) : (M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models V_p) \\ \wedge & \forall i \geq 0, p \in el(\varphi) \wedge (ec(p) \not\equiv true) : (M, \pi^i \models p \wedge \neg ec(p)) \Leftrightarrow (M', \pi'_i \models P_p) \\ \Rightarrow & \forall i \geq 0, p \in el(\varphi) : (M', \pi'_i \models V_p) \Leftrightarrow ((\pi'_i, \pi'_{i+1}) \models T_\varphi(p)), \end{array}$$

which when combined with $\forall i \geq 0 : (proj(\pi'_i), proj(\pi'_{i+1})) \in R$ (π is a path in M), proves (A.7):

$$\begin{array}{ll} & (M, \pi \models \varphi) \wedge proj(\pi') = \pi \\ \wedge & \forall i \geq 0, p \in el(\varphi) : (M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models V_p) \\ \wedge & \forall i \geq 0, p \in el(\varphi) \wedge (ec(p) \not\equiv true) : (M, \pi^i \models p \wedge \neg ec(p)) \Leftrightarrow (M', \pi'_i \models P_p) \\ \Rightarrow & \forall i \geq 0 : (\pi'_i, \pi'_{i+1}) \in R', \end{array}$$

irrespective of whether (6.9) or (6.10) is used for the definition of R'. Lemma 6.2 follows from (A.6), (A.7), and the immediate satisfaction of $\forall i \geq 0, p \in el(\varphi) \wedge (ec(p) \not\equiv true) : (M, \pi^i \models p \wedge \neg ec(p)) \Leftrightarrow (M', \pi'_i \models P_p)$ when π' is constructed according to (6.21). □

The proof of Lemma 6.2 completes the proof of Theorem 6.2 within the TGBA encoding scheme.

A.2.2 Proof of Theorem 6.3

To complete the proof of Theorem 6.3, we need to prove Lemmas 6.4 and 6.5. Let us first prove Lemma 6.4:

$$\begin{array}{ll} & (M, \pi \models \varphi) \wedge proj(\pi') = \pi \\ \wedge & \forall i \geq 0, p \in cl(\varphi) : (M, \pi^i \models p) \Leftrightarrow (M', \pi'_i \models S_\varphi(p)) \\ \wedge & \forall i \geq 0, p \in el(\varphi) \wedge (ec(p) \not\equiv true) : (M, \pi^i \models p \wedge \neg ec(p)) \Leftrightarrow (M', \pi'_i \models P_p) \\ \Rightarrow & \forall c \in C_\varphi, i \geq 0 : \exists j \geq i : M', \pi'_j \models c. \end{array}$$

Proof. Following the proof plan from Section 6.3.2, we expand C_φ in $\forall c \in C_\varphi, i \geq 0 : \exists j \geq i : M', \pi'_j \models c$ and transform the quantification:

$$\begin{array}{ll} & \forall c \in C_\varphi, i \geq 0 : \exists j \geq i : M', \pi'_j \models c \\ \Leftrightarrow & \forall c \in \{\neg v \mid v \in \{P_p \mid p \in el(\varphi) \wedge (ec(p) \not\equiv true)\}\}, i \geq 0 : \exists j \geq i : M', \pi'_j \models c \\ \Leftrightarrow & \forall p \in el(\varphi), i \geq 0 : (ec(p) \not\equiv true) \Rightarrow \exists j \geq i : M', \pi'_j \models \neg P_p. \end{array}$$

Thus we need to prove

$$\begin{array}{ll} & (M,\pi \models \varphi) \wedge proj(\pi') = \pi \\ \wedge & \forall i \geq 0, p \in cl(\varphi) : (M,\pi^i \models p) \Leftrightarrow (M',\pi'_i \models S_\varphi(p)) \\ \wedge & \forall i \geq 0, p \in el(\varphi) \wedge (ec(p) \not\equiv true) : (M,\pi^i \models p \wedge \neg ec(p)) \Leftrightarrow (M',\pi'_i \models P_p) \\ \Rightarrow & \forall p \in el(\varphi), i \geq 0 : (ec(p) \not\equiv true) \Rightarrow \exists j \geq i : M',\pi'_j \models \neg P_p. \end{array} \tag{A.9}$$

Zooming in on $\forall p \in el(\varphi), i \geq 0 : (ec(p) \not\equiv true) \Rightarrow \exists j \geq i : M',\pi'_j \models \neg P_p$, suppose (A.9) does not hold, i.e., for some $p \in el(\varphi)$ with $ec(p) \not\equiv true$, there exists $i_0 \geq 0$ such that $\forall j \geq i_0 : M',\pi'_j \models P_p$. Using the last antecedent in (A.9) we get $\forall j \geq i_0 : (M,\pi^j \models p \wedge \neg ec(p))$. Thus we have $M,\pi^{i_0} \models p$ and $\forall j \geq i_0 : M,\pi^j \not\models ec(p)$, which is a contradiction (since $M,\pi^{i_0} \models p$ implies $\exists j \geq i_0 : M,\pi^j \models ec(p)$ — see Concept 6.4), therefore (A.9) holds. This completes the proof of Lemma 6.4. □

Next, we prove Lemma 6.5:

$$\begin{array}{ll} & \forall c \in C_\varphi, i \geq 0 : \exists j \geq i : M',\pi'_j \models c \\ \Rightarrow & \forall i \geq 0, p \in cl(\varphi) : (M',\pi'_i \models S_\varphi(p)) \Rightarrow (M',\pi'^i \models p). \end{array}$$

Proof. Following the proof plan in Section 6.3.2, after expanding the definition of C_φ and transforming the quantification, we get (A.10).

$$\begin{array}{ll} & \forall p \in el(\varphi), i \geq 0 : (ec(p) \not\equiv true) \Rightarrow \exists j \geq i : M',\pi'_j \models \neg P_p \\ \Rightarrow & \forall p \in cl(\varphi), i \geq 0 : (M',\pi'_i \models S_\varphi(p)) \Rightarrow (M',\pi'^i \models p). \end{array} \tag{A.10}$$

Before we prove (A.10), we prove two things. The first is (A.11).

$$\forall p \in cl(\varphi), i \geq 0 : (M',\pi'_i \models S_\varphi(p)) \Rightarrow ((\pi'_i,\pi'_{i+1}) \models T_\varphi(p)). \tag{A.11}$$

The proof of (A.11) is by induction on the structure of $p \in cl(\varphi)$. The cases are as follows:

- Case $p \in AP(\varphi)$: $(M',\pi'_i \models S_\varphi(p)) \Leftrightarrow (M',\pi'_i \models p) \Leftrightarrow ((\pi'_i,\pi'_{i+1}) \models T_\varphi(p))$.
- Case $p \equiv \neg q$, we have $q \in AP(\varphi)$, and we get

$$\begin{array}{ll} & (M',\pi'_i \models S_\varphi(\neg q)) \\ \Leftrightarrow & (M',\pi'_i \not\models S_\varphi(q)) \\ \Leftrightarrow & ((\pi'_i,\pi'_{i+1}) \not\models T_\varphi(q)) \\ \Leftrightarrow & ((\pi'_i,\pi'_{i+1}) \models T_\varphi(\neg q)). \end{array}$$

- Case $p \equiv q \wedge r$, using the induction hypothesis:

$$\begin{array}{ll} & (M',\pi'_i \models S_\varphi(q \wedge r)) \\ \Leftrightarrow & (M',\pi'_i \models S_\varphi(q)) \wedge (M',\pi'_i \models S_\varphi(r)) \\ \Rightarrow & ((\pi'_i,\pi'_{i+1}) \models T_\varphi(q)) \wedge ((\pi'_i,\pi'_{i+1}) \models T_\varphi(r)) \\ \Leftrightarrow & ((\pi'_i,\pi'_{i+1}) \models T_\varphi(q \wedge r)). \end{array}$$

- Case $p \equiv q \vee r$, using the induction hypothesis:

$$\begin{array}{rl} & (M', \pi'_i \models S_\varphi(q \vee r)) \\ \Leftrightarrow & (M', \pi'_i \models S_\varphi(q)) \vee (M', \pi'_i \models S_\varphi(r)) \\ \Rightarrow & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(q)) \vee ((\pi'_i, \pi'_{i+1}) \models T_\varphi(r)) \\ \Leftrightarrow & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(q \vee r)). \end{array}$$

- Case $p \in el(\varphi)$, using (6.9) or (6.10) as the definition of R':

$$(M', \pi'_i \models S_\varphi(p)) \Leftrightarrow (M', \pi'_i \models V_p)) \Rightarrow ((\pi'_i, \pi'_{i+1}) \models T_\varphi(p)).$$

The second thing to prove is (A.12). This is because the TGBA encoding scheme is transition-oriented, so the proof of (A.10) involves reasoning about transition constraints on formulas in $sub(\varphi)$.

$$\begin{array}{rl} & \forall p \in el(\varphi), i \geq 0 : (ec(p) \not\equiv true) \Rightarrow \exists j \geq i : M', \pi'_j \models \neg P_p \\ \Rightarrow & \forall p \in sub(\varphi), i \geq 0 : ((\pi'_i, \pi'_{i+1}) \models T_\varphi(p)) \Rightarrow (M', \pi'^i \models p). \end{array} \tag{A.12}$$

The proof of (A.12) makes up the bulk of the proof of the soundness of the TGBA encoding scheme, and contains many intricate details. The proof of (A.12) is by induction on the structure of $p \in sub(\varphi)$. Zooming in on $((\pi'_i, \pi'_{i+1}) \models T_\varphi(p)) \Rightarrow (M', \pi'^i \models p)$, the cases are as follows:

- Case $p \in AP(\varphi)$: $((\pi'_i, \pi'_{i+1}) \models T_\varphi(p)) \Leftrightarrow (M', \pi'_i \models p) \Leftrightarrow (M', \pi'^i \models p)$.

- Case $p \equiv \neg q$, we have $q \in AP(\varphi)$, and we get

$$((\pi'_i, \pi'_{i+1}) \models T_\varphi(\neg q)) \Leftrightarrow ((\pi'_i, \pi'_{i+1}) \not\models T_\varphi(q)) \Leftrightarrow (M', \pi'^i \not\models q) \Leftrightarrow (M', \pi'^i \models \neg q).$$

- Case $p \equiv q \wedge r$, using the induction hypothesis:

$$\begin{array}{rl} & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(q \wedge r)) \\ \Leftrightarrow & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(q)) \wedge ((\pi'_i, \pi'_{i+1}) \models T_\varphi(r)) \\ \Rightarrow & (M', \pi'^i \models q) \wedge (M', \pi'^i \models r) \Leftrightarrow (M', \pi'^i \models q \wedge r). \end{array}$$

- Case $p \equiv q \vee r$, using the induction hypothesis:

$$\begin{array}{rl} & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(q \vee r)) \\ \Leftrightarrow & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(q)) \vee ((\pi'_i, \pi'_{i+1}) \models T_\varphi(r)) \\ \Rightarrow & (M', \pi'^i \models q) \vee (M', \pi'^i \models r) \Leftrightarrow (M', \pi'^i \models q \vee r). \end{array}$$

- Case $p \equiv \mathbf{X}q$, using the induction hypothesis, (6.9) or (6.10) as the definition of R', and (A.11):

$$\begin{aligned} & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(\mathbf{X}q)) \\ \Leftrightarrow\ & ((\pi'_i, \pi'_{i+1}) \models next(S_\varphi(p))) \\ \Leftrightarrow\ & (M', \pi'_{i+1} \models S_\varphi(q)) \\ \Rightarrow\ & ((\pi'_{i+1}, \pi'_{i+2}) \models T_\varphi(q)) \\ \Rightarrow\ & (M', \pi'^{i+1} \models q) \\ \Leftrightarrow\ & (M', \pi'^{i} \models \mathbf{X}q). \end{aligned}$$

- Case $p \equiv q\,\mathbf{U}\,r$: first we prove

$$\begin{aligned} & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(q\,\mathbf{U}\,r)) \\ \Rightarrow\ & \forall j \geq i : ((\pi'_j, \pi'_{j+1}) \models T_\varphi(q) \wedge P_{q\,\mathbf{U}\,r} \wedge T_\varphi(\mathbf{X}(q\,\mathbf{U}\,r))) \\ & \quad \vee \exists k : i \leq k \leq j \wedge ((\pi'_k, \pi'_{k+1}) \models T_\varphi(r)) \end{aligned} \tag{A.13}$$

  by induction on j. For $j = i$, the antecedent $((\pi'_i, \pi'_{i+1}) \models T_\varphi(q\,\mathbf{U}\,r))$ gives us

$$\begin{aligned} & ((\pi'_j, \pi'_{j+1}) \models T_\varphi(q\,\mathbf{U}\,r)) \\ \Leftrightarrow\ & ((\pi'_j, \pi'_{j+1}) \models T_\varphi(r) \vee T_\varphi(q) \wedge P_{q\,\mathbf{U}\,r} \wedge T_\varphi(\mathbf{X}(q\,\mathbf{U}\,r))) \\ \Leftrightarrow\ & ((\pi'_j, \pi'_{j+1}) \models T_\varphi(q) \wedge P_{q\,\mathbf{U}\,r} \wedge T_\varphi(\mathbf{X}(q\,\mathbf{U}\,r))) \\ & \quad \vee \exists k : i \leq k \leq j \wedge ((\pi'_k, \pi'_{k+1}) \models T_\varphi(r)). \end{aligned}$$

  For $j > i$, the induction hypothesis gives us two cases to prove. In the first case, the induction hypothesis gives us $((\pi'_{j-1}, \pi'_j) \models T_\varphi(q) \wedge P_{q\,\mathbf{U}\,r} \wedge T_\varphi(\mathbf{X}(q\,\mathbf{U}\,r)))$, thus

$$\begin{aligned} & ((\pi'_{j-1}, \pi'_j) \models T_\varphi(q) \wedge P_{q\,\mathbf{U}\,r} \wedge T_\varphi(\mathbf{X}(q\,\mathbf{U}\,r))) \\ \Rightarrow\ & ((\pi'_j, \pi'_{j+1}) \models T_\varphi(q\,\mathbf{U}\,r)) \\ \Leftrightarrow\ & ((\pi'_j, \pi'_{j+1}) \models T_\varphi(r) \vee T_\varphi(q) \wedge P_{q\,\mathbf{U}\,r} \wedge T_\varphi(\mathbf{X}(q\,\mathbf{U}\,r))) \\ \Rightarrow\ & ((\pi'_j, \pi'_{j+1}) \models T_\varphi(q) \wedge P_{q\,\mathbf{U}\,r} \wedge T_\varphi(\mathbf{X}(q\,\mathbf{U}\,r))) \\ & \quad \vee \exists k : i \leq k \leq j \wedge ((\pi'_k, \pi'_{k+1}) \models T_\varphi(r)). \end{aligned}$$

  In the second case, the induction hypothesis gives use $\exists k : i \leq k \leq j - 1 \wedge ((\pi'_k, \pi'_{k+1}) \models T_\varphi(r))$, thus

$$\begin{aligned} & \exists k : i \leq k \leq j - 1 \wedge ((\pi'_k, \pi'_{k+1}) \models T_\varphi(r)) \\ \Rightarrow\ & ((\pi'_j, \pi'_{j+1}) \models T_\varphi(q) \wedge P_{q\,\mathbf{U}\,r} \wedge T_\varphi(\mathbf{X}(q\,\mathbf{U}\,r))) \\ & \quad \vee \exists k : i \leq k \leq j \wedge ((\pi'_k, \pi'_{k+1}) \models T_\varphi(r)). \end{aligned}$$

  This completes the induction proof for (A.13).

  We now show that given $((\pi'_i, \pi'_{i+1}) \models T_\varphi(q\,\mathbf{U}\,r))$, then $\exists k \geq i : ((\pi'_k, \pi'_{k+1}) \models T_\varphi(r))$. Suppose not, then using (A.13) we get $\forall j \geq i : ((\pi'_j, \pi'_{j+1}) \models T_\varphi(q) \wedge P_{q\,\mathbf{U}\,r} \wedge T_\varphi(\mathbf{X}(q\,\mathbf{U}\,r)))$. However, the antecedent $\forall p \in el(\varphi), i \geq 0 : (ec(p) \not\equiv true) \Rightarrow \exists j \geq$

$i : M', \pi'_j \models \neg P_p$ in (A.12) gives us $\forall i \geq 0 : \exists j \geq i : M', \pi'_j \models \neg P_{q \,\mathbf{U}\, r}$, thus we get a contradiction.

Given $((\pi'_i, \pi'_{i+1}) \models T_\varphi(q \,\mathbf{U}\, r))$, let k_0 be the smallest k such that $k \geq i \wedge ((\pi'_k, \pi'_{k+1}) \models T_\varphi(r))$. From (A.13) we get $\forall j : i \leq j < k_0 \Rightarrow ((\pi'_j, \pi'_{j+1}) \models T_\varphi(q))$. In addition, we have $k_0 \geq i \wedge ((\pi'_{k_0}, \pi'_{k_0+1}) \models T_\varphi(r))$. Thus we have

$$\begin{array}{ll} & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(q \,\mathbf{U}\, r)) \\ \Rightarrow & ((\pi'_{k_0}, \pi'_{k_0+1}) \models T_\varphi(r)) \wedge \forall j : i \leq j < k_0 \Rightarrow ((\pi'_j, \pi'_{j+1}) \models T_\varphi(q)) \\ \Rightarrow & ((M', \pi'^{k_0} \models r) \wedge \forall j : i \leq j < k_0 \Rightarrow (M', \pi'^{j} \models q)) \\ \Leftrightarrow & (M', \pi'^{i} \models q \,\mathbf{U}\, r). \end{array}$$

This completes the proof for the case $p \equiv q \,\mathbf{U}\, r$.

- Case $p \equiv q \,\mathbf{R}\, r$: first we prove

$$\begin{array}{ll} & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(q \,\mathbf{R}\, r)) \\ \Rightarrow & \forall j \geq i : ((\pi'_j, \pi'_{j+1}) \models T_\varphi(r) \wedge T_\varphi(\mathbf{X}(q \,\mathbf{R}\, r))) \\ & \quad \vee \exists k : i \leq k \leq j \wedge ((\pi'_k, \pi'_{k+1}) \models T_\varphi(q) \wedge T_\varphi(r)) \end{array} \tag{A.14}$$

by induction on j. For $j = i$, the antecedent $((\pi'_i, \pi'_{i+1}) \models T_\varphi(q \,\mathbf{R}\, r))$ gives us

$$\begin{array}{ll} & ((\pi'_j, \pi'_{j+1}) \models T_\varphi(q \,\mathbf{R}\, r)) \\ \Leftrightarrow & ((\pi'_j, \pi'_{j+1}) \models T_\varphi(r) \wedge T_\varphi(\mathbf{X}(q \,\mathbf{R}\, r))) \vee T_\varphi(r) \wedge T_\varphi(q) \\ \Leftrightarrow & ((\pi'_j, \pi'_{j+1}) \models T_\varphi(r) \wedge T_\varphi(\mathbf{X}(q \,\mathbf{R}\, r))) \\ & \quad \vee \exists k : i \leq k \leq j \wedge ((\pi'_k, \pi'_{k+1}) \models T_\varphi(q) \wedge T_\varphi(r)). \end{array}$$

For $j > i$, the induction hypothesis gives us two cases. In the first case, the induction hypothesis gives us $((\pi'_{j-1}, \pi'_j) \models T_\varphi(r) \wedge T_\varphi(\mathbf{X}(q \,\mathbf{R}\, r)))$, thus

$$\begin{array}{ll} & ((\pi'_{j-1}, \pi'_j) \models T_\varphi(r) \wedge T_\varphi(\mathbf{X}(q \,\mathbf{R}\, r))) \\ \Rightarrow & ((\pi'_j, \pi'_{j+1}) \models T_\varphi(q \,\mathbf{R}\, r)) \\ \Leftrightarrow & ((\pi'_j, \pi'_{j+1}) \models T_\varphi(r) \wedge T_\varphi(\mathbf{X}(q \,\mathbf{R}\, r))) \vee T_\varphi(r) \wedge T_\varphi(q) \\ \Rightarrow & ((\pi'_j, \pi'_{j+1}) \models T_\varphi(r) \wedge T_\varphi(\mathbf{X}(q \,\mathbf{R}\, r))) \\ & \quad \vee \exists k : i \leq k \leq j \wedge ((\pi'_k, \pi'_{k+1}) \models T_\varphi(q) \wedge T_\varphi(r)). \end{array}$$

In the second case, the induction hypothesis gives us $\exists k : i \leq k \leq j - 1 \wedge ((\pi'_k, \pi'_{k+1}) \models T_\varphi(q) \wedge T_\varphi(r))$, thus

$$\begin{array}{ll} & \exists k : i \leq k \leq j - 1 \wedge ((\pi'_k, \pi'_{k+1}) \models T_\varphi(q) \wedge T_\varphi(r)) \\ \Rightarrow & ((\pi'_j, \pi'_{j+1}) \models T_\varphi(r) \wedge T_\varphi(\mathbf{X}(q \,\mathbf{R}\, r))) \\ & \quad \vee \exists k : i \leq k \leq j \wedge ((\pi'_k, \pi'_{k+1}) \models T_\varphi(q) \wedge T_\varphi(r)). \end{array}$$

This completes the induction proof for (A.14).

From (A.14) we get

$$\begin{aligned}
& ((\pi'_i, \pi'_{i+1}) \models T_\varphi(q \,\mathbf{R}\, r)) \\
\Rightarrow\ & \forall j \geq i : ((\pi'_j, \pi'_{j+1}) \models T_\varphi(r) \wedge T_\varphi(\mathbf{X}(q \,\mathbf{R}\, r))) \\
& \quad \vee \exists k : i \leq k \leq j \wedge ((\pi'_k, \pi'_{k+1}) \models T_\varphi(q) \wedge T_\varphi(r)) \\
\Rightarrow\ & \forall j \geq i : ((\pi'_j, \pi'_{j+1}) \models T_\varphi(r)) \\
& \quad \vee \exists k : i \leq k \leq j \wedge ((\pi'_k, \pi'_{k+1}) \models T_\varphi(q) \wedge T_\varphi(r)) \\
\Rightarrow\ & \forall j \geq i : ((\pi'_j, \pi'_{j+1}) \models T_\varphi(r) \vee T_\varphi(q) \wedge T_\varphi(r)) \\
& \quad \vee \exists k : i \leq k < j \wedge ((\pi'_k, \pi'_{k+1}) \models T_\varphi(q) \wedge T_\varphi(r)) \\
\Rightarrow\ & \forall j \geq i : ((\pi'_j, \pi'_{j+1}) \models T_\varphi(r)) \vee\ \exists k : i \leq k < j \wedge ((\pi'_k, \pi'_{k+1}) \models T_\varphi(q)) \\
\Rightarrow\ & \forall j \geq i : (M', \pi'^j \models r) \vee\ \exists k : i \leq k < j \wedge (M', \pi'^k \models q) \\
\Leftrightarrow\ & \forall j \geq i : (\forall k : i \leq k < j \Rightarrow (M', \pi'^k \not\models q)) \Rightarrow (M', \pi'^j \models r) \\
\Leftrightarrow\ & (M', \pi'^i \models q \,\mathbf{R}\, r).
\end{aligned}$$

- Case $p \equiv \mathbf{G}q$: first we prove

$$((\pi'_i, \pi'_{i+1}) \models T_\varphi(\mathbf{G}q)) \Rightarrow \forall j \geq i : ((\pi'_j, \pi'_{j+1}) \models T_\varphi(q) \wedge T_\varphi(\mathbf{X}(\mathbf{G}q))) \tag{A.15}$$

by induction on j. For $j = i$, the antecedent $((\pi'_i, \pi'_{i+1}) \models T_\varphi(\mathbf{G}q))$ gives us

$$((\pi'_j, \pi'_{j+1}) \models T_\varphi(\mathbf{G}q)) \Leftrightarrow ((\pi'_j, \pi'_{j+1}) \models T_\varphi(q) \wedge T_\varphi(\mathbf{X}(\mathbf{G}q))).$$

For $j > i$, the induction hypothesis gives us $((\pi'_{j-1}, \pi'_j) \models T_\varphi(q) \wedge T_\varphi(\mathbf{X}(\mathbf{G}q)))$, thus

$$\begin{aligned}
& ((\pi'_{j-1}, \pi'_j) \models T_\varphi(q) \wedge T_\varphi(\mathbf{X}(\mathbf{G}q))) \\
\Rightarrow\ & ((\pi'_j, \pi'_{j+1}) \models T_\varphi(\mathbf{G}q)) \\
\Rightarrow\ & ((\pi'_j, \pi'_{j+1}) \models T_\varphi(q) \wedge T_\varphi(\mathbf{X}(\mathbf{G}q))).
\end{aligned}$$

This completes the induction proof for (A.15).

From (A.15) we get

$$\begin{aligned}
& ((\pi'_i, \pi'_{i+1}) \models T_\varphi(\mathbf{G}q)) \\
\Rightarrow\ & \forall j \geq i : ((\pi'_j, \pi'_{j+1}) \models T_\varphi(q) \wedge T_\varphi(\mathbf{X}(\mathbf{G}q))) \\
\Rightarrow\ & \forall j \geq i : ((\pi'_j, \pi'_{j+1}) \models T_\varphi(q)) \\
\Rightarrow\ & \forall j \geq i : (M', \pi'^j \models q) \\
\Leftrightarrow\ & (M', \pi'^i \models \mathbf{G}q).
\end{aligned}$$

- Case $p \equiv \mathbf{F}q$: first we prove

$$\begin{aligned}
& ((\pi'_i, \pi'_{i+1}) \models T_\varphi(\mathbf{F}q)) \\
\Rightarrow\ & \forall j \geq i : ((\pi'_j, \pi'_{j+1}) \models P_{\mathbf{F}q} \wedge T_\varphi(\mathbf{X}(\mathbf{F}q))) \\
& \quad \vee \exists k : i \leq k \leq j \wedge ((\pi'_k, \pi'_{k+1}) \models T_\varphi(q))
\end{aligned} \tag{A.16}$$

by induction on j. For $j = i$, the antecedent $((\pi'_i, \pi'_{i+1}) \models T_\varphi(\mathbf{F}q))$ gives us

$$\begin{aligned} & ((\pi'_j, \pi'_{j+1}) \models T_\varphi(\mathbf{F}q)) \Leftrightarrow ((\pi'_j, \pi'_{j+1}) \models T_\varphi(q) \vee P_{\mathbf{F}q} \wedge T_\varphi(\mathbf{X}(\mathbf{F}q))) \\ \Leftrightarrow \quad & ((\pi'_j, \pi'_{j+1}) \models P_{\mathbf{F}q} \wedge T_\varphi(\mathbf{X}(\mathbf{F}q))) \vee \ \exists k : i \leq k \leq j \wedge ((\pi'_k, \pi'_{k+1}) \models T_\varphi(q)). \end{aligned}$$

For $j > i$, the induction hypothesis gives us two cases. In the first case, the induction hypothesis gives us $((\pi'_{j-1}, \pi'_j) \models P_{\mathbf{F}q} \wedge T_\varphi(\mathbf{X}(\mathbf{F}q)))$, thus

$$\begin{aligned} & ((\pi'_{j-1}, \pi'_j) \models P_{\mathbf{F}q} \wedge T_\varphi(\mathbf{X}(\mathbf{F}q))) \\ \Rightarrow \quad & ((\pi'_j, \pi'_{j+1}) \models T_\varphi(\mathbf{F}q)) \\ \Leftrightarrow \quad & ((\pi'_j, \pi'_{j+1}) \models T_\varphi(q) \vee P_{\mathbf{F}q} \wedge T_\varphi(\mathbf{X}(\mathbf{F}q))) \\ \Rightarrow \quad & ((\pi'_j, \pi'_{j+1}) \models P_{\mathbf{F}q} \wedge T_\varphi(\mathbf{X}(\mathbf{F}q))) \vee \ \exists k : i \leq k \leq j \wedge ((\pi'_k, \pi'_{k+1}) \models T_\varphi(q)). \end{aligned}$$

For the second case we have $\exists k : i \leq k \leq j - 1 \wedge ((\pi'_k, \pi'_{k+1}) \models T_\varphi(q))$, thus

$$\begin{aligned} & \exists k : i \leq k \leq j - 1 \wedge ((\pi'_k, \pi'_{k+1}) \models T_\varphi(q)) \\ \Rightarrow \quad & ((\pi'_j, \pi'_{j+1}) \models P_{\mathbf{F}q} \wedge T_\varphi(\mathbf{X}(\mathbf{F}q))) \vee \ \exists k : i \leq k \leq j \wedge ((\pi'_k, \pi'_{k+1}) \models T_\varphi(q)). \end{aligned}$$

This completes the induction proof for (A.16).

We now show that given $((\pi'_i, \pi'_{i+1}) \models T_\varphi(\mathbf{F}q))$, then $\exists k \geq i : ((\pi'_k, \pi'_{k+1}) \models T_\varphi(q))$. Suppose not, then using (A.16) we get $\forall j \geq i : ((\pi'_j, \pi'_{j+1}) \models P_{\mathbf{F}q} \wedge T_\varphi(\mathbf{X}(\mathbf{F}q)))$. However, the hypothesis $\forall p \in el(\varphi), i \geq 0 : (ec(p) \not\equiv true) \Rightarrow \exists j \geq i : M', \pi'_j \models \neg P_p$ in (A.12) gives us $\forall i \geq 0 : \exists j \geq i : M', \pi'_j \models \neg P_{\mathbf{F}q}$, thus we get a contradiction. Thus we have

$$\begin{aligned} & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(\mathbf{F}q)) \Rightarrow \exists k \geq i : ((\pi'_k, \pi'_{k+1}) \models T_\varphi(q)) \\ \Rightarrow \quad & (\exists k \geq i : (M', \pi'^k \models q)) \Leftrightarrow (M', \pi'^i \models \mathbf{F}q). \end{aligned}$$

- Case $p \equiv \mathbf{GF}q$: first we prove

$$((\pi'_i, \pi'_{i+1}) \models T_\varphi(\mathbf{GF}q)) \Rightarrow \forall j \geq i : ((\pi'_j, \pi'_{j+1}) \models T_\varphi(\mathbf{GF}q)) \tag{A.17}$$

by induction on j. For $j = i$, the antecedent $((\pi'_i, \pi'_{i+1}) \models T_\varphi(\mathbf{GF}q))$ gives us $((\pi'_j, \pi'_{j+1}) \models T_\varphi(\mathbf{GF}q))$.

For $j > i$, the induction hypothesis gives us $((\pi'_{j-1}, \pi'_j) \models T_\varphi(\mathbf{GF}q))$, thus

$$\begin{aligned} & ((\pi'_{j-1}, \pi'_j) \models T_\varphi(\mathbf{GF}q)) \\ \Rightarrow \quad & ((\pi'_{j-1}, \pi'_j) \models T_\varphi(\mathbf{X}(\mathbf{GF}q)) \wedge (T_\varphi(p) \vee P_{\mathbf{GF}q})) \\ \Rightarrow \quad & ((\pi'_j, \pi'_{j+1}) \models T_\varphi(\mathbf{GF}q)). \end{aligned}$$

This completes the induction proof for (A.17).

We now show that given $((\pi'_i, \pi'_{i+1}) \models T_\varphi(\mathbf{GF}q))$, then $\forall j \geq i : \exists k \geq j : ((\pi'_k, \pi'_{k+1}) \models T_\varphi(q))$. Suppose not, then using (A.17) we get

$$\begin{aligned} & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(\mathbf{GF}q)) \\ \Rightarrow \quad & \forall j \geq i : ((\pi'_j, \pi'_{j+1}) \models T_\varphi(\mathbf{GF}q)) \\ \Leftrightarrow \quad & ((\pi'_j, \pi'_{j+1}) \models T_\varphi(\mathbf{X}(\mathbf{GF}q)) \wedge (T_\varphi(p) \vee P_{\mathbf{GF}q})) \\ \Leftrightarrow \quad & ((\pi'_j, \pi'_{j+1}) \models T_\varphi(\mathbf{X}(\mathbf{GF}q)) \wedge P_{\mathbf{GF}q}). \end{aligned}$$

However, the antecedent $\forall p \in el(\varphi), i \geq 0 : (ec(p) \not\equiv true) \Rightarrow \exists j \geq i : M', \pi'_j \models \neg P_p$ in (A.12) gives us $\forall i \geq 0 : \exists j \geq i : M', \pi'_j \models \neg P_{\mathbf{GF}q}$, thus we get a contradiction. Thus we have

$$\begin{aligned} & ((\pi'_i, \pi'_{i+1}) \models T_\varphi(\mathbf{GF}q)) \\ \Rightarrow \quad & \forall j \geq i : \exists k \geq j : ((\pi'_k, \pi'_{k+1}) \models T_\varphi(q)) \\ \Rightarrow \quad & (\forall j \geq i : \exists k \geq j : (M', \pi'^k \models q)) \\ \Leftrightarrow \quad & (M', \pi'^i \models \mathbf{GF}q). \end{aligned}$$

The proof of (A.12) is complete. Combining (A.11) and (A.12) immediately gives us (A.10), proving Lemma 6.5. □

The proofs of Lemma 6.4 and Lemma 6.5 complete the proof of Theorem 6.3 within the TGBA encoding scheme,

Bibliography

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[BCCZ99] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 99)*, pages 193–207, 1999.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.

[BE] Behavior Engineering homepage. www.griffith.edu.au/engineering-information-technology/institute-integrated-intelligent-systems/research/behavior-engineering. Accessed: 2014-07-24.

[Beh07] Behavior Tree Group. Behavior Tree Notation v1.0 (2007). Technical report, ARC Center for Complex Systems, 2007.

[Bet55] E.W. Beth. Semantic Entailment and Formal Derivability. *Mededelingen van de Koninklijke Nederlandse Akademie van Wetenschappen, Afdeling Letterkunde, N.R.*, 18(13):309–342, 1955.

[BK08] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[Bos08] J. Boston. Behavior Trees — How they Improve Engineering Behaviour? In *6th Annual Software and Systems Engineering Process Group Conference (SEPG 2008)*, 2008.

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

[Büc60] J.R. Büchi. On a Decision Method in Restricted Second Order Arithmetic. In *International Congress on Logic, Methodology and Philosophy of Science*, pages 1–11. Stanford University Press, 1960.

[CCGR99] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A New Symbolic Model Verifier. In *11th International Conference on Computer Aided Verification*, LNCS 1633, pages 495–499. Springer, 1999.

[CE81] E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Logic of Programs: Workshop*, LNCS 131, pages 52–71. Springer, 1981.

[CG05] M. Chechik and A. Gurfinkel. A framework for counterexample generation and exploration. In *8th International Conference on Fundamental Approaches to Software Engineering*, LNCS 3442, pages 220–236. Springer, 2005.

[CGH94] E. Clarke, O. Grumberg, and K. Hamaguchi. Another Look at LTL Model Checking. In *6th International Conference on Computer Aided Verification*, LNCS 818, pages 415–427. Springer, 1994.

[CGJ+00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *12th International Conference on Computer Aided Verification*, LNCS 1855, pages 154–169. Springer, 2000.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[CGMZ95] E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking. In *32nd annual ACM/IEEE Design Automation Conference*, pages 427–432. ACM, 1995.

[CGP99] E.M. Clarke, Jr., O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999.

[CH11] R.J. Colvin and I.J. Hayes. A Semantics for Behavior Trees using CSP with Specification Commands. *Science of Computer Programming*, 76(10):891–914, 2011.

[CVWY92] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

[Dij75] E.W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, 1975.

[dMOR+04] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In *16th International Conference on Computer Aided Verification*, LNCS 3114, pages 496–500. Springer, 2004.

[dMOS03] L. de Moura, S. Owre, and N. Shankar. The SAL language manual. Technical Report SRI-CSL-01-02, SRI International, 2003.

[Dro03] R.G. Dromey. From Requirements to Design: Formalizing the Key Steps (Invited Keynote Address). In *1st International Conference on Software Engineering and Formal Methods*, pages 2–11. IEEE Computer Society, 2003.

[Dro06] R.G. Dromey. Formalizing the Transition from Requirements to Design. In *Mathematical Frameworks for Component Software — Models for Analysis and Synthesis*, World Scientific Series on Component-Based Development, pages 156–187. World Scientific Publishing, 2006.

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

[EL86] E.A. Emerson and C-L. Lei. Efficient model checking in fragments of the propositional mu-calculus (extended abstract). In *Proceedings, 1st Annual Symposium on Logic in Computer Science*, pages 267–278. IEEE Computer Society, 1986.

[Eme81] E.A. Emerson. *Branching Time Temporal Logic and the Design of Correct Concurrent Programs*. PhD thesis, Harvard University, 1981.

[FS78] L. Flon and N. Suzuki. Consistent and Complete Proof Rules for the Total Correctness of Parallel Programs. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 184–192. IEEE Computer Society, 1978.

[GH99] A. Gargantini and C. Heitmeyer. Using Model Checking to Generate Tests from Requirements Specifications. In *Software Engineering ESEC/FSE99*, LNCS 1687, pages 146–162. Springer, 1999.

[GPVW95] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple On-the-fly Automatic Verification of Linear Temporal Logic. In *International Symposium on*

Protocol Specification, Testing and Verification, pages 3–18. Chapman and Hall, 1995.

[GS97] S. Graf and H. Saidi. Construction of Abstract State Graph. In *9th International Conference on Computer Aided Verification*, LNCS 1254, pages 72–83. Springer, 1997.

[GWY08] L. Grunske, K. Winter, and N. Yatapanage. Defining the Abstract Syntax of Visual Languages with Advanced Graph Grammars A Case Study Based on Behavior Trees. *Journal of Visual Languages & Computing*, 19(3):343–379, 2008.

[HKD09] T. Han, J.-P. Katoen, and B. Damman. Counterexample Generation in Probabilistic Model Checking. *IEEE Transaction on Software Engineering*, 35(2):241–257, 2009.

[Hol04] G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[HPY96] G.J. Holzmann, D. Peled, and M. Yannakakis. On Nested Depth First Search. *Proceedings of the Second SPIN Workshop*, 32:81–89, 1996.

[JD03] A.L. Juarez Dominguez and N.A. Day. Generating Multiple Diverse Counterexamples for an EFSM. Technical Report CS-2013-06, University of Waterloo, 2003.

[Kil73] G.A. Kildall. A Unified Approach to Global Program Optimization. In *1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 194–206. ACM, 1973.

[Koz83] D. Kozen. Results on the Propositional mu-Calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[LP85] O. Lichtenstein and A. Pnueli. Checking that Finite-State Concurrent Programs Satisfy their Linear Specification. In *12th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 97–107. ACM, 1985.

[LWK12] P.A. Lindsay, K. Winter, and S. Kromodimoeljo. Model-based Safety Risk Assessment using Behavior Trees. In *Proceedings of the 6th Asia Pacific Conference on System Engineering*. Systems Engineering Society of Australia, 2012.

[LWY10] P.A. Lindsay, K. Winter, and N. Yatapanage. Safety Assessment using Behavior Trees and Model Checking. In *8th International Conference on Software Engineering and Formal Methods*, pages 181–190. IEEE Computer Society, 2010.

[LYW12] P.A. Lindsay, N. Yatapanage, and K. Winter. Cut Set Analysis using Behavior Trees and Model Checking. *Formal Aspects of Computing*, 24(2):249–266, 2012.

[MC81] J. Misra and K.M. Chandy. Proofs of Networks of Processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, 1981.

[McM92] K.L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, 1992.

[Min93] S. Minato. Fast Generation of Prime-Irredundant Covers from Binary Decision Diagrams. *IEICE Transactions on Fundamentals of*, E76-A(6):967–973, 1993.

[MIY90] S. Minato, N. Ishiura, and S. Yajima. Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation. In *27th ACM/IEEE Design Automation Conference*, pages 52–57, 1990.

[Mor70] E. Morreale. Recursive Operators for Prime Implicant and Irredundant Normal Form Determination. *IEEE Transactions on Computers*, 19(6):504–509, 1970.

[Nam01] K.S. Namjoshi. Certifying Model Checkers. In *13th International Conference on Computer Aided Verification*, LNCS 2102, pages 2–13. Springer, 2001.

[NNH99] F. Neilson, H.R. Neilson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[Per84] C. Perrow. *Normal Accidents: Living with High Risk Technologies*. Basic Books, 1984.

[PLTP08] P. Papacostantinou, P. Lee, T. Tran, and V. Phillips. Implementing a Behaviour Tree Analysis Tool Using Eclipse Development Frameworks. In *19th Australian Software Engineering Conference*, pages 61–66. IEEE Computer Society, 2008.

[Pnu77] A. Pnueli. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society, 1977.

[PWW96] D. Peled, T. Wilke, and P. Wolper. An Algorithmic Approach for Checking Closure Properties of ω-Regular Languages. In *7th International Conference on Concurrency Theory*, LNCS 1119, pages 596–610. Springer, 1996.

[QS82] J.P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *International Symposium on Programming, 5th Colloquium*, LNCS 137, pages 337–351. Springer, 1982.

[RH01] S. Rayadurgam and M.P.E. Heimdahl. Coverage Based Test-Case Generation Using Model Checkers. In *8th IEEE International Conference on Engineering of Computer-Based Systems (ECBS 2001)*, pages 83–92. IEEE Computer Society, 2001.

[RV07] K.Y. Rozier and M.Y. Vardi. LTL Satisfiability Checking. In *Model Checking Software, 14th International SPIN Workshop*, LNCS 4595, pages 149–167. Springer, 2007.

[RV11] K.Y. Rozier and M.Y. Vardi. A Multi-encoding Approach for LTL Symbolic Satisfiability Checking. In *17th International Symposium on Formal Methods*, LNCS 6664, pages 417–431. Springer, 2011.

[SB05] V. Schuppan and A. Biere. Shortest Counterexamples for Symbolic Model Checking of LTL with Past. In *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 3440, pages 493–509. Springer, 2005.

[SS04] N. Shankar and M. Sorea. Counterexample-Driven Model Checking. Technical Report SRI-CSL-03-04, SRI International, 2004.

[SVV09] M. Schmalz, D. Varacca, and H. Völzer. Counterexamples in Probabilistic LTL Model Checking for Markov Chains. In *20th International Conference on Concurrency Theory*, LNCS 5710, pages 587–602. Springer, 2009.

[tex] TextBE: A Textual Editor for Behavior Engineering. Project URL: code.google.com/a/eclipselabs.org/p/textbe/. Accessed: 2014-07-24.

[WLC+07] L. Wen, K. Lin, R. Colvin, J. Seagrott, N. Yatapanage, and G. Dromey. Integrare — a Collaborative Environment for Behavior-Oriented Design. In

4th International Conference on Cooperative Design, Visualization, and Engineering, LNCS 4674, pages 122–131. Springer, 2007.

Index