ABSTRACT

Title of dissertation: LEARNING TEMPORAL PROPERTIES FROM DATA STREAMS

Samuel Huang
Doctor of Philosophy, 2020

Dissertation directed by: Professor Rance Cleaveland
Department of Computer Science

Verifying that systems behave as expected is a cornerstone of computing. In formal verification approaches, engineers capture their intentions, or specifications, mathematically, often using logic. The verification task is then to confirm that the system satisfies its specifications. In a formal setting this endeavor typically involves the construction of mathematical proofs, which are either constructed automatically, as in the case of so-called model-checking techniques, or by humans with machine assistance, as in the case of theorem-proving-based methodologies.

In practice, formal verification faces a number of obstacles. One involves the construction of formal specifications in the first place. Another is the lack of availability of analyzable system artifacts (source code, executables) about which proofs can be constructed. In this dissertation we propose techniques for inferring formal specifications, in the form of Linear Temporal Logic formulas, from system executions, and models when these are available, as a means of addressing these

concerns.

We first illustrate how guided generation of test cases (i.e. guided exploration of the system's input/output space) can be leveraged to develop and then refine the hypothesis set of invariants that a system satisfies. This approach was successful in revealing a system property required in code specification of an automotive application provided to us but missing in the implementation. An unexpected (undocumented) specification was also discovered through our analysis. The approach has since been applied by other researchers to several other automotive applications.

Second, we develop techniques to mine properties from models of systems and/or their executions. In some cases compact, finite state representations of a system is available. In this scenario we employ a novel automaton-based approach to mine properties matching a user-specified template. In other cases such white-box knowledge is not available and we must work over executions of the system rather than the system itself. In this scenario we apply a novel variant of Linear Temporal Logic (LTL) using finite-sequence semantics to again mine properties based on a specified template.

Lastly, we consider situations where standard formal properties are insufficient due to uncertainty or being overly simplified. Oftentimes properties that are not satisfied 100% of the time can be very interesting during a system inspection or system redesign. For example, natural noise manifesting in data streams from system executions such as missing evidence and changing system behavior could lead

to significant properties being overlooked if a strict “exactitude” were enforced. We explore the notion of “incomplete” properties which we term *partial invariants* by formulating a new “Noisy Linear Temporal Logic” which is an extension of standard LTL. We consider several representations of such noise and show decidability of the language emptiness problem for some of the variants.

LEARNING TEMPORAL PROPERTIES FROM DATA STREAMS

by

Samuel Huang

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2020

Advisory Committee:
Professor Rance Cleaveland, Chair/Advisor
Professor Steve Marcus
Professor Adam Porter
Professor Jim Reggia
Professor David Van Horn

© Copyright by
Samuel Huang
2020

Acknowledgments

The work presented in this document could not be possible without the support of my friends, family, and research community.

I would like to thank my advisor, Rance Cleaveland, for his guidance and support throughout the years. I also thank the members of the committee for being part of a necessary and crucial step in the dissertation process. I also thank my fellow friends and graduate students who were ready and willing to listen to some of my more zany ideas and theories.

I would like to thank my parents and sister for their encouragement throughout the years. I thank my 5-month old son Callum who has always has a smile when I see him, and for providing necessary breaks at the final stretch of the writing process. I would also like to thank my puppy Kira who has encouraged me to venture outdoors every now and then. I would lastly like to thank my wife Tayina for her patience and understanding throughout.

Table of Contents

Acknowledgements ii

Table of Contents iii

List of Tables vii

List of Figures viii

1 Introduction 1
- 1.1 Motivation 1
- 1.2 Description of Problems Addressed 5
  - 1.2.1 Chapter 4 — Model-Based Invariant Extraction from Test Cases 5
  - 1.2.2 Chapter 5 — LTL Query Checking 6
  - 1.2.3 Chapter 6 — Finite LTL 6
  - 1.2.4 Chapter 7 — Finite LTL Query Checking 7
  - 1.2.5 Chapter 8 — Formal Verification of Noisy Sequences 8
- 1.3 Organization of Chapters 8

2 Related Work 10
- 2.1 Formal Methods: Temporal Logic and Model Checking 10
- 2.2 Model/Program Synthesis 11
- 2.3 Interval Algebra 12
- 2.4 Association Rule Mining 14
- 2.5 Temporal Logic Queries 17
- 2.6 Invariant Extraction/Detection 18
- 2.7 Process Mining 20
- 2.8 Probabilistic Model Checking 20
- 2.9 Semantic Anomaly Detection 21
- 2.10 Finite LTL 22
- 2.11 Robust Temporal Logics 24

3 Background 25
- 3.1 Temporal Logic 25
  - 3.1.1 Kripke Structure 26

3.1.2 Linear Temporal Logic (LTL) . . . 27
3.1.2.1 Example queries . . . 36
3.1.3 A Branching Time temporal Logic (CTL*) . . . 36
3.1.3.1 Example queries . . . 39
3.1.4 Computational Tree Logic (CTL) . . . 39
3.1.4.1 Example queries . . . 41
3.2 Model Checking Algorithms . . . 41
3.2.1 LTL Model Checking . . . 42
3.2.2 CTL* Model Checking . . . 46
3.2.3 CTL Model Checking . . . 47
3.3 Association Rule Learning . . . 47
3.3.1 Apriori . . . 48
3.3.2 Alternative Statistics for Association Rules . . . 50
3.3.3 Limitations of Apriori, Alternative Algorithms . . . 51

4 Model-Based Invariant Extraction from Test Cases 52
4.1 Introduction . . . 52
4.2 Background . . . 54
4.2.1 Invariant Inference from Executions . . . 54
4.2.2 Automotive Model-Based Development . . . 56
4.2.3 Instrumentation-Based Verification . . . 59
4.2.4 Reactis . . . 60
4.3 Extracting Requirements . . . 62
4.4 Experimental Configuration . . . 67
4.4.1 Test Application . . . 67
4.4.2 Tool Chain . . . 68
4.4.3 Structural vs. Random Testing . . . 71
4.4.4 Invariant Refinement through Iteration . . . 73
4.5 Experimental Results . . . 75
4.6 Related Work . . . 78
4.7 Conclusion . . . 82

5 LTL Query Checking 84
5.1 Introduction . . . 84
5.2 Büchi Propositional Automaton . . . 86
5.3 The LTL Query Checking Problem . . . 88
5.4 Automaton-Based LTL Query Checking . . . 90
5.4.1 Propositional Queries . . . 91
5.4.2 Büchi Query Automata . . . 92
5.4.3 LTL Query Checking via Büchi Query Automata . . . 94
5.5 Implementing an LTL Query Checker . . . 95
5.5.1 Construct Büchi Automaton B_K . . . 96

5.5.2 Construct Büchi Query Automaton $B_{\neg\phi}[\texttt{var}]$ 96
5.5.3 Construct Product Query Automaton $B_{K,\neg\phi}[\texttt{var}]$ 96
5.5.4 Solve for Shattering Conditions of $B_{K,\neg\phi}[\texttt{var}]$ 97
5.5.5 Implementation and Evaluation 100
5.6 Conclusion 102

6 Finite LTL 104
6.1 Introduction 104
6.2 Finite LTL 106
6.2.1 Syntax of Finite LTL 106
6.2.2 Semantics of Finite LTL 107
6.2.3 Properties of Finite LTL 109
6.3 Normal Forms for Finite LTL 117
6.3.1 Extended Finite LTL and Positive Normal Form 117
6.3.2 Automaton Normal Form 121
6.4 A Tableau Construction for Finite LTL 135
6.4.1 The Construction 136
6.4.2 Discussion of Construction M_ϕ 140
6.5 Implementation and Empirical Results 143
6.6 Conclusion 148

7 Finite LTL Query Checking 149
7.1 Introduction 149
7.2 Definitions and Problem Statements 150
7.2.1 Finite Data Streams 150
7.2.2 Query Checking for Finite Data Streams 152
7.3 From Finite LTL Queries to Finite Automata 154
7.3.1 Finite Query Automata 154
7.3.2 Composing Finite Automata 156
7.4 Solving $QC(\Pi, \phi[\texttt{var}])$ 157
7.4.1 Constructing PNFA B_π for Data Stream π 158
7.4.2 Computing Shattering Formulas for $B_{\pi,\phi[\texttt{var}]}[\texttt{var}]$ 158
7.4.2.1 Shattering Propositional Queries. 159
7.4.2.2 Shattering Finite Query Automata. 161
7.4.3 Solving $\mathrm{QC}(\Pi, \phi[\texttt{var}])$ when $|\Pi| > 1$ 165
7.5 Experimental Results 166
7.6 Conclusion 168

8 Formal Verification of Noisy Sequences 169
8.1 Introduction 169
8.2 Near/Partial Invariants 171
8.3 Noisy Linear Temporal Logic 174
8.3.1 Characterizing Deviation from LTL 175

8.3.2 Syntax of Noisy LTL . . . 176
8.3.3 Semantics of Noisy LTL . . . 178
8.3.4 Example Formulas . . . 182
8.3.5 Properties of Noisy LTL . . . 183
8.3.6 Relating $\mathbf{F}'_{\mathrm{N}}, \mathbf{G}'_{\mathrm{N}}$ to Traditional LTL . . . 187
8.4 Decidability of Noisy LTL . . . 188
8.4.1 Unrolled semantics of Noisy LTL . . . 188
8.4.2 Tableau Construction . . . 202
8.4.3 Observations about Tableau Construction . . . 204
8.4.4 Tableau Examples . . . 206
8.5 Window-based Noisy LTL . . . 219
8.5.1 Decidability for Sliding Window Semantics . . . 222
8.5.2 Window-Based Example Formulas . . . 225
8.6 Conclusion . . . 226

9 Conclusion 227
9.1 Summarized Results of Problems Addressed . . . 227
9.1.1 Chapter 4 — Model-Based Invariant Extraction from Test Cases 227
9.1.2 Chapter 5 — LTL Query Checking . . . 228
9.1.3 Chapter 6 — Finite LTL . . . 229
9.1.4 Chapter 7 — Finite LTL Query Checking . . . 229
9.1.5 Chapter 8 — Formal Verification of Noisy Sequences . . . 230
9.2 Future Work by Chapter . . . 230
9.2.1 Chapter 4 Future Work . . . 231
9.2.2 Chapter 5 Future Work . . . 232
9.2.3 Chapter 6 Future Work . . . 233
9.2.4 Chapter 7 Future Work . . . 234
9.2.5 Chapter 8 Future Work . . . 235
9.3 Final Remarks . . . 236

Bibliography 237

List of Tables

3.1 Derived LTL Operators 31
3.2 Derived CTL operators 41

4.1 Putative invariants mined using full structure criteria. 75
4.2 Putative invariants mined using partial structure criteria. 75
4.3 Jaccard similarity scores for $E^{(1)}_{full}$ and $E^{(2)}_{full}$ 78
4.4 Jaccard similarity scores for $E^{(1)}_{partial}$ and $E^{(2)}_{partial}$ 78

List of Figures

3.1 LTL formula $\phi\,\mathbf{U}\,\psi$ . . . 30
3.2 LTL formula $\phi\,\mathbf{R}\,\psi$ . . . 32
3.3 LTL formula $\mathbf{F}\,\phi$ . . . 33
3.4 LTL formula $\mathbf{G}\,\phi$ . . . 34

4.1 Sample Simulink model . . . 57
4.2 Overview of requirements-extraction process . . . 62
4.3 State machine visualizations for $E^{(1)}_{full}, E^{(2)}_{full}, E^{(1)}_{partial}, E^{(2)}_{partial}$ . . . 79

5.1 Shattering edges in a Büchi query automaton . . . 93
5.2 Statistics for composed Büchi $B_{\mathcal{K}} \times B_{\neg\,\mathbf{G}(\mathtt{var})}$ . . . 101
5.3 Results for computing shattering solutions . . . 103

6.1 Summary of benchmark results . . . 145
6.2 Experimental benchmark: Formulas 0–91. . . . 146
6.3 Experimental benchmark: Formulas 92–183. . . . 147

7.1 Runtime Performance of Finite LTL Query Checking . . . 168

8.1 Adding noise to $a\,\mathbf{U}\,b$. . . . 176
8.2 Sketches of Noisy LTL Operators . . . 181
8.3 Noisy LTL Operator Characteristics. . . . 182
8.4 LTL Unrolled Semantics . . . 189
8.5 Semantic unrollings of noisy operators in noisy LTL . . . 202
8.6 Tableau method applied to state $q = \{a\,\mathbf{U}_{\mathrm{N}}[2]b\}$. . . . 204
8.7 Tableau example 1: $\phi = a\,\mathbf{U}\,b$ . . . 207
8.8 Tableau example 2: $\phi = a\,\mathbf{U}_{\mathrm{N}}\,[c = 2]b$ . . . 208
8.9 Tableau example 3: $\phi = (a\,\mathbf{U}\,b)\,\mathbf{U}\,c$ . . . 210
8.10 Tableau example 4: $\phi = (a\,\mathbf{U}\,b)\,\mathbf{U}_{\mathrm{N}}\,[c = 2]c$ . . . 213
8.11 Tableau example 5: $\phi = a\,\mathbf{R}\,b$ . . . 214
8.12 Tableau example 6: $\phi = a\,\mathbf{R}_{\mathrm{N}}\,[c = 2]b$ . . . 215
8.13 Tableau example 7: $\phi = (a\,\mathbf{R}\,b)\,\mathbf{R}\,c$ . . . 216

8.14 Tableau example 8: $\phi = (a \, \mathbf{R} \, b) \, \mathbf{R}_{\mathrm{N}} \, [c = 2]c$ 219

Chapter 1: Introduction

1.1 Motivation

Systems are pervasive in our world. From a web server handling user requests to a business process which helps to coordinate a company's operations to people forming lines in bakeries, systems are present in our society, but to different extents. Each distinct scenario falls along a continuum indicating how explicitly defined a system is. A web server has code which dictates how it behaves. A business process may be clearly (or not so clearly) documented for employees to read and be expected to follow. People waiting in line at a bakery understand the notion of a queue, but oftentimes no one is standing directing the flow of customers. As the number of systems employed grows, so too do the possible ways in which they can affect us and other systems. Being able to reason about a system and its behavior allows us to better explain events that we observe as well as better predict future events.

Characterizing system properties in a formal sense provides a solid grounding for reasoning over the systems. Formal reasoning allows us to abstractly represent properties that are relevant to a system, and through such an abstraction gain the

ability to perform more costly analysis which might have been inadmissible at a more technical level of representation, albeit usually at the cost of a loss of precision. Such a formalism also allows for comparison between systems that may at first seem disparate, and may provide a greater understanding about how systems can and do interact with one another. In the case of a piece of software where multiple programming languages are used for different subtasks and interface with each other, a common formal language for expressing properties of the each of the components can allow for easier reasoning about the composed system as a whole.

Increasing demands on performance, efficiency, and correctness have led to the creation of systems intended to satisfy these demands. Consequently, this has led to a demand for the analysis and evaluation of these new systems. To aid in this process, system designers will often incorporate logging functionality generated during a system's execution. It has become almost second nature to do this, as Odo from *Star Trek: Deep Space Nine* puts it: "...Humans have a compulsion to keep records and lists and files – so many, in fact, that they have to invent new ways to store them microscopically, otherwise their records would overrun all known civilization." In no small part due to the abundance of system logs, data sources that contain a temporal component have become ubiquitous in today's landscape. As a result, techniques providing formal analysis of data which have a temporal aspect have become a much sought after commodity.

Temporal data is readily available across a broad number of domains and

disciplines. Satellites sending transmissions back and forth from ground stations, UNIX system log files such as `/var/log/secure`, processes such as Business Activity Monitoring, gene expression data, and stock market tickers are only a small representation of temporal data generated from systems. Again, the comprehension of these systems can vary drastically. Some can be very well understood; for example the UNIX system logging source code may be readily available for analysis. Others may have an overarching set of rules that govern the system, such as a specification for satellite communication, or a business process model document, or a set of differential equations guiding the cellular electrophysiology of a heart cell. Others may not be well understood at all, such as a uncharacterized metabolic pathway underlying gene expression data or the behavior of the stock market. In some of these cases we have systems that are transparent and known to us (*white-box* knowledge) and in other cases the systems are more opaque (*black-box*). Despite such diversity of systems, we can still aim to have a common language for reasoning.

Not surprisingly, systems and data acquisition are not without their own troubles; imperfection can easily enter into the equation when the real world is involved. *Cyber-physical systems*, systems operating at the interface of physical systems and software components, are becoming increasingly abundant and present a number of challenges for analysis. Satellite transmissions could not being received, errors could be present when reporting events to a business process monitor, and inherently noisy sampling such as for gene expression data are known to muddle the truthfulness of

the data. Consequently, any technique designed for working over such data streams should be able to account for this noise in order to provide meaningful or otherwise useful results.

In this document we propose a framework leveraging both model checking and machine learning techniques in order to learn and reason about formal temporal properties over data streams (e.g. a system's executions). We believe that by first reasoning over such data streams, we can infer information about the underlying system itself. We begin by showing preliminary work where we perform requirements extraction using a model-based approach from test cases with some white-box knowledge of the system's model. We then explore a scenario where we have white-box knowledge of a system's model and demonstrate how to mine temporal logic formulas satisfied by the model using a novel form of query checking. We then turn to consider a formulation of Linear Temporal Logic using finite semantics, which we envision to be well suited for characterizing real-world observations of black-box systems. Finally, we turn to consider issues of uncertainty in a system model and data streams collected from it. We explore the notion of *partial invariants* of a system which we formalize and generalize as a novel temporal logic which we term noisy LTL.

1.2 Description of Problems Addressed

We provide here a high-level discussion of the research problems addressed along with a summary of proposed solutions, arranged by chapter presenting novel research (Chapters 4 through 8). This is intended to offer an overview of the subsequent material explored herein and to provide some justification for the importance of this document.

1.2.1 Chapter 4 — Model-Based Invariant Extraction from Test Cases

Model-based design of systems has become increasingly prevalent in a number of fields. Such an approach to system design can allow for a natural transition from more informal requirements such as natural language documents or use cases examples. Such a transition, however, can be fraught with inconsistencies as the design process progresses between different representations. Furthermore, design models can be complex and direct analysis is expensive.

We present an automatic method of ascertaining properties of a design model from its test cases (input/output pairs produced during model simulation). These properties, putative invariants to the system itself, offer a fast and inexpensive way to summarize design model behavior which is observed from generated test cases. Using methods which guide test case generation we can produce a suite of test cases which aim to represent the full behavior of the model under study.

1.2.2 Chapter 5 — LTL Query Checking

For a given model $\mathcal{M}$ and LTL formula ϕ, LTL model checking performs the verification task of determining whether or not $\mathcal{M}$ satisfies ϕ. One can relax the formula ϕ to a formula template $\phi[\texttt{var}]$ (a query), allowing a subformula `var` of ϕ to be unspecified. The LTL query checking is the discovery problem to determine a set of solutions to `var` such that when `var` is substituted for solution c, the resulting formula $\phi[c]$ is satisfied by $\mathcal{M}$. This problem can aid in the comprehension and reasoning of $\mathcal{M}$, allowing a user to determine specifics of a property when only an abstracted template query is known.

We present a solution to the LTL query checking problem by adapting the traditional automaton-based LTL model checking algorithm to support LTL queries rather than the usual LTL formula. We provide analysis and empirical results for several sample models.

1.2.3 Chapter 6 — Finite LTL

Many real-world systems generate data streams that have a finite length. Consequently, the infinite semantics of traditional temporal logics such as Linear Temporal Logic (LTL) do not naturally align, and as a result direct application of such logical schemes can be problematic when attempting to perform traditional formal methods tasks such as model checking. A number of different approaches have been

considered in order to bridge this disconnect.

We present a finite-semantic temporal logic inspired by LTL, which we call Finite LTL, which naturally accepts sequences of finite length, thereby obviating the need for a translation from finite-sequence data to infinite-sequence formalisms. A number of results are proven for this logic. We also provide a construction for a finite automaton representing Finite LTL formulas, analogous to the well-studied Büchi automaton tableau constructions from traditional LTL, which enables tasks such as model checking using minor adaptations from the traditional schemes.

1.2.4 Chapter 7 — Finite LTL Query Checking

In many cases white-box knowledge of a model under study is **not** known, but we *are* able to execute the model to gather execution data. As only black-box knowledge is assumed, reasoning about a system's inner workings (to whatever extent possible) can be placed at a premium.

Combining and extending the results from the previous two chapters, this chapter investigates the problem of query checking for Finite LTL over finite length data streams. Rather than being provided a model $\mathcal{M}$ on which to perform model checking as done in Chapter 5, we assume the user is given a set of finite sequences which the model exhibits. We convert these finite data streams into automaton representations which can be used as a model of the data, and then perform query checking by first adapting the tableau construction method from Chapter 6 to pro-

duce query automaton from a Finite LTL query and then proceed with an adapted version of traditional LTL model checking.

1.2.5 Chapter 8 — Formal Verification of Noisy Sequences

Many real-world data streams contain artifacts and abnormalities that perturb the stream away from its predicted baseline behavior. Minor perturbations can oftentimes substantially impact the validity of formal properties expressed over such data streams. This lack of robustness in turn minimized the applicability of properties expressed in formal logics such as LTL.

We present a new temporal logic intended to be robust to such perturbations. This new logic, called Noisy LTL, is an extension of LTL which allows for deviation from the traditional LTL properties by a parameterized degree. We consider several methods to model the degree of deviation, and provide a number of theoretical results relating Noisy LTL to traditional LTL. We also provide a construction to produce a Büchi automaton from a Noisy LTL formula, yielding an immediate solution to the Noisy LTL model checking problem.

1.3 Organization of Chapters

The remainder of this document is organized as follows. Chapter 2 provides a high level discussion of work related to our research. Chapter 3 elaborates substantially on the particular works discussed in Chapter 2 that is fundamental to our research.

Chapter 4 presents work of ours [1] published in Runtime Verification 2010, which discusses model-based invariant extraction from test cases. Chapter 5 discusses a novel, automaton-theoretic approach to the problem of Linear Temporal Logic (LTL) query checking. Chapter 6 defines a finite-semantic version of LTL as well as shows a construction for a finite automaton representing Finite LTL formulas. Chapter 7 discusses the query checking problem as presented for Finite LTL. Chapter 8 investigates how uncertainty can be incorporated into our framework, including through the use of partial invariants as well as establishing a new temporal logic supporting noise. Finally, Chapter 9 summarizes the contents of this proposal and gives concluding remarks, including potential future research directions as a result of this thesis work.

Chapter 2: Related Work

We provide discussion for work related to the contents of this proposal. Several topics contained here are foundational for the direction of our research; they are indicated as such and explained in substantially longer detail in Chapter 3.

2.1 Formal Methods: Temporal Logic and Model Checking

Temporal logic refers to any number of systems or frameworks that involve reasoning over a set of propositions with regards to time. Statements such as "he arrived before I did" or "the pressure rises until the pipe breaks" or "she always answers the phone" are all representable statements, and depending upon the specific language of temporal logic used, can be expressed very succinctly or not at all. Our work is built upon two specific flavors of temporal logic, Linear Temporal Logic (LTL) [2] and Computational Tree Logic (CTL) [3]. For a more in-depth discussion regarding temporal logic the reader is referred to Section 3.1.

Certain temporal logics (including LTL and CTL) are of special interest because they can serve as the basis for a specification in a *verification* task. That is, given a model of a system, we can ask if the model satisfies some specification,

which can be posed using a language based in temporal logic. Verifying specifications such as "the server is always up" or "the math always adds up" are possible by performing this verification task, also referred to as *model checking*. Algorithms exist for a number of well-studied temporal logics, some of which are substantially more efficient than others. For more details on model checking, consult Section 3.2.

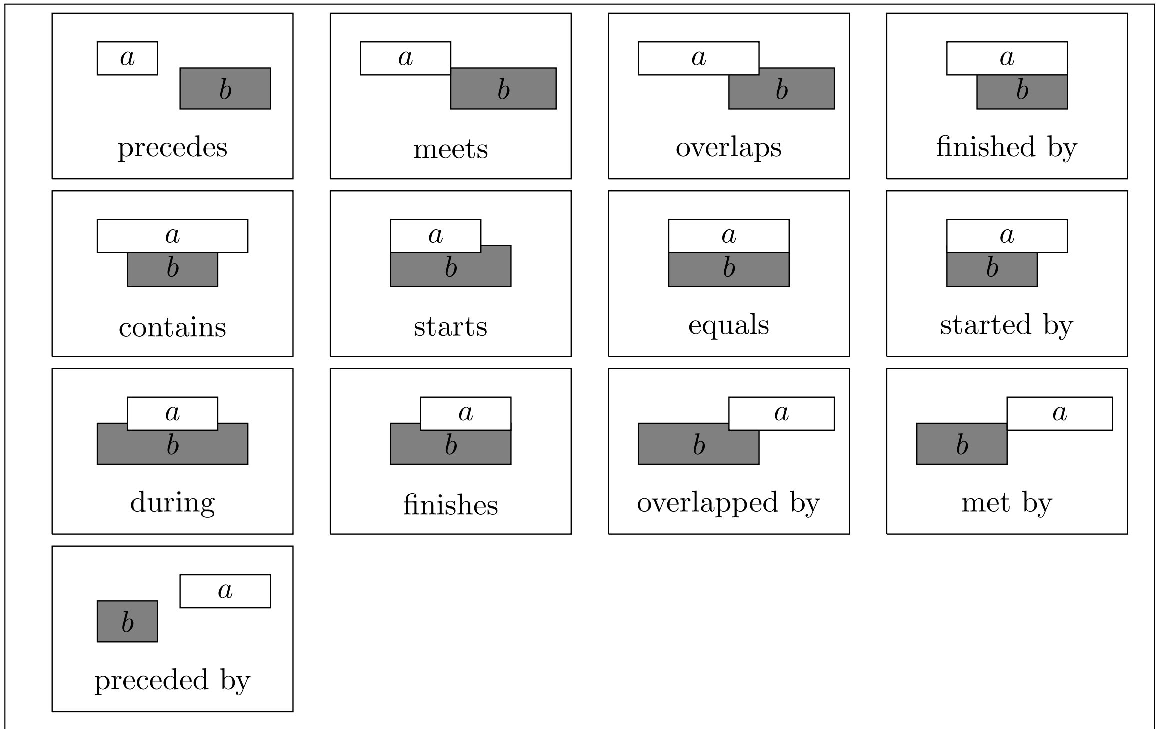
2.2 Model/Program Synthesis

The model synthesis problem is a problem related to model checking but with a different intent. Rather than verifying the satisfaction of a (model, property) pair, it is given a set of properties and attempts to find a succinct model satisfying them. Formally, the model synthesis problem asks "Given a set of properties Φ, what is the model or class of models that satisfy all of Φ?" Often times this is expressed as *program* synthesis [4, 5] instead of model synthesis, and the goal is then usually to build an executable piece of code to accomplish the supplied specification, rather than to build an abstract mathematical model. Model synthesis usually varies on along three axes of choices. Firstly, the form specifications take can vary drastically, including written documentation, logical relations, and even programs themselves that solve the same or related task but are inefficient. The search space of valid solutions also is subject to variation (which usually leads to distinguish model synthesis from program synthesis): automata are sometimes desirable, whereas in other cases a functional program is found. Lastly, the search technique can range

from a number of machine learning-inspired approaches to logical reasoning to an exhaustive search.

2.3 Interval Algebra

Other systems of operators alternative to LTL or CTL exist to describe temporal relationships between entities. In 1983, Allen [6] proposed what is now known as "Allen's interval algebra." This algebra includes a formal system for representing temporal relationships between pairs of time intervals. 13 distinct operators were included to create an exhaustive and distinct set of relationships that were qualitative in nature. Operators such as "a precedes b" (the end point of a occurs before the start point of b) and "a finishes b" (the start point of a occurs after the start point of b and the end points of a and b are equal) are some of the 13 operators defined. The full set is given below:



These 13 operators describe interval positioning in the definitive case; i.e. the case where the interval boundaries are known. In the indefinite case, relationships can be expressed as a boolean disjunction of several of the definite operators. For example, suppose interval a describes the time range when "John was present in the room" and interval b describes the range where "Mary turned on the light switch." Then the statement "John was not in the room when Mary turned on the light switch" has four possible distinct definite relationships that explain a's positioning relative to b. Either a precedes b, a is preceded by b, a meets b, or a is met by b. For the indefinite case there are $2^{13} = 8192$ possible "general" relations formed by considering each subset of definite relations.

In contrast to temporal logics such as LTL and CTL which abstract entities

to time points, the Allen operators assume the entities are represented as intervals. Rosu *et al.* [7] investigated the relationship between the Allen interval algebra and LTL. They constructed an intermediate language termed "Allen linear temporal logic" or ALTL, which has complexity and syntax matching that of Allen's interval algebra but model is similar to LTL. They further showed that ALTL can be efficiently encoded into a fragment of LTL which then can be subject to verification using the same algorithms that already exist for LTL.

The close-ended interval representation provided by the Allen operators is sometimes undesirable, especially when reasoning with uncertainty. In 1992, Freksa [8] proposed a semi-interval notation where one boundary of each interval is known, and the other is not. Semi-interval comparisons are made only based upon the known boundary. Because of this, a number of Allen relations collapse to others, for example the Allen relations "a precedes b," "a meets b," "a overlaps b," "a finished by b," and "a contains b" all map to a single Freksa relation "a older than b" when the end points of a and b are not known. The Freksa relation set contains 10 distinct operators compared to the 13 provided by Allen.

2.4 Association Rule Mining

Association rule mining is a technique for discovery of relationships between items over large datasets (originally targeted for database scale). Initially proposed by Agrawal *et al.* in 1993 [9], it seeks to uncover evidence of *association rules* between

items. These rules take the form $\{a_1, \ldots, a_k\} \implies \{b\}$, which indicate that if items a_1 through a_k are present, then item b is likely to be present as well. The original example considered was a basket analysis of consumer purchases at a grocery store. Items were products found in each customer's shopping cart, and association rules revealed strong ties between groceries. For example, a customer who buys both milk and eggs may also be highly expected to buy bread. The Apriori algorithm was proposed to mine these patterns. It performs in a bottom-up breadth-first manner, considering candidate sets of items of increasing size while verifying that they satisfy criteria for showing a strong correlation from the data. A notion of *confidence* of a rule is developed which corresponds to the representation that rule has over the data set being mined. This confidence value is used to evaluate a rule's strength. Section 3.3 provides a more in-depth discussion about the Apriori algorithm.

In 1995, Agrawal *et al.* [10] published a follow-up result which extended their existing theory and algorithm to be applicable to so-called *sequential patterns*, namely patterns of the form

$$\{a_1, \ldots, a_i\} \rightsquigarrow \{b_1, \ldots, b_j\} \rightsquigarrow \{c_1, \ldots, c_k\} \rightsquigarrow \ldots$$

In contrast to the association rule structure listed above, this sequential pattern occurs over multiple time instances; the implication structure of the rule implies a

passage of time.[1] An example of such a rule is "customers who typically rent 'Star Wars', then 'The Empire Strikes Back', and then (finally) 'Return of the Jedi' ." While the association rule work focuses on determining relationships between items in a single purchase, sequential pattern mining seeks to identify relationships over different transactions. Agrawal showed how mining such sequential patterns could be performed using a modified version of the Apriori algorithm.

Following this initial work by Agrawal *et al.*, others approaches have provided various improvements algorithmic efficiency and supporting parallel execution. Han *et al.* [11] proposed a technique they call the Frequent Pattern growth algorithm (FP-growth algorithm) using a novel data structure they call a FP-tree. Advantages over Apriori include the avoidance of generating all possible candidate itemsets while exploring the space of possible solutions, reduced database scan time due to a highly compressed novel data structure, and a natural possibility to parallelize the algorithm over multiple databases for a substantial performance boost. The Eclat family of algorithms initially proposed by Zaki *et al.* [12, 13] in 1997 take advantage of the structure of the boolean lattice structure that itemsets compose. A clustering is shown to be possible to generate small independent subspaces, which are of the size that can individually computed in main memory rather than on disk.

In 2000, Webb [14] published a new algorithm for association rule learning based on his earlier published OPUS search algorithm [15] from 1995. OPUS identi-

[1] In the literature, the standard notation is $\implies$ but we use $\rightsquigarrow$ here to disambiguate sequential patterns from standard association rules which incorporate no notion of time.

fies ahead of time portions of the search space that could be concluded to contain no solutions to avoid work, which allows for early pruning of the overall search space.

2.5 Temporal Logic Queries

In 2000, Chan [16] developed a theory of "temporal logic queries." In this setting, the user provides a model and a CTL *query*, which is a standard CTL formula that has exactly one placeholder indicated. This placeholder indicates an ambiguity which, once grounded with a logical proposition, turns the CTL query into a valid CTL formula. In particular, Chan defines the semantics of a CTL query to evaluate to the strongest proposition that makes the grounded CTL formula hold for the model given. For example, the CTL query $\mathbf{A}\,\mathbf{G}?$ (here "?" is the placeholder) evaluates to the strongest proposition p that makes $\mathbf{A}\,\mathbf{G}\,p$ hold, namely the strongest invariant of the model. Strength here corresponds to the relative propositions have on the boolean lattice, for example the proposition $a \vee b$ is weaker than both a and b which both in turn are weaker than $a \wedge b$. The solution to queries are naively computed by performing standard model checking over all possible propositions (of which there are exponentially many in the number of atomic propositions). Taking advantage of ordered traversal of the proposition space, Chan presents a subclass of CTL queries which can be solved in time linear to both the model size and the query length. Gurfinkel *et al.* [17] later relaxed the requirement that there be only one placeholder in the query, and presented an implementation of a temporal logic query

checking solving this relaxed problem.

Chan's work was subsequently extended to more expressive branching-time logics via alternating-tree automaton constructions [18] and three-valued model checking [19]; this last paper also describes several applications of the technique in areas such as invariant inference and test generation. Other work has studied the problem for classes of infinite-state systems [20].

In contrast to branching time, linear-time query checking has remained relatively unstudied. Chokler et al. [21] consider several variants of LTL query checking and prove complexity results for these problems; however, no implementation or experimental results were reported.

2.6 Invariant Extraction/Detection

A program *invariant* is a property or condition that holds true for a certain point or points in a program. Example conditions such as being in a range ($a \leq x \leq b$), equality ($x = y$), and an array being in sorted order are all possible invariants. The goal of invariant detection is to identify such program invariants either in an either off-line (static) or online (dynamic) setting. Program invariants can serve to provide an understanding of a program's behavior, which in turn allows for a number of useful applications including (but not limited to) improved documentation, directed generation of test cases, asserting program validity, and automatic theorem proving.

Presented by Ernst in 2007, Daikon [22] is a system for dynamic invariant

detection. It functions by running a given program, observing the values computed by the program, and then reports properties that were not invalidated during the monitored executions. One of the focuses of the work is to make the Daikon system easily adaptable to new domains by providing functionality for adding new properties past the built-in invariant types as well as new data source types. It also assesses uses statistical measures to assess if putative invariants being considering dynamically are actually invariant over the program and not just overfit conclusions from the learning process.

In 2015 Lemieux *et al.* [23] described an approach for mining specifications over system trace logs. They present an implementation of their approach, Texada, which takes as input a set of traces and a LTL template (which they call an LTL property type) and return all grounded instances of the LTL template which are supported by the traces. Notions of noise are also included as they also can return grounded properties which are not supported by the entire set of traces. Although not explicitly indicated by the authors, this project also fits under the work of "template logic queries" as described above.

It is worth noting that the usage of "invariant" has become somewhat multipurpose across subfields of research; its meaning is somewhat contextually dictated. While the uses can be seen as related to some extent, they do have nuanced distinctions. During the presentation of our own work later (notably Chapter 4), we provide a specific meaning for the scope of our research.

2.7 Process Mining

Research in the Business Process Management domain has worked to develop many models of business activity, formal and otherwise. The field of process mining deals in part with extracting knowledge from these models that can help improve, among other things, workflow at a fast and agile pace. Process mining can take several forms. In the *discovery* case, the model is not known *a priori* and the challenge is to construct a representation or understanding of the model through the events captured from outputs such as system logs or other events and observations. In the *conformance* case, the model is instead known ahead of time and the goal is to ensure or verify that the model does not deviate from the intended behavior. Here deviation is measured by variation in output when compared to the formal model design itself. This is effectively model verification as described in the formal methods literature, and indeed one direction is to apply model checking to achieve conformance, as done by Van der Aalst *et al.* [24]. They present work for performing verification over XML-based event logs and a derivative of LTL as the property language to verify.

2.8 Probabilistic Model Checking

PRISM is a *probabilistic model checker* developed by Kwiatkoska *et al.*[25]. It is designed to perform verification tasks over models that exhibit probabilistic behav-

ior such as Markov chains and continuous time Markov Chains, as well as pose probabilistic queries over such models of the form $Pr_{>0.95}(a\,\mathbf{U}\,b)$. In this example the model is verified if at least 95% of the executions satisfy the property $a\,\mathbf{U}\,b$. PRISM supports queries over pCTL and CSL [26, 27], temporal logics that are similar to CTL. Queries about long-run, steady-state dynamics can be posed relative to the underlying Markov chain/CTMC, such as "what is the long run probability that the queue is more than 75% full?" While precise computation using results from the markov chains literature is possible, PRISM has functionality to perform *simulation-based verification*, namely they will execute the model a large number of times and compute a probability for the query to be satisfied.

2.9 Semantic Anomaly Detection

Work done by Raz *et al.*[28] focus on identifying *semantic anomalies* over online data feeds. Raz defines a semantic anomaly to be "unreasonable values of successfully parsed data," meaning that the system and software operate to push data through in a properly formatted manner, but data values make little sense semantically. Raz shows that by first mining from data feeds invariants relative to the data's semantics one can establish a semantic baseline, helping to identify semantic anomalies from unseen data. Motivation for this work was in part to improve incomplete or incorrect specifications for the data stream; Raz provides a process by which the specifications can be updated based upon an adjusted expectation given observed

semantic deviation.

2.10 Finite LTL

This section reviews existing work on the use of finite versions of LTL.

There have been a number of different LTL constructions intended to bridge the gap between finite-length real world data sequences and the automated reasoning of LTL *vis à vis* infinite sequences in the formal-verification community. Some have arisen originating with an intended application in mind (i.e. planning/robotics), while others approach from a more foundational direction. As such, different works have yielded several variants of so-called "finite LTL." With varying semantics syntactically similar logics, there are a number of nuances that are elicited when comparing these different works.

De Giacomo and Vardi [29] provide complexity analysis of their finite semantic logic LTL_f. They also devise a PDL-inspired logic into which LTL_f can be translated in linear time. It was shown for both logics that determining satisfiability of a formula is PSPACE-complete. De Giacomo et al. [30] focus on the interplay between finite and infinite LTL. They are address some risks of directly transferring approaches from the infinite to the finite case, and specifically formalize when a finite trace is "insensitive to infiniteness," where an infinite suffix is constructed to pad a finite sequence. This property is shown to be provable using a standard LTL reasoner.

Li [31] presents a discussion of applying transition systems to satisfiability of LTL_f. They present a recursive construction for a normal form of an LTL_f formula ϕ. However, there does appear to be a minor issue in their normalization of the *Until* and *Release* operators such that they are not logical duals (which historically they are in standard LTL). Roşu [32] poses a sound and complete proof system for his version of finite-trace temporal logic.

Fionda and Greco [33] investigate the complexity of satisfiability for restricted fragments of LTL over finite semantics. They also provide an implementation of a finite LTL reasoner which utilizes their complexity analysis to identify when satisfiability for a specified formula is computable in polynomial time, and performs the computation in the case; otherwise they convert the input formula to a SAT representation and invoke a SAT solver. Notably, their maximal fragment does not include the dual operator of *Next* or any binary operator with temporal modality such as *Until*, and only permits negation to be applied to atomic propositions.

A recent paper by Li et al [34] addresses Mission-Time LTL (MLTL), an LTL logic with time intervals supported for the *Until* and *Release* operators. They develope a satisfiability checking tool for MLTL by first using a novel transformation from MLTL to LTL/LTL_f and then turning this resulting formula into SAT. The authors also observe that there is a need for more solvers of these related languages.

2.11 Robust Temporal Logics

Work has been done to support different types of uncertainty, one of note is characterizing a system's *robustness*, or the ability for a system's characteristics to persist despite some amount of perturbation.

Abbas *et al.* [35] developed an approach to falsify Metric Temporal Logic (an extension of Linear Temporal Logic involving timing constraints) using a notion of robustness as a guiding principle in a random walk through the input space to a model.

More recently, Tabuada *et al.* introduced Robust Linear Temporal Logic [36] as a temporal logic designed to provide support for expressing robustness in a system through the use of applying a 5-valued logic semantics over LTL syntax. Here robustness is indicated by a system only demonstrate small violations of specified properties when a minor perturbation in the environment is observed.

Chapter 3: Background

We provide elaborated background discussion for the relevant work that we build from, presenting topics in a more technical light than they were portrayed in Chapter 2.

The reader interested in learning more about temporal logic, model checking, and its complexity is advised to consult the survey by Schnoebelen [37], which provides a more thorough account than the space we have afforded them here.

3.1 Temporal Logic

Temporal logic is an extension of propositional logic intended to represent properties of propositions quantified over time. We can express properties such as "The system is offline," (basic propositional logic), as well as "The system will eventually be offline," and "the system will never be offline." Historically, such a framework has been useful in posing assertions of both *safety* where something bad never happens (the system will never be offline) and *liveness* (the system will always come back online).

Temporal logic has traditionally been applied on top of formal mathematical

models of systems. Such models are usually Kripke structures, automata-like constructs which the semantics of different TL varieties are commonly defined over. Once a particular temporal logic and a mathematical model are chosen, one can pose a *verification* question such as "does the model satisfy the property?"

Two common temporal logics are Linear Temporal Logic (LTL) and Computational Tree Logic (CTL), whose syntax and semantics we provide here along with the definition of a Kripke structure.

3.1.1 Kripke Structure

Definition 1 (Kripke Structure)**.** *A* Kripke structure M *over a set of atomic propositions* $\mathcal{AP}$ *is a 5-tuple* $M = \langle S, \mathcal{AP}, R, l, s_i \rangle$ *where:*

- S *is a non-empty finite set of states,*
- $\mathcal{AP}$ *is the set of atomic propositions acting as labels,*
- $R \subseteq S \times S$ *is the transition relation which is left-total (for all states* $s \in S$ *there exists some* $x \in S$ *such that* $\langle s, x \rangle \in R$*),*
- $l : S \to 2^{\mathcal{AP}}$ *is the state-labeling function, and*
- $s_i \in S$ *is the initial state.*

We express M *as the 4-tuple* $M = \langle S, R, l, s_i \rangle$ *when* $\mathcal{AP}$ *is clear from context.*

A Kripke structure encodes the behavior of a system, with S representing system states and the transition relation recording the possible execution steps when

a system is in a given state: when the system is in state s it can evolve in one step to state s' iff $(s, s') \in R$. The labeling function l indicates which atomic propositions are true in any given state; if $a \in l(s)$ then a is deemed true at state s, while if $a \notin l(s)$ it is false. A path $p = s_0 s_1 s_2 \ldots \in S^\omega$ is said to be *generated* by a Kripke structure $M = \langle S, \mathcal{AP}, R, l, s_i \rangle$ if there is a sequence of states starting at s_i which then follows the transition relation. [1] Formally, $p \in S^\omega$ is generated by M if $s_0 = s_i$, and $\forall j \in \mathbb{N}, (s_j, s_{j+1}) \in R$. Because R is left-total, each state s is has at least one transition leading from s to some other state, so any finite sequence of states following valid transitions is the prefix of some infinite path generated by M. The labeling function can be lifted to accept infinite sequences of states; for state sequence $p = s_0 s_1 \ldots \in S^\omega$ we define $l(p) = l(s_0) l(s_1) \ldots$ to be the *word* corresponding to state sequence p. The *language generated by M* is the set of words $L(M) = \{l(p) : K \text{ generates } p\}$.

3.1.2 Linear Temporal Logic (LTL)

Initially proposed by Pnueli in 1977 [2], Linear Temporal Logic is a temporal logic that describes properties over sequences. Intuitively, it focuses on properties that describe long-run and future states of tracked properties over a system's execution path. Properties such as "it is eventually true that the system will halt" and "the power is always on" are describable using LTL.

[1] In contrast to S^* which is the set of all *finite* sequences of states from S, S^ω refers to the set of all *infinite* sequences.

Definition 2 (LTL Syntax)**.** *For a set of atomic propositions $\mathcal{AP}$, the set of LTL formulas is defined by the following grammar, where $a \in \mathcal{AP}$:*

$$\phi ::= a \mid \neg\phi \mid \phi \vee \phi \mid \mathbf{X}\,\phi \mid \phi\,\mathbf{U}\,\phi$$

We call the operators $\neg$ and $\vee$ propositional *and $\mathbf{X}$ and $\mathbf{U}$* modal*. We use $\Phi^{\mathcal{AP}}$ to refer to the set of all LTL formulas and $\Gamma^{\mathcal{AP}} \subsetneq \Phi^{\mathcal{AP}}$ for the set of all propositional formulas, i.e. those containing no modal operators. We will write Φ and Γ instead of $\Phi^{\mathcal{AP}}$ and $\Gamma^{\mathcal{AP}}$ when $\mathcal{AP}$ is clear from context.*

The first three syntax rules establish a propositional logic, and the remaining two, termed the "next" ($\mathbf{X}$) and "until" ($\mathbf{U}$) operators are the extension with temporal modality. The semantics of an LTL formula ϕ is defined with regards to infinite-length sequences. Informally, for a particular sequence of sets of atomic propositions drawn from some parent set $\mathcal{AP}$, an LTL formula that is purely propositional satisfies the sequence if the first entry of the sequence evaluates to true over the formula. The "next" temporal operator requires us to look one entry further down in the sequence to evaluate truth, while the "until" temporal operator requires one formula to be true from the first entry to some later entry where a second formula is instead true for one instance. We first formally define notation for infinite-length sequences, and then give the semantics for LTL.

Definition 3 (Infinite Length Sequences)**.** *Let X be a set, with X^{ω} the set of infinite*

sequences of elements of X. Also assume that $\pi \in X^\omega$ has the form $\pi = \pi_0\pi_1\pi_2\ldots$. We define the following notation.

1. *For $i \in \mathbb{N}$, $\pi[i] = \pi_i$.*

2. *For $i \in \mathbb{N}$, $\pi(i)$ is the suffix of π starting at i, taken to be $\pi(i) = \pi_i\pi_{i+1}\ldots$. Note that $\pi(0) = \pi$ and that any suffix of π is itself an infinite sequence taken from X^ω.*

3. *If $x \in X$ and $\pi \in X^\omega$, then $x\pi \in X^\omega$ is the sequence such that $(x\pi)(0) = x$ and $(x\pi)(1) = \pi$.*

The semantics of LTL is defined with respect to the satisfaction relation "$\models$":

Definition 4 (LTL Semantics)**.** *For a set $\mathcal{AP}$, let $\phi \in \Phi^{\mathcal{AP}}$ be an LTL formula and $\pi \in (2^{\mathcal{AP}})^\omega$ be an infinite-length sequence over the power set of $\mathcal{AP}$. The satisfaction relation $\models$ is defined inductively on the structure of ϕ as follows:*

- $\pi \models a$ *if* $a \in \pi[0]$
- $\pi \models \neg\phi$ *if* $\pi \not\models \phi$
- $\pi \models \phi_1 \vee \phi_2$ *if* $\pi \models \phi_1$ *or* $\pi \models \phi_2$
- $\pi \models \mathbf{X}\,\phi$ *if* $\pi\,(1) \models \phi$
- $\pi \models \phi_1\,\mathbf{U}\,\phi_2$ *if* $\exists i \in \mathbb{N} : \pi(i) \models \phi_2$ *and* $\forall j \in \mathbb{N} : (0 \leq j < i) \implies \pi(j) \models \phi_1$

Figure 3.1: LTL formula $\phi \, \mathbf{U} \, \psi$

We write $[\![\phi]\!]$ *for the set* $\{\pi : \pi \models \phi\}$ *and say that LTL formulas* ϕ_1 *and* ϕ_2 *are* logically equivalent*, with notation* $\phi_1 \equiv \phi_2$*, if* $[\![\phi_1]\!] = [\![\phi_2]\!]$*.*

One can think of $\phi_1 \, \mathbf{U} \, \phi_2$ as the requirement that ϕ_1 must persist until some instance where ϕ_2 is true. A sketch to provide further intuition is provided in Figure 3.1.

Having provided above a core syntax and semantics for LTL, we now take note of several commonly derived operators that allow for concise (and usually very meaningful) LTL formulations. Of note are the boolean conjunction and implication ($\wedge$ and $\implies$), as well as the derived temporal logic operators "release," "eventually," "always," and "weak until" (denoted **R**, **F**, **G**, and **W**). We will forgo giving the boolean derivations here; they are the usual form. The definitions for the derived temporal logic operators are given in the Table 3.1.

We can see from the definitions above that the "release" operator is defined to be the logical dual of "until" while the "always" operator is the logical dual of "eventually." We can explicitly write the definitions in terms of set notation, which

Operator	Expression		Derived Syntax
Release	$\phi_1 \, \mathbf{R} \, \phi_2$	$\triangleq$	$\neg\,(\neg\phi_1 \, \mathbf{U} \, \neg\phi_2)$
Eventually	$\mathbf{F} \, \phi$	$\triangleq$	$\mathtt{tt} \, \mathbf{U} \, \phi$
Always	$\mathbf{G} \, \phi$	$\triangleq$	$\neg \, \mathbf{F} \, \neg\phi$
Weak until	$\phi_1 \, \mathbf{W} \, \phi_2$	$\triangleq$	$\phi_2 \, \mathbf{R} \, (\phi_1 \vee \phi_2)$

Table 3.1: Derived LTL Operators

also provides an intuition behind their naming scheme. We start with release:

$$\begin{aligned}
\phi_1 \, \mathbf{R} \, \phi_2 &\triangleq \neg\,(\neg\phi_1 \, \mathbf{U} \, \neg\phi_2) \\
&= \neg\,[\exists i \in \mathbb{N} : (\pi(i) \models \neg\phi_2) \wedge \forall j \in \mathbb{N} : (0 \leq j < i) \implies \pi(j) \models \neg\phi_1] \\
&= \forall i \in \mathbb{N} : (\pi(i) \models \phi_2) \vee \exists j \in \mathbb{N} : (0 \leq j < i) \vee \pi(j) \models \phi_1
\end{aligned}$$

From this representation, we can see that $\phi_1 \, \mathbf{R} \, \phi_2$ is satisfied if for every step of the sequence, the step either satisfies ϕ_2 outright or there is some earlier step where ϕ_2 was satisfied (the release requirement) and no other restriction is placed upon that step. Note that from this definition it is clear that ϕ_1 may never occur, but in this case ϕ_2 is required to always be true. Figure 3.2 illustrates these two cases.

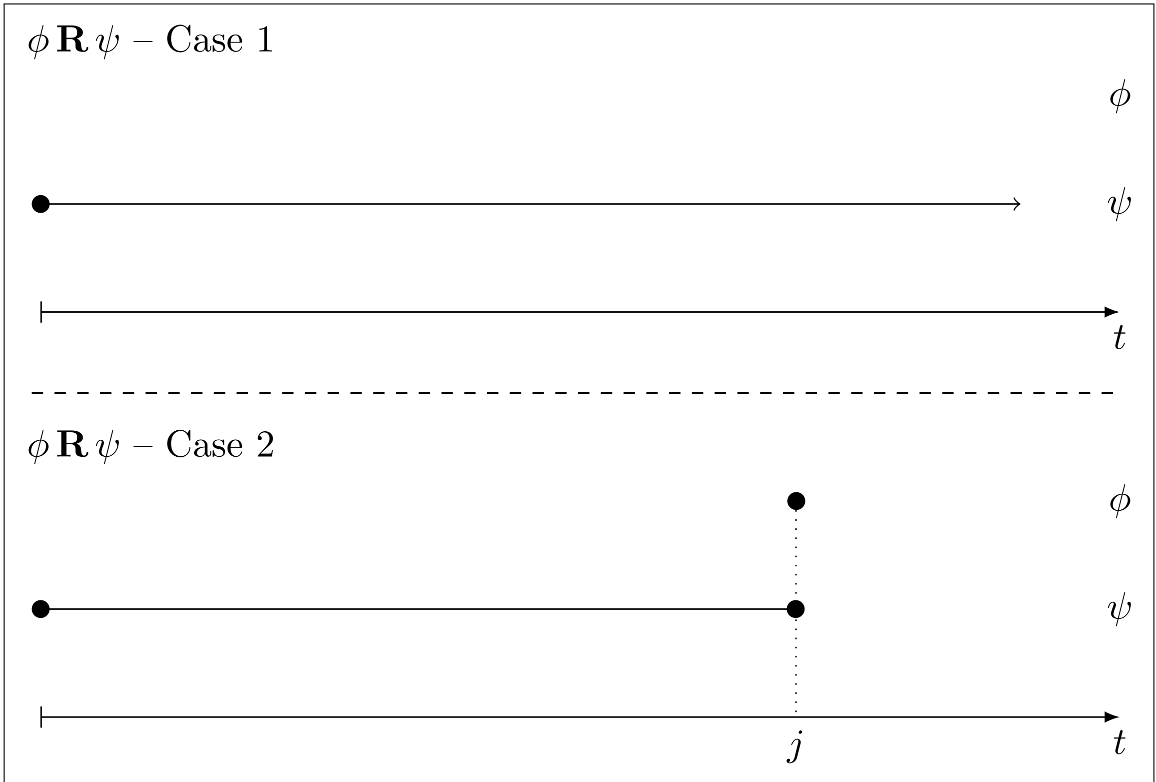


Figure 3.2: LTL formula $\phi\,\mathbf{R}\,\psi$

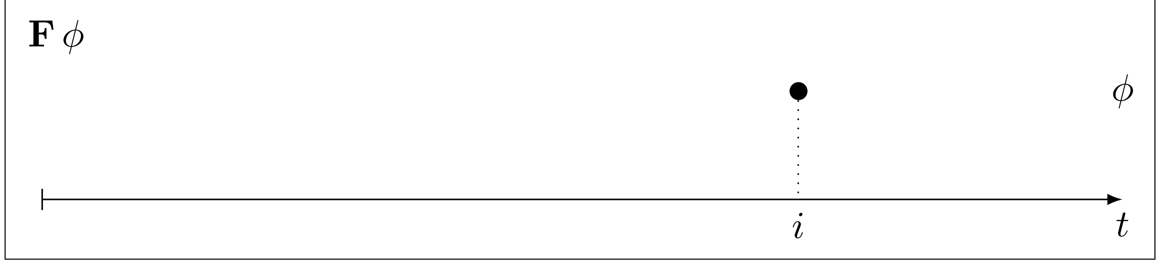


Figure 3.3: LTL formula $\mathbf{F}\,\phi$

Eventually ($\mathbf{F}$) has the following semantics:

$$\mathbf{F}\,\phi \triangleq \mathtt{tt}\,\mathbf{U}\,\phi$$

$$= \exists i \in \mathbb{N} : \pi(i) \models \phi \land \forall j \in \mathbb{N} : (0 \leq j < i) \implies \pi(j) \models \mathtt{tt}$$

$$= \exists i \in \mathbb{N} : \pi(i) \models \phi$$

From this representation we clearly see that $\mathbf{F}\,\phi$ is true exactly when ϕ is satisfied at some future instance. A sketch for eventually is provided in Figure 3.3.

Always ($\mathbf{G}$) has the following semantics:

$$\mathbf{G}\,\phi \triangleq \neg\,\mathbf{F}\,\neg\phi$$

$$= \neg\,(\exists i \in \mathbb{N} : \phi(i) \models \neg\phi)$$

$$= \forall i \in \mathbb{N} : \pi(i) \models \phi$$

From this representation we can see that $\mathbf{G}\,\phi$ is true exactly when ϕ is satisfied at all present and future instances. We can also rewrite $\mathbf{G}\,\phi$ future in order to express

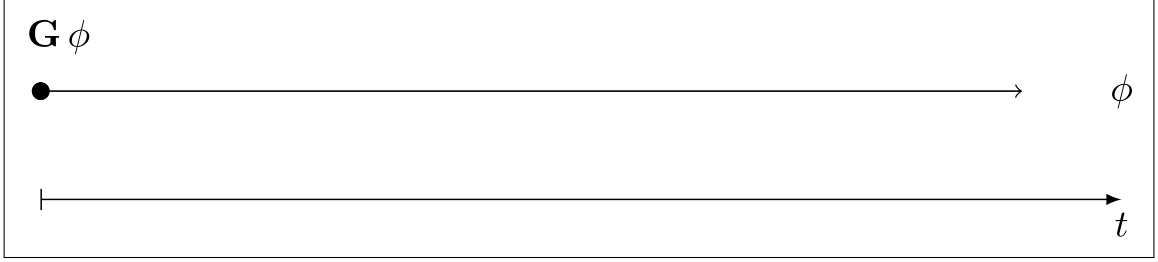


Figure 3.4: LTL formula $\mathbf{G}\,\phi$

$\mathbf{G}$ in terms of $\mathbf{R}$:

$$\begin{aligned}
\mathbf{G}\,\phi &= \forall i \in \mathbb{N} : \pi(i) \models \phi \\
&= \forall i \in \mathbb{N} : \pi(i) \models \phi \vee \exists j \in \mathbb{N} : (0 \leq j < i \wedge \pi(j) \models \texttt{ff}) \\
&= \texttt{ff}\,\mathbf{R}\,\phi
\end{aligned}$$

Using this form, one can think of "always" as a release operation whose release criterion is never met. A sketch for always is provided in Figure 3.4.

Lastly, weak until ($\mathbf{W}$):

$$\begin{aligned}
\phi_1\,\mathbf{W}\,\phi_2 &\triangleq \phi_2\,\mathbf{R}\,(\phi_1 \vee \phi_2) \\
&= \forall i \in \mathbb{N} : \pi(i) \models (\phi_1 \vee \phi_2) \vee \exists j \in \mathbb{N} : (0 \leq j < i) \wedge \pi(j) \models \phi_2 \\
&= \forall i \in \mathbb{N} : \pi(i) \models \phi_1 \vee \pi(i) \models \phi_2 \vee \exists j \in \mathbb{N} : (0 \leq j < i) \wedge \pi(j) \models \phi_2
\end{aligned}$$

Using reasoning similar to what was done for $\phi_1\,\mathbf{R}\,\phi_2$, we can see that for $\phi_1\,\mathbf{W}\,\phi_2$ to be satisfied, for every step i in the sequence, either of the two conditions ϕ_1 or ϕ_2 must be met, or there must exist some earlier index j such that the release

condition ϕ_2 is satisfied. This is similar to the standard until $\phi_1 \, \mathbf{U} \, \phi_2$, with the noticeable exception that ϕ_1 may continue to be held true forever in the case where ϕ_2 is never observed, and $\mathbf{W}$ would be satisfied (which is not permitted by $\mathbf{U}$). The connection to $\mathbf{U}$ can be made more explicit by considering the above semantics when conjoined with $\mathbf{F} \, \phi_2$:

$$
\begin{aligned}
(\phi_1 \, \mathbf{W} \, \phi_2) \land \mathbf{F} \, \phi_2 = (&\forall i \in \mathbb{N} : \pi(i) \models \phi_1 \lor \pi(i) \models \phi_2 \lor \\
&\qquad \exists j \in \mathbb{N} : (0 \leq j < i) \land \pi(j) \models \phi_2) \\
&\land \exists i \in \mathbb{N} : \pi(i) \models \phi_2 \\
= \; &\phi_1 \, \mathbf{U} \, \phi_2
\end{aligned}
$$

With the "promise" of an eventual satisfaction of ϕ_2, the third disjunct in $\phi_1 \, \mathbf{W} \, \phi_2$ is known to occur somewhere (the third disjunct is the same condition as the added $\mathbf{F} \, \phi_2$), and the semantics reduce to $\phi_1 \, \mathbf{U} \, \phi_2$. We can see that the difference between $\phi_1 \, \mathbf{U} \, \phi_2$ and $\phi_1 \, \mathbf{W} \, \phi_2$ is exactly when the sequence is an infinite run of ϕ_1 with no occurrence of ϕ_2.

As a final note, the logical dual of $\mathbf{X}$ is itself:

$$
\begin{aligned}
\neg \, \mathbf{X} \, \neg\phi &= \neg \, (\pi(1) \models \neg\phi) \\
&= \pi(1) \models \phi \\
&= \mathbf{X} \, \phi
\end{aligned}
$$

This observation will be useful later when discussing alternative temporal logics, notably CTL in Section 3.1.4.

3.1.2.1 Example queries

We provide a few example queries posed in LTL to demonstrate the expressiveness:

- $\mathbf{F}\,$terminate – The program will eventually end.
- raining $\mathbf{U}\,$Thursday – It will rain until Thursday
- $\mathbf{G}\,$server running – The server is always running.
- $\mathbf{G}\,$(911 dialed $\implies$ $\mathbf{F}\,$police arrive) – Whenever 911 is dialed, the police will eventually arrive.
- $\mathbf{G}\,$(turn signal $\implies$ (indicator blinks $\mathbf{U}\,$turn is made)) – Whenever the turn signal is activated, the light blinks until the turn is made.

3.1.3 A Branching Time temporal Logic (CTL*)

Computational Tree Logic [38] is a temporal logic that supports analysis of branching paths in a system's execution. For ease of transition from LTL, we will first discuss CTL*, a superset of both CTL and LTL. Whereas LTL allows one to assert properties over an execution of a system (or a system who is linear in its behavior), CTL* supports branching logic by means of the addition of *path quantifiers.* A path quantifier provides the ability to reason over different branching executions from a

particular state of a Kripke structure. It can assert that "some branches" (existential) satisfy a property. LTL has no notion of branching time, it assumes there is one path that is followed. We provide the syntax and semantics for CTL*. CTL* has two types of logic formulas, *state* formulas and *path* formulas. State formulas are typically denoted using σ, while path formulas are represented using ϕ. A CTL* state formula defined over an atomic proposition set $\mathcal{AP}$ has the following syntax:

$$
\begin{aligned}
\sigma \quad ::= \quad & a \in \mathcal{AP} \\
\mid \quad & \neg\sigma \\
\mid \quad & \sigma \vee \sigma \\
\mid \quad & \mathbf{E}\,\phi
\end{aligned}
$$

The syntax for a corresponding path formula is:

$$
\begin{aligned}
\phi \quad ::= \quad & \sigma \\
\mid \quad & \neg\phi \\
\mid \quad & \phi \vee \phi \\
\mid \quad & \mathbf{X}\,\phi \\
\mid \quad & \phi\,\mathbf{U}\,\phi
\end{aligned}
$$

As seen for LTL, we have derived logical operators $\wedge$ and $\implies$, as well as derived path operators **R**, **F** **G**, and **W**, all defined in the same manner. Addi-

tionally, we have a derived state operator **A**, defined to be the dual of **E**. The semantics of both formula types are now given with respect to a Kripke structure $M = \langle S, \mathcal{AP}, R, l, s_I \rangle$. We use the satisfaction relation $\models_M$ to denote this. State formulas are satisfiable only by states in the Kripke structure, whereas path formulas are satisfiable by a sequence of states, i.e. a path. We denote states here as s and paths as π. The semantics for a state formula is as follows:

- $s \models_M a \in \mathcal{AP}$ if $a \in l\,(s)$
- $s \models_M \neg\sigma$ if $s \not\models \sigma$
- $s \models_M \sigma_1 \vee \sigma_2$ if $s \models_M \sigma_1$ or $s \models_M \sigma_2$
- $s \models_M \mathbf{E}\,\phi$ if there exists some path π emanating from s such that $\pi \models_M \phi$

A state satisfies a simple atomic proposition a if a is contained in the set returned from the labeling function from M as applied to s. The semantics for $\neg$ and $\vee$ are defined in the usual fashion. $\mathbf{E}\,\phi$ requires that some path π exists which start at s and emanate following the Kripke structure's state relation R, and for π to satisfy ϕ. The semantics for a path formula is as follows:

- $\pi \models_M \sigma$ if $\pi(0) \models_M \sigma$
- $\pi \models_M \neg\phi$ if $\pi \not\models_M \phi$
- $\pi \models_M \phi_1 \vee \phi_2$ if $\pi \models_M \phi_1$ or $\pi \models_M \phi_2$

- $\pi \models_M \mathbf{X}\,\phi$ if $\pi(1) \models_M \phi$
- $\pi \models_M \phi_1\,\mathbf{U}\,\phi_2$ if $\exists i \in \mathbb{N} : \pi(i) \models_M \phi_2 \wedge \forall j \in \mathbb{N} : (j < i) \implies \pi(j) \models_M \phi_1$

Note that the semantics for $\neg$, $\vee$, $\mathbf{X}$, and $\mathbf{U}$ are identical to those presented for LTL in Section 3.1.2. All of the above semantics are given relative to states/paths satisfying a CTL* property; to use the provided set of semantics when considering "does $M \models \sigma$?" where σ is a CTL* state formula, we equate this to asking "does $s_I \models_M \sigma$?" We do not allow the question "does $M \models \pi$?" for a path formula π; as a path modality ($\mathbf{A}$, $\mathbf{E}$) must be supplied before any path operators have a context to be expressed (we must know which paths we are quantifying over before we can reason about them).

3.1.3.1 Example queries

We provide a few example queries posed in CTL* to demonstrate the expressiveness:

- $\mathbf{E}\,\mathbf{X}\,fail$ – Sometimes you are about to fail.
- $\mathbf{E}\,\mathbf{G}\,(x \implies \mathbf{F}\,y)$ – In some cases whenever x happens, y eventually follows.
- $\mathbf{A}\,\mathbf{F}\,success$ – All paths lead to success.

3.1.4 Computational Tree Logic (CTL)

Computational Tree Logic, first proposed by Clarke and Emerson [3], was designed specifically to support branching logic. Where the syntax for CTL* allowed one to

have a series of path modalities (**X**, **U**, etc) in a row, CTL restricts formulas by requiring that a path quantifier (**E** or **A**) is put in front of each modality. This restriction also removes the need for two types of logical formulas; all formulas in CTL are path formulas. The syntax for a CTL formula is defined as follows:

$$
\begin{aligned}
\sigma \quad ::= & \quad a \in \mathcal{AP} \\
| & \quad \neg\sigma \\
| & \quad \sigma \vee \sigma \\
| & \quad \mathbf{E}\,\mathbf{X}\,\sigma \\
| & \quad \mathbf{E}\,(\sigma\,\mathbf{U}\,\sigma) \\
| & \quad \mathbf{E}\,(\sigma\,\mathbf{R}\,\sigma)
\end{aligned}
$$

As done for LTL in Section 3.1.2 and CTL* in Section 3.1.3, we have the normal derived boolean operators $\wedge$ and $\implies$, as well as the derived path modalities **R**, **F**, **G**, and **W**. Note that explicit syntax for $\mathbf{E}\,(\sigma\,\mathbf{R}\,\sigma)$ (denoted $\mathbf{E}\,\mathbf{R}$) is required in addition to $\mathbf{E}\,(\sigma\,\mathbf{U}\,\sigma)$ (denoted $\mathbf{E}\,\mathbf{U}$) because of the requirement that **E** immediately precede the temporal modality. All derived operators from LTL and CTL* are obtainable here as well, although now are constrained to occur in the modality-follows-quantifier format. Table 3.2 summarizes these relations.

The semantics of the operators follow the same definition as presented in LTL and CTL*. As all formulas are state formulas, the model checking question again

Operator	Expression	Derived Syntax
$\mathbf{A}\,\mathbf{X}$	$\mathbf{A}\,\mathbf{X}\,\sigma$	$= \neg\,\mathbf{E}\,\mathbf{X}\,\neg\sigma$
$\mathbf{A}\,\mathbf{U}$	$\mathbf{A}\,(\sigma_1\,\mathbf{U}\,\sigma_2)$	$= \neg\,\mathbf{E}\,(\neg\sigma_1\,\mathbf{R}\,\neg\sigma_2)$
$\mathbf{A}\,\mathbf{R}$	$\mathbf{A}\,(\sigma_1\,\mathbf{R}\,\sigma_2)$	$= \neg\,\mathbf{E}\,(\neg\sigma_1\,\mathbf{U}\,\neg\sigma_2)$
$\mathbf{E}\,\mathbf{F}$	$\mathbf{E}\,\mathbf{F}\,\sigma$	$= \mathbf{E}\,(\mathtt{tt}\,\mathbf{U}\,\sigma)$
$\mathbf{E}\,\mathbf{G}$	$\mathbf{E}\,\mathbf{G}\,\sigma$	$= \neg\,\mathbf{A}\,\mathbf{F}\,\neg\sigma$
$\mathbf{A}\,\mathbf{F}$	$\mathbf{A}\,\mathbf{F}\,\sigma$	$= \mathbf{A}\,(\mathtt{tt}\,\mathbf{U}\,\sigma)$
$\mathbf{A}\,\mathbf{G}$	$\mathbf{A}\,\mathbf{G}\,\sigma$	$= \neg\,\mathbf{E}\,\mathbf{F}\,\neg\sigma$
$\mathbf{E}\,\mathbf{W}$	$\mathbf{E}\,(\sigma_1\,\mathbf{W}\,\sigma_2)$	$= \mathbf{E}\,(\sigma_2\,\mathbf{R}\,(\sigma_1 \vee \sigma_2))$
$\mathbf{A}\,\mathbf{W}$	$\mathbf{A}\,(\sigma_1\,\mathbf{W}\,\sigma_2)$	$= \mathbf{A}\,(\sigma_2\,\mathbf{R}\,(\sigma_1 \vee \sigma_2))$

Table 3.2: Derived CTL operators

becomes "does $s_I \models_M \sigma$?"

3.1.4.1 Example queries

We provide a few example queries posed in CTL to demonstrate the expressiveness:

- $\mathbf{A}\,\mathbf{G}\,(\textit{shutdown} \implies \mathbf{A}\,\mathbf{G}\,\neg\textit{lights})$ – The lights are always off after a shutdown.
- $\mathbf{A}\,\mathbf{G}\,(\textit{sent} \implies \mathbf{E}\,\mathbf{F}\,\textit{lost})$ – Any message sent could be lost.
- $\mathbf{E}\,\mathbf{G}\,(\textit{spend} \implies \mathbf{E}\,\mathbf{F}\,\textit{earn})$ – Sometimes you have to spend money to make money.

3.2 Model Checking Algorithms

Having provided a definition for a model as well as syntax and semantics of several common temporal logic languages, we turn to the question of model checking. Namely, given an atomic proposition set $\mathcal{AP}$ with Kripke structure $M =$

$\langle S, \mathcal{AP}, R, l, s_I \rangle$ and logical formula ϕ, we ask "does $M \models \phi$?" Recall that this general form of the verification question must first be grounded for the particular choice of temporal logic. In particular, for LTL "$M \models \phi$" becomes "$\pi \models \phi$," where π is an execution of M and ϕ is a path formula expressed in LTL. For CTL, the model checking question becomes "$s_I \models_M \sigma$" where σ is a state formula expressed in CTL and $\models_M$ is the satisfaction relation defined over M. We present here discussion about the theoretical complexity of performing model checking over LTL, CTL*, and finally CTL, including a brief history of work seminal in the field. In the case of LTL, which our work relies most heavily upon, we also provide a discussion about one approach to performing LTL model checking.

3.2.1 LTL Model Checking

The original presentation of LTL from Pnueli in 1977 showed that LTL verification was decidable [2]. In 1985, Sistla and Clarke showed that it is both in the complexity class PSPACE and PSPACE-hard [39], implying that it is PSPACE-complete. This was done via a reduction from model checking to satisfiability.

Lichtenstein and Pnueli [40] provided the first practical algorithm for LTL historically, which had running time of $2^{O(|\phi|)} O(|M|)$. Work done by Vardi and Wolper [41, 42] show an alternative algorithm using their work on Büchi automata over modal logics gives the same running time. It is this algorithm that we will highlight here.

Definition 5 (Büchi automaton)**.** *A* Büchi automaton *is a 5-tuple* $B = \langle S, \Sigma, \delta, s_i, F \rangle$, *where:*

- S *is a finite, non-empty set of states,*
- Σ *is a finite, non-empty set of alphabet symbols,*
- $\delta \subseteq S \times \Sigma \times S$ *is a labeled transition relation,*
- $s_i \in S$ *is the initial state, and*
- $F \subseteq S$ *is a set of accepting states.*

First published by Büchi in 1960, a *Büchi automaton* is an automaton designed for recognizing infinite sequences of symbols. For a Büchi automaton $B = \langle S, \Sigma, \delta, s_i, F \rangle$ and an infinite sequence $w = a_0 a_1 \ldots \in \Sigma^\omega$, a *run* of B on w is a sequence of states $s_0 s_1 \ldots \in S^\omega$ such that $s_0 = s_i$ and $\forall i \in \mathbb{N}, \langle s_i, a_i, s_{i+1} \rangle \in \delta$. A run of B on w corresponds to a "proper" execution of B starting at the initial state s_i and moving from state to state by following the transition relation, taking a transition only if the label of the transition matches the current position in the word w. Such a run is *accepting* if $\forall i \in \mathbb{N}, \exists j \geq i : s_j \in F$. That is to say, a run is accepting if at any point during a run we will, at some future state in the run, encounter a state that is in the accepting set F. We say "B accepts w" if there is an accepting run of B on w. The language of B is defined as $L(B) = \{w \in \Sigma^\omega : B \text{ has an accepting run on } w\}$. We pause here to formally define ω-regular languages.

Definition 6 (ω-regular languages)**.** *An ω-language is a set of infinite-length sequence of symbols. An ω-regular language is an ω-language is defined inductively as follows:*

- *A^ω is ω-regular if A is a non-empty regular language not containing the empty string.*
- *AB is ω-regular if A is a regular language and B is an ω-regular language.*
- *$A \cup B$ is ω-regular if A and B are ω-regular.*

We now present some results regarding Büchi. (1) Büchi automaton are known to accept exactly the set of all ω-regular languages. As ω-regular languages are known to be closed under complementation and intersection, so too are Büchi automata. In particular, an algorithm for computing the intersection of two Büchi automata B_1, B_2 exists which runs in $O(|B_1| \cdot |B_2|)$. (2) For a Büchi automata B, it is decidable whether or not $L(B) = \emptyset$. The computation to determine if $L(B) = \emptyset$ is straight forward; one need only compute the strongly connected components of the graph formed by the states of B and the transition relation δ, and identify if the connected component with the initial state s_i is able to reach any non-trivial connected component containing an accepting state. Determining SCCs is the dominating step of this approach. Using an algorithm such as Tarjan's [43] for computing SCCs leads to overall runtime of $O(|B|)$. (3) Given a Kripke structure M, there is a Büchi automaton B_M such that the language $L(B_M)$ accepts those runs that ex-

actly correspond to all possible executions of M. This can be accomplished directly as follows: For Krikpke structure $M = \langle S, \mathcal{AP}, R, l, s_i \rangle$, let $B_M = \left\langle S, 2^{\mathcal{AP}}, \delta_M, s_i, S \right\rangle$ be the corresponding Büchi automaton, where

$$\delta_M = \{(s, A, s') : (s, s') \in R \text{ and } A = L(s)\}$$

The sets of labels on each state in M is converted to an element of the power set of the labels for B_M, and transitions are labeled according to the label of the outgoing state from M. All states in B_M are marked as accepting because all possible executions from M are allowed. Also observe that $|B_M| = O(|M|)$. (4) For any LTL formula ϕ, there exists some a Büchi automaton B_ϕ such that $L(B_\phi) = [\![\phi]\!]$. Note that in general there is a one-to-many correspondence between LTL formulas and equivalent Büchi automata. Several approaches exist for computing a satisfactory B_ϕ [44, 45, 46, 47], one example (which our work builds from) is the tableau construction [48]. The best techniques yield automata that are $O(3^{|\phi|})$, where $|\phi|$ is the size of the formula, measured by the number of ϕ's subformulas.

From these results, we have that $M \models \phi$ is equivalent to determining if $L(B_\phi) \subseteq L(B_M)$, which is itself equivalent to determining if $L(B_M) \cap L(B_{\neg\phi}) = \emptyset$. The algorithmic sketch for LTL model checking using this equivalence is: (1) Compute B_M from M and $B_{\neg\phi}$ from ϕ. (2) Compute the Büchi automaton that represents the joint language $L(B_M) \cap L(B_{\neg\phi})$. (3) Determine if this automaton accepts any input. (4) If not, then $M \models \phi$, otherwise $M \not\models \phi$. The overall runtime complexity of

this approach is the aforementioned $2^{O(|\phi|)}O\left(|M|\right)$. The interested reader will note that the dominating factor for computation is the construction of the joint Büchi automaton from B_M and $B_{\neg\phi}$.

3.2.2 CTL* Model Checking

Emerson and Lei observed in 1987 [49] that the problem of CTL* model checking was not too dissimilar from LTL model checking. In particular, they showed that CTL* model checking could be reduced in polynomial time to LTL model checking, which as a consequence of LTL model checking being PSPACE-complete implies CTL* model checking is in PSPACE. A corollary to this result is a usable CTL* model checking algorithm that is computable in time $2^{O(|\phi|)}O\left(|M|^2\right)$. Emerson and Lei [49] as well as Kupferman *et al.* [50] observe that this algorithm actually does extra work, in particular it need not invoke an LTL model checker for every state of M (instead only one execution is required) so the running time is reduced by a factor of $|M|$ to become $2^{O(|\phi|)}O\left(|M|\right)$, which (somewhat surprisingly) is the same running time as a known algorithm for LTL model checking and suggests that CTL* model checking is no harder than LTL model checking. In particular, it also has been shown to be PSPACE-complete [49, 51, 39].

3.2.3 CTL Model Checking

CTL model checking is known to be P-complete [37]. It was shown first by Clarke *et al.* [51] in 1986 and then later by Arnold *et al.* [52] in 1988 to be solvable in time $O(|M| \cdot |\phi|)$. Clarke used a dynamic programming algorithm to determine, for all states q in M and all subformulas ψ of ϕ, if $q \models_M \psi$. Arnold gives a result for the more general case of modal μ-calculus (under which CTL, LTL, and CTL* are all expressible) showing the same upper bound of complexity.

3.3 Association Rule Learning

As indicated in Chapter 2, association rule mining is a technique for the discovery of relationships between items over large datasets of transactions. In his initial work in the area, Agrawal focused on discovering relationships between elements in an individual customer's basket [9]. His follow-up work extended this to discovering sequential patterns over multiple baskets, [10], thereby incorporating a temporal aspect to the pattern. We provide discussion here about the theory behind these rules including the statistical measures used to evaluate if a particular rule is "good" in some formal sense.

We refer to the set of k transactions as the dataset $D = \{\langle t_0, I_0\rangle, \ldots, \langle t_k, I_k\rangle\}$, where each $\langle t_i, I_i\rangle$ is an individual transaction dated with timestamp t_i and itemset $I_i \subseteq \mathcal{I}$ drawn from the global set of total possible items $\mathcal{I}$. In the case of standard

association rule mining we ignore the timestamp, in the case of sequential patterns we make use of it. An association rule is expressed in the form $X \implies y$, for $X \subset \mathcal{I}$ and $i \in \mathcal{I}, y \notin X$. A sequential pattern is expressed in the form

$$X_0 \leadsto X_1 \leadsto X_2 \leadsto \ldots \leadsto X_n \qquad \text{for } X_i \subseteq \mathcal{I}$$

The rest of this section focuses on association rules in particular. Much of the theory developed by Agrawal in his initial work is leveraged when performing the computation for mining sequential patterns, the interested reader can consult [10] if they are interested.

3.3.1 Apriori

The initial work by Agrawal established two criterion for evaluating "good" association rules, which equate to association rules that he suspects are of interest to the user. The first criterion he describes is a syntactic criterion, which imposes restrictions on what items drawn from $\mathcal{I}$ can belong to any itemset being considered. As there are a total of $2^{|\mathcal{I}|}$ distinct itemsets overall, for every item that is prohibited from being part of an association rule the total number of itemsets that must be considers drops naively by a factor of 2.[2] Similarly the number of itemsets under consideration decreases when we require that a particular item or set of items be

[2] We say "naively" here because, as may be expected, practical algorithms usually do not need to explore a large portion of this hypothesis space. There are some cases where we come close, however.

present in a rule. The second criterion imposes restrictions on the statistical significance of a rule as observed in a dataset D. The *support* of an itemset $I \subseteq \mathcal{I}$ over a dataset D of transactions is defined to be the fraction of transactions $T \in D$ such that $I \subseteq T.I$. This corresponds to the fraction of customer transactions that give evidence for an itemset being related. Agrawal used these criteria to derive a search algorithm through the space of all itemsets which satisfied both types of restrictions. The user would specify the syntactic constraint as well as a support threshold which determined which itemsets were considered "large" (of interest/worth reporting). The algorithm, called Apriori, performed a modified BFS through the space of all itemsets, using all large itemsets of size 1 as seed locations. When considering a specific large itemset, the algorithm would then consider all 1-augmentations possible to the itemset (all additions of exactly one item to the set) that would keep the set large. This approach takes advantage of the fact that over the boolean lattice of all itemsets, the support function is monotonic non-increasing (as more items are added to an itemset, the support for that itemset can never increase). All maximal large itemsets are then used to produce association rules. For a given itemset I of

size k, k rules are generated of the form

$$
\begin{aligned}
(x_2, x_3, \ldots, x_k) &\implies x_1 \\
(x_1, x_3, \ldots, x_k) &\implies x_2 \\
&\ldots \\
(x_1, x_2, \ldots, x_{k-1}) &\implies x_k
\end{aligned}
$$

3.3.2 Alternative Statistics for Association Rules

Note that the support of a rule is equal to the support of the itemset formed by taking the union of the antecedants and the consequent of the rule. Following this work, other metrics other have been invented to help evaluate the quality of a rule and serve as inspiration for inventing alternative algorithms for rule discovery. The *confidence* of a rule $X \implies y$ is defined to be $\frac{support(X \cup \{y\})}{support(X)}$. This can be thought of as the conditional probability of having item y in the basket if we know the basket already contains all items from X, namely $Pr\left(y|X\right)$. The *lift* of a rule $X \implies y$ is defined to be $\frac{support(X \cup \{y\})}{support(X)support(\{y\})}$. This corresponds to the "surprise" of seeing X and y jointly, as compared to if they were independently occurring. Finally, the *conviction* of a rule $X \implies y$ is defined to be $\frac{1 - support(\{y\})}{1 - confidence(X \implies y)}$. Using intuition from the above functions, this could be rewritten as $\frac{Pr(\neg y)}{Pr(\neg y|X)}$, namely the ratio of y not occurring in a basket when compared to y not being in the basket given all items from X are present. As with support, it is desirable for all of these functions

to have larger values, and as such an algorithm making use of them could define a lower threshold to determine if a rule is "interesting" or not.[3]

3.3.3 Limitations of Apriori, Alternative Algorithms

One of the largest criticisms of Apriori is that despite its attempt to curtail excess search time in the itemset space, its inherent search technique requires enumerating a large number of itemsets to reach its final solution. In particular, for any maximally large item set X, the algorithm will first enumerate all $2^{|X|} - 1$ subsets of X. This approach, termed *candidate generation*, is clearly more and more undesirable as $\mathcal{I}$ grows in size. Efforts have been made to improve efficiency of candidate generation either by pruning more of the search space [12, 11, 14], and to avoid candidate generation altogether [53].

[3] Many formulations of an association rule since Agrawal's original 1993 work (including his 1995 follow-up) relaxes the constraint that the consequent of an association rule be a single item. Many of these statistical functions were originally defined with the generalized notion of an association rule. We use the original restricted version of a rule in our definition for these functions for consistency; the definitions can easily be adjusted for the general case.

Chapter 4: Model-Based Invariant Extraction from Test Cases

4.1 Introduction

Software development, maintenance and evolution activities are frequently complicated by the lack of accurate and up-to-date requirements specifications. In addition to providing developers with guidance on their design and implementation decisions, good requirements documentation can also give an overview of system purpose and functionality. Such an overview, if accurate, gives maintainers and future development teams a clear snapshot of expected system behavior and can be used to guide and assess system-modification decisions necessitated by bug fixes and upgrades.

Implementations can deviate from their requirements specifications for a number of reasons. Miscommunication among the requirements, design and development teams is one; churn in the requirements is another. So-called implicit requirements can also arise during development, especially with experienced programmers familiar with the problem domain; such programmers may rely on their intuitions about what ought to be required rather than what is actually in the requirements documentation. Regardless of the source, such deviations confound development, main-

tenance and evolution efforts, especially when teams are geographically distributed and possess differing levels of experience about the system domain.

In this chapter, we propose and assess a methodology, based on machine learning, for automatically extracting requirements from executable software artifacts. The motivation of the work is to make requirements documents more accurate and complete. Our approach is intended for use with software following a read-execute-write behavioral model: input variables are assigned values, computations performed, and values written to output variables. The method first uses an automated test-generator, which based on the structure of the model and pre-determined coverage criteria, generates inputs on which the model then executes and produces outputs with each pair of input-output vectors defining a *test*. Machine learning tools are then applied to the set of tests to infer relationships among the input and output variables that remain constant over the entire test set (*invariants*). In a subsequent validation step, an automated validation tool is used to check which of the proposed invariants are indeed invariant; invariants passing this step are then proposed as requirements.

The rest of the chapter is structured as follows. Section 4.2 gives background on data mining, invariant inference, and the artifacts and verification technique used to conduct the studies in this chapter. Section 4.3 then outlines our approach in more detail, while Sections 4.4 and 4.5 present the results of a pilot study involving a production automotive application. Section 4.6 discusses related work, and the

final section contains our conclusions and ending discussion.

4.2 Background

Our work is inspired by Raz *et al.* [28, 54], which used data-mining tools to deduce invariants from the execution traces of running systems for the purposes of anomaly detection. Our motivation differs in that our work is aimed at reconstructing requirements from program test data arising in the context of model-based development of automotive systems. In this section we review the results of Raz et al. and also describe the model-based development environment for automotive software in which the results of our work are assessed. An approach for verifying automotive software models, *Instrumentation-Based Verification* [55], is also briefly described.

4.2.1 Invariant Inference from Executions

Invariants are commonly employed in program verification and express a relation between variables that holds for all executions of a piece of code. For example the invariant $(x > y)$ means that the value of variable x is always greater than the value of variable y. Invariants have a long history in software specification and development, as they define relationships that must hold among program variables even as these variables change values.

The work of Raz *et al.* was motivated by the desire to study the emergent behavior of systems when access to software and other development artifacts for the

systems was impossible. The technical approach taken was to observe input / output sequences at the system interface and to use data-mining tools to infer invariants on the input and output variables. Several such tools are capable of discovering so-called *association rules* from time-series data given to them; these rules take the form of implications involving variables in the data that appear to hold throughout the data set. For example, in a data set recording values at different time instants for two variables, `speed` and `active`, which reflect the vehicle speed and the status (active or not) of a vehicle cruise control, one possible association rule that could be inferred is the following.

```
'speed < 30.0' -> 'active = false'
```

In other words, the data set might support the conclusion that whenever the speed is below 30.0, the cruise control is inactive.

In the case of Raz *et al.*, inferred association rules are viewed as invariants that, if true, yield insight into system behavior. Because the invariants deduced by these tools are only based on a subset of system executions, they may in fact not be invariants when considering the entire system. In Raz *et al.* this issue was addressed by presenting inferred invariants in a template form to an expert, who would use his / her understanding of the system to decide whether these candidate invariants were actual invariants on system behavior or had merely been flagged as invariants by the automated tools based on the characteristics of the analyzed traces. If accepted, this invariant would be then used to build up a model of program execution and then

when anomalous behavior was observed, it would be flagged either as an error or used to update the set of invariants and consequently the model of proper operation [28].

In this chapter, we interpret requirements to be invariants that hold true on all possible runs of the software system. Such requirements can constitute a document of formal properties for the model under inspection (containing properties such as the relationship between inputs, for example), which can serve as the basis for a comparison between the model's observed behavior and its intended behavior. This may reveal inadequacies in the design, or perhaps uncover hidden invariants that were not intended to be present.

Unlike [28], whose primary goal was anomaly detection, our primary aim is to efficiently identify a complete set of invariants that are true requirements with minimal and targeted effort for the expert. This includes deducing previously unknown implicit requirements, eliminating candidate invariants that are not requirements, and demonstrating that our procedure is robust in that multiple execution runs will produce nearly-identical results, thus avoiding potential challenges of resolving different mined invariant sets to reach some sense of a consensus.

4.2.2 Automotive Model-Based Development

The work in this chapter grew out of a project devoted to improving the efficiency of software development processes for automotive applications. The pilot study in particular involves an external-lighting control feature in a Bosch production

application. As automotive software is increasingly developed using model-based development, the software-artifact analyzed in later takes the form of a model in the MATLAB® / Simulink® / Stateflow®[1] modeling notation. This section discusses some of the uses of such models in the automotive industry.

Modern automobiles contain significant amounts of software. One estimate put the average amount of source code in high-end models at 100 million lines of code, with the amount growing by an order of magnitude on average every decade [56]. At the same time, the business importance of software is growing, with new (and profitable) vehicle features increasingly relying on software for their functionality.

For these reasons, automotive companies, and their suppliers such as Bosch, have strong incentives to improve the efficiency of their software development processes. At the same time, safety, warranty, recall and liability concerns also require that this software be of high quality and dependability.

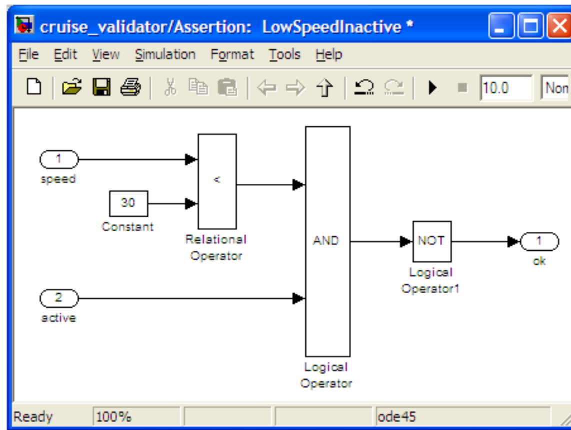


Figure 4.1: Sample Simulink model.

[1]MATLAB®, Simulink® and Stateflow® are trademarks of The MathWorks, Inc.

One approach that is garnering rapidly growing acceptance in the industry is *model-based development* (MBD). In MBD traditional specification and design documents are supplemented with executable models in notations such as MATLAB / Simulink / Stateflow or ASCET®[2] that precisely define the expected behavior of eventual software and system implementations. These models are often developed upstream of the software-development teams by controls engineers, and the notations are often based on block-diagram notations favored by members of the controls community.

Figure 4.1 gives an example Simulink model. From a program-ming-language perspective, Simulink (and its Statecharts-like sub-language, Stateflow) may be seen as a synchronous dataflow language. At time instances determined by the so-called sampling rate of the system, inputs (ovals with no incoming edges) are read, and values transmitted to blocks. Each block computes a function on its inputs when all have arrived and outputs the results. Model outputs (ovals with no outgoing edges) store the final results of this staged computation. In keeping with the synchrony assumption, it is assumed that the calculation of these model outputs is instantaneous with the arrival of inputs.

Because Simulink and related models are executable, they may be simulated and debugged, and they may also be used as test oracles for downstream software development. They are also increasingly used to drive the automatic generation of source code; so-called *autocoding* tools such as TargetLink from dSPACE and

[2] ASCET® is a trademark of the ETAS Group.

Embedded Coder from The MathWorks produce ready-to-deploy code in many instances directly from models. As the gap between design models and implementation narrows, design models also become increasingly attractive as test oracles.

4.2.3 Instrumentation-Based Verification

Because of the centrality of models in model-based development processes, it is important that they behave correctly, i.e. in accordance with functional requirements specified for them. Several model-checking tools, including commercial ones such as The MathWorks' DesignVerifier, have been developed for this purpose. Such tools take models to be verified (we call these *design models* in what follows, because they are often the outputs of design processes) and requirements specifications, typically in a temporal logic, and attempt to prove automatically that model behavior conforms to the requirements.

A related approach, called *Instrumentation-Based Verification* (IBV) [55], advocates the formalization of requirements instead as so-called *monitor models* in the same modeling notation used for the other models in the MBD process. Each discrete requirement has its own monitor model, whose purpose is to monitor the data flowing through the design model and determine if the associated requirement is being violated or not via a boolean-valued output. (For example, the Simulink model in Fig. 4.1 is the monitor model associated with the association rule given in Section 4.2.1.) The design model is then instrumented with the monitor models,

and structural-coverage testing performed to determine if any monitor models can report an error. The advantages of IBV are that a separate notation for formalizing requirements need not be learned; that monitor models can be executed and debugged; that the monitor models (and the requirements they express) are likely to be updated with the design models, and that testing-based approaches scale better than model checkers. The disadvantage is that IBV cannot produce the iron-clad guarantees of correctness that model checkers can when the latter do indeed terminate. Commercial tools like Reactis®[3] provide support for IBV by supporting the instrumentation process and automating the generation of test suites that maximize coverage of models.

4.2.4 Reactis

The experimental work described later in this chapter makes heavy use of the aforementioned Reactis tool, so this section gives more detail about it.

Reactis is a model-based testing tool. Given an open-loop model (i.e. one with unconnected inputs and outputs) in the MathWorks' Simulink / Stateflow notation, Reactis generates test cases in the form of sequences of input vectors for the model. The goal of the generated tests is to provide full coverage of the model according to different model-based adaptations of structural coverage criteria. In general, for reasons of undecidability, full coverage cannot be guaranteed; Reactis thus uses

[3]Reactis® is a registered trademark of Reactive Systems, Inc. (RSI). In the interest of full disclosure, one of the authors is a co-founder of this company.

different heuristics in to try to maximize the coverage of the test cases it creates. The tool also evaluates the model while it is constructing the test suites and stores the model-generated output vectors in the test cases.

Reactis additionally provides support for the Instrumen-tation-Based Verification (IBV) technique mentioned in the previous subsection. To use this feature of Reactis, a user first creates a Simulink library containing the monitor models for the requirements of interest. S/he then uses features in Reactis to instrument the model to be verified with the monitor models. The Reactis test generator subsequently is used to generate test cases that cover the instrumented model, including the constructs contained in the monitor models. While the tests are being constructed Reactis also evaluates the monitor model outputs, and if any reports "false" then the resulting test is evidence that a requirement is violated. The structural coverage criteria guarantee that the test generator will attempt to generate tests that cause outputs of "false" from the monitor models because covering boolean variables entails ensuring the variable contains each value at some point in some test.

The Reactis test-generation algorithm employs a three-phase approach. In the first phase, which is optional, pre-existing test cases may be loaded for inclusion in the test suite being constructed. In the second phase, a collection of random test cases is generated using Monte Carlo simulation in order to create an initial collection of tests. User-provided parameters govern how many random tests are added to the test suite. The third phase uses a collection of heuristics to extend

and modify tests already in the test suite in order to cover uncovered parts of the models logic. The core of the technique is covered by US Patent #7,644,398.

It is important to note that Reactis test suites include randomly generated test cases that are subsequently refined. For this reason, different executions of the tool, even on the same model, will in general yield very different test suites.

4.3 Extracting Requirements

We propose an approach to inferring requirements from executable software artifacts and to provide a preliminary evaluation for this approach. The specific setting in which we study this question is model-based development for automotive systems; in particular, our interest is in taking Simulink models and producing requirements from these models, so that they may be compared with existing requirements specifications so that gaps and inconsistencies in the latter may be identified.

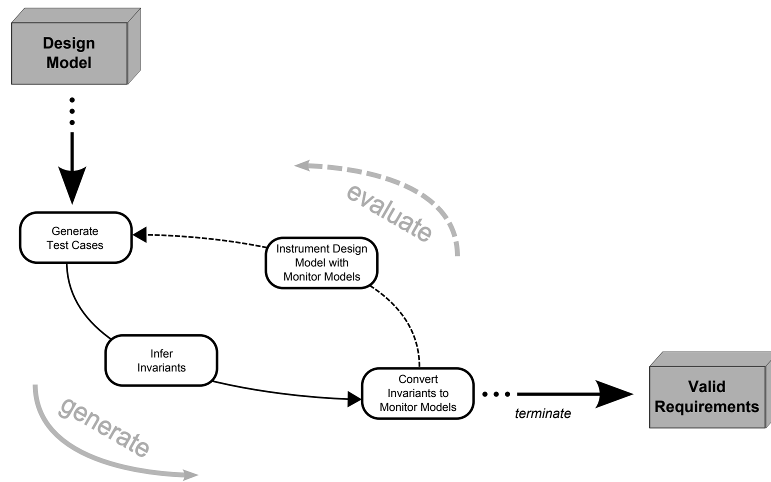


Figure 4.2: Overview of requirements-extraction process.

The steps in our methodology are illustrated in Figure 4.2. As is the case with Raz *et al.* [28], our approach relies on the use of data mining tools to propose invariants from test data generated from simulated execution runs of the model. However, our approach differs in several key aspects.

- As our interest is in requirements generation, we wish to use structurally thorough test suites as a basis for data mining, where Raz *et al.* does not. Because we also have access to the entire model, we can use coverage criteria as a basis for determining how *complete* the test data is.
- We use Instrumentation-Based Verification (IBV) to automate the assessment of the candidate invariants that are produced, rather than relying on human experts to make this determination.

Because we use coverage testing to generate the tests, we hypothesize that the invariants that are proposed are more likely to be actual requirements. Because we automate the invariant-assessment task, we also expect that the task of generating and assessing invariants will be less labor-intensive. These points are investigated in more depth in the next section. Our requirements-extraction strategy involves the following large steps; a more detailed division is given in Section 4.4.2.

Step 1: Generate Test Cases. Test data is generated from design models by running a sequence of generated inputs on the design models using automated, coverage-based test-generation tools (in our case, Reactis).

An invariant for a test suite expresses a constraint that holds for all inputs and outputs of that test suite. If the test suites only execute a certain part of the system or the model, the invariants that are inferred from these test cases might only hold true for just that part but not for others.

We have addressed this issue by using notions of structural coverage for automatic test generation. The heuristics that our tool for automated test generation uses are based on covering as much of the model's structure as possible, using metrics adapted to the model setting from traditional source-code coverage frameworks. In other words, our test suites are chosen so that every block is executed at least once and every condition and decision evaluated to true and false each at least once.

Step 2: Invariant Inference. Given the test suites obtained in step 1, invariants are discovered using association rule mining. This putative rule set constitutes the set of association rules that are currently suspected to be invariants of the model under inspection. Under some circumstances a rule may be found that is subsumed by another rule, in these cases we omit reporting the more specific one.

Invariant inference algorithms can be adjusted through a number of parameters/restrictions, which result in different invariant sets being reported. We require here that all invariants reported have a *strength* value equal to 1.0, the maximum value a strength score can take [57]. This equivalently means that any invariants reported have no counter examples in the observed data, namely it is a true invariant for the particular set of test cases for which the inference process is performed over.

Other metrics also exist for scoring association rules, such metrics include confidence, support and lift [57]. Many of these are a function of the association rule's abundance in the observed data; this is a trait that we wish to avoid because we do not wish to bias our results by reporting only the more common invariants.

Step 3: Validating Invariants. Model-coverage metrics merely assure that critical elements (blocks, conditions, decisions, etc.) have been executed at least once in a test-suite. They of course cannot enforce full coverage of all possible behavior (i.e. path coverage). This is why the inferred invariants cannot be assumed to be true requirements, and must be further validated in order to be reported as such.

In Raz *et al.*'s approach, this validation was carried out by a human agent who, based on his domain knowledge, made a decision as to whether the invariants were actually requirements or not. We automate this validation step by converting each candidate requirement into a monitor model and use IBV to determine if the proposed invariant can be invalidated. This is done by instrumenting the design model with the constructed monitor models, and running a technique supporting IBV (again, in our case, Reactis implements such a technique) to check if the design model satisfies the monitor model (i.e. is it possible for the monitor model to output *false* while executing the design model). While performing this verification, the validation tool will generate a new test-suite that will be driven by two heuristics: 1) provide maximum model coverage, and 2) attempt to violate the monitor model. As a result of the second heuristic, the test suite that will now be generated will be

different from the original test suite that was used to infer the invariant captured by the monitor model.

This instrumentation is done in some sense parallel to the execution of the original model; for a given execution the values on the input ports are effectively copied and fed into each monitor model. This prevents the introduced, instrumented code to otherwise affect the base design model's standard execution path.

If the monitor model is not satisfied, then the putative invariant that it represents does not hold true for all traces and is thus discarded. If the monitor model is satisfied, then we can say, with a high level of confidence, that a valid invariant has been inferred from the design model and can be seen as a requirement.

The validation step taken also allows one to easily validate any existing invariants that are presented initially along with the design model; one can convert these initial invariants into monitor models to perform immediate validation using the same framework as described above. The confidence level of these initial invariants does not affect the strategy here; in either case one can immediately use the initial invariants as monitor models. If they are high confidence, then they should not be refuted by any test cases generated.

As indicated in Figure 4.2, our technique can be iterated. By the nature of IBV, the extra step of validating the invariants involves instrumenting the original design model with monitor model representations of the invariants themselves. The result of this instrumentation is a well-defined design model which has at its core

the original design model whose behavior has not been altered due to the manner in which the instrumentation takes place. Thus, we can repeat steps 1-3 on this new, instrumented model and obtain a different set of invariants. One would suspect that this set is in some ways similar to the set obtained in the first iteration, our empirical results shown in Section 4.5 shows that this intuition is accurate in the cases we consider.

4.4 Experimental Configuration

We evaluate the requirements extraction process in the previous section using an automotive application pilot study. This section details the experimental set-up used, including the application and tool chain used to implement the steps of the procedure, the specific questions studied, and the analysis framework for assessing the results. The section following then reports the results themselves.

4.4.1 Test Application

The model used to evaluate our framework is a Simulink diagram encoding the design of an automotive software function. The design of the model was taken from existing C source code developed by Bosch. The model consists of approximately 75 blocks and has two inputs and two outputs. Existing documentation was present that described, among other things, a state machine conceptualization of the C code. The challenge of requirements extraction corresponds to inferring valid edges

between the set of states on this machine. Over this state machine, which is known to contain 9 states, there are a total of 42 known transitions, which we refer to as the total number of invariants that can be discovered. We refer to this automotive model as $\mathcal{D}$ in the following sections.

4.4.2 Tool Chain

The specific tasks that need to be performed in order to implement our requirements-extraction approach include: (1) generation of full-coverage test suites from $\mathcal{D}$; (2) production of proposed invariants from test data; (3) creation of monitor models from invariants; (4) instrumentation of $\mathcal{D}$ with monitor models; (5) generation of coverage test-suites from instrumented $\mathcal{D}$.

As indicated in Section 4.2, the Reactis tool generates high-coverage test suites from Simulink models and also supports the instrumentation of models with monitor models and subsequent validation testing. Thus, this tool was used for tasks 1 and 5.

To generate invariants (task 2), we used the Magnum Opus data-mining tool [14], mainly based on its relative ease of use and its ability to generate so-called *association rules* (i.e. invariants in the form of implications) efficiently. Magnum Opus is capable of inferring invariants whose individual terms are exactly assignments (=) to properties, not ranges (< or >). This is sufficient for our task, as the inputs and outputs for the models studied have nominal domains, i.e. they are not ordered in

any particular manner, and thus range assertions make little sense. Specifically, we discover rules of the form

$$\bigwedge a = a_i \longrightarrow \bigwedge b = b_i,$$

namely where the premise and consequent are both conjunctions of terms which are ground-out variables. Magnum Opus may seem restrictive in some cases, especially when variables with ordinal domains become involved. For our model under test, all the variables under consideration are nominal - ranges in our model have no semantic meaning to them and thus any invariant involving one can be expressed equivalently as a set of invariants containing only equality without the loss of any semantic meaning. Because we have existing ground truth data regarding $\mathcal{D}$ in the form of a state machine, we rewrite all invariants discovered to contain state information on both the premise and the consequent of rule. When considering invariants of this form, there is a one-to-one correspondence with state transitions on the state machine. This allows us to easily check what rules are recovered and what are not. For example, a rule such as `'button=pressed' -> 'new_state=2'` would be expanded to the set of rules

```
'state=1' & 'button=pressed' -> 'new_state=2'
'state=2' & 'button=pressed' -> 'new_state=2'
'state=3' & 'button=pressed' -> 'new_state=2'
```

if the state variable `state` had three possible values, `1`, `2`, and `3`.

To streamline the process, we also wrote scripts that: translate Reactis-generated test data into the Magnum Opus format; automatically translate Magnum Opus association rules into monitor models; and automatically create the file Reactis uses to wire the resulting monitor models into the design model $\mathcal{D}$ (tasks 3 and 4). The result is a fully automated system that requires no intermediate involvement by a human.

To use the resulting tool-chain, a user first runs Reactis on $\mathcal{D}$ to create a test suite (set of sequences of input/output vectors). The suite is then automatically translated into Magnum Opus format, and that tool then run infer invariants. Another conversion transforms these invariants into monitor models, along with the proper information for wiring monitor-model inputs into $\mathcal{D}$. Finally, the user runs Reactis a second time on the instrumented $\mathcal{D}$ ($\mathcal{D}$ + monitor models); Reactis creates a second test suite that attempts to cover the instrumented model (and also tries to invalidate the monitor models), reporting when it terminates which monitor models were found to be violated in the second round of testing. Violated monitor models correspond to invariants that are in fact not valid invariants and thus should not be considered requirements.

Note that, as discussed in Section 4.3, this process can be iterated. Furthermore, because the test suite created during the validation phase of the monitor models is constructed using the same heuristics as that of the standard test suites, it can be used as the basis for a second round of invariant inference and by being

combined with the first round's data. Because the second batch of tests includes any counterexamples that where constructed to invalidate some of the invariants generated from the first batch of tests, the already-violated invariants will not reappear in subsequent iterations of this procedure due to our criterion that proposed invariants must satisfy all test data known at the time.

4.4.3 Structural vs. Random Testing

One hypothesis we wish to test in our experiments is that using full-coverage tests as a basis for invariant inference yields better invariants than tests that do not have coverage guarantees (such as those used by Raz *et al.*). We quantify the notion of "better" in two dimensions: how *accurate* are the invariants (i.e. what proportion of a set of proposed invariants are found to be valid in the validation-testing phase), and how *complete* are they (what proportion of the total set of invariants, which are known from the requirements documentation, are generated).

To conduct this assessment empirically, we first produce a test suite guaranteeing maximal coverage. The resulting test suite is then mined for invariants. Finally, we validate these proposed invariants by encoding them as monitor models and generating new test cases with these asserted monitor models present on top of the original design model. This experimental configuration, which we call E_{full}, is performed five times in isolation to each other, to increase statistical confidence in the results.

As a baseline comparison, we then generate a suite of test cases randomly with no structural constraints imposed. In the exact same way as E_{full}, this test suite is then mined for invariants which are then converted into monitor models and validated using Reactis. We refer to this configuration as $E_{partial}$, indicating that full coverage is not guaranteed for the design model. As before for E_{full}, five separate experiments of configuration $E_{partial}$ are performed.

We hypothesize that because coverage is complete for E_{full} runs and incomplete for those generated by $E_{partial}$ runs, one expects less of the state space of the design model to be covered, and thus that E_{full} runs will generate more accurate and more complete invariant sets than those generated by random testing. This can be measured by inspecting the total number of valid invariants that each process produces, which will be described in Section 4.5. Along with this total, the number of invalid invariants can also be considered. Because we cover more variation of the state space in our test cases, we expect fewer spurious invariants to be inferred by E_{full} than by $E_{partial}$ during the entire process.

However, the number of invalid invariants is not as essential for someone investigating the overall performance of our system, as putative invariants that are invalid will be detected automatically by our framework during the subsequent validation stage. The fraction of known requirements that are generated is a better measurement in this respect. In this pilot study, we have access to these known requirements. However, if we did not, we could measure the reproducibility of the

output, i.e. the final set of proposed invariants. Put another way, any run should be fairly similar to any other run within the same configuration. Intuitively, this is not an expected property of the $E_{partial}$ experiments, but it is desirable for E_{full}. To assess how similar any particular invariant set is to another, we use set a set-similarity statistic, specifically the Jaccard coefficient [58]. Often used in clustering and other applications where similarity scores are needed [59], this metric is a measurement of the overlap of two sets, with a score of 0 (lowest) signifying no overlap, and a score of 1 (highest) signifying set equivalence. We compute these similarity scores between all pairs of runs within each configuration. We expect to observe a higher similarity between pairs of individual E_{full} experiments than the similarity between pairs of individual $E_{partial}$ experiments. If each individual run does produce a roughly complete set of invariants, then the Jaccard similarities should approach 1.

4.4.4 Invariant Refinement through Iteration

The second hypothesis we wish to test is that iterating our procedure produces more accurate and complete sets of invariants. To assess this, following the validation phase of each of the previous experiments, we perform the entire process again, accumulating the test cases produced in the first iteration. For E_{full}, we use as a new test suite the suite generated during the previous validation step together with the original test suite used for invariant-generation. For $E_{partial}$, we generate

another suite of randomly selected test cases, as we wish to maintain the property that $E_{partial}$ test suites have no structural guidance. We shall notationally refer to the configuration and results of the E_{full} experiments after only one iteration as $E^{(1)}_{full}$ (which corresponds to exactly the configuration discussed in Section 4.4.3), and results of these experiments extending over two iterations to be $E^{(2)}_{full}$. A similar scheme applies to $E_{partial}$, where we refer to $E^{(1)}_{partial}$ and $E^{(2)}_{partial}$. It should be noted that although the validation phases for $E^{(2)}_{partial}$ involves generating test suites guided by coverage criterion, we discard this when producing new data for the second iteration, as we wish to preserve the "coverage-blindness" of the $E_{partial}$ test suites.

As before, each run using one of the second iteration configurations are repeated five times. For further analysis, we again report the number of valid invariants mined, as well as the pairwise Jaccard similarity measurements between experimental runs in belonging to the same configuration. We expect that the number of valid invariants to increase from results in $E^{(1)}_{full}$ to $E^{(2)}_{full}$. This is expected due to the increase in the amount of testing, in particular because the newly introduced test cases for the second iteration can potentially include counterexamples and other new portions of the state space that were not well represented in the first iteration. For this reason, the number of valid invariants detected by $E^{(2)}_{partial}$ is also expected to exceed the number found by $E^{(1)}_{partial}$, but again because no structural guidance is given, the likelihood of counter examples and other unexplored portions of the state machine being encountered is lower, so the increase should not be as significant.

4.5 Experimental Results

This section presents the results of our empirical study on $\mathcal{D}$. Table 4.1 shows the results of running $E^{(1)}_{full}$ and $E^{(2)}_{full}$ experiments, while Table 4.2 shows the results of experiments using $E^{(1)}_{partial}$ and $E^{(2)}_{partial}$ configurations.

	$E^{(1)}_{full}$					$E^{(2)}_{full}$				
Run #	Putative	Invalid	Net	Acc.	Comp.	Putative	Invalid	Net	Acc.	Comp.
1	26	8	18	0.69	0.43	40	1	39	0.97	0.93
2	34	6	28	0.82	0.67	40	2	38	0.95	0.90
3	30	9	21	0.70	0.50	38	1	37	0.97	0.88
4	33	7	26	0.79	0.62	42	1	41	0.98	0.98
5	34	7	27	0.79	0.64	38	0	38	1.00	0.90
Avg	31.4	7.4	24.0	0.76	0.57	39.6	1.0	38.6	0.97	0.92

Table 4.1: Results from $E^{(1)}_{full}$ and $E^{(2)}_{full}$. The "Putative" columns reports the total number of potential invariants mined directly after the inference phase, but before validation. "Invalid" reports the number of invariants that were then found to be spurious from the validation phase. "Net" reports the number of remaining, validated invariants. "Acc." is the accuracy ratio, i.e. the ratio of "Net" to "Putative". "Comp." is the completeness ratio, i.e. the ratio of "Net" to the total number of 42 invariants contained in the original state-machine specification. The average of each column is reported in the last row.

	$E^{(1)}_{partial}$					$E^{(2)}_{partial}$				
Run #	Putative	Invalid	Net	Acc.	Comp.	Putative	Invalid	Net	Acc.	Comp.
1	19	11	8	0.42	0.19	29	13	16	0.55	0.38
2	22	11	11	0.50	0.26	27	10	17	0.63	0.40
3	26	12	14	0.54	0.33	34	9	25	0.74	0.60
4	26	13	13	0.50	0.31	32	15	17	0.53	0.40
5	25	6	19	0.76	0.45	35	3	32	0.91	0.76
Avg	23.6	10.6	13.0	0.54	0.31	31.4	10.0	21.4	0.67	0.51

Table 4.2: Results from $E^{(1)}_{partial}$ and $E^{(2)}_{partial}$. The meaning of each column is the same as in Table 4.1.

The data in the tables supports both hypotheses made in Sections 4.4.3 and 4.4.4. In particular, in the first iteration of the structural-coverage method, the accuracy

ratios (ratio of proposed invariants that the validation step determines are indeed invariant) are in the range $0.69 - 0.82$, with an average over the 5 runs of 0.76; the corresponding figures for the first iteration of the randomly generated method are $0.42 - 0.76$, with an average of 0.54. In other words, about $\frac{3}{4}$ of the invariants inferred from full-coverage test data are valid in the first iteration, on average, while only just over $\frac{1}{2}$ are using randomly generated data. The differences in completeness (ratio of net invariants to total number of known invariants, based on requirements documentation) is also pronounced, with coverage test-data yielding numbers in the range $0.43 - 0.67$ for an average of 0.57 and random test data producing results in the range $0.19 - 0.45$ with an average of 0.31. That is, structural-coverage test data in the first iteration yields about half of the known invariants, on average, while random test data yields less than one-third.

These differences persist and are accentuated when the results of the second iteration are considered. In the structural-coverage case (Table 4.1) the accuracy and completeness ratios rise to 0.97 and 0.92, respectively, while in the random case the corresponding figures are 0.67 and 0.51. In other words, structural-coverage test data yields a negligible number of incorrect invariants and infers 92% of the total possible invariants, while one-third of the invariants produced from random test data are determined to be invalid in the second iteration and just over one-half of known invariants are discovered.

The data in these tables also supports the second hypothesis: that iteration

of the process yields more accurate and more complete sets of invariants. In the structural-coverage case, the average accuracy ratio increases from 0.76 to 0.97, and the average completeness ratio rises from 0.57 to 0.92. The corresponding figures for the random-test case show a similar (but less substantial) improvement: from 0.54 to 0.67 (accuracy), and from 0.31 to 0.51 (completeness).

Table 4.3 and Table 4.4 present the Jaccard pairwise similarities between individual runs of the same type. The average Jaccard similarity for $E^{(1)}_{partial}$ is 0.51, and it increases to 0.58 when a second iteration is added in $E^{(2)}_{partial}$. However, the structurally-guided coverage testing shows better results. The average for $E^{(1)}_{full}$ is 0.65, which increases to 0.87 when adding a second iteration. These findings, coupled with the completeness-ratio results from Table 4.1 and Table 4.2, support our hypothesis, showing that randomly guided test cases lead to both fewer invariants being detected, as well as high variation in the set of those that are detected, when compared to test cases generated using some coverage criterion as a guide. This gives credence to utilizing an iterative, structurally-guided approach when inferring invariants.

Regarding the effort needed to conduct these experiments, space limitations prevent us from reporting fully. However, it should be noted that no run of Reactis or Magnum Opus ever required more than 3.5 minutes to complete, and that the computing platforms used were commercial laptops. The data suggest obvious time savings for validating invariants over manual inspection by an expert user.

$E^{(1)}_{full}$	**1**	**2**	**3**	**4**	**5**
1	1	0.53	0.86	0.52	0.67
2		1	0.63	0.64	0.72
3			1	0.62	0.71
4				1	0.61
5					1

$E^{(2)}_{full}$	**1**	**2**	**3**	**4**	**5**
1	1	0.88	0.85	0.90	0.83
2		1	0.92	0.88	0.81
3			1	0.86	0.83
4				1	0.88
5					1

	Min	Avg	Max
$E^{(1)}_{full}$	0.52	0.65	0.87
$E^{(2)}_{full}$	0.81	0.87	0.92

Table 4.3: Jaccard similarity scores for $E^{(1)}_{full}$ and $E^{(2)}_{full}$. The minimum, average, and maximum values are also given for $E^{(1)}_{full}$ and $E^{(2)}_{full}$ for summarization.

$E^{(1)}_{partial}$	**1**	**2**	**3**	**4**	**5**
1	1	0.46	0.47	0.62	0.35
2		1	0.47	0.60	0.58
3			1	0.59	0.43
4				1	0.52
5					1

$E^{(2)}_{partial}$	**1**	**2**	**3**	**4**	**5**
1	1	0.74	0.52	0.74	0.50
2		1	0.45	0.70	0.48
3			1	0.56	0.63
4				1	0.48
5					1

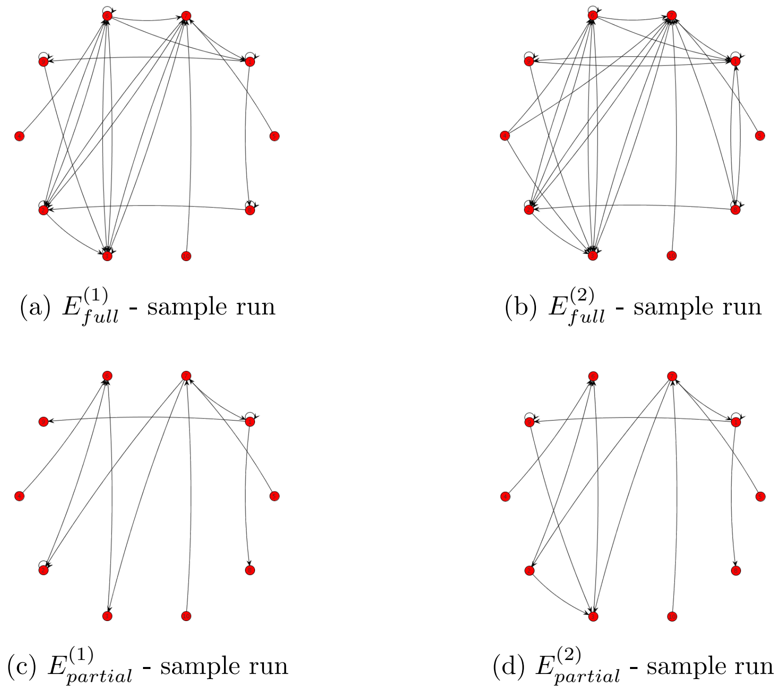
	Min	Avg	Max
$E^{(1)}_{partial}$	0.35	0.51	0.62
$E^{(2)}_{partial}$	0.45	0.58	0.74

Table 4.4: Jaccard similarity scores for $E^{(1)}_{partial}$ and $E^{(2)}_{partial}$, with accompanying summarized statistics.

State machine visualizations are shown in Figure 4.3 for representative invariant sets of each of the four configurations, $E^{(1)}_{full}$, $E^{(2)}_{full}$, $E^{(1)}_{partial}$, and $E^{(2)}_{partial}$.

4.6 Related Work

In *specification mining* [60, 61, 62, 63, 64, 65, 66], the interaction behavior of running programs is extracted [67] by machine learning algorithms [68] wherein a state machine, that is supposed to represent a model of the program's specification, is con-



(a) $E^{(1)}_{full}$ - sample run

(b) $E^{(2)}_{full}$ - sample run

(c) $E^{(1)}_{partial}$ - sample run

(d) $E^{(2)}_{partial}$ - sample run

Figure 4.3: State machine visualization of representative invariant sets for each of the 4 configurations. The specific invariant set chosen for each configuration here has the median number of net invariants discovered, as reported in Table 4.1. Edge labels, which would correspond to specific requirements of inputs, are omitted for clarity.

structed and which can subsequently be analyzed using a variety of static techniques [69]. Our approach, in contrast, concentrates on deriving individual requirements rather than constructing a total specification of the system, though we also provide the facility to visualize the full specification from the requirements. In addition, our approach is distinguished by its provisions of guarantees regarding coverage of behavior. With respect to invariant detection, we use Magnum Opus for its easy setup and its support for association rules but there are several other tools which can mine rules from dynamic systems.

The Weka project[70] is a toolkit that is capable of performing data mining-related tasks, including the Apriori algorithm [9], one of the earliest algorithms used for association rule mining. The particular types of association rules mined are similar to Magnum Opus in the sense that they do not support invariants involving ranges.

The Daikon system proposed by Ernst *et al.* [22] performs dynamic detection of "likely invariants." It focus primarily on programs written in C, C++, Java, and Perl. An early step of Daikon's system uses code instrumenters to obtain trace data which is passed into its inference algorithm. This trace data does not guarantee good coverage over the program under inspection; Ernst *et al.* note that multiple runs may be required by the program under inspection and combined. In general, Daikon does not support a mechanism for generating such a set of high-coverage runs itself, this is left to the user to determine. The invariants proposed by Daikon are checked for redundancy, but it is difficult to validate invariant correctness for test cases other than those given by the presented trace data. Through the use of Reactis, our framework only will produce test cases with a specified coverage level (able to be set by the user). Furthermore, by converting the existing invariants into monitor models and validating in an iterative process, we are able to provide stronger assurances in the correctness of our final invariant set.

Hangal *et al.*'s IODINE framework [71] dynamically mines low-level invariants on hardware designs. Rather than employ a machine learning approach, they uti-

lize a series of *analyzers* that monitor signals in the hardware design and reports observations that are of interest, such as equality between signals, mutual exclusion between signals, etc. These analyzers report what are termed invariants. In contrast to our work, the invariant types that are discoverable are determined by the analyzers selected or considered to be used, which can be difficult to identify if one is searching for invariants that are either unknown or otherwise not considered. In our case, test cases are generated through the same general framework regardless (using structural coverage), and the data mining tool employed is the determining factor of what types of invariants are discovered.

Mesbah *et al.* [72] proposed a method of automatic testing of the user interfaces of AJAX-based applications. Their approach reveals invariants in the DOM-tree of an AJAX application as well as constructs a state machine of the application, over which other invariants (which they equate to requirements) are identified. The notion of differing states in their context corresponds to the various paths of action events that can be taken through user interactions such as button clicks. Their work focuses on using these invariants to detecting faults for testing purposes, rather than attempting to construct a well-covered set of invariants which corresponds to a largely complete view of the state machine, as our work does.

Cheng *et al.* [73] use data mining techniques to extract putative invariants over a program's dynamic execution, and augment an existing bounded model checker that uses SAT formulations of the code statically. This approach is shown to speed

up software verification when compared to performing bounded model checking without the invariants obtained from the data mining. The main focus of Cheng *et al.* is to accelerate the software verification process by adding invariants found during program execution. Our work is driven to infer a largely complete set of such invariants that could characterize the model itself over the variety of dynamic executions that could possibly occur.

4.7 Conclusion

This chapter has presented a framework for requirements reconstruction from executable software artifacts. The kinds of systems it targets follow a read-compute-write behavior model, with inputs being read, computations conducted, and results written. Control software, such as that found in embedded applications, represents a class of applications that fall into this classification, as do others. The method relies on the application of machine-learning / data-mining techniques to test data, in the form of sequences of input-output vectors that structurally cover the artifact, obtained for the software to derive proposed requirements in the form of invariants expressing relationships between inputs and outputs. The method then uses an automated validation step to identify spurious invariants. The method was piloted on an automotive lighting-control application in which the artifact in question was a Simulink model. The experimental data suggest that using full-coverage test data yields better invariant / requirement sets than random test data, and that itera-

tively applying the approach further improves these invariants. In particular, using a single iteration with random samples showed only a 31% recovery of all known invariants, whereas this number is increased to 57% with the application of structural coverage, and further increased to 92% when a second iteration is performed.

The findings of this chapter was presented at Runtime Verification in 2010 [1].

Chapter 5: LTL Query Checking

5.1 Introduction

Temporal logics [74] are widely used to specify desired properties of system behavior. Such logics permit the description of how systems should execute over time; tools such as model checkers [75, 76] can then be used automatically to determine whether or not certain types of system possess given temporal properties.

The practical utility of model checking and other temporal-logic-based verification technologies relies on the ability of users to define correctly the properties they are interested in. To assist users in this regard, researchers have looked into various forms of automated *temporal-property reconstruction* [77, 78, 79, 80] as a means of helping users to devise temporal specifications from given system specifications. Users may then use these as specifications for the system (useful when systems subsequently have new functionality added, as the new system can be checked against the old specification to ensure backward compatibility); they may also review them as a means of gaining insight into the behavior of a system that may not have been formally specified or verified. One of the most influential lines of work in this

area is so-called *temporal logic query checking* by William Chan [16], which aims to solve the following general problem: given a system, and a temporal formula with a missing (propositional) subformula, "solve" for the missing subformula. As originally formulated by Chan, the temporal logic in question was a subset of the branching-time temporal logic CTL [81], for which he gave efficient algorithms for computing most-precise missing formulas. Others have considered different variants of this problem, by considering multiple missing subformulas, for instance, or different logics [18, 21, 19].

In this chapter we consider the problem of *query checking for Linear Temporal Logic* (LTL) [81]. LTL differs from branching-time logics in that one specifies properties of executions, rather than states in a system, and it is often viewed as an easier formalism to master for this reason. It is also the basis for specification languages, such as FORSPEC [82], used in digital hardware design. We show how to adapt automaton-based model-checking techniques in order to yield a solution to the query-checking problem that, while computationally complex in the worst case, exploits structure in the space of possible query solutions to yield better performance. We first develop some needed mathematical preliminaries and then present our technique and report on empirical results from our existing implementation.

5.2 Büchi Propositional Automaton

For the remainder of this chapter we will employ an alternative formulation of Büchi automata than that presented in Chapter 3. Let B be a Büchi automaton whose alphabet is the power set of some other set, i.e. $B = \langle S, 2^{\mathcal{AP}}, \delta, s_i, F \rangle$. Then, transitions in B are labeled by subsets of $\mathcal{AP}$, which can be represented as a propositional formula over $\mathcal{AP}$. Such a formula would correspond to exactly one line in a truth table containing possible valuations of the atomic propositions in $\mathcal{AP}$. For a propositional formula $\gamma \in \Gamma^{\mathcal{AP}}$, the interpretation of the transition $s \xrightarrow{\gamma} s'$ is that $s \xrightarrow{A} s'$ for every $A \subseteq \mathcal{AP}$ satisfying γ.

Definition 7 (Satisfying propositional formulas with $\mathcal{AP}$)**.** *For atomic proposition set $\mathcal{AP}$, define $A \models \gamma$ for $A \subseteq \mathcal{AP}$ and $\gamma \in \Gamma^{\mathcal{AP}}$ inductively as follows:*

- $A \models a$ *iff* $a \in A$
- $A \models \neg\gamma$ *iff* $A \not\models \gamma$
- $A \models \gamma_1 \vee \gamma_2$ *iff* $A \models \gamma_1$ *or* $A \models \gamma_2$

We write $[\![\gamma]\!]$ *for* $\{A \subseteq \mathcal{AP} : A \models \gamma\}$.

Note that $A \models \gamma$ if and only if $\pi \models \gamma$ for all π such that $\pi[0] = A$. A set $A \subseteq \mathcal{AP}$ can be expressed as a propositional formula over $\mathcal{AP}$ with the formula $(\bigwedge_{a \in A} a) \wedge (\bigwedge_{a \notin A} \neg a)$. Note that $[\![A]\!] = \{A\}$ in this case.

Definition 8 (Büchi propositional automaton)**.** *Given* $\mathcal{AP}$*, a* Büchi propositional automaton *is a 4-tuple* $\langle S, \delta, s_I, F\rangle$*, where:*

- S *is a finite non-empty set of states, with* $s_I \in S$ *and* $F \subseteq S$*.*
- $\delta \subseteq (S \times \Gamma \times S)$ *is the transition relation.*

Based on our interpretation of subsets of $\mathcal{AP}$ as propositions it is easy to see that every Büchi automaton is also a Büchi propositional automaton. An arbitrary Büchi propositional automaton $B = \langle S, \delta, s_I, F\rangle$ may also be translated into a traditional Büchi automaton $B' = \left\langle S, 2^{\mathcal{AP}}, \delta', s_I, F\right\rangle$ by defining

$$\delta' = \{(s, A, s') \mid \exists\gamma.\, (s, \gamma, s') \in \delta \wedge A \in [\![\gamma]\!]\}$$

We define $L(B) \triangleq L(B')$. The traditional tableau-based constructions for converting LTL formulas into Büchi automata may easily be adapted to generate Büchi propositional automata with the property that for every pair of automaton states q, q' there is exactly one γ such that $(q, \gamma, q') \in \delta$.

Finally, we give a construction for Büchi propositional automaton B_{12} with $L(B_{12}) = L(B_1) \cap L(B_2)$ for the special case of Büchi propositional automata B_1 and B_2, with every state in B_1 accepting.

Theorem 1. *Let* $B_1 = (S_1, \delta_1, s_1, S_1)$ *and* $B_2 = (S_2, \delta_2, s_2, F_2)$ *be Büchi propositional*

automata. Then $L(B_{12}) = L(B_1) \cap L(B_2)$*, where*

$$B_{12} = (S_1 \times S_2, \delta_{12}, (s_1, s_2), S_1 \times F_2)$$

and $((s_1, s_2), \gamma_1 \wedge \gamma_2, (s_1', s_2')) \in \delta_{12}$ *iff* $(s_1, \gamma_1, s_1') \in \delta_1$ *and* $(s_2, \gamma_2, s_2') \in \delta_2$*.*

5.3 The LTL Query Checking Problem

In LTL query checking we are interested in Kripke structures and LTL formula *queries*, which are formulas containing a missing propositional subformula. The goal in LTL query checking is to construct solutions for the missing subformula. This section defines the problem precisely and proves results that will be used later in our algorithmic solution.

LTL queries correspond to LTL formulas with a missing propositional subformula, which we denote `var`. It should be noted that `var` stands for an unknown proposition over $\mathcal{A}$; it is *not* a propositional variable. The syntax of queries is as follows.

$$\phi := \texttt{var} \mid a \in \mathcal{AP} \mid \neg\phi \mid \phi \vee \phi \mid \mathbf{X}\,\phi \mid \phi\,\mathbf{U}\,\phi$$

In this chapter we only consider the case of a single propositional unknown, although the definitions can naturally be extended to multiple such unknowns. We often write $\phi[\texttt{var}]$ for LTL query with unknown `var`, and $\phi[\phi']$ for the LTL formula obtained by replacing all occurrences of `var` by LTL formula ϕ'. We also say that an occurrence

of `var` within $\phi[\texttt{var}]$ is *positive* if it appears within an even number of instances of $\neg$, and *negative* otherwise. If all occurrences of `var` in $\phi[\texttt{var}]$ are positive we say `var` is *positive* in $\phi[\texttt{var}]$; if all are negative we say `var` is *negative* in $\phi[\texttt{var}]$; if there are both positive and negative occurrences of `var` in $\phi[\texttt{var}]$ then `var` is *mixed* in $\phi[\texttt{var}]$.

The query-checking problem may now be formulated as follows.

Given: Finite-state Kripke structure K, LTL query $\phi[\texttt{var}]$

Compute: All $\gamma \in \Gamma$ (i.e. all propositional formulas over $\mathcal{A}$) with $K \models \phi[\gamma]$.

If γ is such that $K \models \phi[\gamma]$, then we call γ a *solution* for K and $\phi[\texttt{var}]$, and in this case we say that $\phi[\texttt{var}]$ is solvable for K. Computing all solutions for query checking problem K and $\phi[\texttt{var}]$ cannot be done explicitly, since the number of propositional formulas is infinite. However, if we define $\gamma_1 \equiv \gamma_2$ to hold if $[\![\gamma_1]\!] = [\![\gamma_2]\!]$, then it is clear that there are only finitely many distinct equivalence classes for Γ (exactly $2^{2^{|\mathcal{AP}|}}$ in fact). We also say that γ_1 is at least as strong (weak) as γ_2 if $[\![\gamma_1]\!] \subseteq [\![\gamma_2]\!]$ ($[\![\gamma_2]\!] \subseteq [\![\gamma_1]\!]$). We now have the following.

Theorem 2. *Let K be a finite-state Kripke structure and $\phi[\texttt{var}]$ an LTL query.*

1. *If **var** is positive in $\phi[\texttt{var}]$ then there is a finite set (modulo $\equiv$) of strongest solutions for $\phi[\gamma]$.*

2. *If **var** is negative in $\phi[\texttt{var}]$ then there is a finite set (modulo $\equiv$) of weakest solutions to $\phi[\gamma]$.*

In some cases these sets of maximal solutions contain a single solution.

Definition 9. *Let $\phi[\mathtt{var}]$ be an LTL query. Then $\phi[\mathtt{var}]$ is:*

- conjunctively covariant *iff for all γ_1, γ_2, $\phi[\gamma_1 \wedge \gamma_2] \equiv \phi[\gamma_1] \wedge \phi[\gamma_2]$; and*
- conjunctively contravariant *iff for all γ_1, γ_2, $\phi[\gamma_1 \vee \gamma_2] \equiv \phi[\gamma_1] \wedge \phi[\gamma_2]$.*

Theorem 3. *Let K be a finite-state Kripke structure, and let $\phi[\mathtt{var}]$ be solvable for K. Then the following hold.*

1. *If $\mathtt{var}$ is positive in $\phi[\mathtt{var}]$ and $\phi[\mathtt{var}]$ is conjunctively covariant, then there is a unique strongest solution (modulo $\equiv$) for $\phi[\mathtt{var}]$.*
2. *If $\mathtt{var}$ is negative in $\phi[\mathtt{var}]$ and $\phi[\mathtt{var}]$ is conjunctively contravariant, then there is a unique weakest solution (modulo $\equiv$) for $\phi[\mathtt{var}]$.*

As examples, note that $\mathbf{G}\,\mathtt{var}$ is conjunctively covariant and solvable for every K, and that $\mathtt{var}$ is positive; it is guaranteed to have a unique strongest solution for any K. So does $\mathbf{G}\,\mathbf{F}\,\mathtt{var}$. On the other hand, $\mathbf{G}(\mathtt{var} \implies \mathbf{F}\,\phi')$ is conjunctively contravariant and solvable for every K, and $\mathtt{var}$ appears negatively. Thus, every K has a unique weakest solution for this query.

5.4 Automaton-Based LTL Query Checking

In this section we show how LTL query checking can be formulated as a problem on Büchi propositional automata whose propositional labels may contain instances of

`var`. In this chapter we only consider LTL queries in which `var` is either negative or positive; the mixed case will not be dealt with. The approach is based on LTL model checking in that we generate Büchi propositional automata from both a Kripke structure and the negation of an LTL query and compose them; we then search for solutions to `var` that make the language of the composition automaton empty. To formalize these notions, we introduce the following definitions.

5.4.1 Propositional Queries

Definition 10 (Propositional query)**.** *Let* $\boldsymbol{var}$ *be an unknown proposition. Then* propositional queries *are generated by the following grammar.*

$$\gamma ::= \boldsymbol{var} \mid a \in \mathcal{AP} \mid \neg\gamma \mid \gamma \vee \gamma$$

We write $\gamma[\boldsymbol{var}]$ *for a generic instance of a propositional template, and* $\Gamma[\boldsymbol{var}]$ *for the set of all propositional templates involving* $\boldsymbol{var}$*.*

It is easy to see that propositional queries form a subset of LTL queries, and that notions of $\gamma[\gamma']$, positive and negative occurrences of `var`, etc., carry over immediately. A *shattering formula* for query $\gamma[\texttt{var}]$ is a propositional formula γ' with the property that $[\![\gamma[\gamma']]\!] = \emptyset$; that is, γ' "makes" $\gamma[\texttt{var}]$ unsatisfiable. We call $\gamma[\texttt{var}]$ *shatterable* if it has a shattering formula. The following is a consequence of the fact that the set of propositional formulas form a Boolean algebra.

Theorem 4. *Let $\gamma[\boldsymbol{var}]$ be shatterable.*

1. *If* $\boldsymbol{var}$ *is positive in* $\gamma[\boldsymbol{var}]$ *then there is a unique (modulo $\equiv$) weakest shattering formula for* $\gamma[\boldsymbol{var}]$.

2. *If* $\boldsymbol{var}$ *is negative in* $\gamma[\boldsymbol{var}]$ *then there is a unique strongest (modulo $\equiv$) shattering formula for* $\gamma[\boldsymbol{var}]$.

Intuitively, if $\gamma[\texttt{var}]$ is shatterable and `var` is positive, then $\gamma[\texttt{var}]$ can be rewritten as $\texttt{var} \wedge \gamma'$ for some propositional formula γ' (i.e. γ' contains no occurrences of `var`). In this case the weakest shattering formula for $\gamma[\texttt{var}]$ is $\neg\gamma'$. A dual argument holds in the case that `var` is negative in $\gamma[\texttt{var}]$.

5.4.2 Büchi Query Automata

Büchi query automata are propositional automata with propositional queries labeling transitions.

Definition 11 (Büchi query automaton)**.** *Let* $\boldsymbol{var}$ *be a propositional unknown. A* Büchi query automaton $B[\boldsymbol{var}]$ *has form* $\langle S, \delta, q_I, F\rangle$*, with finite state set* S*, initial state* $s_i \in S$*, accepting states* $F \subseteq S$*, and transition relation* $\delta \subseteq S \times \Gamma[\boldsymbol{var}] \times S$.

Intuitively, a Büchi query automaton is like an LTL query in that it contains a propositional unknown, `var`, that can be used to change the language accepted by the automaton. Specifically, if `var` is set to a condition γ' that shatters the edge label $\gamma[\texttt{var}]$, then any query-automaton edge of form $(s, \gamma[\texttt{var}], s')$ is no longer

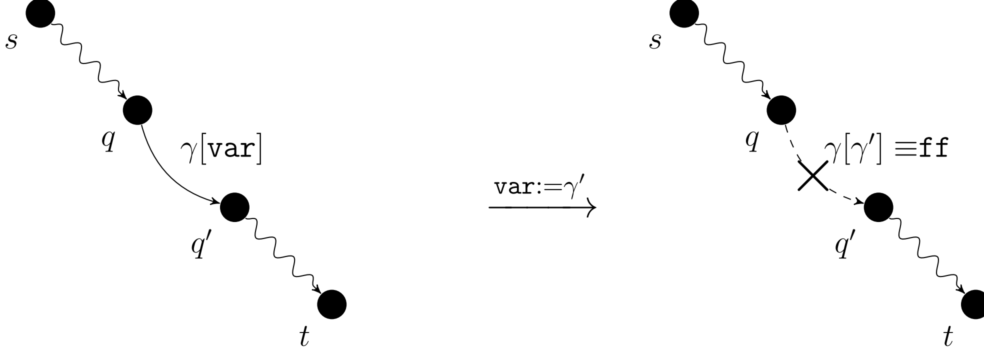


Figure 5.1: Shattering edges in a Büchi query automaton. Proposition γ' shatters $\gamma[\texttt{var}]$, and consequently the edge $(q, \gamma[\texttt{var}], q')$ is removed.

available for use in constructing runs of the automaton. Figure 5.1 illustrates this phenomenon. Thus, by varying `var` we can affect the language accepted by the query automaton.

Formally, if $B[\texttt{var}]$ is a Büchi query automaton and $\gamma \in \Gamma$, then define $B[\gamma]$ to be the Büchi propositional automaton obtained by replacing all occurrences of `var` by γ in any edge label within $B[\texttt{var}]$. We say that γ *shatters* $B[\texttt{var}]$ if $L(B[\gamma]) = \emptyset$, i.e. if γ renders the language of $B[\texttt{var}]$ empty. Notions of positive and negative occurrences of `var` in $B[\texttt{var}]$, etc., carry over in the obvious manner.

We now note the following correspondence between LTL queries and Büchi query automata.

Theorem 5. *Let $\phi[\texttt{var}]$ be an LTL query. Then there exists a Büchi query automaton $B_\phi[\texttt{var}]$ such that the following hold.*

1. *For all $\gamma \in \Gamma$, $[\![\phi[\gamma]]\!] = L(B_\phi[\gamma])$.*

2. *If `var` is positive in $\phi[\texttt{var}]$ then `var` is positive in $B_\phi[\texttt{var}]$.*

3. *If* $\texttt{var}$ *is negative in* $\phi[\texttt{var}]$ *then* $\texttt{var}$ *is negative in* $B_\phi[\texttt{var}]$.

The construction of $B_\phi[\texttt{var}]$ is a straightforward adaptation of the construction of Büchi propositional automata from LTL formulas ϕ.

5.4.3 LTL Query Checking via Büchi Query Automata

We now explain our approach to LTL query checking. Given finite-state Kripke structure K and LTL query $\phi[\texttt{var}]$, we perform the following.

1. Construct Büchi (propositional) automaton B_K.
2. Construct Büchi query automaton $B_{\neg\phi}[\texttt{var}]$.
3. Construct the product query automaton, $B_{K,\neg\phi}[\texttt{var}]$.
4. Solve for shattering conditions for $B_{K,\neg\phi}[\texttt{var}]$.

Because of Theorem 5 we know the following. If $\phi[\texttt{var}]$ is conjunctively covariant and `var` is positive in $\phi[\texttt{var}]$, then `var` is negative in $B_{K,\neg\phi}[\texttt{var}]$, and the strongest solution for `var` in $\phi[\texttt{var}]$ with respect to K coincides with the weakest shattering condition for $B_{K,\neg\phi}[\texttt{var}]$. The dual result holds in case `var` is negative in $\phi[\texttt{var}]$. Thus, solving for shattering conditions in $B_{K,\neg\phi}[\texttt{var}]$ yields appropriate query solutions for K and $\phi[\texttt{var}]$.

5.5 Implementing an LTL Query Checker

Based on the developments given earlier in the chapter, to develop a query checker for finite-state Kripke structures and LTL queries $\phi[\texttt{var}]$ it suffices to construct the product query automaton $B_{K,\neg\phi}[\texttt{var}]$ and then search for γ that shatter $B_{K,\neg\phi}[\texttt{var}]$. In this section we highlight some of the algorithmic aspects of this strategy and report on preliminary results of a prototype implementation.

At the outset, we can note that there is one immediate algorithmic solution: enumerate γ and test to see if $L(B[\gamma]) = \emptyset$ by computing the strongly connected components of $B[\gamma]$ and seeing if the start state can reach a successful component (i.e. one with an accepting state and at least one edge from the component back to itself). As there are $2^{2^{|\mathcal{AP}|}}$ semantically distinct such γ, this procedure terminates; indeed, this is the basis of the approach outlined in [21]. The complexity of this approach is prohibitive, however, as a sample implementation of ours has shown: even Kripke structures with 10s of states and 10 atomic propositions failed to complete successfully. This is to be expected, given that there are $2^{2^{10}} \geq 1.75 \times 10^{308}$ semantically distinct propositions in this case.

Instead, the approach outlined below pursues two different strategies to reduce the computational effort associated with shattering. One involves exploiting the lattice structure of $2^{2^{\mathcal{AP}}}$ to reduce the number of propositions that must be considered; the second combines this idea with a weakening of the problem to require the com-

putation of a single shattering proposition, rather than all such propositions. The next sections provide further details regarding our approach.

5.5.1 Construct Büchi Automaton B_K

Given a Kripke structure K, constructing the corresponding Büchi automaton B_K is done using the traditional method as described above. There is no query component to the model input, it should be noted.

5.5.2 Construct Büchi Query Automaton $B_{\neg\phi}[\texttt{var}]$

The LTL3BA package performs translations from standard LTL formulas to Büchi propositional automata. For a given query $\phi[\texttt{var}]$ we convert the formula into a Büchi query automaton by treating `var` as a normal atomic proposition. By default, LTL3BA attempts to remove non-determinism from the output Büchi query automaton, which can increase the number of edges in the automaton containing `var` on their labels. We configure LTL3BA so that removal of non-determinism is not required in order to avoid this extra overhead.

5.5.3 Construct Product Query Automaton $B_{K,\neg\phi}[\texttt{var}]$

As mentioned before, there is a well-known product construction for composing two Büchi automata into a single one accepting the intersection of the languages of the component automata. We adapt this composition operation to automaton B_K

and query automaton $B_{\neg\phi}[\texttt{var}]$, yielding composite query automaton $B_{K,\neg\phi}[\texttt{var}]$, as follows. States in $B_{\neg\phi}[\texttt{var}]$ are pairs of states from B_K and $B_{\neg\phi}[\texttt{var}]$. Tuple $((q_1, q_2), A \wedge \gamma[\texttt{var}], (q_1', q_2'))$ is a transition in $B_{K,\neg\phi}[\texttt{var}]$ iff (q_1, A, q_1') is a transition in B_K and $(q_2, \gamma[\texttt{var}], q_2')$ is a transition in $B_{\neg\phi}[\texttt{var}]$. It should be noted that the transition label in this case, $A \wedge \gamma[\texttt{var}]$, has a special property: for any $\texttt{var}$, either $[\![A \wedge \gamma[\texttt{var}]]\!] = \{A\}$, or $[\![A \wedge \gamma[\texttt{var}]]\!] = \emptyset$. This is a consequence of the fact that our treatment of A as a proposition means that $[\![A]\!] = \{A\}$. The initial state of $B_{K,\neg\phi}[\texttt{var}]$ is the pair consisting of the start states of B_K and $B_{\neg\phi}[\texttt{var}]$, respectively; states are accepting in $B_{K,\neg\phi}[\texttt{var}]$ if and only if the state component coming from $B_{\neg\phi}[\texttt{var}]$ is accepting.

5.5.4 Solve for Shattering Conditions of $B_{K,\neg\phi}[\texttt{var}]$

Given $B_{K,\neg\phi}[\gamma]$, we now must find a proposition γ such that $L(B_{K,\neg\phi}[\gamma]) = \emptyset$. One approach [21] is to enumerate all possible γ and compute whether or not $L(B_{K,\neg\phi}[\gamma]) = \emptyset$ for each such γ. Because of the number of possible γ, this approach is infeasible for all but trivial $\mathcal{AP}$.

Our approach instead focuses on determining when sets of edges in $B_{K,\neg\phi}[\texttt{var}]$ can be shattered via a common proposition γ in such a way that $L(B_{K,\neg\phi)}[\gamma])$ is empty. Our procedure may be summarized as follows.

1. Pre-process $B_{K,\neg\phi}[\texttt{var}]$ to eliminate all strongly connected components that have no outgoing edges from the component and that do not contain any

accepting states. Call the reduced query automaton $B'[\texttt{var}]$.

2. Identify all unique edge labels $S = \{\gamma_1[\texttt{var}], \ldots, \gamma_n[\texttt{var}]\}$ in $B'[\texttt{var}]$.

3. Process Γ appropriately to determine how $B'[\texttt{var}]$ can be shattered.

We now expand on the last step of the above procedure. In this work our interest is only for LTL queries $\phi[\texttt{var}]$ in which var appears only positively or only negatively; we do not consider queries in which var is mixed. Based on the construction of $B_{K,\neg\,\texttt{var}}[\texttt{var}]$ it follows that var is either positive in all of the $\gamma_i[\texttt{var}]$ or negative in all of the $\gamma_i[\texttt{var}]$. In what follows we assume that var is positive; the negative case is dual.

The first step in processing the $\gamma_i[\texttt{var}]$ (var is positive) is to determine if $\gamma_i[\texttt{var}]$ is shatterable, and if so, to compute its weakest shattering condition γ_i'. Propositional queries $\gamma_i[\texttt{var}]$ that are not shatterable are removed from future consideration, as they cannot contribute to shattering $B'[\texttt{var}]$. In what follows we assume that each $\gamma_i[\texttt{var}]$ is shatterable, with weakest shattering condition γ_i'.

The next step S is to search for subsets of S that, when all shattered, shatter $B'[\texttt{var}]$. More specifically, suppose $S' \subseteq S$ and γ'' is such that γ'' shatters each $\gamma'[\texttt{var}] \in S'$. In $B'[\gamma'']$ none of the edges labeled by elements of S' would be present; if enough edges are eliminated, $L(B[\gamma'']) = \emptyset$, and γ'' would shatter $B'[\texttt{var}]$. In this case we say that S' *shatters* $B'[\texttt{var}]$. This search procedure is facilitated by the following observations.

1. If S' shatters $B'[\texttt{var}]$ and $S' \subseteq S''$, S'' also shatters $B'[\texttt{var}]$.

2. If S' does not shatter $B'[\texttt{var}]$ and $S'' \subseteq S'$, S'' does not shatter $B'[\texttt{var}]$.

These observations can be exploited to develop a modified breadth-first search (BFS) strategy for finding all minimal subsets of S that shatter $B'[\texttt{var}]$. The BFS algorithm maintains a work set, $W \subseteq 2^S$, of subsets of S that need processing. Initially, $W = \{\emptyset\}$. The algorithm then repeatedly does the following. It selects a minimum-sized $S' \in W$ and checks if S' shatters $B[\texttt{var}]$. If it does, then it removes all supersets of S' from W and adds S' to the set of minimal shattering subsets of S. If it does not, then every superset of S' that contains one more element than S' is added to W. The procedure terminates when W is empty. Note that the approach does not add to W when S' is found to be a shattering set; the correctness of this approach is based on the first observation above.

The BFS algorithm in the worst-case can still require examination of all subsets of S, so we also consider a different algorithm whose goal is to compute a single minimal shattering subset of S. This approach, which we call GREEDY_SET_SEARCH (GSS), first locates a (not necessarily minimal) shattering set using a depth-first search strategy as follows. The procedure maintains a set $R \subseteq S$ that is initially $\emptyset$. It then repeatedly checks to see if R shatters $B'[\texttt{var}]$; if so, it terminates, otherwise, it adds a new element from S into R. The observations above guarantee that the above procedure will terminate after at most $|S|$ iterations. The second stage of the procedure then locates a minimal subset of the shattering set R returned by the

first stage as follows. Each edge (except the last one added) is removed from R, and the set without this edge is checked for shattering. If the newly modified set R', consisting of R with this single edge removed, shatters $B'[\texttt{var}]$ then the edge is permanently removed from R; otherwise, the edge is left in R. When this procedure terminates the resulting value of R is guaranteed to be a minimal shattering subset of S.

5.5.5 Implementation and Evaluation

We have developed prototype implementations of the BFS and GSS algorithms. Kripke structures are read in as directed graph data containing node labels, and LTL formulas are represented as simple strings. As stated previously, the LTL3BA routine was used to generate Büchi query automata from LTL queries.

For a proof-of-concept assessment of the techniques we use a modified version of NuSMV (extended from version 2.6.0) to extract the explicit Kripke structures from a sample `.smv` model files included in the NuSMV distribution. For each choice of model used, we considered property queries that were conceivably of interest based upon grounded properties known to be true of the systems already. These always took one of the following forms: $\mathbf{G}\, a$, $\mathbf{G}\,\mathbf{F}\, a$ or $\mathbf{G}\,(a \rightarrow \mathbf{F}\, b)$. The models we considered in our evaluation are the following.

- Counter[k] - An implementation of a k-bit counter.

- Semaphore[k] - An implementation of a semaphore access control scheme for

Model $\mathcal{M}$	States	Transitions	$\lvert\mathcal{AP}\rvert$	shatterable edges	shatterable labels
counter[3]	17	26	3	9	8
counter[4]	33	50	4	17	16
counter[5]	65	98	5	33	32
counter[10]	2049	3074	10	1025	1024
semaphore[2]	25	98	9	33	12
semaphore[3]	65	314	13	105	32
semaphore[4]	161	917	17	305	80
semaphore[5]	385	2498	21	833	192
semaphore[6]	897	6530	25	2177	448
semaphore[7]	2049	16514	29	5505	1024
production-cell	163	245	76	82	81

Figure 5.2: Statistics for composed Büchi product query automata for specified model $\mathcal{M}$ taking $\phi = \mathbf{G}(\texttt{var})$.

k different processes.

- Production cell - A production cell control model, first presented as an SMV model by Winter [83]. The original intent of this model concerned safety and liveness specifications.

Figure 5.2 contains relevant data about sizes of these models, and about the size of the Büchi query automata formed when composing the models with the query automaton $B_{\neg \mathbf{G}\,\texttt{var}}$. For our purposes, the following measures are relevant: (1) number of states, (2) number of transitions, (3) number of atomic propositions in the Kripke structure, (4) number of transition labels containing variable labels in the composite automaton, and (5) number of unique transition labels.

Figure 5.3 contains performance data for both BFS and GSS. Algorithms were implemented in Java, and experiments were conducted on a single machine with a 3.5 GHz processor containing 32 GB of memory. Individual experiments were allowed

to run for up to 2 hours before being stopped and considered timed out. BFS yielded minimal success, as most datasets timed out. The GSS approach to find a single minimal shattering set proved much more effective.

5.6 Conclusion

In this chapter we have considered the problem of query checking for Linear Temporal Logic (LTL). An LTL query checker takes a query, or LTL formula with a missing propositional subformula, together with a Kripke structure and computes a solution for the missing subformula. We have shown how this problem may be solved using automata-theoretic techniques that rely on the use of Büchi automata and the computation of so-called shattering conditions that make the languages of these automata empty. An implementation and preliminary performance data are also given.

The findings of this chapter was presented at AVoCS 2017 [84].

Dataset	Time (s)	# Queries
counter[3]	0.2	257
counter[4]	7.2	65537
counter[5]	timeout	2^{32} (*)
counter[10]	timeout	2^{1024} (*)
semaphore[2]	1.3	4097
semaphore[3]	timeout	2^{32} (*)
semaphore[4]	timeout	2^{80} (*)
semaphore[5]	timeout	2^{192} (*)
semaphore[6]	timeout	2^{448} (*)
semaphore[7]	timeout	2^{1024} (*)
production-cell	timeout	2^{81} (*)

(a)

Dataset	Time (s)	# Queries
counter[3]	0.2	17
counter[4]	0.2	33
counter[5]	0.6	65
counter[10]	22.4	2049
semaphore[2]	0.2	25
semaphore[3]	0.4	65
semaphore[4]	1.4	161
semaphore[5]	6.9	385
semaphore[6]	44.1	897
semaphore[7]	296.9	2049
production-cell	1.3	163

(b)

Figure 5.3: Timing results for finding (a) all shattering sets via breadth first search, and (b) one minimal shattering set via GREEDY SET SEARCH. The number of total shattering queries that are made for each experiment are also reported. Query counts marked with a (*) are estimates based on our understanding of the models.

Chapter 6: Finite LTL

6.1 Introduction

Since its introduction into Computer Science by Amir Pnueli in a landmark paper [85], propositional linear temporal logic, or LTL, has played a prominent role as a specification formalism for discrete systems. LTL includes constructs for describing how a system's state may change over time; this fact, coupled with its simplicity and decidability properties for both model and satisfiability checking, have made it an appealing framework for research into system verification and analysis. The notation has also served as a springboard for the study of other temporal logics in computing.

Traditional LTL formulas are interpreted with respect to infinite sequences of states, where each state assigns a truth value to the atomic propositions appearing in the formula. Such infinite sequences are intended to be viewed as runs of a system, with each state representing a snapshot of the system as it executes. However, applications of so-called *finite* variants of LTL have emerged as well; in finite LTL finite, rather than infinite, state sequences are the models. Domains as varied as

robotic path planning and automated run-time monitoring have used finite versions of LTL to precisely specify the desired behavior of systems.

This chapter defines a construction for a specific finite variant of LTL, which we call Finite LTL, that renders formulas into finite-state automata that accepts the models, or "satisfying state sequences," of the corresponding formula. Such constructions have been given for traditional LTL (e.g. [86]) and have played a pivotal role in practical techniques for both model checking (i.e. determining if every execution of a system satisfies and LTL formula) and satisfiability checking (i.e. determining of a given formula has any models). To the best of our knowledge, no such construction has been given in the literature for Finite LTL; our goal in providing such a construction in this chaper is to provide researchers with a basis for automata-theoretic techniques for studying Finite LTL as well. The construction exploits specific features of Finite LTL to simplify the traditional tableau constructions found for classical LTL; in particular, the state construction relies on semantics-preserving syntactic formula transformations, and the acceptance condition for the resulting automaton can be determined syntactically based purely on the formulas associated with a state.

This fact, along with the other aforementioned work, motivates our work. Across the surveyed literature, if any construction for an automaton representing a finite semantic LTL formula was provided, the acceptance criterion was recursively defined based on reachability from the initial state. In our case, we have also

provided a purely syntactic method of determining state acceptance.

In this chapter, we have adopted a version of Finite LTL that is similar to that as used in [30, 29]. Our choice is based firstly on a desire to appeal to the mature and well-studied similarities this representation has to the case of standard (infinite) LTL, and secondly to facilitate supplementing the logic to support the task of query checking [16] for finite sets of finite data streams.

6.2 Finite LTL

This section introduces the specific syntax and semantics of Finite LTL. For reference, refer to Section 3.1.2 for the definition of standard LTL. In what follows, fix a (nonempty) finite set $\mathcal{AP}$ of atomic propositions.

6.2.1 Syntax of Finite LTL

Definition 12 (Finite LTL Syntax)**.** *The set of Finite LTL formulas is defined by the following grammar, where* $a \in \mathcal{AP}$*.*

$$\phi ::= a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \mathbf{X}\,\phi \mid \phi_1\,\mathbf{U}\,\phi_2$$

We call the operators $\neg$ *and* $\wedge$ propositional *and* $\mathbf{X}$ *and* $\mathbf{U}$ modal*. We use* $\Phi^{\mathcal{AP}}$ *to refer to the set of all Finite LTL formulas and* $\Gamma^{\mathcal{AP}} \subsetneq \Phi^{\mathcal{AP}}$ *for the set of all propositional formulas, i.e. those containing no modal operators. We often write* Φ

and Γ *instead of* $\Phi^{\mathcal{AP}}$ *and* $\Gamma^{\mathcal{AP}}$ *when* $\mathcal{AP}$ *is clear from context.*

Finite LTL formulas may be constructed from atomic propositions using the traditional propositional operators $\neg$ and $\wedge$, as well as the modalities of "next" ($\mathbf{X}$) and "until" ($\mathbf{U}$). We also use the following derived notations:

$$\begin{aligned} \mathit{false} &= a \wedge \neg a & \phi_1 \,\mathbf{R}\, \phi_2 &= \neg((\neg\phi_1)\,\mathbf{U}(\neg\phi_1)) \\ \mathit{true} &= \neg \mathit{false} & \overline{\mathbf{X}}\,\phi &= \neg\,\mathbf{X}(\neg\phi) \\ \phi_1 \vee \phi_2 &= \neg((\neg\phi_1) \wedge (\neg\phi_2)) & \mathbf{F}\,\phi &= \mathit{true}\,\mathbf{U}\,\phi \\ & & \mathbf{G}\,\phi &= \neg\,\mathbf{F}(\neg\phi) \end{aligned}$$

The constants *false* and *true*, and the operators $\wedge$ and $\vee$, and $\mathbf{U}$ and $\mathbf{R}$, are duals in the usual logical sense, with $\mathbf{R}$ sometimes referred to as the "release" operator. We introduce $\overline{\mathbf{X}}$ ("weak next") as the dual for $\mathbf{X}$. That this operator is needed is due to the semantic interpretation of Finite LTL with respect to finite sequences, which means that, in contrast to regular LTL, $\mathbf{X}$ is not its own dual. This point is elaborated on later. Finally, the duals $\mathbf{F}$ and $\mathbf{G}$ capture the usual notions of "eventually" and "always", respectively.

6.2.2 Semantics of Finite LTL

The semantics of Finite LTL is formalized as relation $\pi \models \phi$, where $\pi \in (2^{\mathcal{AP}})^*$ is a finite sequence whose elements are subsets of $\mathcal{AP}$. Such a subset $A \subseteq \mathcal{AP}$ represents a state $\sigma_A \in \mathcal{AP} \to \{0,1\}$, or assignment of truth values to atomic propositions, in the usual fashion: $\sigma_A(a) = 1$ if $a \in A$, and $\sigma_A(a) = 0$ if $a \notin A$. We first introduce

some notation on finite sequences.

Definition 13 (Finite-Sequence Terminology)**.** *Let X be a set, with X^* the set of finite sequences of elements of X. Also assume that $\pi \in X^*$ has form $x_0 \ldots x_{i-1}$ for some $i \in \mathbb{N} = \{0, 1, \ldots\}$. We define the following notations.*

1. *$\varepsilon \in X^*$ is the empty sequence.*

2. *$|\pi| = i$ is the number of elements in π. Note that $|\varepsilon| = 0$.*

3. *For $j \in \mathbb{N}$, $\pi_j = x_j \in X$, provided $j < |\pi|$, and is undefined otherwise.*

4. *For $j \in \mathbb{N}$, the* suffix*, $\pi(j)$, of π beginning at j is taken to be $\pi(j) = x_j \ldots x_{i-1} \in X^*$, provided $j \leq |\pi|$, and is undefined otherwise. Note that $\pi(0) = \pi$ and that $\pi(|\pi|) = \varepsilon$.*

5. *If $x \in X$ and $\pi \in X^*$ then $x\pi \in X^*$ is the sequence such that $(x\pi)_0 = x$ and $(x\pi)(1) = \pi$.*

Definition 14 (Finite LTL Semantics)**.** *Let ϕ be a Finite LTL formula, and let $\pi \in (2^{\mathcal{AP}})^*$. Then the satisfaction relations, $\pi \models \phi$, for Finite LTL is defined inductively on the structure of ϕ as follows.*

- *$\pi \models a$ iff $|\pi| \geq 1$ and $a \in \pi_0$*

- *$\pi \models \neg\phi$ iff $\pi \not\models \phi$*

- *$\pi \models \phi_1 \wedge \phi_2$ iff $\pi \models \phi_1$ and $\pi \models \phi_2$*

- $\pi \models \mathbf{X}\,\phi$ *iff* $|\pi| \geq 1$ *and* $\pi(1) \models \phi$

- $\pi \models \phi_1\,\mathbf{U}\,\phi_2$ *iff* $\exists j\colon 0 \leq j \leq |\pi|\colon \pi(j) \models \phi_2$ *and* $\forall k\colon 0 \leq k < j\colon \pi(k) \models \phi_1$

We write $[\![\phi]\!]$ *for the set* $\{\pi \mid \pi \models \phi\}$*. We also say that* ϕ_1 *and* ϕ_2 *are* logically equivalent*, notation* $\phi_1 \equiv \phi_2$*, if* $[\![\phi_1]\!] = [\![\phi_2]\!]$*.*

Intuitively, π can be seen as an execution sequence of a system, with π_0, if it exists, taken to be the current state and π_i for $i > 0$, if they exist, referring to states i time steps in the future. In this interpretation ε can be seen as representing an execution with no states in it. Then $\pi \models \phi$ holds if the sequence π satisfies ϕ. Formula $a \in \mathcal{AP}$ can only be satisfied by non-empty π, as the presence or absence of a in the first state π_0 contained in π is used to determine whether a is true ($a \in \pi_0$) or not ($a \notin \pi_0$). Negation and conjunction are defined as usual. The $\mathbf{X}$ operator is the next operator; a sequence π satisfies $\mathbf{X}$ if it is non-empty (and thus has a notion of "next") and the suffix of π beginning after π_0 satisfies ϕ. Finally, $\mathbf{U}$ captures a notion of *until*: π satisfies $\phi_1\,\mathbf{U}\,\phi_2$ when it has a suffix satisfying ϕ_2 and every suffix of π that strictly includes this suffix satisfies ϕ_1.

6.2.3 Properties of Finite LTL

Despite the close similarity of Finite LTL and LTL, the former nevertheless possesses certain semantic subtleties that we will address in this section. Many of these aspects of the logic have to do with properties of ε, the empty sequence, as a potential model of formulas. (Indeed, some other finite versions of LTL explicitly exclude non-empty

sequences as possible models.) The inclusion of ε as a possible model for formulas simplifies the tableau construction given later in this chapter, however; indeed, the definition of the acceptance condition that we give demands it. Accordingly, this section also shows how the possibly counter-intuitive features of Finite LTL can be addressed with proper encodings.

$\mathbf{X}$ is not self-dual. In traditional LTL the $\mathbf{X}$ operator is *self-dual*. That is, for any ϕ, $\mathbf{X}\,\phi$ and $\neg\,\mathbf{X}(\neg\phi)$ are logically equivalent. This fact simplifies the treatment of notions such as positive normal form, since no new operator needs to be introduced for the dual of $\mathbf{X}$.

In Finite LTL $\mathbf{X}$ does not have this property. To see why, consider the formula $\mathbf{X}\,\mathit{true}$. If $\mathbf{X}$ were self-dual then we should have that $\mathbf{X}\,\mathit{true} \equiv \neg(\mathbf{X}\,\neg\mathit{true})$, i.e. that $[\![\mathbf{X}\,\mathit{true}]\!] = [\![\neg\,\mathbf{X}(\neg\mathit{true})]\!]$. However this fact does not hold. Consider $[\![\mathbf{X}\,\mathit{true}]\!]$. Based on the semantics of Finite LTL, $\pi \models \mathbf{X}\,\mathit{true}$ iff $|\pi| \geq 1$ and $\pi(0) \models \mathit{true}$. Since any sequence satisfies true, it therefore follows that

$$[\![\mathbf{X}\,\mathit{true}]\!] = \{\pi \in (2^{\mathcal{AP}})^* \mid |\pi| \geq 1\};$$

note that $\varepsilon \notin [\![\mathbf{X}\,\mathit{true}]\!]$. Now consider $[\![\neg\,\mathbf{X}\,\neg\mathit{true}]\!]$. From the semantics of Finite LTL one can see that $[\![\neg\mathit{true}]\!] = \emptyset = [\![\mathbf{X}\,\neg\mathit{true}]\!]$. It then follows that

$$[\![\neg\,\mathbf{X}\,\neg\mathit{true}]\!] = (2^{\mathcal{AP}})^*,$$

and thus $\varepsilon \in [\![\neg\,\mathbf{X}\,\neg \mathit{true}]\!]$.

The existence of duals for logical operators is used extensively in the tableau construction, so for this reason we have introduced $\overline{\mathbf{X}}$ as the dual for $\mathbf{X}$. Using the semantics of $\mathbf{X}$ and $\neg$ it can be seen that $\pi \models \overline{\mathbf{X}}\,\phi$ iff either $|\pi| = 0$ or $\pi(1) \models \phi$. Indeed, $[\![\overline{\mathbf{X}}\,\phi]\!] = [\![\mathbf{X}\,\phi]\!] \cup \{\varepsilon\}$; we sometimes refer to $\overline{\mathbf{X}}$ as the *weak next* operator for this reason.

Including / excluding ε. The discussion about $\mathbf{X}$ and $\overline{\mathbf{X}}$ above leads to the following lemma.

Lemma 1 (Empty-sequence Formula Satisfaction). *Let $\pi \in (2^{\mathcal{AP}})^*$.*

1. *$\pi \neq \varepsilon$ iff $\pi \models \mathbf{X}\,\mathit{true}$.*
2. *$\pi = \varepsilon$ iff $\pi \models \overline{\mathbf{X}}\,\mathit{false}$.*

Proof. Immediate from the semantics of $\mathbf{X}$, $\overline{\mathbf{X}}$. □

This lemma suggests a way for including / excluding ε as a model of formula.

Corollary 1. *Let $\pi \in (2^{\mathcal{AP}})^*$ and $\phi \in \Phi^{\mathcal{AP}}$. Then the following hold.*

1. *$\pi \models \phi \wedge \mathbf{X}\,\mathit{true}$ iff $\pi \models \phi$ and $\pi \neq \varepsilon$; and*
2. *$\pi \models \phi \vee \overline{\mathbf{X}}\,\mathit{false}$ iff $\pi \models \phi$ or $\pi = \varepsilon$.*

Literals and ε. A *literal* is a formula that has form either a or $\neg a$ for $a \in \mathcal{AP}$. A positive-normal-form result for a logic asserts that any formula can be converted into an equivalent one in which all negations appear only as literals.

The semantics of Finite LTL dictates that for $\pi \models a$ to hold, where $a \in \mathcal{AP}$, π must be non-empty. Specifically, the semantics requires that $|\pi| \geq 1$ and $a \in \pi_0$. Based on the semantics of $\neg$, it therefore follows that $\pi \models \neg a$ iff *either* $|\pi| = 0$ *or* $a \notin \pi_0$. It follows that $\varepsilon \models \neg a$ for any $a \in \mathcal{AP}$.

This may seem objectionable at first glance, since $\neg a$ can be seen as asserting that a is false "now" (i.e. in the current state), and ε has no current state. Given the semantics of Finite LTL, however, the conclusion is unavoidable. However, using Corollary 1 we can give a formula that captures what might be the desired meaning of $\neg a$, namely, that a satisfying sequence must be non-empty. Consider $(\neg a) \wedge \mathbf{X}\, \mathit{true}$. From the corollary, it follows that $\pi \models (\neg a) \wedge \mathbf{X}\, \mathit{true}$ iff π is non-empty and $a \notin \pi_0$.

The relationship between ε and literals also influences the semantics of formulas involving temporal operators. For example, consider $\mathbf{F}\, \neg a$ for literal $\neg a$, which intuitively asserts that a is eventually false. More formally, based on the definition of $\mathbf{F}$ in terms of $\mathbf{U}$ and the semantics of $\mathbf{U}$, it can be seen that $\pi \models \mathbf{F}\, \neg a$ iff there exists i such that $0 \leq i \leq |\pi|$ and $\pi(i) \models \neg a$. Since for any π, $\pi(|\pi|) = \varepsilon$, it therefore follows that $\pi(|\pi|) \models \neg a$ for any π, and thus that every π satisfies $\pi \models \mathbf{F}\, \neg a$. This can be seen as offending intuition. However, Corollary 1 again offers a helpful encoding. Consider the formula $\mathbf{F}((\neg a) \wedge \mathbf{X}\, \mathit{true})$. It can be seen that $\pi \models \mathbf{F}((\neg a) \wedge \mathbf{X}\, \mathit{true})$ iff there is an i such that $0 \leq i < |\pi|$ and $\pi(i) \models \neg a$, meaning that there must exist an i such that $a \notin \pi_i$.

A similar observation highlights a subtlety in the formula $\mathbf{G}\, a$ when $a \in \mathcal{AP}$.

It can be seen that $\pi \models \mathbf{G}\, a$ iff for all i such that $0 \leq i \leq |\pi|$, $\pi(i) \models a$. Since $\pi(|\pi|) = \varepsilon$ and $\varepsilon \not\models a$, it therefore follows that $\mathbf{G}\, a$ is unsatisfiable. This also seems objectionable, although Corollary 1 again offers a workaround. Consider $\mathbf{G}(a \vee \overline{\mathbf{X}}\, \mathit{false})$. In this case $\pi(|\pi|) \models a \vee \overline{\mathbf{X}}\, \mathit{false}$, and for all i such that $0 \leq i < |\pi|$, $\pi(i) \models a$ iff $a \in \pi_i$. This formula captures the intuition that for π to satisfy a, a must be satisfied in every subset of $\mathcal{AP}$ in π.

Propositional Formulas. We close this section with a discussion about the semantics of propositional formulas (i.e. those not involving any propositional operators) in Finite LTL. Later in this chapter we rely extensively on traditional propositional identities, including De Morgan's Laws and distributivity, and the associated normal forms — positive normal form and disjunctive normal form in particular — that they enable. In what follows we show that for the set $\Gamma^{\mathcal{AP}}$ of propositional formulas in Finite LTL, logical equivalence coincides with traditional propositional equivalence. The only subtlety in establishing this fact has to do with the fact that in Finite LTL, ε is allowed as a potential model.

We begin by recalling the traditional semantics of propositional formulas.

Definition 15 (Semantics for Finite LTL Propositional Subset)**.** *Given a (finite, non-empty) set $\mathcal{AP}$ of atomic propositions, the* propositional semantics *of formulas in $\Gamma^{\mathcal{AP}}$ is given as a relation $\models_p \subseteq 2^{\mathcal{AP}} \times \Gamma^{\mathcal{AP}}$ defined as follows.*

1. *$A \models_p a$, where $a \in \mathcal{AP}$, iff $a \in A$.*

2. $A \models_p \neg\gamma$ *iff* $A \not\models_p \gamma$.

3. $A \models_p \gamma_1 \wedge \gamma_2$ *iff* $A \models_p \gamma_1$ *and* $A \models_p \gamma_2$.

We write $[\![\gamma]\!]_p$ *for* $\{A \subseteq \mathcal{AP} \mid A \models_p \gamma\}$ *and* $\gamma_1 \equiv_p \gamma_2$ *when* $[\![\gamma_1]\!]_p = [\![\gamma_2]\!]_p$.

In this section we show that for any $\gamma_1, \gamma_2 \in \Gamma^{\mathcal{AP}}$, $\gamma_1 \equiv \gamma_2$ iff $\gamma_1 \equiv_p \gamma_2$: in other words, logical equivalence of propositional formulas in Finite LTL coincides exactly with traditional propositional logical equivalence. In traditional LTL, this fact follows immediately from the fact that for infinite sequence π, $\pi \models \gamma$ iff $\pi_0 \models_p \gamma$. In the setting of Finite LTL we have a similar result for non-empty π, but care must be taken with ε.

Lemma 2 (Non-empty Sequence Propositional Satisfaction)**.** *Let* $\pi \in (2^{\mathcal{AP}})^*$ *be such that* $|\pi| > 0$, *and let* $\gamma \in \Gamma^{\mathcal{AP}}$. *Then* $\pi \models \gamma$ *iff* $\pi_0 \models_p \gamma$.

Proof. Follows by induction on the structure of γ. □

The next lemma establishes a correspondence between ε satisfying propositional Finite LTL formulas and the propositional semantics of such formulas.

Lemma 3 (Empty Sequence Propositional Satisfaction)**.** *Let* $\gamma \in \Gamma^{\mathcal{AP}}$ *be a propositional formula. Then* $\varepsilon \models \gamma$ *iff* $\emptyset \models_p \gamma$.

Proof. The result follows by structural induction on γ. There are three cases to consider

1. $\gamma = a$ for some $a \in \mathcal{AP}$. In this case $\varepsilon \not\models a$ and $\emptyset \not\models_p a$, so the desired bi-implication follows.

2. $\gamma = \neg\gamma'$ for some $\gamma' \in \Gamma^{\mathcal{AP}}$. In this case the induction hypothesis says that $\varepsilon \models \gamma'$ iff $\emptyset \models_p \gamma'$. The reasoning proceeds as follows.

$$\begin{aligned} \varepsilon \models \gamma \text{ iff } \varepsilon \models \neg\gamma' & \quad \gamma = \neg\gamma' \\ \text{iff } \varepsilon \not\models \gamma' & \quad \text{Semantics of Finite LTL} \\ \text{iff } \emptyset \not\models_p \gamma' & \quad \text{Induction hypothesis} \\ \text{iff } \emptyset \models_p \neg\gamma' & \quad \text{Propositional semantics} \\ \text{iff } \emptyset \models_p \gamma & \quad \gamma = \neg\gamma' \end{aligned}$$

3. $\gamma = \gamma_1 \wedge \gamma_2$ for some $\gamma_1, \gamma_2 \in \Gamma^{\mathcal{AP}}$. In this case the induction hypothesis guarantees the result for γ_1 and γ_2. We reason as follows.

$$\begin{aligned} \varepsilon \models \gamma \text{ iff } \varepsilon \models \gamma_1 \wedge \gamma_2 & \quad \gamma = \gamma_1 \wedge \gamma_2 \\ \text{iff } \varepsilon \models \gamma_1 \text{ and } \varepsilon \models \gamma_2 & \quad \text{Semantics of Finite LTL} \\ \text{iff } \emptyset \models_p \gamma_1 \text{ and } \emptyset \models_p \gamma_2 & \quad \text{Induction hypothesis (twice)} \\ \text{iff } \emptyset \models_p \gamma_1 \wedge \gamma_2 & \quad \text{Propositional semantics} \\ \text{iff } \emptyset \models_p \gamma & \quad \gamma = \gamma_1 \wedge \gamma_2 \end{aligned}$$

□

We can now state the main result of this section.

Theorem 6 (Propositional / Finite LTL Semantic Correspondence)**.** *Let* $\gamma_1, \gamma_2 \in \Gamma^{\mathcal{AP}}$. *Then* $\gamma_1 \equiv \gamma_2$ *iff* $\gamma_1 \equiv_p \gamma_2$.

Proof. We break the proof into two pieces.

1. Assume that $\gamma_1 \equiv \gamma_2$; we must show that $\gamma_1 \equiv_p \gamma_2$, i.e. that for any $A \subseteq \mathcal{AP}, A \models_p \gamma_1$ iff $A \models_p \gamma_2$. We reason as follows.

$$\begin{aligned} A \models_p \gamma_1 &\text{ iff } \pi \models \gamma_1 \text{ all } \pi \text{ such that } |\pi| > 0 \text{ and } \pi_0 = A && \text{Lemma 2} \\ &\text{iff } \pi \models \gamma_2 \text{ all } \pi \text{ such that } |\pi| > 0 \text{ and } \pi_0 = A && \gamma_1 \equiv \gamma_2 \\ &\text{iff } A \models_p \gamma_2 && \text{Lemma 2} \end{aligned}$$

2. Assume that $\gamma_1 \equiv_p \gamma_2$; we must show that $\gamma_1 \equiv \gamma_2$, i.e. that for any $\pi \in (2^{\mathcal{AP}})^*, \pi \models \gamma_1$ iff $\pi \models \gamma_2$. So fix $\pi \in (2^{\mathcal{AP}})^*$; we first consider the case when $|\pi| > 0$.

$$\begin{aligned} \pi \models \gamma_1 &\text{ iff } \pi_0 \models_p \gamma_1 && \text{Lemma 2} \\ &\text{iff } \pi_0 \models_p \gamma_2 && \gamma_1 \equiv_p \gamma_2 \\ &\text{iff } \pi \models \gamma_2 && \text{Lemma 2} \end{aligned}$$

We now consider the case when $|\pi| = 0$, meaning $\pi = \varepsilon$.

$$\begin{aligned} \varepsilon \models \gamma_1 &\text{ iff } \emptyset \models_p \gamma_1 && \text{Lemma 3} \\ &\text{iff } \emptyset \models_p \gamma_2 && \gamma_1 \equiv_p \gamma_2 \\ &\text{iff } \varepsilon \models \gamma_2 && \text{Lemma 3} \end{aligned}$$

□

Because of this lemma, propositional formulas in Finite LTL enjoy the usual properties of propositional logic. In particular, in a logic extended with $\vee$ formulas can be converted into positive normal form, and disjunctive normal form, while preserving their semantics, including their behavior with respect to ε.

6.3 Normal Forms for Finite LTL

The purpose of this chapter is to define a construction for converting formulas in Finite LTL into non-deterministic finite automata (NFAs) with the property that the language of the NFA for a formula consists exactly of the finite sequences that satisfy the formula. Such automata have many uses: they provide a basis for model checking against Finite LTL specifications and for checking satisfiability of Finite LTL formulas. The approach is adapted from the well-known tableau construction [87] for LTL. Our presentation relies on showing how Finite LTL formulas may be converted into logically equivalent formulas in a specific *normal form*; this normal form will then be used in the construction given in the next section.

6.3.1 Extended Finite LTL and Positive Normal Form

Our construction works with Finite LTL formulas in *positive normal norm* (PNF), in which negation is constrained to be applied to atomic propositions. The PNF formulas in Finite LTL as given in Definition 12 are not as expressive as full Finite LTL; there are formulas ϕ in Finite LTL such that $\phi \not\equiv \phi'$ for any PNF ϕ' in Finite

LTL. However, if we extend Finite LTL by including duals of all operators in Finite LTL, we can obtain a logic whose formulas are as expressive as those in Finite LTL.

Definition 16 (Extended Finite LTL Syntax)**.** *The set of Extended Finite LTL formulas is given by the following grammar, where* $a \in \mathcal{AP}$*.*

$$\phi ::= a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \mathbf{X}\,\phi \mid \phi_1\,\mathbf{U}\,\phi_2 \mid \phi_1 \vee \phi_2 \mid \overline{\mathbf{X}}\,\phi \mid \phi_1\,\mathbf{R}\,\phi_2$$

We use $\Phi_e^{\mathcal{AP}}$ *to refer to the set of all Extended Finite LTL formulas, and* $\Gamma_e^{\mathcal{AP}}$ *for the set of propositional Extended Finite LTL formulas (i.e. formulas that do not include any use of* $\mathbf{X}, \mathbf{U}, \overline{\mathbf{X}}$ *or* $\mathbf{R}$*).*

Extended Finite LTL extends Finite LTL by including the duals of $\wedge$, $\mathbf{X}$ and $\mathbf{U}$, namely, $\vee$, $\overline{\mathbf{X}}$ and $\mathbf{R}$, respectively. Note that $\Phi^{\mathcal{AP}} \subsetneq \Phi_e^{\mathcal{AP}}$: every Finite LTL formula is syntactically an Extended Finite LTL formula, but not vice versa.

The semantics of Extended Finite LTL is given as follows.

Definition 17 (Extended Finite LTL Semantics)**.** *Let* ϕ *be an Extended Finite LTL formula, and let* $\pi \in (2^{\mathcal{AP}})^*$*. Then the semantics of Extended Finite LTL is given as a relation* $\pi \models_e \phi$ *defined as follows.*

- $\pi \models_e a$ *iff* $|\pi| \geq 1$ *and* $a \in \pi_0$*.*
- $\pi \models_e \neg\phi$ *iff* $\pi \not\models_e \phi$*.*
- $\pi \models_e \phi_1 \wedge \phi_2$ *iff* $\pi \models_e \phi_1$ *and* $\pi \models_e \phi_2$*.*

- $\pi \models_e \mathbf{X}\,\phi$ *iff* $|\pi| \geq 1$ *and* $\pi(1) \models_e \phi$.

- $\pi \models_e \phi_1 \,\mathbf{U}\, \phi_2$ *iff* $\exists j\colon 0 \leq j \leq |\pi|\colon \pi(j) \models_e \phi_2$ *and* $\forall k\colon 0 \leq k < j\colon \pi(k) \models_e \phi_1$.

- $\pi \models_e \phi_1 \vee \phi_2$ *iff either* $\pi \models_e \phi_1$ *or* $\pi \models_e \phi_2$.

- $\pi \models_e \overline{\mathbf{X}}\,\phi$ *iff either* $|\pi| = 0$ *or* $\pi(1) \models_e \phi$.

- $\pi \models_e \phi_1 \,\mathbf{R}\, \phi_2$ *iff* $\forall j\colon 0 \leq j \leq |\pi|\colon \pi(j) \models_e \phi_2$ *or* $\exists k\colon 0 \leq k < j\colon \pi(k) \models_e \phi_1$.

We define $[\![\phi]\!]_e = \{\pi \in (2^{\mathcal{AP}})^* \mid \pi \models_e \phi\}$ *and* $\phi_1 \equiv_e \phi_2$ *iff* $[\![\phi_1]\!]_e = [\![\phi_2]\!]_e$.

The next lemmas establish relationships between Finite LTL and Extended Finite LTL. The first shows that the semantics of Extended Finite LTL, when restricted to Finite LTL formulas, matches the semantics of Finite LTL.

Lemma 4 ((Extended) Finite LTL Semantic Correspondence)**.** *Let* ϕ *be a formula in Finite LTL and* $\pi \in (2^{\mathcal{AP}})^*$. *Then* $\pi \models \phi$ *iff* $\pi \models_e \phi$.

Proof. Immediate. □

The next result establishes duality properties between the new operators in Extended Finite LTL and the existing ones in Finite LTL.

Lemma 5 (Dualities in Extended Finite LTL)**.** *Let* ϕ, ϕ_1, ϕ_2 *be formulas in Extended Finite LTL, and let* $\pi \in (2^{\mathcal{AP}})^*$. *Then the following hold.*

1. $\pi \models_e \phi_1 \vee \phi_2$ *iff* $\pi \models_e \neg((\neg\phi_1) \wedge (\neg\phi_2))$.

2. $\pi \models_e \overline{\mathbf{X}}\,\phi$ *iff* $\pi \models_e \neg\,\mathbf{X}\,\neg\phi$.

3. $\pi \models_e \phi_1 \mathbf{R}\, \phi_2$ *iff* $\pi \models_e \neg((\neg\phi_1)\, \mathbf{U}(\neg\phi_2))$.

Proof. Follows from the definition of $\models_e$. □

The next lemma establishes that although Extended Finite LTL includes more operators than Finite LTL, any Extended Finite LTL formula can be translated into a logically equivalent Finite LTL formula. Thus, the two logics have the same expressive power.

Lemma 6 (Co-expressiveness for (Extended) Finite LTL)**.** *Let ϕ be an Extended Finite LTL formula. Then there is a Finite LTL formula ϕ' such that $[\![\phi]\!]_e = [\![\phi']\!]$.*

Proof. Follows from Lemmas 4 and 5. The latter lemma in particular establishes that each non-Finite LTL operator in ϕ ($\vee$, $\overline{\mathbf{X}}$, $\mathbf{R}$) can be replaced by appropriately negated versions of its dual. Specifically, $\phi_1 \vee \phi_2$ can be replaced by $\neg((\neg\phi_1) \wedge (\neg\phi_2))$, $\overline{\mathbf{X}}\, \phi'$ by $\neg\, \mathbf{X}\, \neg\phi$, and $\phi_1 \mathbf{R}\, \phi_2$ by $\neg((\neg\phi_1)\, \mathbf{U}(\neg\phi_2))$. □

Although Extended Finite LTL does not enhance the expressive power of Finite LTL, it does enjoy a property that Finite LTL does not: its formulas may be converted in *positive normal form*. This fact will be useful in defining the tableau construction; the relevant mathematical results are presented here.

Definition 18 (Positive Normal Form (PNF))**.** *The set of* positive normal form *(PNF) formulas of Extended Finite LTL is defined inductively as follows.*

- *If $a \in \mathcal{AP}$ then a and $\neg a$ are in positive normal form.*

- *If ϕ is in positive form then $\mathbf{X}\,\phi$ and $\overline{\mathbf{X}}\,\phi$ are in positive normal form.*

- *If ϕ_1 and ϕ_2 are in positive normal normal then $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $\phi_1 \,\mathbf{U}\, \phi_2$ and $\phi_1 \,\mathbf{R}\, \phi_2$ are in positive normal form.*

We now have the following.

Lemma 7 (PNF and Extended Finite LTL)**.** *Let $\phi \in \Phi_e^{\mathcal{AP}}$ be an Extended Finite LTL formula. Then there is a $\phi' \in \Phi_e^{\mathcal{AP}}$ in PNF such that $\phi \equiv_e \phi'$.*

Proof. Follows from the fact that $\neg\neg\phi \equiv_e \phi$ and the existence of dual operators in Extended Finite LTL, which enable identities such as $\neg(\phi_1 \,\mathbf{U}\, \phi_2) \equiv_e (\neg\phi_1)\,\mathbf{R}(\neg\phi_2)$ to be used to "drive negations" down to atomic propositions. □

6.3.2 Automaton Normal Form

Propositional logic exhibits a number of logical equivalences that support the conversion of arbitrary formulas into various *normal forms* that are then the basis for algorithmic analysis, including satisfiability checking. Disjunctive Normal Form (DNF) is one such well-known normal form. In this section we show how Extended LTL formulas in PNF can be converted into a normal form related to DNF, which we call *Automaton Normal Form* (ANF); ANF will be a key vehicle for the automaton construction in the next section. We begin by reviewing the basics of DNF in the setting of the propositional fragment of Extended LTL.

We first lift the definitions of $\vee$ and $\wedge$ to finite sets of formulas as follows.

Definition 19 (Conjunction / Disjunction for Sets of Formulas)**.** *Let* $P = \{\phi_1, \ldots, \phi_n\}$*,* $n \geq 0$ *be a finite set of Extended LTL formulas. Then* $\bigwedge P$ *and* $\bigvee P$ *are defined as follows.*

$$\bigwedge P = \begin{cases} \mathit{true} & \text{if } n = 0 \text{ (i.e. } P = \emptyset \text{)} \\ \phi_1 & \text{if } n = 1 \text{ (i.e. } P = \{\phi_1\}\text{)} \\ \phi_1 \wedge (\bigwedge\{\phi_2, \ldots, \phi_n\}) & \text{if } n \geq 2 \end{cases}$$

$$\bigvee P = \begin{cases} \mathit{false} & \text{if } n = 0 \text{ (i.e. } P = \emptyset \text{)} \\ \phi_1 & \text{if } n = 1 \text{ (i.e. } P = \{\phi_1\}\text{)} \\ \phi_1 \vee (\bigvee\{\phi_2, \ldots, \phi_n\}) & \text{if } n \geq 2 \end{cases}$$

We now define disjunctive normal form as follows.

Definition 20 (Disjunctive Normal Form (DNF))**.**

1. *A* literal *is a formula of form* a *or* $\neg a$ *for some* $a \in \mathcal{AP}$*.*

2. *A* DNF clause *is a formula* C *of form* $\bigwedge\{\ell_1, \ldots, \ell_n\}$*,* $n \geq 0$*, where each* ℓ_i *is a literal.*

3. *A formula in* $\Gamma_e^{\mathcal{AP}}$ *is in* disjunctive normal form (DNF) *if it has form* $\bigvee\{C_1, \ldots C_k\}$*,* $k \geq 0$*, where each* C_i *is a DNF clause.*

The following is a well-known result in propositional logic that, due to Theorem 6, is also applicable to the propositional fragment of Extended Finite LTL.

Theorem 7 (DNF Conversion for Extended Finite LTL)**.** *Let $\gamma \in \Gamma_e^{\mathcal{AP}}$. Then there is a DNF formula $\gamma' \in \Gamma_e^{\mathcal{AP}}$ such that $\gamma \equiv_e \gamma'$.*

Automaton normal form (ANF) can be seen as an extension of DNF in which each clause is allowed to have a single subformula of form $\mathbf{X}\,\phi$ or $\overline{\mathbf{X}}\,\phi$, where ϕ is an formula in full Extended Finite LTL. A clause in an ANF formula can be seen as defining whether or not a sequence π satisfies the formula in terms of conditions that must hold on the first element of the sequence, if there is one, (the literals in the clause), and the rest of the sequence (the "next-state" formula in the clause). This feature will be exploited in the automaton construction in the next section. The formal definition of ANF is as follows.

Definition 21 (Automaton Normal Form (ANF))**.**

1. *An* ANF clause C *has form* $(\bigwedge\{\ell_1, \ldots \ell_k\}) \wedge \mathbf{N}(\bigwedge\{\phi_1, \ldots, \phi_n\})$*, where each* ℓ_i *is a literal,* $\mathbf{N} \in \{\mathbf{X}, \overline{\mathbf{X}}\}$ *and each* $\phi_j \in \Phi_e^{\mathcal{AP}}$ *is an arbitrary Extended Finite LTL formula.*

2. *A formula in Extended Finite LTL is in* automaton normal form (ANF) *iff it has form* $\bigvee\{C_1, \ldots, C_k\}$*,* $k \geq 0$*, where each* C_i *is an ANF clause.*

We often represent clauses as $(\bigwedge \mathcal{L}) \wedge \mathbf{N}(\bigwedge \mathcal{F})$*, where* $\mathcal{L}$ *is a finite set of literals and* $\mathcal{F}$ *a finite set of Extended LTL formulas. If* $C = (\bigwedge \mathcal{L}) \wedge \mathbf{N}(\bigwedge \mathcal{F})$ *we write*

$$\begin{aligned} \mathit{lits}(C) &= \mathcal{L} \\ \mathit{nf}(C) &= \mathcal{F} \end{aligned}$$

for the set of literals and the set of "next formulas" following the next operator ($\mathbf{X}$ *or* $\overline{\mathbf{X}}$*) in* C*.*

The next lemma establishes a key feature of formulas in ANF *vis à vis* the sequences in $(2^{\mathcal{AP}})^*$ that model it.

Lemma 8 (Sequence Satisfaction and ANF)**.**

1. *Let* C *be an ANF clause. Then for any* $\pi \in (2^{\mathcal{AP}})^*$ *such that* $|\pi| > 0$*,* $\pi \models_e C$ *iff* $\pi_0 \models_p \bigwedge lits(C)$ *and* $\pi(1) \models_e \bigwedge nf(C)$*.*

2. *Let* $\phi = \bigvee_i C_i$ *be in ANF. Then for every* $\pi \in (2^{\mathcal{AP}})^*$*,* $\pi \models_e \phi$ *iff* $\pi \models_e C_i$*.*

Proof. For Part 1, let $\pi \in (2^{\mathcal{AP}})^*$ be such that $|\pi| > 0$. Also let $\mathcal{L} = lits(C)$ and $\mathcal{F} = nf(C)$. We reason as follows.

$$
\begin{aligned}
\pi \models_e C \text{ iff } & \pi \models_e (\textstyle\bigwedge \mathcal{L}) \wedge \mathbf{N}(\bigwedge \mathcal{F}) && \text{Definition 21} \\
\text{iff } & \pi \models_e \textstyle\bigwedge \mathcal{L} \text{ and } \pi \models_e \mathbf{N}(\bigwedge \mathcal{F}) && \text{Semantics of } \wedge \\
\text{iff } & \pi_0 \models_p \textstyle\bigwedge \mathcal{L} \text{ and } \pi \models_e \mathbf{N}(\bigwedge \mathcal{F}) && \text{Lemma 2, } \textstyle\bigwedge L \in \Gamma_e^{\mathcal{AP}} \\
\text{iff } & \pi_0 \models_p \textstyle\bigwedge \mathcal{L} \text{ and } \pi(1) \models_e \bigwedge \mathcal{F} && \text{Semantics of } \mathbf{N} \in \{\mathbf{X}, \overline{\mathbf{X}}\}
\end{aligned}
$$

Part 2 follows immediately from the semantics of $\bigvee$. □

The import of this lemma derives especially from its first statement. This asserts that determining if an ANF clause is satisfied by a non-empty sequence can be broken down into a propositional determination about its initial state (π_0) and the literals in the clause, and a determination about the rest of the sequence ($\pi(1)$) and

the "next formulas" of the clause. This observation is central to the construction of automata from formulas that we give later.

In the rest of this section we will show that for any Extended Finite LTL formula ϕ there is a logically equivalent one in ANF. We start by stating some logical identities that will be used later.

Lemma 9 (Distributivity of $\mathbf{X}$, $\overline{\mathbf{X}}$)**.** *Let* $\phi_1, \phi_2 \in \Phi_e^{\mathcal{AP}}$*.*

1. $(\mathbf{X}\,\phi_1) \wedge (\mathbf{X}\,\phi_2) \equiv_e \mathbf{X}(\phi_1 \wedge \phi_2)$.
2. $(\overline{\mathbf{X}}\,\phi_1) \wedge (\overline{\mathbf{X}}\,\phi_2) \equiv_e \overline{\mathbf{X}}(\phi_1 \wedge \phi_2)$.
3. $(\mathbf{X}\,\phi_1) \vee (\mathbf{X}\,\phi_2) \equiv_e \mathbf{X}(\phi_1 \vee \phi_2)$.
4. $(\overline{\mathbf{X}}\,\phi_1) \vee (\overline{\mathbf{X}}\,\phi_2) \equiv_e \overline{\mathbf{X}}(\phi_1 \vee \phi_2)$.

Proof. Immediate from the semantics of Extended Finite LTL □

The next lemma establishes that in a certain sense, $\mathbf{X}$ "dominates" $\overline{\mathbf{X}}$ in the context of conjunction.

Lemma 10 ($\mathbf{X}$ Dominates $\overline{\mathbf{X}}$)**.** *The following holds for any Extended Finite LTL formulas* ϕ_1, ϕ_2*.*

$$(\mathbf{X}\,\phi_1) \wedge (\overline{\mathbf{X}}\,\phi_2) \equiv_e \mathbf{X}(\phi_1 \wedge \phi_2)$$

Proof. Follows from the fact that if $\pi \models (\mathbf{X}\,\phi_1) \wedge (\overline{\mathbf{X}}\,\phi_2)$ then $|\pi| > 0$. □

The final lemma is key to our ANF transformation result. It states that operators $\mathbf{U}$ and $\mathbf{R}$ may be rewritten using operators $\wedge$, $\vee$, $\mathbf{X}$ and $\overline{\mathbf{X}}$.

Lemma 11 (Unrolling $\mathbf{U}$ and $\mathbf{R}$). *The following holds for any Extended Finite LTL formulas ϕ_1, ϕ_2.*

1. $\phi_1 \, \mathbf{U} \, \phi_2 \equiv_e \phi_2 \vee (\phi_1 \wedge \mathbf{X}(\phi_1 \, \mathbf{U} \, \phi_2))$.

2. $\phi_1 \, \mathbf{R} \, \phi_2 \equiv_e \phi_2 \wedge (\phi_1 \vee \overline{\mathbf{X}}(\phi_1 \, \mathbf{R} \, \phi_2))$.

Proof. We prove Part 1 by showing that $[\![\phi_1 \, \mathbf{U} \, \phi_2]\!]_e = [\![\phi_2 \wedge (\phi_1 \vee \overline{\mathbf{X}}(\phi_1 \, \mathbf{R} \, \phi_2))]\!]_e$.

$$\llbracket \phi_1 \,\mathbf{U}\, \phi_2 \rrbracket_e$$

$= \{\pi \mid \pi \models_e \phi_1 \,\mathbf{U}\, \phi_2\}$ — Def. of $\llbracket - \rrbracket_e$

$= \{\pi \mid \exists j \colon 0 \leq j \leq |\pi| \colon \pi(j) \models_e \phi_2 \text{ and } \forall i \colon 0 \leq i < j \colon \pi(i) \models_e \phi_1\}$ — Semantics of $\mathbf{U}$

$= \{\pi \mid \pi(0) \models_e \phi_2\} \cup$

$\{\pi \mid \exists j \colon 1 \leq j \leq |\pi| \colon \pi(j) \models_e \phi_2 \text{ and } \forall i \colon 0 \leq i < j \colon \pi(i) \models_e \phi_1\}$ — Set theory

$= \llbracket \phi_2 \rrbracket_e \cup$

$\{\pi \mid \exists j \colon 1 \leq j \leq |\pi| \colon \pi(j) \models_e \phi_2 \text{ and } \forall i \colon 0 \leq i < j \colon \pi(i) \models_e \phi_1\}$ — $\pi(0) = \pi$, def. of $\llbracket - \rrbracket_e$

$= \llbracket \phi_2 \rrbracket_e \cup (\{\pi \mid \pi(0) \models \phi_1\} \cap$

$\{\pi \mid \exists j \colon 1 \leq j \leq |\pi| \colon \pi(j) \models_e \phi_2 \text{ and } \forall i \colon 1 \leq i < j \colon \pi(i) \models_e \phi_1\})$ — Set theory

$= \llbracket \phi_2 \rrbracket_e \cup (\llbracket \phi_1 \rrbracket_e \cap$

$\{\pi \mid \exists j \colon 1 \leq j \leq |\pi| \colon \pi(j) \models_e \phi_2 \text{ and } \forall i \colon 1 \leq i < j \colon \pi(i) \models_e \phi_1\})$ — $\pi(0) = \pi$, def. of $\llbracket - \rrbracket_e$

$= \llbracket \phi_2 \rrbracket_e \cup (\llbracket \phi_1 \rrbracket_e \cap$

$\{\pi \mid \exists j' \colon 0 \leq j' \leq |\pi| - 1 \colon \pi(j'+1) \models_e \phi_2 \text{ and } \forall i' \colon 0 \leq i < j' \colon \pi(i'+1) \models_e \phi_1\})$

$j = j' + 1, i = i' + 1$

$= \llbracket \phi_2 \rrbracket_e \cup (\llbracket \phi_1 \rrbracket_e \cap$

$\{\pi \mid \exists j' \colon 0 \leq j' \leq |\pi(1)| \colon \pi(1)(j') \models_e \phi_2 \text{ and } \forall i' \colon 0 \leq i < j' \colon \pi(1)(i') \models_e \phi_1\})$

$\pi(j' + 1) = \pi(1)(j'), |\pi($

$|\pi| - 1$

$= \llbracket \phi_2 \rrbracket_e \cup (\llbracket \phi_1 \rrbracket_e \cap \{\pi \mid \pi(1) \models_e \phi_1 \,\mathbf{U}\, \phi_2\})$ — Semantics of $\mathbf{U}$

$= \llbracket \phi_2 \rrbracket_e \cup (\llbracket \phi_1 \rrbracket_e \cap \{\pi \mid \pi \models_e \mathbf{X}(\phi_1 \,\mathbf{U}\, \phi_2)\})$ — Semantics of $\mathbf{X}$

$= \llbracket \phi_2 \rrbracket_e \cup (\llbracket \phi_1 \rrbracket_e \cap \llbracket \mathbf{X}(\phi_1 \,\mathbf{U}\, \phi_2) \rrbracket_e$ — Def. of $\llbracket - \rrbracket_e$

$= \llbracket \phi_2 \vee (\phi_1 \wedge \mathbf{X}(\phi_1 \,\mathbf{U}\, \phi_2)) \rrbracket_e$ — Semantics of $\wedge, \vee$

To prove Part 2, we can rely the duality of $\mathbf{R}$ and $\mathbf{U}$ and Part 1. Again, it suffices to show that $[\![\phi_1 \,\mathbf{R}\, \phi_2]\!] \equiv_e [\![\phi_2 \wedge (\phi_1 \vee \overline{\mathbf{X}}(\phi_1 \,\mathbf{R}\, \phi_2))]\!]_e$. We reason as follows.

$$
\begin{array}{rlll}
[\![\phi_1 \,\mathbf{R}\, \phi_2]\!] & \equiv_e & [\![\neg((\neg\phi_1)\,\mathbf{U}(\neg\phi_2))]\!] & \text{Lemma 5(3)} \\
& \equiv_e & (2^{\mathcal{AP}})^* - [\![(\neg\phi_1)\,\mathbf{U}(\neg\phi_2)]\!]_e & \text{Semantics of } \neg \\
& \equiv_e & (2^{\mathcal{AP}})^* - [\![(\neg\phi_2) \vee ((\neg\phi_1) \wedge \mathbf{X}((\neg\phi_1)\,\mathbf{U}(\neg\phi_2)))]\!]_e & \text{Part 1} \\
& \equiv_e & [\![\neg((\neg\phi_2) \vee ((\neg\phi_1) \wedge \mathbf{X}((\neg\phi_1)\,\mathbf{U}(\neg\phi_2))))]\!]_e & \text{Semantics of } \neg \\
& \equiv_e & [\![\phi_2 \wedge (\phi_1 \vee \neg\,\mathbf{X}(\neg\phi_1)\,\mathbf{U}(\neg\phi_2)))]\!]_e & \text{Lemma 5(1)} \\
& \equiv_e & [\![\phi_2 \wedge (\phi_1 \vee \overline{\mathbf{X}}\,\neg((\neg\phi_1)\,\mathbf{U}(\neg\phi_2)))]\!]_e & \text{Lemma 5(2)} \\
& \equiv_e & [\![\phi_2 \wedge (\phi_1 \vee \overline{\mathbf{X}}(\phi_1 \,\mathbf{R}\, \phi_2))]\!]_e & \text{Lemma 5(3)}
\end{array}
$$

□

The remainder of this section will be devoted to proving the following theorem.

Theorem 8 (Conversion to ANF)**.** *Let ϕ be an Extended Finite LTL formula in PNF. Then there exists a transformation anf such that $anf(\phi)$ is in ANF and the following hold.*

1. $\phi \equiv_e anf(\phi)$.

2. *Suppose $anf(\phi) = \bigvee C_i$. Then for each C_i and each $\phi' \in nf(C_i)$, ϕ' is a subformula of ϕ.*

This theorem states that any PNF Extended LTL formula ϕ can be converted into ANF formula $anf(\phi)$, and in such away that each clause's "next-state subformula"

consists of a conjunction of subformulas of ϕ. As any Extended LTL formula can be converted into PNF, this ensures that any Extended LTL formula can be converted into ANF.

To prove this theorem, we define several formula transformations that, when applied in sequence, yield a formula in ANF with the desired properties. The first transformation ensures that all occurrences of $\mathbf{U}$ and $\mathbf{R}$ are *guarded* in the resulting formula, in the following sense.

Definition 22 (Guardedness)**.** *Let ϕ be an Extended Finite LTL formula.*

1. *Let ϕ' be a subformula of ϕ. Then ϕ' is* guarded *in ϕ iff for every occurrence of ϕ' in ϕ is within an occurrence of a subformula of ϕ of form $\mathbf{N}\phi''$, where $\mathbf{N} \in \{\mathbf{X}, \overline{\mathbf{X}}\}$.*

2. *Formula ϕ is* guarded *iff every subformula of ϕ of form $\phi_1 \mathbf{U} \phi_2$ or $\phi_1 \mathbf{R} \phi_2$ appears guarded in ϕ.*

As an example of the above definition, consider formula $\phi = (a \mathbf{U} b) \wedge \mathbf{X}(a \mathbf{U} b)$. This formula is not guarded, because the left-most occurrence of $(a \mathbf{U} b)$ does not appear within an occurrence of a subformula of form $\mathbf{X} \phi''$. However, $\phi' = (b \vee (a \wedge \mathbf{X}(a \mathbf{U} b))) \wedge \mathbf{X}(a \mathbf{U} b)$ is guarded, and indeed $\phi' \equiv_e \phi$ due to Lemma 11(1).

We now define a transformation gt on formulas; the intent of this transformation is that $gt(\phi)$ is guarded, and $gt(\phi) \equiv_e \phi$.

Definition 23 (Guardedness Transformation)**.** *Extended Finite LTL formula transformation gt is defined inductively as follows.*

$$gt(\phi) = \begin{cases} a & \text{if } \phi = a \\ \neg(gt(\phi')) & \text{if } \phi = \neg\phi' \\ gt(\phi_1) \wedge gt(\phi_2) & \text{if } \phi = \phi_1 \wedge \phi_2 \\ gt(\phi_1) \vee gt(\phi_2) & \text{if } \phi = \phi_1 \vee \phi_2 \\ \phi & \text{if } \phi = \mathbf{X}\,\phi' \text{ or } \phi = \overline{\mathbf{X}}\,\phi' \\ gt(\phi_2) \vee (gt(\phi_1) \wedge \mathbf{X}\,\phi) & \text{if } \phi = \phi_1\,\mathbf{U}\,\phi_2 \\ gt(\phi_2) \wedge (gt(\phi_1) \vee \overline{\mathbf{X}}\,\phi) & \text{if } \phi = \phi_1\,\mathbf{R}\,\phi_2 \end{cases}$$

We have the following.

Lemma 12 (Properties of gt)**.** *Let ϕ be an Extended Finite LTL formula. Then:*

1. *$gt(\phi)$ is guarded.*
2. *$gt(\phi) \equiv_e \phi$.*
3. *If ϕ is in PNF, then so is $gt(\phi)$.*
4. *Let $\mathbf{N}\phi'$ be a subformula of $gt(\phi)$, where $\mathbf{N} \in \{\mathbf{X}, \overline{\mathbf{X}}\}$. Then ϕ' is a subformula of ϕ.*

Proof. Immediate from the definition of gt and Lemma 11. □

The next transformation we describe converts guarded Extended Finite LTL formulas into *pseudo-ANF*.

Definition 24 (Pseudo ANF)**.**

1. *An* ANF pseudo-literal *has form* $a, \neg a$ *or* $\mathbf{N}\phi$*, where* $\mathbf{N} \in \{\mathbf{X}, \overline{\mathbf{X}}\}$ *and* $\phi \in \Phi_e^{\mathcal{AP}}$.

2. *An* ANF pseudo-clause C *has form* $\bigwedge\{\alpha_1, \ldots, \alpha_n\}$*,* $n \geq 0$*, where each* α_i *is an ANF pseudo-literal.*

3. *A formula is in* Pseudo-ANF *if it has form* $\bigvee\{C_1, \ldots, C_n\}$*,* $n \geq 0$*, where each* C_i *is an ANF pseudo-clause.*

Note that every literal is also an ANF pseudo-literal. Moreover, an ANF pseudo-clause differs from an ANF clause in that the former may have multiple (or no) instances of pseudo-literals of form $\mathbf{N}\phi$, while the latter is required to have exactly one, of form $\mathbf{N} \bigwedge \mathcal{F}$. We have the following.

Lemma 13 (Conversion to Pseudo ANF)**.** *Let* ϕ *be a guarded Extended Finite LTL formula in PNF. Then there exists a formula* $pa(\phi)$ *such that:*

1. $pa(\phi)$ *is in Pseudo ANF.*

2. $pa(\phi) \equiv_e \phi$.

Proof. Transformation pa is a version of the classical DNF transformation for propositional formulas in which that ANF pseudo-literals are treated as literals. □

The final transformation, an, converts formulas in pseudo-ANF into semantically equivalent formulas in ANF.

Definition 25 (Pseudo-ANF to ANF Conversion)**.** *The transformation an is defined as follows:*

1. *Let* $C = \bigwedge P$*, where* $P = \{\alpha_1, \ldots, \alpha_n\}$ *is a set of ANF pseudo-literals and* $n \geq 0$*, be an ANF pseudo-clause. Also let* $L(P)$ *be the literals in* P *and* $N(P) = P - L(P) = \{\boldsymbol{N}_1\phi_1, \ldots, \boldsymbol{N}_i\phi_i\}$ *for some* $0 \leq i \leq n$*, each* $\boldsymbol{N}_i \in \{\mathbf{X}, \overline{\mathbf{X}}\}$*, be the non-literals in* P*. Then* $ct(C)$ *is defined as follows.*

$$ct(C) = \begin{cases} (\bigwedge L(P)) \wedge \mathbf{X}(\bigwedge\{\phi_1, \ldots, \phi_i\}) & \textit{if } \mathbf{N}_j = \mathbf{X} \textit{ for some } 1 \leq j \leq i \\ (\bigwedge L(P)) \wedge \overline{\mathbf{X}}(\bigwedge\{\phi_1, \ldots, \phi_i\}) & \textit{otherwise} \end{cases}$$

2. *Let* $\phi = \bigvee\{C_1, \ldots, C_n\}$*,* $n \geq 0$*, be an Extended Finite LTL formula in Pseudo ANF. Then transformation* $an(\phi) = \bigvee\{ct(C_1), \ldots, ct(C_n)\}$*.*

The next lemma and its corollary establish that *ct* and *an* convert pseudo-ANF clauses and formulas, respectively, into ANF clauses and formulas.

Lemma 14 (Conversion from Pseudo ANF to ANF Clauses)**.** *Let* C *be a pseudo-ANF clause. Then* $ct(C)$ *is an ANF clause, and* $C \equiv_e ct(C)$*.*

Proof. Follows from Lemmas 9 and 10. □

Corollary 2 (Conversion from Pseudo ANF to ANF Formulas)**.** *Let* ϕ *be a pseudo-ANF formula. Then* $an(\phi)$ *is in ANF, and* $\phi \equiv_e an(\phi)$*.*

Proof. Follows from Lemma 14. □

We now have the machinery necessary to prove Theorem 8.

Theorem 8. Let ϕ be an Extended Finite LTL formula in PNF. We must show how to convert it into an ANF formula $\mathit{anf}(\phi) = \bigvee C_i$ such that $\phi \equiv_e \mathit{anf}(\phi)$, and such that for each C_i and each $\phi' \in \mathit{nf}(C_i)$, ϕ' is a subformula of ϕ.

We define $\mathit{anf}(\phi) = \mathit{an}(\mathit{pa}(\mathit{gt}(\phi)))$; obviously $\mathit{anf}(\phi)$ is in ANF. We now reason as follows.

$$
\begin{array}{lll}
\phi & \equiv_e \; gt(\phi) & \text{Lemma 12; note } gt(\phi) \text{ is PNF} \\
& \equiv_e \; pa(gt(\phi)) & \text{Lemma 13} \\
& \equiv_e \; an(pa(gt(\phi))) & \text{Corollary 2} \\
& \equiv_e \; anf(\phi) & \text{Definition of } anf
\end{array}
$$

Thus $\mathit{anf}(\phi)$ is in ANF, and $\mathit{anf}(\phi) \equiv_e \phi$.

For the second part, we note that in the construction of $\mathit{anf}(\phi)$ we first compute $gt(\phi)$, which has the property that every subformula of form $\mathbf{N}\phi''$ is such that ϕ'' is a subformula of ϕ. The definition of pa guarantees that this property is preserved in $pa(gt(\phi))$. Finally, the definition of an ensures the desired result. □

Example 1 (Conversion to ANF)**.** *We close this section with an example showing how our conversion to ANF works. Consider* $\phi = a\,\mathbf{U}(b\,\mathbf{R}\,c)$*; we show how to*

compute $an(pa(gt(\phi)))$*. Here is the result of* $gt(\phi)$*.*

$$
\begin{aligned}
gt(\phi) &= gt(a\,\mathbf{U}(b\,\mathbf{R}\,c)) \\
&= gt(b\,\mathbf{R}\,c) \vee (gt(a) \wedge \mathbf{X}\,\phi) \\
&= (gt(c) \wedge (gt(b) \vee \overline{\mathbf{X}}(b\,\mathbf{R}\,c))) \vee (a \wedge \mathbf{X}\,\phi) \\
&= (c \wedge (b \vee \overline{\mathbf{X}}(b\,\mathbf{R}\,c))) \vee (a \wedge \mathbf{X}\,\phi)
\end{aligned}
$$

Note that this formula is guarded. We now consider $pa(gt(\phi))$*.*

$$
\begin{aligned}
pa(gt(\phi)) &= pa((c \wedge (b \vee \overline{\mathbf{X}}(b\,\mathbf{R}\,c))) \vee (a \wedge \mathbf{X}\,\phi)) \\
&= pa(((c \wedge b) \vee (c \wedge \overline{\mathbf{X}}(b\,\mathbf{R}\,c))) \vee (a \wedge \mathbf{X}\,\phi)) \\
&= \bigvee\{c \wedge b, c \wedge \overline{\mathbf{X}}(b\,\mathbf{R}\,c), a \wedge \mathbf{X}\,\phi\}
\end{aligned}
$$

Note that two of the three clauses in $pa(gt(\phi))$ *are already ANF clauses; the only that is not is* $c \wedge b$*. This leads to the following.*

$$
\begin{aligned}
anf(\phi) &= an(pa(gt(\phi))) \\
&= an(\bigvee\{c \wedge b, c \wedge \overline{\mathbf{X}}(b\,\mathbf{R}\,c), a \wedge \mathbf{X}\,\phi\}) \\
&= \bigvee\{ct(c \wedge b), ct(c \wedge \overline{\mathbf{X}}(b\,\mathbf{R}\,c)), ct(a \wedge \mathbf{X}\,\phi)\} \\
&= \bigvee\{c \wedge b \wedge \overline{\mathbf{X}}\,true, c \wedge \overline{\mathbf{X}}(b\,\mathbf{R}\,c), a \wedge \mathbf{X}\,\phi\}
\end{aligned}
$$

Note that this formula is in ANF. Also note that $ct(c \wedge b) = c \wedge b \wedge \overline{\mathbf{X}}\,true$ *due*

to the fact that in pseudo-ANF clause $c \wedge b$ has no next-state pseudo-literals. The definition of ct ensures that $\overline{\mathbf{X}} \bigwedge \emptyset = \overline{\mathbf{X}}$ true is added to ensure that the result satisfies the syntactic requirements of being an ANF clause.

6.4 A Tableau Construction for Finite LTL

In this section we show how Finite LTL formulas may be converted into finite-state automata whose languages consist of exactly the sequences making the associated formula true. Based on Lemma 7 we know that any Finite LTL formula can be converted into an Extended Finite LTL formula in PNF, so in the sequel we show how to build finite automata from Extended Finite LTL formulas in PNF. We begin by recalling the definitions of non-deterministic finite-state automata.

Definition 26 (Non-deterministic Finite Automata (NFA)). *1. A* non-deterministic finite automaton *(NFA) is a tuple $(Q, \Sigma, q_I, \delta, F)$, where:*

- *Q is a finite set of* states*;*
- *Σ is a finite non-empty set of* alphabet *symbols;*
- *$q_I \in Q$ is the* start state*;*
- *$\delta \subseteq Q \times \Sigma \times Q$ is the* transition relation*; and*
- *$F \subseteq Q$ is the set of* accepting states.

2. Let $M = (Q, \Sigma, q_I, \delta, F)$ be a NFA, let $q \in Q$, and let $w \in \Sigma^*$. Then q accepts *w* in *M iff one of the following hold.*

- *$w = \varepsilon$ and $q \in F$*
- *$w = \sigma w'$ for some $\sigma \in \Sigma, w' \in \Sigma^*$ and there exists $(q, \sigma, q') \in \delta$ such that q' accepts w' in M.*

3. *Let $M = (Q, \Sigma, q_I, \delta, F)$ be a NFA. Then $L(M)$, the* language *of M, is*

$$L(M) = \{w \in \Sigma^* \mid q_I \text{ accepts } w \text{ in } M\}.$$

We now state the theorem we will prove in the rest of this section.

Theorem 9 (NFAs from Extended LTL Formulas)**.** *Let $\phi \in \Phi_e^{\mathcal{AP}}$ be in PNF. Then there is a NFA M_ϕ such that $L(M_\phi) = [\![\phi]\!]_e$.*

6.4.1 The Construction

In this section we describe our construction for building NFA M_ϕ from PNF Extended Finite LTL formula ϕ. We have been referring to this construction as a tableau construction, and indeed it makes essential use of identities, such as those in Lemmas 5– 11, that also underpin classical tableau constructions. However, because of our use of ANF we are able to avoid other aspects of tableau constructions, such as the need for maximally consistent subsets as automaton states.

In what follows we use $S(\phi)$ to refer to the set of (not necessarily proper) subformulas of ϕ. States in M_ϕ will be associated with subsets of $S(\phi)$, and defining accepting states will require checking if $\varepsilon \models_e \phi'$ for arbitrary $\phi' \in S(\phi)$. The

next lemma establishes that this latter check can be computed on the basis of the syntactic structure of ϕ'.

Lemma 15 (Empty-sequence Check). *Let $\phi \in \Phi_e^{\mathcal{AP}}$ be in PNF. Then $\varepsilon \models_e \phi$ iff one of the following hold.*

1. $\phi = \neg a$ *for some* $a \in \mathcal{AP}$
2. $\phi = \phi_1 \wedge \phi_2$, $\varepsilon \models_e \phi_1$, *and* $\varepsilon \models_e \phi_2$
3. $\phi = \phi_1 \, \mathbf{U} \, \phi_2$ *and* $\varepsilon \models_e \phi_2$
4. $\phi = \phi_1 \vee \phi_2$ *and either* $\varepsilon \models_e \phi_1$ *or* $\varepsilon \models_e \phi_2$
5. $\phi = \overline{\mathbf{X}} \, \phi'$
6. $\phi = \phi_1 \, \mathbf{R} \, \phi_2$ *and* $\varepsilon \models_e \phi_2$

Proof. Immediate from the definition of $\models_e$. □

We now define our construction for M_ϕ.

Definition 27 (NFA M_ϕ). *Let $\phi \in \Phi_e^{\mathcal{AP}}$ be in PNF. Then we define NFA $M_\phi = (Q_\phi, \Sigma_{\mathcal{AP}}, q_{I,\phi}, \delta_\phi, F_\phi)$ as follows.*

- $Q_\phi = 2^{S(\phi)}$
- $\Sigma_{\mathcal{AP}} = 2^{\mathcal{AP}}$
- $q_{I,\phi} = \{\phi\}$

- *Let* $q, q' \in Q_\phi$ *(so* $q, q' \subseteq S(\phi)$*) and* $A \in \Sigma_{\mathcal{AP}}$*(so* $A \subseteq \mathcal{AP}$*). Also let* $\mathit{anf}(\bigwedge q) = \bigvee\{C_1, \ldots C_n\}$ *be the ANF conversion of* $\bigwedge q$*. Then* $(q, A, q') \in \delta$ *iff there exists* C_i *such that:*
  - $A \models_p \bigwedge \mathit{lits}(C_i)$*; and*
  - $q' = \mathit{nf}(C_i)$*.*
- $F_\phi = \{q \in Q_\phi \mid \varepsilon \models_e \bigwedge q\}$

Theorem 9 states that the above construction is correct. In the rest of this section, we will prove this claim. We first establish the following useful lemma.

Lemma 16 (Well-Formedness of M_ϕ)**.** *Let* $\phi \in \Phi_e^{\mathcal{AP}}$ *be in PNF, and let* $M_\phi = (Q_\phi, \Sigma_{\mathcal{AP}}, q_{I,\phi}, \delta_\phi, F_\phi)$*. Fix arbitrary* $q \in Q$*, and let*

$$\mathit{anf}(\bigwedge q) = \bigvee C_i,$$

Then for each C_i*,* $\mathit{nf}(C_i) \in Q_\phi$*.*

Proof. Follows from Lemma 12(4) and the fact that every subformula of every $\phi' \in \mathit{nf}(C_i)$ of form $\mathbf{N}\phi''$, $\phi'_1 \,\mathbf{U}\, \phi'_2$ or $\phi'_1 \,\mathbf{R}\, \phi'_2$ is also a subformula of ϕ. □

This lemma in effect says that every clause occurring in $\mathit{anf}(\bigwedge q)$ (recall q is a set of subformulas of ϕ) gives rise to transitions in M_ϕ, because the "next-state" formulas in such a clause involve subformulas of ϕ.

Theorem 9. We now prove Theorem 9 as follows. Let $\phi \in \Phi_e^{\mathcal{AP}}$ and $M_\phi = (Q_\phi, \Sigma_{\mathcal{AP}}, q_{I,\phi}, \delta_\phi, F_\phi)$. We recall that $\Sigma_{\mathcal{AP}} = 2^{\mathcal{AP}}$, and thus $(\Sigma_{\mathcal{AP}})^* = (2^{\mathcal{AP}})^*$. Thus, the words accepted by M_ϕ come form the same set as the sequences to interpret Extended Finite LTL formulas. To emphasize this connection, we use $A \in \Sigma_{\mathcal{AP}}$ and $\pi \in (\Sigma_{\mathcal{AP}})^*$ in the following. We will in fact prove a stronger result: for every $\pi \in (\Sigma_{\mathcal{AP}})^*$ and $q \in Q$, q accepts π in M_ϕ iff $\pi \models_e \bigwedge q$. The desired result then follows from the fact that this statement holds in particular for the start state, $q_{I,\phi}$, that $\bigwedge q_{I,\phi} = \phi$, and that as a result, $L(M_\phi) = [\![\phi]\!]_e$.

The proof proceeds by induction on π. For the base case, assume that $\pi = \varepsilon$ and fix $q \in Q$. We reason as follows.

$$
\begin{aligned}
q \text{ accepts } \varepsilon \text{ in } M \text{ iff } q \in F_\phi \quad & \text{Definition of acceptance} \\
\text{iff } \varepsilon \models_e \textstyle\bigwedge q \quad & \text{Definition of } F_\phi
\end{aligned}
$$

In the induction case, assume $\pi = A\pi'$ for some $A \in \Sigma_{\mathcal{AP}}$ (so $A \subseteq \mathcal{AP}$) and $\pi' \in (\Sigma_{\mathcal{AP}})^*$. The induction hypothesis states for any $q' \in Q$, q' accepts π' in M_ϕ iff $\pi' \models_e \bigwedge q'$ (recall $q' \subseteq S(\phi)$). Now fix $q \in Q$; we must prove that q accepts π in M_ϕ

iff $\pi \models_e \bigwedge q$. We reason as follows.

$\pi \models_e \bigwedge q$	
iff $A\pi' \models_e \bigwedge q$	$\pi = A\pi'$
iff $A\pi' \models_e \mathit{anf}(\bigwedge q)$	Theorem 8
iff $A\pi' \models_e \bigvee C_i$	$\mathit{anf}(\bigwedge q) = \bigvee C_i$ in ANF
iff $A\pi' \models_e C_i$ some i	Lemma 8(2)
iff $A \models_p \bigwedge \mathcal{L}$ and $\pi' \models_e \bigwedge \mathcal{F}$	Lemma 8(1), $\mathcal{L} = \mathit{lits}(C_i)$, $\mathcal{F} = \mathit{nf}(C)$
iff $A \models_p \bigwedge \mathcal{L}$ and $\pi' \models_e \bigwedge q'$ some $q' \in Q_\phi$	Lemma 16
iff $(q, A, q') \in \delta_\phi$ and $\pi' \models_e \bigwedge q'$ some $q' \in Q_\phi$	Definition of δ_ϕ
iff $(q, A, q') \in \delta_\phi$ and q' accepts w' in M	Induction hypothesis
iff q accepts w in M	Definition 26(2)

□

6.4.2 Discussion of Construction M_ϕ

We now comment of some aspects of M_ϕ, both from the standpoint of its complexity but also in terms of heuristics for improving the construction in practice.

Size of $|M_\phi|$. The key drivers for the size of $|M_\phi|$ are the size of its state space, Q_ϕ, and its transition relation, δ_ϕ. The next theorem characterizes these.

Theorem 10 (Bounds on Size of M_ϕ)**.** *Let $\phi \in \Phi_e^{\mathcal{AP}}$ be in PNF, and let $M_\phi = (Q_\phi, \Sigma_{\mathcal{AP}}, q_{I,\phi}, \delta_p hi, F_\phi)$. Then we have the following.*

1. $|Q_\phi| \leq 2^{|\phi|}$.
2. $|\delta_\phi| \leq 4^{|\phi|} \cdot 2^{|\mathcal{AP}|}$.

Proof. For the first statement, we note that there is a state in M_ϕ for each subset of $S(\phi)$, and that there are at most $2^{|\phi|}$ such subsets. The second follows from the fact that each pair of states can have at most $2^{|\mathcal{AP}|}$ transitions between them. □

It is worth noting that in the above result, the bound on the number of states is tight: it is $2^{|\phi|}$, not e.g. $2^{O(|\phi|)}$, which some tableau constructions for LTL yield. Also note that if ϕ contains multiple instances of the same subformula, then $|S(\phi)| < |\phi|$; this explains the inequality in Statement (1).

Optimizing M_ϕ. The size results in Theorem 10 are consistent with other tableau constructions; they are in the worst case exponential in the size of the formulas for which automata are constructed. This worst-case behavior cannot be avoided over-all, but it can often be mitigated heuristically. In the remainder of this section we consider different methods for doing so.

On-the-fly Construction of Q_ϕ. The construction in Definition 27 may be seen as pre-computing all possible states of M_ϕ. In practice many of these states are

unreachable, and adding them to Q_ϕ and computing their transitions is unnecessary work. One method for avoiding this work is to construct Q_ϕ in a demand-driven, or *on-the-fly* manner. Specifically, one starts with the state $q_{I,\phi}$ and adds this to Q_ϕ. Then one repeatedly does the following: select a state q in the current Q_ϕ whose transitions have not been computed, compute q's transitions, adding states into Q_ϕ so that each transition has a target in Q_ϕ. Stop when there are no states in Q_ϕ whose transitions have not been computed. The result of this strategy is that only states reachable from $q_{I,\phi}$ will be added into Q_ϕ.

Symbolic Representation of Transitions. In Definition 27 transition labels are represented concretely, as sets of atomic propositions. One can instead allow transition labels that are *symbolic*: these labels have form $\gamma \in \Gamma_e^{\mathcal{AP}}$ for some propositional Extended LTL formula γ. A transition labeled by such a gamma can be seen as summarizing all transitions in M_ϕ labeled by $A \subseteq \mathcal{AP}$ such that $A \models_p \gamma$. The construction given in Definition 27 suggests an immediate method for doing this: rather than labeling transitions by $A \subseteq \mathcal{AP}$ such that $A \models_p \bigwedge \mathit{lits}(C_i)$, instead label a single transition by $\mathit{lits}(C_i)$.

Relaxation of ANF. Our definition of ANF says that a formula is in ANF iff it has form $\bigvee C_i$, where each clause C_i has form $(\bigwedge \mathcal{L}) \wedge \mathbf{N}(\bigwedge \mathcal{F})$. The method we give for converting formulas into ANF involves the use of a routine for converting formulas into DNF, which can itself be exponential. We adopted this mechanism for ease of exposition, and also because in the worst case this exponential overhead

is unavoidable. However, requiring the propositional parts of clauses to be of form $\bigwedge \mathcal{L}$, where $\mathcal{L}$ consists only of literals, is unnecessarily restrictive: all that is needed for the construction of M_ϕ is to require clauses to be of form $\gamma \wedge \mathbf{N}(\bigwedge \mathcal{F})$, where $\gamma \in \Gamma_e^{\mathcal{AP}}$ is a proposition formula in Extended Finite LTL. Relaxing ANF in this manner eliminates the need for full DNF calculations in general, and can lead to time and space savings when transitions are being represented symbolically.

6.5 Implementation and Empirical Results

We have implemented our tableau construction as a C++ package. The user specifies a formula $\phi \in \Phi_e^{\mathcal{AP}}$ and the corresponding NFA M_ϕ is constructed as defined in Definition 27. For convenience, we also support the boolean implication operator $\rightarrow$ by immediately performing the syntactic rewrite $\phi \rightarrow \psi \equiv \neg\phi \vee \psi$ upon parsing the formula. Our current implementation utilizes an on-the-fly construction of Q_ϕ such that only states reachable from the initial state are constructed and recursed upon. We also support the option to represent transitions symbolically by bundling edges together as a single transition whose label is the disjunction of all merged edge labels.

We used the Spot [88] platform (v.2.8.1) to handle the parsing of input formulas as well as the syntactic representation of the constructed output graph. To support proper parsing and construction of a Finite LTL formula, we added support for the $\overline{\mathbf{X}}$ (Weak Next) operator. Additionally, several Spot-provided automatic formula

rewrites are based on standard LTL identities and do not hold under finite semantics; these were disabled (such as $\mathbf{X}\,\mathtt{tt} \equiv \mathtt{tt}$).

We performed our tableau construction on the benchmark set of 184 standard LTL formulas (92 formulas and their negations) used by Duret-Lutz [89]. As our semantics for Finite LTL differs from traditional LTL, a direct comparison of of times needed for automaton construction are not appropriate. We imposed a 60 minute time-out for each formula, marked by "t/o" when applicable (this only occurred for formulas 24 and 109). We also report formula complexity ($|\phi|$), which is the number of subformulas in ϕ. Experiments were carried out on a single machine with an Intel Core i5-6600K (4 cores), with 32 GB RAM and a 64-bit version of GNU/Linux.

Figures 6.2 and 6.3 contain the results of our experiments. For each formula in the testbed, four pieces of data are reported: the size of the formula, the number of states in the resulting NFA, the number of transitions in the NFA, and the time needed to build the automaton. In most cases the construction time is negligible, although in two cases — Formulas 24 and 109 — our tool did not terminate before the 60-minute time-out we imposed. The reasons for this behavior are under further investigation. We have presented a summarized set of the results in Figure 6.1. Formulas are ordered by complexity.

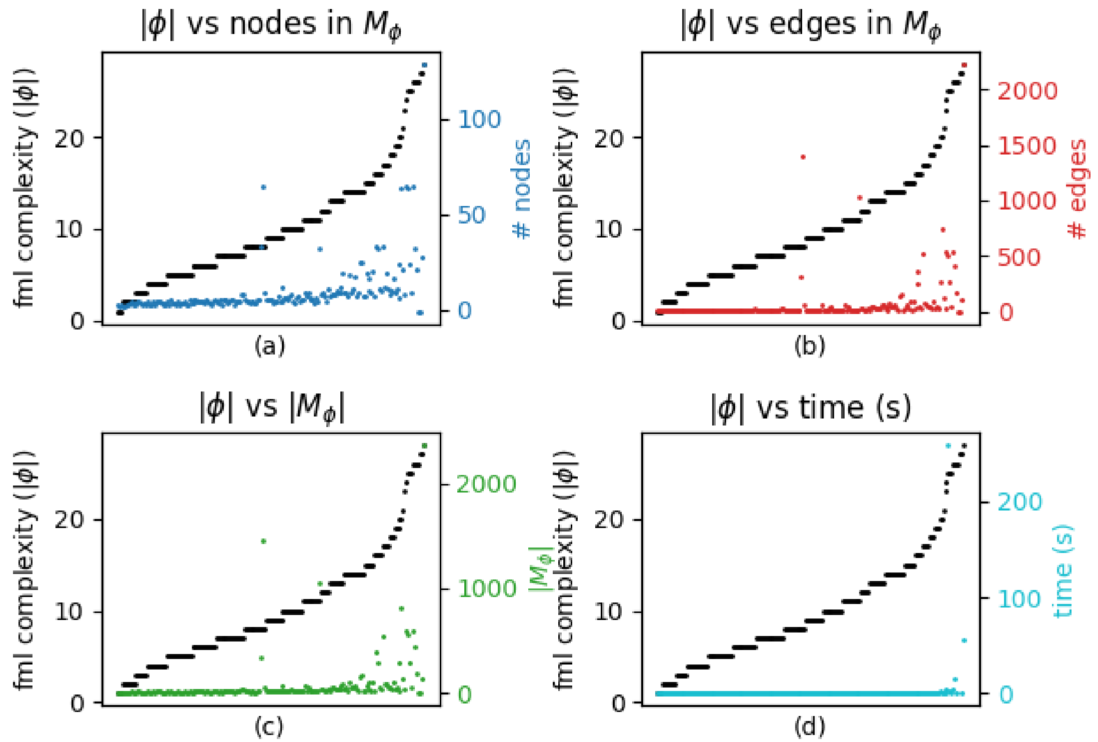


Figure 6.1: Summarized results of experiments. Formulas are ordered by complexity. (a) Formula complexity vs nodes in automaton M_ϕ. (b) Formula complexity vs edges in automaton M_ϕ. (c) Formula complexity vs nodes + edges in automaton M_ϕ. (d) Formula complexity vs computation time.

Figure 6.2: Experimental benchmark: Formulas 0–91.

ID	$\vert\phi\vert$	states	edges	time(s)
0	2	1	1	0.0031
1	3	2	3	0.0033
2	5	4	7	0.0033
3	6	5	9	0.0034
4	5	2	3	0.0034
5	6	3	6	0.0034
6	7	4	9	0.0039
7	8	5	10	0.0034
8	8	4	11	0.0045
9	9	5	16	0.0037
10	1	2	3	0.0031
11	2	1	1	0.0031
12	7	3	5	0.0032
13	8	4	11	0.0036
14	6	5	11	0.0035
15	7	5	11	0.0037
16	10	4	11	0.0048
17	11	5	13	0.0038
18	7	2	4	0.0037
19	8	3	5	0.0035
20	23	8	29	0.0039
21	24	65	534	3.9052
22	25	64	496	258.4280
23	26	65	535	4.1924
24	27	-	-	t/o
25	28	129	2234	55.1272
26	1	1	1	0.0031
27	2	2	3	0.0031
28	4	4	7	0.0037
29	5	5	9	0.0035
30	4	2	3	0.0032
31	5	3	6	0.0035
32	6	4	9	0.0042
33	7	5	10	0.0040
34	6	4	11	0.0043
35	7	5	16	0.0036
36	5	3	5	0.0033
37	6	4	11	0.0034
38	6	4	7	0.0034
39	7	5	9	0.0034
40	10	6	15	0.0035
41	11	9	56	0.0152
42	8	4	11	0.0053
43	9	5	10	0.0035
44	8	4	11	0.0059
45	9	5	16	0.0036

ID	$\vert\phi\vert$	states	edges	time(s)
46	4	2	4	0.0036
47	5	2	3	0.0034
48	10	6	13	0.0037
49	11	7	15	0.0048
50	7	3	7	0.0049
51	8	3	6	0.0038
52	12	8	30	0.0187
53	13	7	16	0.0048
54	18	8	40	0.3139
55	19	9	31	0.0527
56	10	5	9	0.0034
57	11	9	41	0.0067
58	11	5	11	0.0034
59	12	9	29	0.0067
60	16	7	18	0.0036
61	17	33	515	0.1389
62	13	8	36	0.0100
63	14	9	30	0.0070
64	13	8	40	0.0106
65	14	9	42	0.0062
66	9	5	10	0.0036
67	10	7	19	0.0039
68	11	6	15	0.0042
69	12	7	13	0.0036
70	15	7	18	0.0036
71	16	33	365	0.1412
72	13	8	31	0.0862
73	14	7	14	0.0042
74	17	12	76	0.4799
75	18	10	30	0.0042
76	10	8	33	0.0068
77	11	5	10	0.0035
78	14	18	77	0.0145
79	15	19	58	0.0072
80	14	9	41	0.0339
81	15	10	25	0.0041
82	16	32	249	0.6428
83	17	19	59	0.0072
84	26	32	411	15.4769
85	27	27	111	0.0223
86	7	4	13	0.0039
87	8	3	6	0.0039
88	13	10	29	0.0050
89	14	11	42	0.0079
90	9	3	7	0.0039
91	10	4	9	0.0039

Figure 6.3: Experimental benchmark: Formulas 92–183.

ID	$\|\phi\|$	states	edges	time(s)
92	15	16	93	0.0744
93	16	11	43	0.0077
94	20	24	267	2.0705
95	21	21	117	0.0196
96	9	4	13	0.0043
97	10	3	7	0.0034
98	16	10	29	0.0052
99	17	11	42	0.0120
100	11	3	7	0.0040
101	12	4	10	0.0039
102	18	16	93	0.0812
103	19	11	43	0.0129
104	25	24	267	2.4522
105	26	21	162	0.1342
106	19	6	20	0.0043
107	20	64	740	0.2893
108	25	9	36	0.0043
109	26	-	-	t/o
110	1	2	3	0.0034
111	2	2	3	0.0030
112	2	3	6	0.0039
113	3	4	11	0.0035
114	5	4	11	0.0042
115	6	3	6	0.0034
116	6	5	11	0.0035
117	7	5	16	0.0039
118	3	4	10	0.0035
119	4	4	11	0.0038
120	2	4	6	0.0040
121	3	3	6	0.0033
122	9	4	6	0.0039
123	10	6	13	0.0034
124	6	4	12	0.0039
125	7	5	11	0.0040
126	13	9	32	0.0049
127	14	17	55	0.0064
128	2	2	3	0.0040
129	3	2	3	0.0034
130	8	5	9	0.0036
131	9	7	10	0.0039
132	9	4	8	0.0034
133	10	7	10	0.0040
134	14	7	26	0.0062
135	15	9	15	0.0037
136	13	9	28	0.0060
137	14	9	15	0.0042

ID	$\|\phi\|$	states	edges	time(s)
138	4	2	4	0.0036
139	5	2	3	0.0033
140	6	3	5	0.0037
141	7	2	4	0.0037
142	7	4	12	0.0035
143	8	5	11	0.0041
144	4	4	16	0.0037
145	5	3	5	0.0032
146	4	4	8	0.0032
147	5	3	4	0.0034
148	8	33	312	0.0903
149	9	12	52	0.0051
150	13	12	52	0.0218
151	14	25	132	0.0081
152	13	19	63	0.0280
153	14	25	150	0.0088
154	10	4	8	0.0044
155	11	6	13	0.0040
156	6	2	2	0.0035
157	7	4	6	0.0036
158	2	3	5	0.0033
159	3	3	5	0.0031
160	3	2	3	0.0037
161	4	4	11	0.0038
162	4	3	5	0.0037
163	5	4	11	0.0037
164	14	7	13	0.0038
165	15	20	90	0.0653
166	4	2	3	0.0033
167	5	2	4	0.0032
168	6	4	6	0.0038
169	7	4	10	0.0036
170	4	3	5	0.0036
171	5	2	4	0.0038
172	5	5	10	0.0034
173	6	5	20	0.0041
174	4	2	4	0.0035
175	5	3	6	0.0036
176	10	5	9	0.0034
177	11	4	7	0.0040
178	11	32	1024	0.1093
179	12	11	25	0.0037
180	7	8	23	0.0046
181	8	65	1395	0.1501
182	7	8	28	0.0062
183	8	9	50	0.0045

6.6 Conclusion

This chapter has defined the logic Finite LTL, which uses the syntax of LTL but employs a semantics based on finite, rather than infinite, state sequences. It also has provided a tableau-inspired construction for converting formulas into non-deterministic finite automata whose languages consist of exactly the sequences making the corresponding formulas true. In the construction states are equivalent to subsets of subformulas of the input formula, and accepting states are defined as those where all contained subformulas satisfy the empty sequence. We have also observed that the check for satisfaction by the empty sequence can be done purely syntactically on the basis of the structure of the formula. We also have described a prototype implementation of the approach and given empirical results on an existing benchmark for standard LTL. Our methodology, while heavily inspired by the standard (infinite semantics) LTL community, transforms a Finite LTL formula directly into an NFA, and does not rely upon any intermediate representation from the standard LTL realm. The continued development of such "native" approaches will allow us to appropriately and accurately reason over domains with finite sequences.

Chapter 7: Finite LTL Query Checking

7.1 Introduction

A central problem in system analysis may be phrased as the *behavioral understanding problem*: given concrete observations of a system's behavior, infer high-level properties characterizing this behavior. Such properties can be used for a variety of purposes, including system specification, software understanding (when the system in question is software), and root-cause failure analysis. Several researchers have studied variants of this problems in a variety of contexts, from software engineering [1] to data mining [10] and artificial intelligence [90, 91].

This chapter considers the following variant of the behavioral understanding problem: given a finite set of system executions encoded as *data streams*, and a temporal-logic *query*, or "formula with a hole," infer formulas that, when substituted for hole, yield a temporal-logic formula satisfied by all data streams in the given set. For example, if the query in question has form $\mathbf{G}\,\mathtt{var}$, where `var` is the "hole" and $\mathbf{G}$ is the "always operator", then a solution ϕ would be a formula that is invariant across all data streams. Other temporal formulas can be used to characterize when error

conditions are tripped, or when temporal correspondences hold between different basic properties captured in the data streams. Using our query-solving technology, an engineer can collect different system executions, which might be in the form of system logs, or experimentally observed data, and then pose queries to develop insights into the mechanisms underpinning system behavior.

Drawing on the concepts explored in Chapter 6, in this chapter we take our representation of Finite LTL and use it as a base specification language for performing query checking.

7.2 Definitions and Problem Statements

We present here some formal definitions of chapter-specific concepts as well as some the exact problem statement of Finite LTL query checking. As Chapter 6 is foundational to the content presented herein, the reader is encouraged to refer to that chapter for all necessary definitions and results.

7.2.1 Finite Data Streams

We now give a formal definition of data stream that is used throughout the remainder of this chapter.

Definition 28 (Data Stream)**.** *Let $\mathcal{AP}$ be a set of atomic propositions and $\mathbb{N}$ the*

set of natural numbers. Then a data stream *over $\mathcal{AP}$ is a finite sequence*

$$(t_0, A_0) \dots (t_{n-1}, A_{n-1}) \in (\mathbb{N} \times 2^{\mathcal{AP}})^*$$

such that the following holds for all $0 \leq i \leq j < n$: $t_i \leq t_j$. We sometimes refer to (t_i, A_i) in a data stream as an observation *and t_i as the* time stamp *of the observation. We use $\Pi^{\mathcal{AP}}$ to represent the set of all data streams over $\mathcal{AP}$.*

Intuitively, atomic propositions are observations that can be made about the state of a system as it executes. Data stream $\pi = (t_0, A_0) \dots (t_{n-1}, A_{n-1})$ can then be seen as the result of observing the system for a finite period of time, where at each time instant t_i the atomic propositions $A_i \subseteq \mathcal{AP}$ are true while those in $\mathcal{AP}/A_i$ are false. The condition imposed by Definition 28 on time stamps requires that time advance monotonically throughout the data stream. We use ε to denote the empty data stream. We write $|\pi| = n$ for the length of data stream $\pi = (t_0, A_0) \dots (t_{n-1}, A_{n-1})$, $\pi_i = (t_i, A_i)$ for the ith-indexed step in in π, and $\pi(i) = (t_i, A_i) \dots (t_{n-1}, A_{n-1})$ for the suffix of π obtained by removing the first i elements from the data stream. Note that π_i is only defined when $i < |\pi|$, while $\pi(i)$ is defined when $i \leq |\pi|$, and that $\pi(0) = \pi$ and $\pi(|\pi|) = \varepsilon$.

In the rest of the chapter we will focus on so-called *normalized* data streams, which are defined as follows.

Definition 29. *Data stream π over AP is* normalized *if and only if for all i such*

that $0 \leq i < |\pi|$, π_i *has form* (i, A_i).

In a normalized data stream, the time stamps of the elements in the sequence begin at 0 and increase by 1 at every step. In such data streams we can omit the explicit time stamp and instead represent the stream as a finite sequence $A_0 \ldots A_{n-1}$. For normalized data streams represented as $\pi = A_0 \ldots A_{n-1}$ we abuse notation and write $\pi(i)$ as follows: $\pi(i) = A_i \ldots A_{n-1}$. This definition makes $\pi(i)$ normalized.[1]

7.2.2 Query Checking for Finite Data Streams

In our previous work on LTL Query Checking [84] as presented in Chapter 5, we were interested in solving LTL *queries* over Kripke structures. In that setting a query is a formula containing a missing propositional subformula; the goal of LTL query-checking in this case is to construct solutions for the missing subformula. In this chapter, we instead are interested in Finite LTL formulas and finite data streams obtained by observing the behavior of the system in question. This section defines the problem precisely and proves results used later in the chapter. When considering a set of finite data streams, we operate on an underlying assumption that the streams are all derived from the same source, e.g. are different executions of the same system. In what follows we restrict $\mathcal{AP}$ to be finite as well as non-empty.

Finite LTL queries correspond to Finite LTL formulas with a missing propositional subformula, which we denote `var`. It should be noted that `var` stands for an

[1] This detail, while necessary to point out, is not important in what follows, since the properties we consider in this chapter are insensitive to specific time-stamp values.

unknown *propositional formula*; it is *not* analogous to a (fresh) atomic proposition. The syntax of queries is as follows:

$$\phi := \texttt{var} \mid a \in \mathcal{AP} \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \mathbf{X}\,\phi \mid \phi_1\,\mathbf{U}\,\phi_2$$

In this chapter we only consider the case of a single propositional unknown, although the definitions can naturally be extended to multiple such unknowns, as well as missing subformulas lifted to arbitrary Finite LTL formulas rather than only propositional formulas. We often write $\phi[\texttt{var}]$ for an LTL query with unknown $\texttt{var}$, and $\phi[\gamma]$ for the LTL formula obtained by replacing all occurrences of $\texttt{var}$ by LTL propositional formula γ. If $\gamma[\texttt{var}]$ is a query containing no modalities then we call $\gamma[\texttt{var}]$ a *propositional query*. We write $\Phi[\texttt{var}]$ for the set of Finite LTL queries and $\Gamma[\texttt{var}] \subsetneq \Phi[\texttt{var}]$ for the set of propositional queries. We also lift the notion of logical equivalence, $\equiv$, to LTL queries as follows: $\phi_1[\texttt{var}] \equiv \phi_2[\texttt{var}]$ iff $\phi_1[\gamma] \equiv \phi_2[\gamma]$ for all $\gamma \in \Gamma$.

The query-checking problem $\mathrm{QC}(\Pi, \phi[\texttt{var}])$ may be formulated as follows.

Given: Finite set Π of normalized data streams, Finite LTL query $\phi[\texttt{var}]$

Compute: All propositional formulas γ such that for all $\pi \in \Pi$, $\pi \models \phi[\gamma]$

If γ is such that $\pi \models \phi[\gamma]$ for all π in Π, then we call γ a *solution* for Π and $\phi[\texttt{var}]$, and in this case we say that $\phi[\texttt{var}]$ is solvable for π. Computing all solutions for $\mathrm{QC}(\Pi, \phi[\texttt{var}])$ cannot be done explicitly, since the number of propositional formulas

is infinite. However, since $\mathcal{AP}$ is finite it turns out that logical equivalence, $\equiv$, induces a finite number of equivalence classes on Γ. We can use these equivalence classes as finite representations of the set of solutions to a given query.

As an example Finite LTL query, consider $\mathbf{G}\,\texttt{var}$. A solution to this query would yield a formula that is invariant at every observation in every data stream in Π. Another example of a finite LTL query $\phi[\texttt{var}]$ is $\mathbf{G}\,(\texttt{var} \rightarrow \mathbf{F}\,err)$. Assuming err is an atomic proposition representing the occurrence of an error condition, a solution to this query would give conditions guaranteed to trigger a future system error. Such information could be useful in subsequent root-cause analyses of why the error occurred.

7.3 From Finite LTL Queries to Finite Automata

This section discusses how, given a Finite LTL query, how to compute a corresponding automaton, similar to the methodology presented in Chapter 5 for standard LTL.

7.3.1 Finite Query Automata

A *Finite Query Automaton* is like a PNFA whose edge transition labels are propositional queries and whose acceptance condition depends on the propositional unknown embedded in the queries.

Definition 30. *Let* $\boldsymbol{var}$ *be an unknown proposition. A* Finite Query Automaton (FQA) $B[\boldsymbol{var}]$ *is a quintuple* $(Q, \mathcal{AP}, q_0, \delta[\boldsymbol{var}], F[\boldsymbol{var}])$*, where:*

- *Q is a finite set of states;*
- *$\mathcal{AP}$ is a finite, non-empty set set of atomic proposition;*
- *$q_0 \in Q$ is the initial state;*
- *$\delta[\boldsymbol{var}] \subseteq Q \times \Gamma[\boldsymbol{var}] \times Q$ is the transition relation;*
- *$F[\boldsymbol{var}] \in \Gamma \to 2^Q$, the acceptance condition, is required to satisfy $F[\boldsymbol{var}](\gamma_1) = F[\boldsymbol{var}](\gamma_2)$ whenever $\gamma_1 \equiv \gamma_2$.*

If $\gamma \in \Gamma$ is a propositional formula then we write $B[\gamma]$ for the instantation *of $B[\boldsymbol{var}]$ with γ for the PNFA $(Q, \mathcal{AP}, q_0, \delta[\gamma], F[\gamma])$, where $F[\gamma] = F[\boldsymbol{var}](\gamma)$ and*

$$\delta[\gamma] = \{(q, \gamma'[\gamma], q') \mid (q, \gamma'[\boldsymbol{var}], q') \in \delta[\boldsymbol{var}]\}$$

An FQA is intended to be the automaton analog of a Finite LTL query, where `var` is the unknown proposition to be solved for. An instantiation of an FQA with γ is then the PNFA obtained by replacing `var` by propositional formula γ. Note that this replacement can have two effects on the language of the instantiation: one via the transition relation, and the other via the accepting / non-accepting status of states. This latter explains why the accepting condition of an FQA is a function mapping propositional formulas to sets of states.

Our method for query-solving is automaton-theoretic; it is based on constructing a FQA $B_{\phi[\texttt{var}]}[\texttt{var}]$ from an LTL query $\phi[\texttt{var}]$. Our method for computing

$B_{\phi[\mathtt{var}]}[\mathtt{var}]$ is uses a modification of the tableau construction presented in the previous section; we only sketch the details here. It can be shown that any query $\phi[\mathtt{var}]$ can be put into PNF, where $\neg$ can only be applied to atomic propositions or instances of var. Automaton normal form can also be extended to PNF queries, where each clause has form $\bigwedge_i \ell_i \wedge \oplus\phi'[\mathtt{var}]$, with each ℓ_i either a literal, an occurrence of var, or an occurrence of $\neg\,\mathtt{var}$; $\oplus \in \{\mathbf{X}, \overline{\mathbf{X}}\}$; and $\phi'[\mathtt{var}]$ a query consisting of a conjunction of subformulas from $\phi[\mathtt{var}]$. We may then define $B_{\phi[\mathtt{var}]}[\mathtt{var}]$ to be $(Q, \mathcal{AP}, \delta[\mathtt{var}], \phi[\mathtt{var}], F[\mathtt{var}])$, where each $q \in Q$ is a query $\phi_q[\mathtt{var}] \in Q$ consisting of a conjunction of subformulas of $\phi[\mathtt{var}]$, each $(q, \gamma[\mathtt{var}], q') \in \delta[\mathtt{var}]$ is a propositional query based on the tableau construction, and $F[\mathtt{var}](\gamma) = \{q \mid \varepsilon \models \phi_q[\gamma]\}$. We have the following.

Theorem 11. *Let $\phi[\mathtt{var}]$ be a PNF Finite LTL query, and let γ be a propositional formula. Then $L(B_{\phi[\mathtt{var}]}[\gamma]) = [\![\phi[\gamma]]\!]$.*

7.3.2 Composing Finite Automata

We close this section by defining a language-intersection composition operation, $\otimes$, on PNFAs.

Definition 31. *Let B_i, where $i \in \{1, 2\}$, be PNFAs $(Q_i, \mathcal{AP}, q_i, \delta_i, F_i)$. Then $B_1 \otimes B_2$ is PNFA $(Q_1 \times Q_2, \mathcal{AP}, (q_1, q_2), \delta_{1,2}, F_1 \times F_2)$ where*

$$\delta_{1,2} = \{((q_1', q_2'), \gamma_1 \wedge \gamma_2, (q_1'', q_2'')) \mid (q_1', \gamma_1, q_1'') \in \delta_1 \text{ and } (q_2', \gamma_2, q_2'') \in \delta_2\}.$$

Operation $\otimes$ can be extended to the case when one of the B_i is a QFA in an obvious manner. WLOG assume B_1 is PNFA $(Q_1, \mathcal{AP}, q_1, \delta_1, F_1)$ and B_2 is FQA $(Q_2, \mathcal{AP}, q_2, \delta_2[\mathtt{var}], F_2[\mathtt{var}])$. Then $B_1 \otimes B_2[\mathtt{var}]$ is the QFA

$$(Q_1 \times Q_2, \mathcal{AP}, (q_1, q_2), \delta_{1,2}[\mathtt{var}], F_{1,2}[\mathtt{var}])$$

where $\delta_{1,2}[\mathtt{var}]$ is defined as $\delta_{1,2}$ in Definition 31 and $F_{1,2}[\mathtt{var}](\gamma) = F_1 \times F_2[\mathtt{var}](\gamma)$. We have the following.

Theorem 12. *Let B_1 be a PNFA.*

1. *If B_2 is a PNFA then $L(B_1 \otimes B_2) = L(B_1) \cap L(B_2)$.*
2. *If $B_2[\mathtt{var}]$ is a FQA then for $\gamma \in \Gamma$, $L((B_1 \otimes B_2[\mathtt{var}])[\gamma]) = L(B_1) \cap L(B_2[\gamma])$.*

7.4 Solving $QC(\Pi, \phi[\mathtt{var}])$

In this section we present our approach to solving the general query-checking problem $QC(\Pi, \phi[\mathtt{var}])$: given a finite set Π of data streams and Finite LTL query $\phi[\mathtt{var}]$, compute all γ such that for each $\pi \in \Pi$, $\pi \models \phi[\gamma]$. We first consider the case when $\Pi = \{\pi\}$ is a singleton set. We then study the multi-stream case.

The basic approach for the single-stream case is as follows. Given (normalized) data stream π, we first construct PNFA B_π such that $L(B_\pi) = \{\pi\}$. Then we construct FQA $B_{\neg\phi[\mathtt{var}]}[\mathtt{var}]$ from LTL query $\phi[\mathtt{var}]$; this QFA is such that for any γ, $L(B_{\neg\phi[\mathtt{var}]}[\gamma]) = \{\pi' \mid \pi' \not\models \phi[\gamma]\}$. Then, if $L(B_\pi) \cap L(B_{\neg\phi[\mathtt{var}]}[\gamma]) = \emptyset$, we have that

$\pi \models \phi[\gamma]$, and γ is a solution to query $\phi[\texttt{var}]$ and $\{\pi\}$. To solve $QC(\{\pi\}, \phi[\texttt{var}])$, it suffices to construct $B_{\pi,\phi[\texttt{var}]}[\texttt{var}] = B_\pi \otimes B_{\neg\phi[\texttt{var}]}[\texttt{var}]$ and then compute all γ such that $L(B_{\pi,\phi[\texttt{var}]}[\gamma]) = \emptyset$.

The key operations remaining to be addressed in this approach are the construction of B_π and the search for γ such that $L(B_{\pi,\phi[\texttt{var}]}[\gamma]) = \emptyset$. We call such γ *shattering formulas* in what follows. We consider each of these in turn.

7.4.1 Constructing PNFA B_π for Data Stream π

The construction of B_π from normalized data stream π is given below. Let $\pi = A_0 \ldots A_{n-1}$. Also, if $A \subseteq \mathcal{AP}$ defined $\langle A \rangle \in \Gamma$ as follows:

$$\langle A \rangle = \bigwedge \{a \mid a \in A\} \wedge \bigwedge \{\neg b \in \mathcal{AP} \mid b \notin A\}.$$

Then we define $B_\pi = (Q, \mathcal{AP}, q_0, \delta_\pi, \{q_n\})$, where $Q = \{q_0, \ldots, q_n\}$ and $\delta_\pi = \{(q_i, \langle A_i \rangle, q_{i+1}) \mid i < |\pi|\}$. We have the following.

Lemma 17. *Let π be a normalized data stream. Then $L(B_\pi) = \{\pi\}$.*

Proof. Immediate. Note that if $\pi = \varepsilon$ then $Q = \{q_0\}$, $\delta_\pi = \emptyset$ and $F = \{q_0\}$. □

7.4.2 Computing Shattering Formulas for $B_{\pi,\phi[\texttt{var}]}[\texttt{var}]$

We now consider how to compute shattering formulas for FQA $B_{\pi,\phi[\texttt{var}]}[\texttt{var}]$.

7.4.2.1 Shattering Propositional Queries.

Our approach to shattering $B[\texttt{var}]$ relies on selecting values γ' for var that cause some transitions in $B[\texttt{var}]$ to be disabled because the labels of such transitions, which have form $\gamma[\texttt{var}]$, become unsatisfiable when instantiated to $\gamma[\gamma']$.[2] In effect, this disables the transition labeled by $\gamma[\gamma']$ in $B[\gamma']$, as the transition can never be taken while processing an observation from a data stream. When such a γ exists, we call $\gamma[\texttt{var}]$ *shatterable* and the corresponding γ' a *shattering formula* for $\gamma[\texttt{var}]$.

In the rest of this section we formalize an account of shatterability for propositional queries $\gamma[\texttt{var}]$ that leads to an efficient representation of the shattering formulas for a $\gamma[\texttt{var}]$. We start by considering special cases of $\gamma[\texttt{var}]$. Specifically, we say that var is *positive* in propositional query $\gamma[\texttt{var}]$ iff every occurrence of var is within the scope of an even number of negations and *negative* iff every occurrence is within the scope of an odd number of negations. In what follows we also write $\gamma_1 \leq \gamma_2$ if γ_1 is at least as weak as γ_2.

Theorem 13. *Let $\gamma[\texttt{var}]$ be shatterable.*

1. *If **var** is positive in $\gamma[\texttt{var}]$, then there is a unique (modulo $\equiv$) weakest shattering formula γ' for $\gamma[\texttt{var}]$, and for every γ'' such that $\gamma' \leq \gamma''$, γ'' is also a shattering formula for $\gamma[\texttt{var}]$.*

2. *If **var** is negative in $\gamma[\texttt{var}]$, then there is a unique (modulo $\equiv$) strongest shat-*

[2] A formula γ is unsatisfiable iff $\gamma \equiv$ *false*.

tering formula γ' for $\gamma[\texttt{var}]$, and for every γ'' such that $\gamma'' \leq \gamma'$, γ'' is also a shattering formula for γ'.

Proof. Case 1 follows from the fact that if $\texttt{var}$ is positive in $\gamma[\texttt{var}]$ then $\gamma[\texttt{var}]$ can be rewritten as $\gamma''[\texttt{var}]$, where $\gamma[\texttt{var}] \equiv \gamma''[\texttt{var}]$ and $\gamma''[\texttt{var}]$ has form $\texttt{var} \wedge \gamma'''$ with $\gamma''' \in \Gamma$ (i.e. γ''' has no occurrences of $\texttt{var}$). It can easily be shown that γ' in this case is $\neg\gamma'''$. The other case is dual and is omitted. □

Characterizing the shatterability of general $\gamma[\texttt{var}]$, in which $\texttt{var}$ may appear both positively and negatively, is more complex and relies on the following.

Definition 32. *Propositional query $\gamma[\texttt{var}]$ is in* shattering normal form (SNF) *iff it has form $\gamma_1 \vee (\texttt{var} \wedge \gamma_2) \vee ((\neg\, \texttt{var}) \wedge \gamma_3)$, where each $\gamma_i \in \Gamma$ and is in DNF.*

Lemma 18. *For every $\gamma[\texttt{var}] \in \Gamma[\texttt{var}]$ there is a $\gamma'[\texttt{var}] \in \Gamma[\texttt{var}]$ in SNF such that $\gamma[\texttt{var}] \equiv \gamma'[\texttt{var}]$.*

Proof. Observe that if we view $\texttt{var}$ as an atomic proposition then we can also treat $\gamma[\texttt{var}]$ as a propositional formula and convert it into DNF in such a way that every clause has either no occurrences of $\texttt{var}$, or one occurrence of $\texttt{var}$, or one occurrence $\neg\,\texttt{var}$. We finish building $\gamma'[\texttt{var}]$ by grouping the clauses containing $\texttt{var}$ and then factoring out $\texttt{var}$, and similarly for $\neg\,\texttt{var}$. □

Theorem 14. *Let $\gamma[\texttt{var}] = \gamma_1 \vee (\texttt{var} \wedge \gamma_2) \vee ((\neg\, \texttt{var}) \wedge \gamma_3)$ be in SNF.*

1. *$\gamma[\texttt{var}]$ is shatterable iff γ_1 is unsatisfiable and $\neg\gamma_2 \leq \gamma_3$.*

2. *If* γ_1 *is unsatisfiable then* γ' *shatters* $\gamma[\textbf{\texttt{var}}]$ *iff* $\neg\gamma_2 \leq \gamma' \leq \gamma_3$.

Proof. Follows from the definition of shatterability and Theorem 13. □

As a consequence of this theorem and Lemma 18, we have that the set of shattering formulas for any shatterable propositional query $\gamma[\texttt{var}]$ can be represented as an interval $[\gamma_1, \gamma_2] = \{\gamma' \mid \gamma_1 \leq \gamma' \leq \gamma_2\}$. We sometimes refer to $[\gamma_1, \gamma_2]$ as a *shattering interval*. Moreover, we have finitary representations of the equivalences classes modulo $\equiv$ of γ_1 and γ_2. Also note that when $\gamma[\texttt{var}]$ is such that `var` is positive this interval has form $[\gamma', \mathit{false}]$, while if `var` is negative then the interval has form $[\mathit{true}, \gamma']$; here the γ' are the shattering formulas guaranteed by Theorem 13.

7.4.2.2 Shattering Finite Query Automata.

We now turn to the question of shattering FQA $B_{\pi,\phi[\texttt{var}]}[\texttt{var}]$. Our approach relies on using the results in the previous section to shatter transitions in $B[\texttt{var}]$ and compute PNFAs that we then check for for language-emptiness. We explain each of these operations in turn.

Transition shattering. The purpose of this step is to select a collection of individually shatterable transitions in $B[\texttt{var}]$ — i.e. transitions of form $(q, \gamma[\texttt{var}], q')$ where $\gamma[\texttt{var}]$ is shatterable — and compute a joint shattering interval, if one exists. That this is possible is due to the following.

Lemma 19 ([92])**.** *Let* $[\gamma_1', \gamma_1'']$ *and* $[\gamma_2', \gamma_2'']$ *be shattering intervals for* $\gamma_1[var]$ *and*

$\gamma_2[\boldsymbol{var}]$, respectively. Then $[\gamma_1' \wedge \gamma_2', \gamma_1'' \vee \gamma_2'']$ is the shattering interval for query $\gamma_1[\boldsymbol{var}] \wedge \gamma_2[\boldsymbol{var}]$.

In what follows we write $[\gamma_1', \gamma_1''] \wedge [\gamma_2', \gamma_2'']$ for $[\gamma_1' \wedge \gamma_2', \gamma_1'' \vee \gamma_2'']$. This lemma gives us an immediate means for computing the shattering interval of any non-empty finite subset of shatterable propositional queries, and allows us to compute shattering intervals that shatter a given collection of transition labels in $B_{\pi,\phi[\mathtt{var}]}[\mathtt{var}]$, or determine that such a set cannot be shattered).

Accepting States. Recall that states in $B_{\pi,\phi[\mathtt{var}]}[\mathtt{var}]$ have form (q, r), where q is a state in B_π and r is a state in $B_{\neg\phi[\mathtt{var}]}[\mathtt{var}]$ and therefore has form $\phi_r[\mathtt{var}]$, where $\phi_r[\mathtt{var}]$ is an LTL query. As seen above, the propositional formula γ can affect the transitions in PNFA $B_{\pi,\phi[\mathtt{var}]}[\gamma]$ by disabling some of those in $B_{\pi,\phi[\mathtt{var}]}[\mathtt{var}]$; it can also affect the accepting states. We explore this issue in what follows.

Define an equivalence relation $\sim_\varepsilon \subseteq \Gamma \times \Gamma$ as follows: $\gamma_1 \sim_\varepsilon \gamma_2$ iff it is the case that $\varepsilon \models \gamma_1$ iff $\varepsilon \models \gamma_2$. It is easy to see that $\sim_\varepsilon$ induces two equivalence classes on Γ: $[\mathit{true}]_{\sim_\varepsilon}$, consisting of γ such that $\varepsilon \models \gamma$, and $[\mathit{false}]_{\sim_\varepsilon}$, consisting of γ' such that $\varepsilon \not\models \gamma'$. We have the following.

Lemma 20. *Let $\gamma_1, \gamma_2 \in \Gamma$.*

1. *If $\gamma_1 \equiv \gamma_2$ then $\gamma_1 \sim_\varepsilon \gamma_2$.*

2. *If $\gamma_1 \sim_\varepsilon \gamma_2$ then for any Finite LTL query $\phi[\boldsymbol{var}]$, $\varepsilon \models \phi[\gamma_1]$ iff $\varepsilon \models \phi[\gamma_2]$.*

Proof. (1) is immediate. (2) relies on the fact that determining if $\varepsilon \models \phi$ for Finite LTL formula ϕ can be computed inductively on the structure of ϕ. □

Based on this lemma, we observe that for any $\gamma \in \Gamma$, $B_{\pi,\phi[\texttt{var}]}[\gamma]$ can have only one of two possible sets of accepting states, depending on whether $\pi \models \gamma$. Specifically, define F_π to be set of accepting states in B_π and R to be the set of states in $B_{\neg\phi[\texttt{var}]}[\texttt{var}]$. Also define the following.

$$\begin{aligned} F_\varepsilon &= F_\pi \times \{r \in R \mid \varepsilon \models \phi_r[\gamma] \text{ some } \gamma \in \Gamma\} \\ F_{\neg\varepsilon} &= F_\pi \times \{r \in R \mid \varepsilon \not\models \phi_r[\gamma] \text{ some } \gamma \in \Gamma\} \end{aligned}$$

Then, for any $\gamma \in \Gamma$, the accepting states of $B_{\pi,\phi[\texttt{var}]}[\gamma]$ are either F_ε or $F_{\neg\varepsilon}$.

Computing Shattering Formulas for $B_{\pi,\phi[\texttt{var}]}[\gamma]$. We now describe an algorithm for computing a representation of the shattering conditions for $B_{\pi,\phi[\texttt{var}]}[\gamma]$.

Input: $B_{\pi,\phi[\texttt{var}]}[var]$

Output: Set SC of shattering formulas, represented implicitly

1. Eliminate all *spurious* transitions in $B_{\pi,\phi[\texttt{var}]}[\texttt{var}]$, i.e. transitions of the form $((q,r), \texttt{var}[\gamma], (q',r'))$ such that $\gamma[\gamma']$ is unsatisfiable for all $\gamma \in \Gamma$.

2. Replace all *unshatterable* transitions $((q,r), \gamma[\texttt{var}], (q',r'))$ by $((q,r), \mathit{true}, (q',r'))$. Call the remaining set of transitions δ.

3. $TC := \{\gamma[\texttt{var}] \mid ((q,r), \gamma[\texttt{var}], (q',r')) \in \delta \text{ some } q, q', r, r'\}$.

4. For each $\gamma[\texttt{var}] \in TC$ compute the shattering interval $I_{\gamma[var]}$.

5. $SC := \emptyset$, $US := \Gamma - (\bigcup_{\gamma[\texttt{var}] \in TC} I_{\gamma[\texttt{var}]})$

6. If $US \cap [true]_{\sim_\varepsilon} \neq \emptyset$ and $L(B_{US,\varepsilon}) = \emptyset$, where $B_{US,\varepsilon}$ is PNFA whose states are the states of $B_{\pi,\phi[\texttt{var}]}[\gamma]$, whose transition relation is $\{((q,r), true, (q',r')) \mid ((q,r), \gamma[\texttt{var}], (q',r')) \in \delta \text{ some } q, q', r, r'\}$ and whose accepting states are F_ε then add $US \cap [true]_{\sim_\varepsilon}$ to SC

7. If $US \cap [false_{\sim_\varepsilon}] \neq \emptyset$ and $L(B_{US,\neg\varepsilon}) = \emptyset$, where $B_{US,\varepsilon}$ is PNFA whose states are the states of $B_{\pi,\phi[\texttt{var}]}[\gamma]$, whose transition relation is $\{((q,r), true, (q',r')) \mid ((q,r), \gamma[\texttt{var}], (q',r')) \in \delta \text{ some } q, q', r, r'\}$ and whose accepting states are $F_{\neg\varepsilon}$ then add $US \cap [false]_{\sim_\varepsilon}$ to SC.

8. For each nonempty subset $T \subseteq TC$ do

   (a) Compute $I_T = \bigwedge_{\gamma[\texttt{var}] \in TC} I_{\gamma[\texttt{var}]}$.

   (b) If $I_T \cap US \cap [true]_{\sim_\varepsilon} \neq \emptyset$ and $L(B_{T,\varepsilon}) = \emptyset$, where $B_{T,\varepsilon}$ is PNFA whose states are the states of $B_{\pi,\phi[\texttt{var}]}[\gamma]$, whose transition relation is $\{((q,r), true, (q',r')) \mid ((q,r), \gamma[\texttt{var}], (q',r')) \in \delta \text{ some } q, q', r, r', \gamma[\texttt{var}] \notin T\}$ then add $I_T \cap US \cap [true]_{\sim_\varepsilon}$ to SC

   (c) If $I_T \cap US \cap [false]_{\sim_\varepsilon} \neq \emptyset$ and $L(B_{T,\neg\varepsilon}) = \emptyset$, where $B_{T,\neg\varepsilon}$ is PNFA with states states of $B_{\pi,\phi[\texttt{var}]}[\gamma]$, whose transition relation is $\{((q,r), true, (q',r')) \mid ((q,r), \gamma[\texttt{var}], (q',r')) \in \delta \text{ some } q, q', r, r', \gamma[\texttt{var}] \notin T\}$ then add $I_T \cap [false]_{\sim_\varepsilon}$ to SC.

9. Return SC.

It can be shown that γ shatters $B_{\pi,\phi[\texttt{var}]}[\texttt{var}]$ iff $\gamma \models c \in SC$.

7.4.3 Solving $\mathrm{QC}(\Pi, \phi[\texttt{var}])$ when $|\Pi| > 1$

Having considered the case of query-checking a finite LTL query $\phi[\texttt{var}]$ for the single data stream case of $\Pi = \{\pi\}$, we now consider the generalized case of a finite set $\Pi = \{\pi_i\}$ of data streams. Naively one can compute individual solutions for each $\pi_i \in \Pi$. The solution for $\mathrm{QC}(\Pi, \phi[\texttt{var}])$ is all solutions found across solutions for each sub-problem $\mathrm{QC}(\{\pi_i\}, \phi[\texttt{var}])$. One could consider applying a in iterative approach which refines a running solution to the overall $\mathrm{QC}(\Pi, \phi[\texttt{var}])$ problem by first fixing an ordering for $\{\pi_i\}$. Let $|\Pi| = n$ and S_i be the solution for the first i data streams, namely $QC(\{\pi_1, \ldots, \pi_i\}, \phi[\texttt{var}])$. First, $S_1 = \mathrm{QC}(\{\pi_1\}, \phi[\texttt{var}])$ is computed as indicated in the above subsections. Then, to compute each subsequent S_i, we perform the individual $\mathrm{QC}(\{\pi_i\}, \phi[\texttt{var}])$, but restrict shattering intervals to be contained inside of S_{i-1}. This results in the shattering intervals of individual transitions to be narrowed and potentially wholly unshatterable under the restricted space, which allows us to remove these transitions from consideration. S_n the overall solution for $\mathrm{QC}(\Pi, \phi[\texttt{var}])$.

Another option is to leverage the fact that our methodology inherently converts data streams into finite automata. One could turn each data stream π_i into its own PNFA B_{π_i}, and then compute a composed automaton $B_c = \cap_i B_{\pi_i}$ for which $L(B_c)$ is equivalent (or related to) to $\cap_i L(B_{\pi_i})$, which would require performing a single (albeit more complex) query checking problem.

7.5 Experimental Results

We have implemented our methodology described in previous sections in C/C++. We make use of the Spot [88] platform (v.2.8.1) to handle the parsing of input formulas. Calls were also made to the Python SymPy symbolic computing package [93] to perform simplifications to convert propositional formulas into desired normal forms. The resulting code is able to support and represent finite state machines as well as finite query automaton, and perform all relevant operations as discussed above, including composition, language emptiness checking, and shattering computations.

We first evaluated our methodology using a dataset containing synthetic data generated representing product sales as influenced by promotions. This dataset contained sales volumes of fictitious 100 products as well as the status of 1000 promotions, each of which influences a subset of the product set. Three years of daily data was captured, totalling 1095 days total. We created a datastream with 1095 time points containing atomic propositions of the forms

$$prod_i \qquad i \in \{1, \ldots, 100\}$$

$$promo_i \qquad i \in \{1, \ldots, 1000\}$$

For a time point t of the data stream, $prod_i$ is true if product i's volume of sales as reported on day t was greater than as reported on day $t-1$, or false otherwise. $promo_i$ is true on time point t if promotion i is active on day t, and false otherwise.

We considered queries of the form

$$\phi_1[\texttt{var}] = \mathbf{G}(prod_i \rightarrow \mathbf{F}\,\texttt{var})$$

where `var` is restricted to be drawn only from product propositions. Loosely, this is the property stating that whenever product i increases in value from the previous day, some price change triggers in other products on the same day. This is intended to capture a correlation between products indicating that over all observed combinations of potential active promotions, the sales of two products are correlated positively. The more directed query

$$\phi_2[\texttt{var}] = \mathbf{G}(promo_i \wedge prod_j \rightarrow \mathbf{F}\,\texttt{var})$$

was also considered, again with the same restriction on `var`. This query could be useful when planning promotional strategy, as knowing that, were promotion i to be active, a raise in sales of product j would also lead to a raise in sales for other products (namely the solution to `var`), possibly precluding the need to commit additional resources to promote those other products through additional needs/promotions.

As $|\mathcal{AP}| = 1100$ is clearly too large for us to directly apply query checking, we sampled down the number of atomic propositions to random subsets and performed query checking on these finite streams. Figure 7.1 reports average running times

over 1000 samples for a fixed number of promotions and products. A hard cut-off of 10 minutes was imposed for all experiments. Configurations where all experiments timed out were reported with "t/o."

# products	# promotions	ϕ_1[`var`] avg time	ϕ_2[`var`] avg time
2	0	0.928 ± 0.201	-
3	0	1.192 ± 0.063	-
4	0	2.147 ± 0.140	-
5	0	36.700 ± 1.490	-
2	1	1.241 ± 0.089	2.582 ± 0.466
3	1	2.223 ± 0.206	8.348 ± 5.563
4	1	36.238 ± 12.061	t/o
2	2	2.011 ± 0.217	7.398 ± 5.300
3	2	31.315 ± 17.614	t/o
2	3	8.144 ± 10.030	t/o

Figure 7.1: Average running times to perform query checking for down-sampled data streams. ϕ_2[`var`] is not applicable when no promotionsa re present in stream. Averages taken over 1000 downsamples. All times in seconds.

Experiments were carried out on a single machine with 4 processors Intel© Core i5-6600K, with 32 GB RAM and a 64-bit version of GNU/Linux.

7.6 Conclusion

We have presented our work on performing LTL query checking over finite data streams using an automaton-theoretic approach. Solving such queries can aid in the comprehension of system behavior underlying observed execution traces, which might allow for one to better understand or diagnose a system when it cannot be accessed directly.

Chapter 8: Formal Verification of Noisy Sequences

8.1 Introduction

Material presented in Chapter 5 dealt with query checking on standard LTL, while Chapters 6 through 7 explored a variant of LTL with finite semantics in an attempt to better support real-world scenarios. However, both standard LTL and Finite LTL have a major drawback: their requirements are strict and rigid, and are intolerant to potential noise entering through execution data. Even a small deviation can be the deciding factor as to whether or not a system supports a property. This is true of other well-studied temporal logics as well. Consider the CTL formula $\mathbf{A}\,\mathbf{G}\,\mathbf{A}\,(\phi_1\,\mathbf{U}\,\phi_2)$. This formula is satisfied if *all* paths *always* hold that *all* executions satisfy $\phi_1\,\mathbf{U}\,\phi_2$. Such a stringent requirement is violated in any number of ways. Consider a data stream D that satisfies $\phi_1\,\mathbf{U}\,\phi_2$. This means that there is a range of time points, starting from the initial point in the stream, where ϕ_1 is satisfied at each of those points, and then ϕ_2 is satisfied immediately following this range of events. If ϕ_1 were not present in even one of the time points in the initial range, $\phi_1\,\mathbf{U}\,\phi_2$ would no longer hold true for D. Consequently, any set of data streams

D belongs to would not satisfy the formula $\mathbf{A}\,\mathbf{G}\,\mathbf{A}\,(\phi_1\,\mathbf{U}\,\phi_2)$. It is unsatisfying for such a small perturbation to impact the overall conclusion of a (potentially) much larger data set in such a drastic way. Yet, a scenario where this situation would occur is not rare. We list here several scenarios which could easily lead to such a circumstance:

- **Missing evidence** - Observances of atomic propositions are missing from data streams. Imperfect data acquisition, abstractions to the data stream representation, and system failures could all contribute to such a situation. The property $\phi\,\mathbf{U}\,\psi$ may never be discovered if several data streams have a missing ϕ annotation during stretches before ψ is encountered.

- **Changing/hybrid behavior** - Complex systems can lead to complex behavior. From a batch of system logs generated over time, a log file may satisfy the property $a\,\mathbf{U}\,b$ half of the time, and $a\,\mathbf{U}\,c$ the other half. This would suggest two different behavioral patterns, especially if b and c are not suspected to be related in any way. This overall property could be summarized to $a\,\mathbf{U}\,(b \vee c)$, but is weaker than either of the individual formulas.

- **Overfitting to data** - The set of data streams provided should not be expected to be every possible execution imaginable from the system, only a sample. We may conclude a stronger property than is actually true of the system because a counter-example was not present in the sample set. For example, we may conclude $a\,\mathbf{U}\,(b \wedge c)$ for the set of data streams we are given,

when in system satisfies only the weaker property $a \mathbf{U} b$. In this case, we may have always seen both b and c occurring to terminate the sequence of a's in each example data stream, but c's appearance is not mandated by the system.

While exactness may be desirable in cases where errors are intolerable, many scenarios would benefit from the extra knowledge that could be discoverable if "approximate" properties were expressible. Such properties would specify absolute properties (i.e. it is satisfiable by a data set) expressing approximate characteristics (i.e. the property is weaker than needed to to be satisfiable by the entire data set). This would translate to a stronger property being true all a portion of the time. From a system design perspective, being able to discover the set of properties at are true "most of the time" could lead to isolating properties intended to be invariant but are instead sometimes broken. Switching to a more black-box setting, being able to make statements such as "behavior ϕ was observed 75% of the time" is useful information which may provide further insight into a system under investigation. In this case, if the degree of deviation from an absolute is substantial, this might be indication that more complex behavior (such as $\phi \vee \psi$) underlies the mechanism.

8.2 Near/Partial Invariants

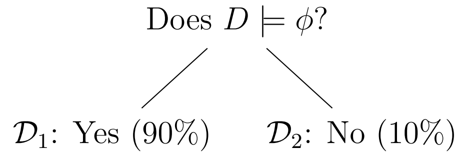
For some data sets and domains, expressing properties which apply to only a portion of the data set are of interest. Such properties can capture complex design choices where a property is true only in a fraction of the data set, as well as represent under-

lying complex behavior when the system is unknown or not fully characterized. This can be a growing concern in systems which themselves compose different subsystems for different situations. Consider the example of an online payment system. One might want the property "$\mathbf{G}\,(\text{payment sent} \implies \mathbf{F}\,\text{payment received})$" to be true over all data streams; namely for all executions, it is always the case that a payment sent eventually results in the same payment being received. However, an inspection of the logs from the system reveals that this property is true only for a portion of the streams. It may further reveal that the cases where the payment requirement was satisfied contained only data streams where the user paid by credit card, whereas in the cases where the property was not present users paid exclusively with PayPal. This would suggest something is wrong with the transaction system for PayPal, and suggests a direction for a system designer to look in order to determine the cause of the problem.

Similarly, for an organization using blockchain technology, one might be interested in identifying any potential double-spending, where a digital resource is spent in multiple transactions by the same user, typically by first making a copy of the digital resource. The simplified property "$\mathbf{G}(\text{resource spent} \implies \mathbf{X}\neg\mathbf{F}\,\text{resource spent})$" requires that if a digital resource is spent once, it is never spent again in the future (this model presumes that once the resource changes hands it has a new unique id allowing itself to be spent at most one time). Were the organization to utilize two blockchains internally, it might be the case that one chain becomes compromised or

otherwise exhibits double-spending but the other does not.

We introduce the notion of a *partial invariant*. These invariants are properties that are satisfied by only a portion of the evidence, in our case only a portion of the data streams. We use *near invariant* to refer to those properties that satisfy "almost all" data streams. We refer to properties that satisfy all evidence as *full* or *total* invariants. The current design envisions using cut-off thresholds to deem what is considered as a near or partial invariant, based upon what portion of the data streams must be satisfied. Consider the following scenario where the data streams have been partitioned according to whether or not they satisfy a property ϕ:



In this case there perhaps is a strong motivation to report ϕ as a partial (possibly near) invariant, as set $\mathcal{D}_1$ which is 90% of the total data streams observed satisfy ϕ. Performing a partitioning in this manner also allows us to further analyze the 10% portion that is exactly the set $\mathcal{D}_2$. In the case of the online payment situation given above, it might be found that all data streams in $\mathcal{D}_2$ satisfy a property stating that PayPal was used.

Partial invariants can be desirable for a number of reasons. We several examples include:

- **Design bug** – The system designer may have intended for the partial invari-

ant to be a total invariant, but some design issue led to a corner case being missed. In Chapter 4 we were able to infer a transition that was missing in a reconstructed state machine due to an error in system code.

- **Unexpected/Emergent behavior** – A partial invariant may represent behavior that manifested unexpectedly due to other characteristics of the system. This could correspond to undocumented specifications, or an unexpected emergent interaction between composed subsystems.

- **Inherently complex nature** – Partial invariants capture inherently more complex behavior than a total invariant because they occur only a portion of the time. Specifying the "equivalent" total invariant makes the invariant description itself much more complex. When designing a system, detecting or properly documenting them can be challenging, and may be entirely overlooked.

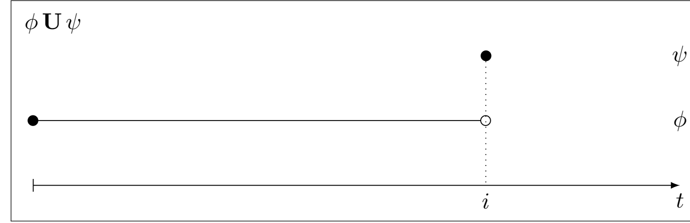
8.3 Noisy Linear Temporal Logic

With partial invariants in mind, we formalize the notion as a new temporal logic, which we call noisy Linear Temporal Logic (noisy LTL). We first provide some intuition behind the logic, and then detail the syntax and semantics as well as provide some example formulas and prove some useful properties about the logic. We then consider an alternative to the logic which may appeal to specific applications.

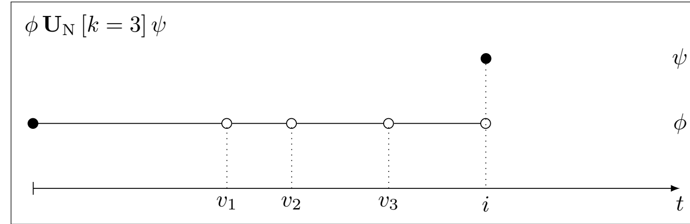
8.3.1 Characterizing Deviation from LTL

Schematically one can envision the semantics of traditional LTL as pertaining to requirements over ranges and points of time. For example, the LTL formula $a\,\mathbf{U}\,b$ requires that for some finite interval of the naturals $[0, i)$, a must be satisfied, until, at time point i, b must then be satisfied at time point i. With the goal to introduce a deviation from a requirement about a range, a natural adjustment that could be made is to allow for the range requirement to be enforced over "most" of the range rather than "all" of the range. Continuing the above example, we could allow up to 3 time points in the range where property a is not required inside of $[0, i)$. We call such change a *lax deviation*. Alternatively, introducing deviation on a requirement about a point could entail requiring a multiplicity of the point before the requirement is satisfied. Starting again from the property $a\,\mathbf{U}\,b$, one could require that the b property be satisfied an extra 3 times before the a range ends. We call such a change a *redundant deviation*. Figure 8.1 illustrate these deviations.

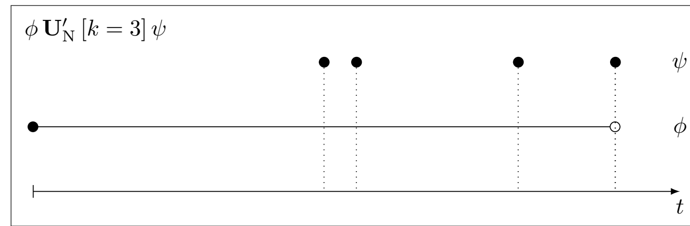
These sorts of types of deviations are exactly what noisy LTL captures. We will now show the syntax and semantics for the logic, where variants of existing LTL operators represent either a lax or redundant deviation.



(a) Sketch of $a\,\mathbf{U}\,b$.



(b) Laxness in $a\,\mathbf{U}\,b$.



(c) Redundancy in $a\,\mathbf{U}\,b$.

Figure 8.1: Adding noise to $a\,\mathbf{U}\,b$.

8.3.2 Syntax of Noisy LTL

We have positioned noisy LTL as an extension of traditional LTL, as such the syntax is a superset of traditional LTL.

Definition 33 (Noisy LTL Syntax)**.** *For a given $\mathcal{AP}$, the following grammar defines*

the set of noisy LTL formulas, with $a \in \mathcal{AP}$.

$$
\begin{aligned}
\phi ::= & \; a \mid \neg\phi \mid \phi \wedge \phi \mid \mathbf{X}\,\phi \mid \phi\,\mathbf{U}\,\phi && \textit{LTL (base)} \\
& \mid \boldsymbol{ff} \mid \boldsymbol{tt} \mid \phi \vee \phi \mid \phi\,\mathbf{R}\,\phi \mid \mathbf{F}\,\phi \mid \mathbf{G}\,\phi && \textit{LTL (derived)} \\
& \mid \phi\,\mathbf{U}_{\mathrm{N}}[c]\phi \mid \phi\,\mathbf{R}_{\mathrm{N}}[c]\phi && \textit{left-noisy operators} \\
& \mid \phi\,\mathbf{U}'_{\mathrm{N}}[c]\phi \mid \phi\,\mathbf{R}'_{\mathrm{N}}[c]\phi \mid \mathbf{F}'_{\mathrm{N}}[c]\phi \mid \mathbf{G}'_{\mathrm{N}}[c]\phi && \textit{right-noisy operators}
\end{aligned}
$$

We extend the term modal *to include the left- and right- noisy operators. We use $\Phi_N^{\mathcal{AP}}$ to refer to the set of all noisy LTL formulas and $\Gamma_N^{\mathcal{AP}} \subsetneq \Phi_N^{\mathcal{AP}}$ for the set of all propositional formulas (those formulas containing no modal operators). We will often write Φ and Γ instead of $\Phi_N^{\mathcal{AP}}$ and $\Gamma_N^{\mathcal{AP}}$ when it is clear from context.*

The first two rows of the grammar are the syntax for traditional LTL, along with derived operators presented in the previous section. Note that we have chosen a set of operators such that the language is closed under logical duality. The third row of the syntax contains operators that allow noise on the left-handed subformula, while the fourth row contains operators allowing noise of the right-handed subformula. We call operators from the third and fourth rows *noisy operators*, they are all denoted with a subscript "N". Right-handed noisy operators are additionally marked with a $'$. All noisy operators contain a *noise parameter c* which takes an integer as a value, intended to capture deviation from the traditional operator to which the noisy operator corresponds.

As is the case for traditional LTL, we have syntactically defined $\{\mathbf{U}_{\mathrm{N}}, \mathbf{R}_{\mathrm{N}}\}$ to be logical duals, as well as $\{\mathbf{U}'_{\mathrm{N}}, \mathbf{R}'_{\mathrm{N}}\}$, so both $\mathbf{R}_{\mathrm{N}}$ and $\mathbf{R}'_{\mathrm{N}}$ are included to make the logic closed under duality. The same is true for $\{\mathbf{F}'_{\mathrm{N}}, \mathbf{G}'_{\mathrm{N}}\}$, which are themselves derived from $\mathbf{U}'_{\mathrm{N}}$ (or equivalently $\mathbf{R}'_{\mathrm{N}}$). This is summarized as follows:

$$\phi_1 \,\mathbf{R}_{\mathrm{N}}[c]\phi_2 = \neg((\neg\phi_1)\,\mathbf{U}_{\mathrm{N}}[c](\neg\phi_2))$$

$$\phi_1 \,\mathbf{R}'_{\mathrm{N}}[c]\phi_2 = \neg((\neg\phi_1)\,\mathbf{U}'_{\mathrm{N}}[c](\neg\phi_2))$$

$$\mathbf{F}'_{\mathrm{N}}[c]\phi = \mathtt{tt}\,\mathbf{U}'_{\mathrm{N}}[c]\phi$$

$$\mathbf{G}'_{\mathrm{N}}[c]\phi = \neg\,\mathbf{F}'_{\mathrm{N}}[c](\neg\phi)$$

It might seem strange that there are no left-noisy operators for $\mathbf{F}$ or $\mathbf{G}$ $(\mathbf{F}_{\mathrm{N}}, \mathbf{G}_{\mathrm{N}})$. However, as the natural extension of $\mathbf{F}\,\phi = \mathtt{tt}\,\mathbf{U}\,\phi$ would be to make $\mathbf{F}_{\mathrm{N}}[c]\phi = \mathtt{tt}\,\mathbf{U}_{\mathrm{N}}[c]\phi$, this would result in a not-so-interesting operator as all the noise is on the left-handed subformula, namely $\mathtt{tt}$. A similar reason holds for $\mathbf{G}_{\mathrm{N}}[c]\phi = \mathtt{ff}\,\mathbf{R}_{\mathrm{N}}[c]\phi$. In fact, if the semantics (presented in the next section) of $\mathbf{U}_{\mathrm{N}}$ and $\mathbf{R}_{\mathrm{N}}$ are inspected, one can observe that $\{\mathbf{F}_{\mathrm{N}}, \mathbf{G}_{\mathrm{N}}\}$ collapse to $\{\mathbf{F}, \mathbf{G}\}$.

8.3.3 Semantics of Noisy LTL

We will now present the semantics of noisy LTL, starting with the semantics for the fragment representing standard LTL.

Definition 34 (Noisy LTL Semantics)**.** *Let ϕ be a noisy LTL formula and let $\pi \in$*

$(2^{\mathcal{AP}})^\omega$. *Then, the satisfaction relation* $\pi \models \phi$, *is defined inductively on the structure of* ϕ *as follows:*

- $\pi \models a \in \mathcal{AP}$ *iff* $a \in \pi_0$
- $\pi \models \neg\phi$ *iff* $\pi \not\models \phi$
- $\pi \models \phi_1 \wedge \phi_2$ *iff* $\pi \models \phi_1$ *and* $\pi \models \phi_2$
- $\pi \models \mathbf{X}\,\phi$ *iff* $\pi(1) \models \phi$
- $\pi \models \phi_1\,\mathbf{U}\,\phi_2$ *iff*

$$\exists i \in \mathbb{N} :$$
$$\pi(i) \models \phi_2 \wedge \forall j \in \mathbb{N} :$$
$$(0 \leq j < i) \implies \pi(j) \models \phi_1$$

- $\pi \models \phi_1\,\mathbf{U}_{\mathrm{N}}[c]\phi_2$ *iff*

$$\exists\,\mathcal{V} \subset \mathbb{N}, |\,\mathcal{V}\,| \leq c, \exists i \in \mathbb{N} :$$
$$\pi(i) \models \phi_2 \wedge \forall j \in \mathbb{N} - \mathcal{V} :$$
$$(0 \leq j < i) \implies \pi(j) \models \phi_1$$

- $\pi \models \phi_1 \, \mathbf{R}'_{\mathrm{N}}[c]\phi_2$ *iff*

$$\exists \mathcal{V} \subset \mathbb{N}, |\, \mathcal{V} \,| \leq c, \forall i \in \mathbb{N} - \mathcal{V}: \\ \pi(i) \models \phi_2 \vee \exists j \in \mathbb{N}: \\ (0 \leq j < i) \wedge \pi(j) \models \phi_1$$

The semantics for derived standard LTL ($\mathbf{R}$*,* $\mathbf{F}$*,* $\mathbf{G}$*) are the same as for standard LTL. The semantics for the remaining noisy operators follow from their syntactic relations to the above two noisy operators, and are explicitly given here:*

- $\pi \models \phi_1 \, \mathbf{R}_{\mathrm{N}}[c]\phi_2$ *iff*

$$\forall \mathcal{V} \subset \mathbb{N}, |\, \mathcal{V} \,| \leq c, \forall i \in \mathbb{N}: \\ \pi(i) \models \phi_2 \vee \exists j \in \mathbb{N} - \mathcal{V}: \\ (0 \leq j < i) \wedge \pi(j) \models \phi_1$$

- $\pi \models \phi_1 \, \mathbf{U}'_{\mathrm{N}}[c]\phi_2$ *iff*

$$\forall \mathcal{V} \subset \mathbb{N}, |\, \mathcal{V} \,| \leq c, \exists i \in \mathbb{N} - \mathcal{V}: \\ \pi(i) \models \phi_2 \wedge \exists j \in \mathbb{N}: \\ (0 \leq j < i) \implies \pi(j) \models \phi_1$$

- $\pi \models \mathbf{F}'_{\mathrm{N}}[c]\phi$ *iff*

$$\forall \mathcal{V} \subset \mathbb{N}, |\, \mathcal{V} \,| \leq c, \exists i \in \mathbb{N} - \mathcal{V}: \\ \pi(i) \models \phi$$

- $\pi \models \mathbf{G}'_{\mathrm{N}}[c]\phi$ *iff*

$$\exists \mathcal{V} \subset \mathbb{N}, |\, \mathcal{V} \,| \leq c, \forall i \in \mathbb{N} - \mathcal{V}: \\ \pi(i) \models \phi$$

Items above the dashed line are identical to those of traditional LTL, while those

below the line are semantics for the new noisy operators. We write $[\![\phi]\!]$ *for the set* $\{\pi : \pi \models \phi\}$ *and say that* ϕ_1 *and* ϕ_2 *are logically equivalent, with notation* $\phi_1 \equiv \phi_2$, *if* $[\![\phi_1]\!] = [\![\phi_2]\!]$.

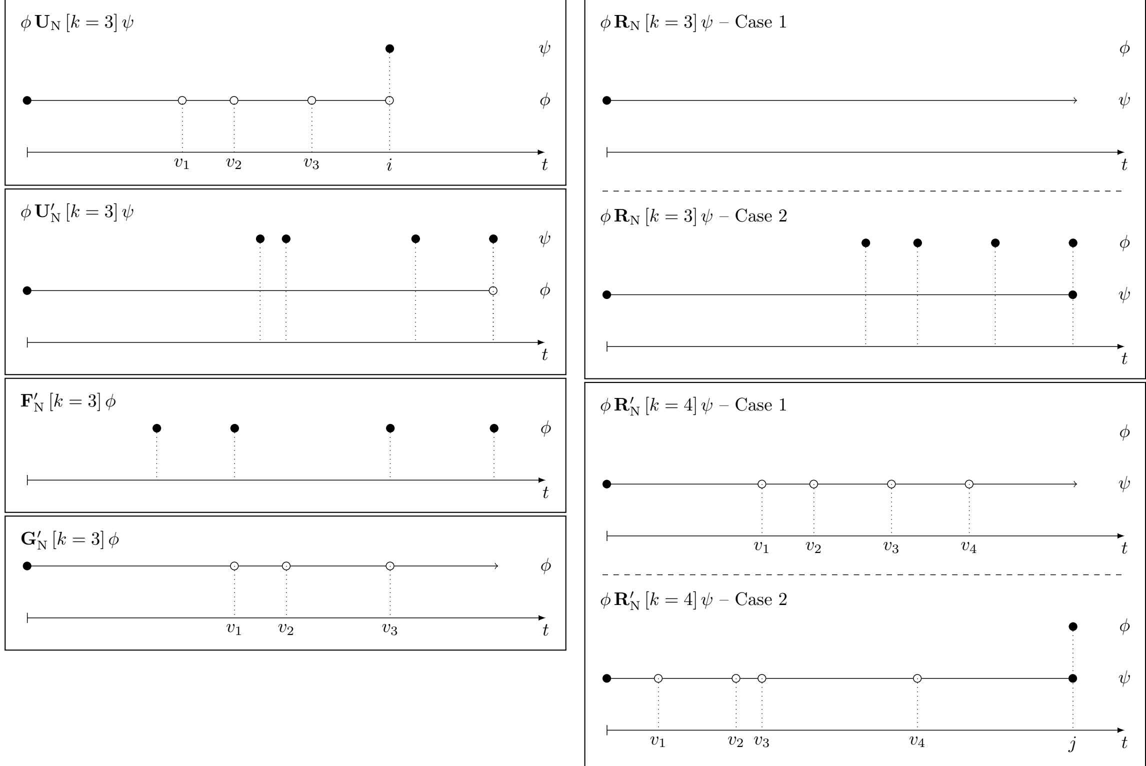


Figure 8.2: Sketches of Noisy LTL Operators

Graphical sketches of the noisy operators are provided in Figure 8.2. Intuitively, the larger the noise parameter, the more deviation there is. The crux of the noisy semantics is the $\mathcal{V}$ entity, which is a subset of the natural numbers, marking indices of interest for one of the two subformulas based on the operator at which deviations can occur. For lax operators (i.e. $\mathbf{U}_\mathrm{N}$, $\mathbf{R}'_\mathrm{N}$, $\mathbf{G}'_\mathrm{N}$), $\mathcal{V}$ is existentially quantified and specifies a set of indices (of size at most c) where the requirement for the noisy

subformula is *not enforced*. For operators expressing redundant noise ($\mathbf{R}_{\mathrm{N}}$, $\mathbf{U}'_{\mathrm{N}}$, $\mathbf{F}'_{\mathrm{N}}$), $\mathcal{V}$ is universally quantified and serves to provide a *minimum coverage* requirement such that the noisy subformula must occur at least $c+1$ times to avoid being potentially eclipsed by indices in $\mathcal{V}$.

The duality of the modal operators serves to tether lax and redundant noise. The base semantics establish $\mathbf{U}_{\mathrm{N}}$ to be lax, and as a consequence the dual operator $\mathbf{R}_{\mathrm{N}}$ expresses redundant noise. The same is true starting from $\mathbf{R}'_{\mathrm{N}}$ and dualizing to $\mathbf{U}'_{\mathrm{N}}$. Figure 8.3 presents a breakdown of the noisy operators based on which subformula is noisy and what type of noise is captured. In principle we could also consider another class of noisiness where we add redundant deviation to a ranges and lax deviation to points. This would allow us to effectively swap $\mathbf{U} \leftrightarrow \mathbf{R}$ as presented in Figure 8.3 (i.e. we would have a left-handed redundantly noisy Until operator), but the applications for this seems minimal.

	Noisy subformula	
	ϕ	ψ
lax	$\phi\,\mathbf{U}_{\mathrm{N}}\,\psi$	$\phi\,\mathbf{R}'_{\mathrm{N}}\,\psi$
redundant	$\phi\,\mathbf{R}_{\mathrm{N}}\,\psi$	$\phi\,\mathbf{U}'_{\mathrm{N}}\,\psi$

lax	redundant
$\mathbf{G}'_{\mathrm{N}}\,\phi$	$\mathbf{F}'_{\mathrm{N}}\,\phi$

Figure 8.3: Noisy LTL Operator Characteristics.

8.3.4 Example Formulas

We provide a several examples of noisy LTL formulas with interpretation.

- `MSFT_up` $\mathbf{U}_{\mathrm{N}}[c=2]$`GOOG_up` - Stock in Microsoft (usually) goes up until stock

in Google goes up.

- $\texttt{new_job}\,\mathbf{R}_{\mathrm{N}}[c = 5]\neg\texttt{open_new_server}$ - More than 5 jobs must be added to the job queue before another server is created to handle requests.
- $\mathbf{G}(\texttt{at_bat}\,\mathbf{U}'_{\mathrm{N}}[c = 2]\texttt{strike})$ - 3 strikes and you're out.
- $\texttt{win}\,\mathbf{R}'_{\mathrm{N}}[c = 3]\texttt{alive}$ - The player has 3 lives to beat the game.
- $\mathbf{F}'_{\mathrm{N}}[c = 3]\texttt{failsafe_button}$ - The button must be pressed, but it is faulty and may misfire up to k times. Pressing $k+1$ times guarantees it will be observed.
- $\mathbf{G}'_{\mathrm{N}}[c = 2]\texttt{attends_meetings}$ - He only missed at most 2 meetings all year.

8.3.5 Properties of Noisy LTL

We have constructed noisy LTL to be an extension of traditional LTL, with the same base syntax and semantics for the operators in common. Because the noisy operators are intended to express deviation from the corresponding base operators, semantically they are related.

Lemma 21 (Grounding out noise)**.** *For $c = 0$ and arbitrary noisy LTL formulas ϕ and ψ, noisy temporal logic reduces to standard LTL:*

- $\phi\,\mathbf{U}_{\mathrm{N}}[c = 0]\psi \equiv \phi\,\mathbf{U}\,\psi$.
- $\phi\,\mathbf{U}'_{\mathrm{N}}[c = 0]\psi \equiv \phi\,\mathbf{U}\,\psi$.
- $\phi\,\mathbf{R}_{\mathrm{N}}[c = 0]\psi \equiv \phi\,\mathbf{R}\,\psi$.
- $\phi\,\mathbf{R}'_{\mathrm{N}}[c = 0]\psi \equiv \phi\,\mathbf{R}\,\psi$.

As well as for derived operators:

- $\mathbf{F}'_{\mathrm{N}}[c=0]\phi \equiv \mathbf{F}\,\phi$.
- $\mathbf{G}'_{\mathrm{N}}[c=0]\phi \equiv \mathbf{G}\,\phi$.

Proof. Immediate upon comparison of the semantics of the noisy temporal operator with $c = 0$ in each case to their non-noisy correspondent. □

We also have some straight-forward simplifications when the noise parameter c is taken to be a negative integer.

Lemma 22 (Negative values of c)**.** *Let i be an integer less than 0 and ϕ, ψ be arbitrary noisy LTL formulas. We have the following:*

- $\phi\,\mathbf{U}_{\mathrm{N}}[c=i]\psi = \boldsymbol{ff}$
- $\phi\,\mathbf{U}'_{\mathrm{N}}[c=i]\psi = \boldsymbol{tt}$
- $\phi\,\mathbf{R}_{\mathrm{N}}[c=i]\psi = \boldsymbol{tt}$
- $\phi\,\mathbf{R}'_{\mathrm{N}}[c=i]\psi = \boldsymbol{ff}$

As well as for derived operators:

- $\mathbf{F}'_{\mathrm{N}}[c=i]\phi = \boldsymbol{tt}$
- $\mathbf{G}'_{\mathrm{N}}[c=i]\phi = \boldsymbol{ff}$

Proof. Observe for $\{\mathbf{U}_{\mathrm{N}}, \mathbf{R}'_{\mathrm{N}}, \mathbf{G}'_{\mathrm{N}}\}$ we are existentially quantifying $\mathcal{V}$ with size at most i (which cannot exist for negative i). For $\{\mathbf{R}_{\mathrm{N}}, \mathbf{U}'_{\mathrm{N}}\,\mathbf{F}'_{\mathrm{N}}\}$, we are universally quantifying $\mathcal{V}$ over sets of size at most i (vacuously satisfying the semantics). □

We can also relate two noisy LTL formulas with similar structure, differing only in the noise parameter.

Lemma 23 (Subsumption of Noisy Operators)**.** *For noisy LTL formulas ϕ, ψ and integers $u \leq v$, we have the following tautologies:*

- $\phi\,\mathbf{U}_{\mathrm{N}}[c=u]\psi \rightarrow \phi\,\mathbf{U}_{\mathrm{N}}[c=v]\psi$
- $\phi\,\mathbf{U}'_{\mathrm{N}}[c=u]\psi \leftarrow \phi\,\mathbf{U}'_{\mathrm{N}}[c=v]\psi$
- $\phi\,\mathbf{R}_{\mathrm{N}}[c=u]\psi \leftarrow \phi\,\mathbf{R}_{\mathrm{N}}[c=v]\psi$
- $\phi\,\mathbf{R}'_{\mathrm{N}}[c=u]\psi \rightarrow \phi\,\mathbf{R}'_{\mathrm{N}}[c=v]\psi$

And by extension, the following tautologies as well.

- $\mathbf{F}'_{\mathrm{N}}[c=u]\phi \leftarrow \mathbf{F}'_{\mathrm{N}}[c=v]\phi$
- $\mathbf{G}'_{\mathrm{N}}[c=u]\phi \rightarrow \mathbf{G}'_{\mathrm{N}}[c=v]\phi$

Proof. Something "less lax" can satisfy a "more lax" requirement. Conversely, something "more redundant" can satisfy a "less redundant" requirement. □

Lemma 24 (Subsumption of $\mathbf{U}$ and $\mathbf{R}$)**.** *For noisy LTL formulas ϕ, ψ and integer $i \in \mathbb{N}$, the following are tautologies for all $j \in \mathbb{N}$:*

- $\phi\,\mathbf{U}'_{\mathrm{N}}[c=i]\psi \rightarrow \phi\,\mathbf{U}_{\mathrm{N}}[c=j]\psi$
- $\phi\,\mathbf{R}_{\mathrm{N}}[c=i]\psi \rightarrow \phi\,\mathbf{R}'_{\mathrm{N}}[c=j]\psi$

Proof. Combining Lemmas 21 and 23, we have the following two tautologies:

$$\phi\,\mathbf{U}'_{\mathrm{N}}[c=i]\psi \rightarrow \phi\,\mathbf{U}\,\psi \qquad \text{and} \qquad \phi\,\mathbf{U}\,\psi \rightarrow \phi\,\mathbf{U}_{\mathrm{N}}[c=j]\psi$$

Together, this yields the lemma's claim. A similar argument holds for the $\mathbf{R}$-related subsumption. □

This result was surprising to us at first, as it allows us to relate left- and right-handed noise. However, given that for both $\mathbf{U}$ and $\mathbf{R}$ exactly one of the noisy variants is lax ($\mathbf{U}_{\mathrm{N}}, \mathbf{R}'_{\mathrm{N}}$) while the other is redundant ($\mathbf{R}_{\mathrm{N}}, \mathbf{U}'_{\mathrm{N}}$), we can see how they can be

connected through the “neutral” non-noisy operator. This leads us to a corollary which will be useful later for showing decidability.

Corollary 3 (Uniqueness of $\phi \mathbf{U} \psi$ and $\phi \mathbf{R} \psi$)**.** *For a conjunction of noisy LTL clauses $C = \{c_i\}$, there exists a refined conjunction C' that is semantically equivalent and has at most one occurrence of any of $\{\mathbf{U}, \mathbf{U}_{\mathrm{N}}, \mathbf{U}'_{\mathrm{N}}\}$ for each pair of subformulas ϕ, ψ. The same is true for $\{\mathbf{R}, \mathbf{R}_{\mathrm{N}}, \mathbf{R}'_{\mathrm{N}}\}$. All of the above is also true for disjunctions.*

Proof. Consider the sets

$$\text{All-Until}(\phi, \psi) := \{\phi\, \mathcal{U}\, \psi : \mathcal{U} \in \{\mathbf{U}, \mathbf{U}_{\mathrm{N}}[i], \mathbf{U}'_{\mathrm{N}}[i]\}, i \in \mathbb{N}\}$$

and

$$\text{All-Release}(\phi, \psi) := \{\phi\, \mathcal{R}\, \psi : \mathcal{R} \in \{\mathbf{R}, \mathbf{R}_{\mathrm{N}}[i], \mathbf{R}'_{\mathrm{N}}[i]\}, i \in \mathbb{N}\}$$

From Lemmas 21, 23, and 24, we have shown a total ordering for each of these sets for arbitrary ϕ, ψ by formula strength (ϕ is stronger than ψ if $\phi \rightarrow \psi$) for each of these sets. Note that if ϕ is stronger than ψ, then $\phi \wedge \psi = \phi$, while $\phi \vee \psi = \psi$. Thus, if we ever encounter two formulas from the same class of All-Until(ϕ, ψ) or All-Release(ϕ, ψ) as part of a conjunction or disjunction, one of the formulas can be removed without affecting the semantics. Applying this iteratively gives us the refined set of conjuncts (or disjuncts). □

The above reuslts can also be extended to the classes for Eventually $\{\mathbf{F}, \mathbf{F}'_{\mathrm{N}}\}$ and Always $\mathbf{G}, \mathbf{G}'_{\mathrm{N}}\}$, as they are syntactically derived from Until and Release (respectively).

8.3.6 Relating $\mathbf{F}'_{\mathrm{N}}, \mathbf{G}'_{\mathrm{N}}$ to Traditional LTL

Noisy LTL has been defined as a syntactic and semantic extension of traditional LTL. We further have determined that a fragment of the noisy LTL language is a purely syntactic extension of traditional LTL, specifically the fragment containing traditional LTL + $\{\mathbf{F}'_{\mathrm{N}}, \mathbf{G}'_{\mathrm{N}}\}$. Consider noisy LTL formula $\phi = \mathbf{F}'_{\mathrm{N}}[c]\psi$ with non-negative c and traditional LTL formula ψ. Then a sequence π satisfies ϕ under noisy LTL semantics if and only if ψ occurs at least $c+1$ distinct times in π. One could express the same property in traditional LTL as:

$$\phi = \underbrace{\mathbf{F}(\psi \wedge \mathbf{X}\,\mathbf{F}(\psi \wedge \mathbf{X}\,\mathbf{F}(\psi \wedge \ldots \mathbf{X}\,\mathbf{F}\,\psi)\ldots))}_{c+1\ \psi\text{'s}}$$

From duality, we can similarly express $\mathbf{G}'_{\mathrm{N}}$ using traditional LTL. $\phi = \mathbf{G}'_{\mathrm{N}}[c]\psi$ can be rewritten as:

$$\phi = \underbrace{\mathbf{G}(\psi \vee \mathbf{X}\,\mathbf{G}(\psi \vee \mathbf{X}\,\mathbf{G}(\psi \vee \ldots \mathbf{X}\,\mathbf{G}\,\psi)\ldots))}_{c+1\ \psi\text{'s}}$$

We can relax the restriction on ψ to be a formula from the fragment LTL + $\{\mathbf{F}'_{\mathrm{N}}, \mathbf{G}'_{\mathrm{N}}\}$ by observing that we can apply the above syntactic rewrites recursively as needed on

the subformulas of ψ. The related LTL pattern $\phi = \mathbf{F}(p_1 \wedge \mathbf{F}(p_2 \wedge \mathbf{F}(p_3 \wedge \ldots \mathbf{F}\, p_c) \ldots))$ for $p_i \in \mathcal{AP}$ was studied in [94] as an example of a formula template whose minimal Büchi automaton is known.

8.4 Decidability of Noisy LTL

We show that, given a formula ϕ in noisy LTL, we can construct a Büchi automaton B_ϕ such that the language $L(B_\phi)$ accepted by B_ϕ contains exactly the set of ω-regular sequences that satisfy ϕ, i.e. $L(B_\phi) = [\![\phi]\!]$, as is done for traditional LTL. We utilize a tableau style construction [86] for this problem, adapted for noisy LTL. Constructing such an automaton allows us to prove decidability of the language emptiness problem for noisy LTL.

8.4.1 Unrolled semantics of Noisy LTL

A core requirement of utilizing a tableau construction is to have a set of recursive semantics for each modal operator (excluding $\mathbf{X}$) which recursively *unrolls* the formula by one time step based on semantics. The resulting unrolling provide several useful properties, including that each non-$\mathbf{X}$ modal operator is *guarded* by a $\mathbf{X}$ in the parse tree of the formula. A table containing the recursive semantics for LTL is shown in Figure 8.4. Our aim is to produce another such table containing all unrolled semantics for noisy LTL operators. We will first show the unrolling for $\mathbf{U}_{\mathrm{N}}[c]$.

Operator	Unrolled Semantics
$\phi\,\mathbf{U}\,\psi$	$\psi \vee (\phi \wedge \mathbf{X}(\phi\,\mathbf{U}\,\psi))$
$\phi\,\mathbf{R}\,\psi$	$\psi \wedge (\phi \vee \mathbf{X}(\phi\,\mathbf{R}\,\psi))$
$\mathbf{F}\,\phi$	$\phi \vee \mathbf{X}\,\mathbf{F}\,\phi$
$\mathbf{G}\,\phi$	$\phi \wedge \mathbf{X}\,\mathbf{G}\,\phi$

Figure 8.4: LTL Unrolled Semantics

Lemma 25 (Unrolled semantics for $\phi\,\mathbf{U}_{\mathrm{N}}\,[c]\psi$)**.** *For non-negative integer $c \geq 0$ and noisy LTL formulas ϕ and ψ, the formula $\phi\,\mathbf{U}_{\mathrm{N}}[c]\psi$ has the following unrolled semantics:*

$$\phi\,\mathbf{U}_{\mathrm{N}}[c]\psi \iff \psi \vee \Big(\phi \wedge \mathbf{X}(\phi\,\mathbf{U}_{\mathrm{N}}[c]\psi)\Big) \vee \mathbf{X}(\phi\,\mathbf{U}_{\mathrm{N}}[c-1]\psi)$$

Proof. First we handle the special case where $c = 0$. Consider the proposed unrolled semantics:

$$\phi\,\mathbf{U}_{\mathrm{N}}[c=0]\psi \iff \psi \vee \Big(\phi \wedge \mathbf{X}(\phi\,\mathbf{U}_{\mathrm{N}}[c=0]\psi)\Big) \vee [\mathbf{X}(\phi\,\mathbf{U}_{\mathrm{N}}[c=-1]\psi)]$$

From Lemma 21 we have that $\phi\,\mathbf{U}_{\mathrm{N}}[c=0]\psi = \phi\,\mathbf{U}\,\psi$, and from Lemma 22 we have that $\phi\,\mathbf{U}_{\mathrm{N}}[c=-1]\psi = \mathtt{ff}$, so the unrolling reduces to the known unrolled semantics for $\mathbf{U}$:

$$\phi\,\mathbf{U}\,\psi \iff \psi \vee \Big(\phi \wedge \mathbf{X}(\phi\,\mathbf{U}\,\psi)\Big)$$

For the remaining case where $c \geq 1$, we prove implication in both directions.

($\leftarrow$) Assume $\pi \models \psi \vee (\phi \wedge \mathbf{X}(\phi\,\mathbf{U}_{\mathrm{N}}[c]\psi)) \vee \mathbf{X}(\phi\,\mathbf{U}_{\mathrm{N}}[c-1]\psi)$. Then, one of the three disjuncts must be true. We will perform a case analysis for each, showing that the semantics for $\pi \models \phi\,\mathbf{U}_{\mathrm{N}}[c]\psi$ is satisfied:

- $\pi \models \psi$: Then, let $i = 0$ and $\mathcal{V} \subseteq \mathbb{N}$ be arbitrary. Clearly, we have that $\pi(i) \models \psi \wedge \forall j \in \mathbb{N} - \mathcal{V} : (0 \leq j < i) \implies \pi(j) \models \phi$, which is equivalent to $\pi \models \phi\,\mathbf{U}_{\mathrm{N}}[c]\psi$.
- $\pi \models \phi \wedge \mathbf{X}(\phi\,\mathbf{U}_{\mathrm{N}}[c]\psi)$: Then, $\pi \models \phi$ and $\pi(1) \models \phi\,\mathbf{U}_{\mathrm{N}}[c]\psi$. Expanding $\pi(1) \models \phi\,\mathbf{U}_{\mathrm{N}}[c]\psi$ we have

  $$\exists i \geq 1, \mathcal{V} \subseteq \mathbb{N}, |\,\mathcal{V}\,| \leq c : \pi(i) \models \psi \wedge \forall j \in \mathbb{N} - \mathcal{V} : (1 \leq j < i) \implies \pi(j) \models \phi$$

  so combined with $\pi \models \phi$ we have

  $$\exists i \geq 1, \mathcal{V} \subseteq \mathbb{N}, |\,\mathcal{V}\,| \leq c : \pi(i) \models \psi \wedge \forall j \in \mathbb{N} - \mathcal{V} : (0 \leq j < i) \implies \pi(j) \models \phi$$

  which is a strictly stronger statement than the same statement where is bounded $i \geq 0$, which are the semantics of $\phi\,\mathbf{U}_{\mathrm{N}}[c]\psi$, so $\pi \models \phi\,\mathbf{U}_{\mathrm{N}}[c]\psi$.
- $\pi \models \mathbf{X}(\phi\,\mathbf{U}_{\mathrm{N}}[c-1]\psi)$: Then, $\pi(1) \models \phi\,\mathbf{U}_{\mathrm{N}}[c-1]\psi$, equivalently

  $$\exists i \geq 1, \mathcal{V} \subseteq \mathbb{N}, |\,\mathcal{V}\,| \leq c-1 : \pi(i) \models \psi \wedge \forall j \in \mathbb{N} - \mathcal{V} : (1 \leq j < i) \implies \pi(j) \models \phi$$

  Let $i', \mathcal{V}'$ be an instance set satisfying these semantics. Let $\mathcal{V}'' = \mathcal{V}' \cup \{0\}$

be a new violation set formed by adding the index 0 to $\mathcal{V}'$. Note that $|\,\mathcal{V}''\,| \leq c$. So, we have that for the choice of $i', \mathcal{V}''$:

$$\pi(i') \models \psi \wedge \forall j \in \mathbb{N} - \mathcal{V}'' : (0 \leq j < i') \implies \pi(j) \models \phi$$

Note that we can expand the range for j from $(1 \leq j < i')$ to $(0 \leq j < i')$ because $0 \in \mathcal{V}''$. Thus, we have shown the existence of a choice for $i, \mathcal{V}$ to enforce $\pi \models \phi\, \mathbf{U}_{\mathrm{N}}[c]\psi$.

We have handled all 3 cases and thus have shown that

$$\pi \models \psi \vee [\phi \wedge \mathbf{X}(\phi\, \mathbf{U}_{\mathrm{N}}[c]\psi)] \vee \mathbf{X}(\phi\, \mathbf{U}_{\mathrm{N}}[c-1]\psi) \implies \pi \models \phi\, \mathbf{U}_{\mathrm{N}}[c]\psi$$

$(\rightarrow)$ Now assume $\pi \models \phi\, \mathbf{U}_{\mathrm{N}}[c]\psi$. To prove

$$\pi \models \phi\, \mathbf{U}_{\mathrm{N}}[c]\psi \implies \pi \models \psi \vee [\phi \wedge \mathbf{X}(\phi\, \mathbf{U}_{\mathrm{N}}[c]\psi)] \vee \mathbf{X}(\phi\, \mathbf{U}_{\mathrm{N}}[c-1]\psi)$$

we will show that one of the three disjuncts must be true. Because $\pi \models \phi\, \mathbf{U}_{\mathrm{N}}[c]\psi$, we have that

$$\exists i \geq 0, \mathcal{V} \subseteq \mathbb{N}, |\,\mathcal{V}\,| \leq c : \pi(i) \models \psi \wedge \forall j \in \mathbb{N} - \mathcal{V} : (0 \leq j < i) \implies \pi(j) \models \phi$$

Let $i', \mathcal{V}'$ be an instance set satisfying these semantics. We will perform a case

analysis on $i', \mathcal{V}'$, and π, proving one of the disjuncts to be true in each case. First assume $i' = 0$. If this is the case, then $\pi(0) \models \psi \equiv \pi \models \psi$ (the first disjunct) so we are done. Now assume $i' \geq 1$. For an arbitrary π, either $\pi \models \phi$ or $\pi \not\models \phi$. We consider each case:

- $\pi \models \phi$:
    * If $0 \in \mathcal{V}'$, let $\mathcal{V}'' = \mathcal{V}' - \{0\}$. Note that $|\,\mathcal{V}\,|$ can still be upper bounded by c. Then, $\pi \models \phi$ and $\pi(i') \models \psi \wedge \forall j \in \mathbb{N} - \mathcal{V}'' : (0 \leq j < i') \implies \pi(j) \models \phi$, where the second statement can be relaxed to

      $$\pi(i') \models \psi \wedge \forall j \in \mathbb{N} - \mathcal{V}'' : (1 \leq j < i') \implies \pi(j) \models \phi$$

      Existentially quantifying i' to i and $\mathcal{V}''$ to $\mathcal{V}$ gives us

      $$\exists i \geq 1, \mathcal{V} \subseteq \mathbb{N}, |\,\mathcal{V}\,| \leq c : \pi(i) \models \psi \wedge \forall j \in \mathbb{N} - \mathcal{V} : (1 \leq j < i) \implies \pi(j) \models \phi$$

      which are the semantics for $\pi(1) \models \phi\,\mathbf{U}_{\mathrm{N}}[c]\psi$, or equivalently, $\pi \models \mathbf{X}(\phi\,\mathbf{U}_{\mathrm{N}}[c]\psi)$. Thus $\pi \models \phi \wedge \mathbf{X}(\phi\,\mathbf{U}_{\mathrm{N}}[c]\psi)$, and so we have proven the second disjunct to be true.
    * If $0 \notin \mathcal{V}'$, then let $\mathcal{V}'' = \mathcal{V}'$, whose cardinality is still bounded above by c. We can apply the same argument for the above case using this choice of $\mathcal{V}''$ to again show that the second disjunct is true.

– $\pi \not\models \phi$: Then we must have $0 \in \mathcal{V}'$ in order to satisfy the semantics of $a\,\mathbf{U}_{\mathrm{N}}[c]\psi$. Let $\mathcal{V}'' = \mathcal{V}' - \{0\}$. Note that $|\,\mathcal{V}''\,| = |\,\mathcal{V}'\,| - 1$, so $|\,\mathcal{V}''\,| \leq c-1$. We have

$$\pi(i') \models \psi \wedge \forall j \in \mathbb{N} - \mathcal{V}'' : (1 \leq j < i') \implies \pi \models \phi$$

which can be existentially quantified over i' and $\mathcal{V}''$ to obtain

For all cases we have shown that at least one of the three disjuncts holds, and thus

$$\pi \models \phi\,\mathbf{U}_{\mathrm{N}}[c]\psi \implies \pi \models \psi \vee [\phi \wedge \mathbf{X}(\phi\,\mathbf{U}_{\mathrm{N}}[c]\psi)] \vee \mathbf{X}(\phi\,\mathbf{U}_{\mathrm{N}}[c-1]\psi)$$

This proves the overall unrolling. □

We can see that these semantics are very similar to the semantics of $\phi\,\mathbf{U}\,\psi$: simply a third disjunct was added. Intuitively this third disjunct captures the noise of the formula. We can dualize the unrolling for $\phi\,\mathbf{U}_{\mathrm{N}}[c]\psi$ to obtain an unrolling for $\phi\,\mathbf{R}_{\mathrm{N}}[c]\psi$:

$$\phi\,\mathbf{R}_{\mathrm{N}}[c]\psi \iff \psi \wedge \Big(\phi \vee \mathbf{X}(\phi\,\mathbf{R}_{\mathrm{N}}[c]\psi)\Big) \wedge \mathbf{X}(\phi\,\mathbf{R}_{\mathrm{N}}[c-1]\psi)$$

The semantic unrollings for right-handed noise will require a few more nota-

tional conveniences.

Definition 35. *Let* $\xi[c] := \mathbf{X}(\mathit{tt}\,\mathbf{R}'_{\mathrm{N}}[c-1]\,\mathit{tt})$ *be a noisy LTL formula template such that for a choice of* c*,* $\xi[c]$ *is a noisy LTL formula.*

Lemma 26 (Encoding $c \geq 1$ in noisy LTL)**.** *For* $c \in \mathbb{N}$*,* $[\![\xi[c]]\!] = [\![\mathit{tt}]\!]$ *if and only if* $c \geq 1$*, while* $[\![\xi[c]]\!] = [\![\mathit{ff}]\!]$ *iff* $c = 0$*.*

Proof. Lemma 22 gives us that if $c = 0$, then $[\![\xi[c]]\!] = [\![\mathtt{ff}]\!]$. For $c \geq 1$, we have the semantics of $\xi[c]$ as:

$$\begin{aligned} \pi \models \xi[c] \iff \exists \mathcal{V} \subset \mathbb{N}, |V| \leq c, \forall i \in \mathbb{N} - \mathcal{V} : \\ \pi(i) \models \mathtt{tt} \vee \exists j \in \mathbb{N} : \\ (0 \leq j < i) \wedge \pi(j) \models \mathtt{tt} \end{aligned}$$

which is logically equivalent to $\mathtt{tt}$. □

Definition 36 (Shifting Violation Set indices)**.** *For an arbitrary violation set* $\mathcal{V} \subseteq \mathbb{N}$ *and a* $\delta \in \mathbb{N}$*, let* $\mathrm{shift}(\mathcal{V}, \delta) = \{i + \delta : i \in \mathcal{V}\}$ *be the set formed by shifting all indices in* $\mathcal{V}$ *to the right by* δ *time points.*

Note that $\mathrm{shift}(\mathcal{V}, \delta)$ is itself a violation set of the same cardinality as $\mathcal{V}$, i.e. violation sets are closed under the shift operator (requiring non-negative δ).

Lemma 27 (Unrolled semantics for $\phi\,\mathbf{R}'_{\mathrm{N}}[c]\psi$)**.** *For non-negative integer* $c \geq 0$ *and noisy LTL formulas* ϕ *and* ψ*, the formula* $\phi\,\mathbf{R}'_{\mathrm{N}}[c]\psi$ *has the following unrolled*

semantics:

$$\phi\,\mathbf{R}'_{\mathrm{N}}[c]\psi \iff \Big(\phi\wedge\psi\Big)\vee\Big(\phi\wedge\mathbf{X}(\xi[c])\Big)\vee\Big(\psi\wedge\mathbf{X}(\phi\,\mathbf{R}'_{\mathrm{N}}[c]\psi)\Big)\vee\mathbf{X}(\phi\,\mathbf{R}'_{\mathrm{N}}[c-1]\psi)$$

Proof. Recall the semantics for $\phi\,\mathbf{R}'_{\mathrm{N}}[c]\psi$:

$$\pi \models \phi\,\mathbf{R}'_{\mathrm{N}}[c]\psi \iff \exists\,\mathcal{V}\subset\mathbb{N}, |V|\leq c, \forall i\in\mathbb{N}-\mathcal{V}:$$

$$\pi(i)\models\psi\vee\exists j\in\mathbb{N}:$$

$$(0\leq j<i)\wedge\pi(j)\models\phi$$

We will show equivalence by proving implication in both directions. First, we will show

$$\pi \models \phi\,\mathbf{R}'_{\mathrm{N}}[c]\psi \leftarrow \pi\models\Big(\phi\wedge\psi\Big)\vee\Big(\phi\wedge\mathbf{X}(\xi[c])\Big)\vee\Big(\psi\wedge\mathbf{X}(\phi\,\mathbf{R}'_{\mathrm{N}}[c]\psi)\Big)\vee\mathbf{X}(\phi\,\mathbf{R}'_{\mathrm{N}}[c-1]\psi)$$

Assume the premise; one of the four disjuncts must be true. We will perform a case analysis on each disjunct:

- $\phi\wedge\psi$: Then, $\pi\models\phi$ and $\pi\models\psi$, so $\pi(0)\models\phi$ and $\pi(0)\models\psi$. Let $\mathcal{V}'=\emptyset$. Then, we have

$$\forall i\in\mathbb{N}-\mathcal{V}':\pi(i)\models\psi\vee\exists j\in\mathbb{N}:(0\leq j<i)\wedge\pi(j)\models\phi$$

Note here that when $i \geq 1$, $j = 0$ is a solution to the existence claim. Observe that $|\,\mathcal{V}'\,| = 0 \leq c$ for any $c \in \mathbb{N}$. Existentially quantifying $\mathcal{V}'$, we have

$$\exists\,\mathcal{V} \subseteq \mathbb{N}, |\,\mathcal{V}\,| \leq c, \forall i \in \mathbb{N} - \mathcal{V} : \pi(i) \models \psi \vee \exists j \in \mathbb{N} : (0 \leq j < i) \wedge \pi(j) \models \phi$$

which exactly states $\pi \models \phi\,\mathbf{R}'_{\mathrm{N}}[c]\psi$.

- $\phi \wedge \mathbf{X}(\xi[c])$: Then, $\pi \models \phi$ and $\pi \models \mathbf{X}(\xi[c])$. From Lemma 26 we have $c \geq 1$. Let $\mathcal{V}' = \{0\}$. Note that $|\,\mathcal{V}'\,| = 1 \leq c$. Then, the following is true:

$$\forall i \in \mathbb{N} - \mathcal{V}' : \exists j \in \mathbb{N} : (0 \leq j < i) \wedge \pi(j) \models \phi$$

We can add a disjunct to the existential term with no impact to logical validity:

$$\forall i \in \mathbb{N} - \mathcal{V}' : \pi(i) \models \psi \vee \exists j \in \mathbb{N} : (0 \leq j < i) \wedge \pi(j) \models \phi$$

And finally we can existentially quantify $\mathcal{V}'$:

$$\exists\,\mathcal{V} \subseteq \mathbb{N}, |\,\mathcal{V}\,| \leq c, \forall i \in \mathbb{N} - \mathcal{V} : \pi(i) \models \psi \vee \exists j \in \mathbb{N} : (0 \leq j < i) \wedge \pi(j) \models \phi$$

which again exactly states $\pi \models \phi\,\mathbf{R}'_{\mathrm{N}}[c]\psi$.

- $\psi \wedge \mathbf{X}(\phi\,\mathbf{R}'_{\mathrm{N}}[c]\psi)$: Then, $\pi(0) \models \psi$ and $\pi \models \mathbf{X}(\phi\,\mathbf{R}'_{\mathrm{N}}[c]\psi)$, which means that

$\pi(1) \models \phi\,\mathbf{R}'_{\mathrm{N}}[c]\psi$ and so we have

$$\exists\,\mathcal{V} \subseteq \mathbb{N}, |\,\mathcal{V}\,| \leq c, \forall i \in \mathbb{N} - \mathcal{V} : \pi(1)(i) \models \psi \vee \exists j \in \mathbb{N} : (0 \leq j < i) \wedge \pi(1)(j) \models \phi$$

and equivalently

$$\exists\,\mathcal{V} \subseteq \mathbb{N}, |\,\mathcal{V}\,| \leq c, \forall i \in \mathbb{N} - \mathcal{V} : \pi(i+1) \models \psi \vee \exists j \in \mathbb{N} : (0 \leq j < i) \wedge \pi(j+1) \models \phi$$

Let $\mathcal{V}'$ be a violation set that satisfies this property. Let $\mathcal{V}'' = \mathrm{shift}(\mathcal{V}', 1)$, with $|\,\mathcal{V}''\,| \leq c$. $\mathcal{V}''$ corresponds to the indices in $\mathcal{V}'$ over $\pi(0) = \pi$, not $\pi(1)$. The following holds for $\mathcal{V}''$:

$$\forall i \in \mathbb{N} - \mathcal{V}'' : \pi(i) \models \psi \vee \exists j \in \mathbb{N} : (0 \leq j < i) \wedge \pi(j) \models \phi$$

By construction $0 \notin \mathcal{V}''$, but the above holds for $i = 0$ because $\pi(0) \models \psi$. For $i > 0$, we can observe that the above semantics for $\pi(1) \models \phi\,\mathbf{R}'_{\mathrm{N}}[c]\psi$ covers us. Existentially quantifying $\mathcal{V}''$ gives us

$$\exists\,\mathcal{V} \in \mathbb{N}, |\,\mathcal{V}\,| \leq c, \forall i \in \mathbb{N} - \mathcal{V} : \pi(i) \models \psi \vee \exists j \in \mathbb{N} : (0 \leq j < i) \wedge \pi(j) \models \phi$$

which, as above, exactly states $\pi \models \phi\,\mathbf{R}'_{\mathrm{N}}[c]\psi$.

- $\mathbf{X}(\phi\,\mathbf{R}'_{\mathrm{N}}[c-1]\psi)$: Then, $\pi(1) \models \phi\,\mathbf{R}'_{\mathrm{N}}[c-1]\psi$, so we have

  $$\exists\,\mathcal{V} \subseteq \mathbb{N}, |\,\mathcal{V}\,| \leq c{-}1, \forall i \in \mathbb{N}{-}\mathcal{V} : \pi(i{+}1) \models \psi \vee \exists j \in \mathbb{N} : (0 \leq j < i) \wedge \pi(j{+}1) \models \phi$$

  Let $\mathcal{V}'$ be a violation set that satisfies this property. Let $\mathcal{V}'' = \{0\} \cup \mathrm{shift}(\mathcal{V}', 1)$ with $|\,\mathcal{V}''\,| \leq c$. $\mathcal{V}''$ corresponds to the indices in $\mathcal{V}'$ over $\pi(0) = \pi$, not $\pi(1)$ and includes 0 as an index. The following holds for $\mathcal{V}''$:

  $$\forall i \in \mathbb{N} - \mathcal{V}'' : \pi(i) \models \psi \vee \exists j \in \mathbb{N} : (0 \leq j < i) \wedge \pi(j) \models \phi$$

  and existentially quantifying $\mathcal{V}''$ gives us

  $$\exists\,\mathcal{V} \subseteq \mathbb{N}, |\,\mathcal{V}\,| \leq c, \forall i \in \mathbb{N} - \mathcal{V} : \pi(i) \models \psi \vee \exists j \in \mathbb{N} : (0 \leq j < i) \wedge \pi(j) \models \phi$$

  which, as in all three above cases, exactly states $\pi \models \phi\,\mathbf{R}'_{\mathrm{N}}[c]\psi$.

In each of the four cases, we have shown that $\pi \models \phi\,\mathbf{R}'_{\mathrm{N}}[c]\psi$. Now, we will show the converse, i.e.

$$\pi \models \phi\,\mathbf{R}'_{\mathrm{N}}[c]\psi \rightarrow \pi \models [\phi \wedge \psi] \vee [\phi \wedge \mathbf{X}(\xi[c])] \vee [\psi \wedge \mathbf{X}(\phi\,\mathbf{R}'_{\mathrm{N}}[c]\psi)] \vee \mathbf{X}(\phi\,\mathbf{R}'_{\mathrm{N}}[c-1]\psi)$$

Assume the new premise, $\pi \models \phi\,\mathbf{R}'_{\mathrm{N}}[c]\psi$. Let $\mathcal{V}'$ be a violation set that satisfies this property. We will show that at least one of the disjuncts of the consequent must hold based on case analysis of the structure of π and $\mathcal{V}'$. First we consider if $\pi \models \phi$

or not:

- $[\pi \models \phi]$: Now consider if $\pi(0) \models \psi$:
  - $\pi(0) \models \psi$: Then we have both $\pi \models \phi$ and $\pi \models \psi$, thus

$$\pi \models (\phi \wedge \psi)$$

    which is the first disjunct, so we are done with this case.

  - $\pi(0) \not\models \psi$: Then for $i = 0$ the body of the semantics (assumed above to be true) is not satisfied:

$$\pi(0) \models \psi \vee \exists j \in \mathbb{N} : (0 \leq j < 0) \wedge \pi(j) \models \phi \iff \texttt{ff}$$

    which means that $0 \in \mathcal{V}'$ in order to avoid this expression. Because $\mathcal{V}'$ has at least one element, its size is greater than or equal to 1, and thus $\pi \models \xi[c]$. Because we have assumed also that $\pi \models \phi$, we have

$$\pi \models \phi \wedge \mathbf{X}(\xi[c])$$

    which is the second disjunct, so we are done with this case.

- $[\pi \not\models \phi]$: Now consider if $0 \in \mathcal{V}'$:

– $0 \notin \mathcal{V}'$: Then, for $i = 0$, the body of the semantics must be true:

$$\pi(0) \models \psi \vee \exists j \in \mathbb{N} : (0 \leq j < 0) \wedge \pi(j) \models \phi$$

$$\pi(0) \models \psi \vee \texttt{ff} \qquad \text{(No such } j\text{)}$$

$$\pi(0) \models \psi$$

$$\pi \models \psi$$

Let $\mathcal{V}'' = \{i - 1 : i \in \mathcal{V}'\}$ and $\pi' = \pi(1)$. $\mathcal{V}''$ is still a subset of $\mathbb{N}$ because 0 is not a member of $\mathcal{V}'$. Consider the following:

$$\forall i \in \mathbb{N} - \mathcal{V}'' : \pi'(i) \models \psi \vee \exists j \in \mathbb{N} : (0 \leq j < i) \wedge \pi'(j) \models \phi$$

$$\equiv \forall i \in \mathbb{N} - \mathcal{V}'' : \pi(1)(i) \models \psi \vee \exists j \in \mathbb{N} : (0 \leq j < i) \wedge \pi(1)(j) \models \phi$$

$$\equiv \forall i \in \mathbb{N} - \mathcal{V}'' : \pi(i + 1) \models \psi \vee \exists j \in \mathbb{N} : (0 \leq j < i) \wedge \pi(j + 1) \models \phi$$

We want to show that this is true based on our assumption that $\pi \models \phi \, \mathbf{R}'_{\mathrm{N}}[c]\psi$. Because $\mathcal{V}''$ is a 1 time point shift from $\mathcal{V}'$ and $\pi(1)$ is the suffix of π (shifted to the same time point), this is the same expression body of $\phi \, \mathbf{R}'_{\mathrm{N}}[c]\psi$ except that $\pi(0) \models \phi$ is never considered here. However, because we have assumed that $\pi \not\models \phi$, we have $\pi(0) \not\models \phi$ and thus we can safely discount this consideration, and thus we have shown that the

above holds. Existentially quantifying $\mathcal{V}''$ gives us

$$\exists \mathcal{V} \subseteq \mathbb{N}, |\,\mathcal{V}\,| \leq c, \forall i \in \mathbb{N}-\mathcal{V} : \pi(i+1) \models \psi \vee \exists j \in \mathbb{N} : (0 \leq j < i) \wedge \pi(j+1) \models \phi$$

$$\equiv \exists \mathcal{V} \subseteq \mathbb{N}, |\,\mathcal{V}\,| \leq c, \forall i \in \mathbb{N}-\mathcal{V} : \pi(1)(i) \models \psi \vee \exists j \in \mathbb{N} : (0 \leq j < i) \wedge \pi(1)(j) \models \phi$$

$$\equiv \pi(1) \models \phi\, \mathbf{R}'_{\mathrm{N}}[c]\psi \equiv \pi \models \mathbf{X}(\phi\, \mathbf{R}'_{\mathrm{N}}[c]\psi)$$

Thus, we have shown that $\pi \models \psi$ and $\pi \models \mathbf{X}(\phi\, \mathbf{R}'_{\mathrm{N}}[c]\psi)$, so collectively

$$\pi \models \psi \wedge \mathbf{X}(\phi\, \mathbf{R}'_{\mathrm{N}}[c]\psi)$$

which is the third disjunct, so we are done with this case.

– $0 \in \mathcal{V}'$: Let $\mathcal{V}'' = \{i - 1 : i \in \mathcal{V}' - \{0\}\}$ be a new violation set with the removal of 0 as an element and a 1 time step shift. $\mathcal{V}''$ is a subset of $\mathbb{N}$ by construction. Note that $|\,\mathcal{V}''\,| = |\,\mathcal{V}'\,| - 1 \leq c$. Also, let $\pi' = \pi(1)$. We can apply a similar argument as the above case where $0 \notin \mathcal{V}'$ (the only difference is that we use $c - 1$ instead of c) to show that

$$\pi(1) \models \phi\, \mathbf{R}'_{\mathrm{N}}[c-1]\psi \equiv \pi \models \mathbf{X}(\phi\, \mathbf{R}'_{\mathrm{N}}[c-1]\psi)$$

which is the fourth disjunct, so we are done with this case.

We have shown that over all cases, we can arrive at one of the disjuncts of the

consequent, thus we have shown overall that the overall disjunction holds and thus are down with this direciton.

Having shown both directions, we are done. □

We can compute the semantics for $\mathbf{U}'_{\mathrm{N}}[c]$ by dualizing the unrolling of $\mathbf{R}'_{\mathrm{N}}[c]$, and then apply the syntactic definitions of $\mathbf{F}'_{\mathrm{N}}[c]$ and $\mathbf{G}'_{\mathrm{N}}[c]$ to obtain their unrollings. The full table of syntactic unrollings for noisy operators is presented in Figure 8.5.

Operator	Unrolled Semantics
$\phi\,\mathbf{U}_{\mathrm{N}}[c]\psi$	$\psi \vee \Big(\phi \wedge \mathbf{X}(\phi\,\mathbf{U}_{\mathrm{N}}[c]\psi)\Big) \vee \mathbf{X}(\phi\,\mathbf{U}_{\mathrm{N}}[c-1]\psi)$
$\phi\,\mathbf{R}_{\mathrm{N}}[c]\psi$	$\psi \wedge \Big(\phi \vee \mathbf{X}(\phi\,\mathbf{R}_{\mathrm{N}}[c]\psi)\Big) \wedge \mathbf{X}(\phi\,\mathbf{R}_{\mathrm{N}}[c-1]\psi)$
$\phi\,\mathbf{R}'_{\mathrm{N}}[c]\psi$	$\Big(\phi \wedge \psi\Big) \vee \Big(\phi \wedge \mathbf{X}(\mathtt{tt}\,\mathbf{R}'_{\mathrm{N}}[c-1]\,\mathtt{tt})\Big) \vee \Big(\psi \wedge \mathbf{X}(\phi\,\mathbf{R}'_{\mathrm{N}}[c]\psi)\Big) \vee \mathbf{X}(\phi\,\mathbf{R}'_{\mathrm{N}}[c-1]\psi)$
$\phi\,\mathbf{U}'_{\mathrm{N}}[c]\psi$	$\Big(\phi \vee \psi\Big) \wedge \Big(\phi \vee \mathbf{X}(\mathtt{ff}\,\mathbf{U}'_{\mathrm{N}}[c-1]\,\mathtt{ff})\Big) \wedge \Big(\psi \vee \mathbf{X}(\phi\,\mathbf{U}'_{\mathrm{N}}[c]\psi)\Big) \wedge \mathbf{X}(\phi\,\mathbf{U}'_{\mathrm{N}}[c-1]\psi)$
$\mathbf{F}'_{\mathrm{N}}[c]\psi$	$\Big(\psi \vee \mathbf{X}(\mathbf{F}'_{\mathrm{N}}[c]\psi)\Big) \wedge \mathbf{X}(\mathbf{F}'_{\mathrm{N}}[c-1]\psi)$
$\mathbf{G}'_{\mathrm{N}}[c]\psi$	$\Big(\psi \wedge \mathbf{X}(\mathbf{G}'_{\mathrm{N}}[c]\psi)\Big) \vee \mathbf{X}(\mathbf{G}'_{\mathrm{N}}[c-1]\psi)$

Figure 8.5: Semantic unrollings of noisy operators in noisy LTL

8.4.2 Tableau Construction

We refer the reader to literature [86, 89] regarding tableau-style constructions, and present an abridged description here. With a complete set of recursive semantics, we can perform a tableau-style construction of a Büchi automaton B_ϕ from a noisy LTL formula ϕ. This will allow us to indirectly reason about ϕ by performing computation on ϕ's representative automaton, much as is done for LTL. In particular, we show decidability of the language emptiness for a noisy LTL formula ϕ. Let $S(\phi)$ be the set of subformulas of ϕ. We create a Büchi automaton B_ϕ where the set of states of B_ϕ

is $2^{S(\phi)}$; that is to say that each state q in the automaton corresponds to a subset of the subformulas, i.e. $q \in 2^{S(\phi)}$. We mark the state corresponding to $\{\phi\}$ as the initial state and perform a traversal of the graph, creating edges as we visit new states. Upon visiting a state $q = \{s_i\}$, we compute the conjunction of all s_i, i.e. $\psi = \bigwedge s_i$. We then put ψ into a semantically equivalent formula ψ' of a normal form which is a disjunction of conjuncts, and where each disjunct has as conjuncts (negated) literals and exactly one conjunction of subformulas guarded by $\mathbf{X}$. This normal form is then used to construct an edge in B_ϕ for each disjunct, where the source state of the new edge if q, the destination state is the state corresponding to the set of conjoined subformulas that are guarded, and the edge label is the conjunction of the (negated) literals. To achieve this normal form we apply propositional logic manipulation (De Morgan's law, distributivity of $\wedge/\vee$), apply the unrolling semantics of modal operators, and group guarded terms using distributivity of $\mathbf{X}$ with $\wedge$.

As an example, let the visited state be $q = \{a\,\mathbf{U}_{\mathrm{N}}[2]b\}$ (a single subformula). Then $\psi = a\,\mathbf{U}_{\mathrm{N}}[2]b$ and is put in the normal form

$$\psi' = \Big(b \wedge \mathbf{X}(\mathtt{tt})\Big) \vee \Big(a \wedge \mathbf{X}(a\,\mathbf{U}_{\mathrm{N}}[2]b)\Big) \vee \Big(\mathtt{tt} \wedge \mathbf{X}(a\,\mathbf{U}_{\mathrm{N}}[1]b\Big)$$

The result lets us create three edges from q, as shown in Figure 8.6. Note that the labels on the edges are propositional formulas, which can be seen as a compressed representation of individual transitions containing a single subset of the $\mathcal{AP}$. Edge e

can be taken if the next element of the path sequence π satisfies (using propositional semantics) the propositional label of e.

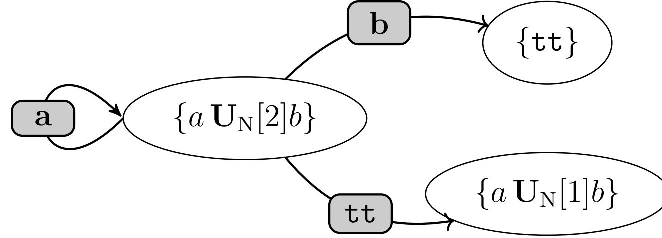


Figure 8.6: Tableau method applied to state $q = \{a\,\mathbf{U}_{\mathrm{N}}[2]b\}$.

Having shown how to construct a corresponding Büchi B_ϕ from a noisy LTL formula ϕ, solving the language emptiness problem for ϕ can be performed by first compute B_ϕ and then determining if B_ϕ has any accepting inputs. This can be done by first computing the set of strongly connected components of B_ϕ, and then determining if the connected component containing the input state can reach any non-trivial connected component containing an accepting state. Construction of the Büchi could in the worse case have us traverse all $2^{S(\phi)}$ states corresponding to subsets of subformulas of ϕ. We note that our notion of subformula here is extended; if we have noisy LTL formula $\phi\mathcal{O}[c = i]\psi$ with $\mathcal{O} \in \{\mathbf{U}_{\mathrm{N}}, \mathbf{R}_{\mathrm{N}}, \mathbf{U}'_{\mathrm{N}}, \mathbf{R}'_{\mathrm{N}}\}$, then all formulas of the set $\{\phi\mathcal{O}[c = j]\psi : 0 \leq j < c\}$ are considered subformulas. This extends to the unary noisy operators $\{\mathbf{F}'_{\mathrm{N}}, \mathbf{G}'_{\mathrm{N}}\}$ from their syntactic construction.

8.4.3 Observations about Tableau Construction

Some observations about the unrolling semantics presented in Section 8.4.1 can also be leveraged to provide a better worse-case analysis. We can achieve some savings

when computing strongly connected components of Büchi automaton B_ϕ. While noramlly an SCC algorithm starts by grouping all states into a single partition, we can avoid this general assumption by considering how transitions are created in B_ϕ. Each unrolling summarized in Figure 8.5 of a noisy operator contains only noise parameters c or $c-1$, and thus the iterative unrolling of subformulas will result in a monotonic decrease of noise parameters until the subformulas ground out to traditional LTL as shown in Lemma 21.

From this, we can reason that B_ϕ will never have a transition created starting from a lower noise formula to a higher noise formula of the same type with matching subformulas. Put another way, we know that tableau-style constructions never generate transitions from state $q = \{s_i\}$ to state $q' = \{t_j\}$ if some $s \in q$ is a proper subformula of some $t \in q'$, and by our extended notion of subformula we can observe a monotonic decrease of noise. All of this is to say that because we cannot re-enter areas of more elevated noise, when we "step down" a level we cannot reenter through some series of transitions and thus have moved to another strongly connected component. Strongly connected components exist only in one "level," and thus we can limit our computation of strongly connected components by ignoring all transitions that bridge levels of noise.

8.4.4 Tableau Examples

We provide a number of example formulas and their corresponding Büchi automaton. We have implemented the tableau construction described above in C++. We made use of the Spot [88] platform (v.2.8.1) to handle the parsing of input formulas as well as the syntactic representation of the constructed Büchi automata. To support the proper parsing and construction of a noisy LTL formula, we added support for all noisy operators introduced in Definition 33.

Example 1: $\phi = a \, \mathbf{U} \, b$

For $\phi = a \, \mathbf{U} \, b$, we have a very simple two-state system. First, a state for ϕ:

$$
\begin{aligned}
\phi = \phi_1 = a \, \mathbf{U} \, b &\equiv b \vee (a \wedge \mathbf{X}(a \, \mathbf{U} \, b)) \\
&\equiv [b \wedge \mathbf{X}(\texttt{tt})] \vee [a \wedge \mathbf{X}(a \, \mathbf{U} \, b)] \\
&\equiv [b \wedge \mathbf{X}(\texttt{tt})] \vee [a \wedge \mathbf{X}(\phi_1)]
\end{aligned}
$$

thus there are two out-edges from this state defined by each disjunct. One of the out-edges transitions to the state for tt:

$$\texttt{tt} \equiv \mathbf{X}(\texttt{tt})$$

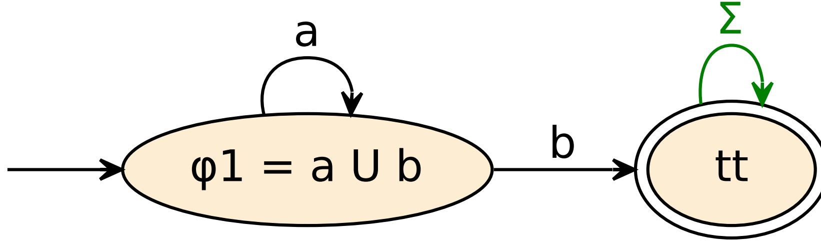


Figure 8.7: Büchi for $\phi = a\,\mathbf{U}\,b$

thus there is a single out-edge from this state (leading back to itself) which takes unrestricted input (all elements of the input alphabet Σ). Because this state has no outgoing edges (which would correspond to no pending obligations), it can be marked an accepting state. A graphical representation is given in Figure 8.7.

Example 2: $\phi = a\,\mathbf{U}_{\mathrm{N}}[c = 2]b$

Next we consider a lax left noisy example of Until, namely $\phi = a\,\mathbf{U}_{\mathrm{N}}[c = 2]b$. We start with $\phi_1 = \phi$:

$$\begin{aligned}\phi_1 &\equiv [b] \vee [a \wedge \mathbf{X}(a\,\mathbf{U}_{\mathrm{N}}[c = 2]b)] \vee [\mathbf{X}(a\,\mathbf{U}_{\mathrm{N}}[c = 1]b)] \\ &\equiv [b \wedge \mathbf{X}(\mathtt{tt})] \vee [a \wedge \mathbf{X}(\phi_1)] \vee [\mathbf{X}(a\,\mathbf{U}_{\mathrm{N}}[c = 1]b)]\end{aligned}$$

which leads us to $\phi_2 = a\,\mathbf{U}_{\mathrm{N}}[c = 1]b$:

$$\begin{aligned}\phi_2 &\equiv [b] \vee [a \wedge \mathbf{X}(a\,\mathbf{U}_{\mathrm{N}}[c = 1]b)] \vee [\mathbf{X}(a\,\mathbf{U}_{\mathrm{N}}[c = 0]b)] \\ &\equiv [b \wedge \mathbf{X}(\mathtt{tt})] \vee [a \wedge \mathbf{X}(\phi_2)] \vee [\mathbf{X}(a\,\mathbf{U}\,b)]\end{aligned}$$

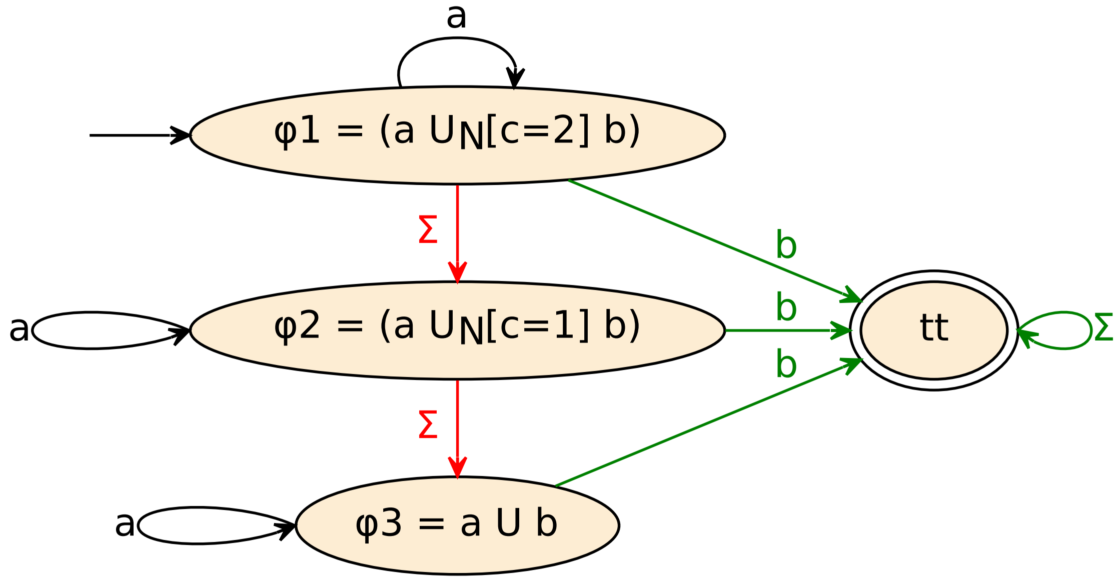


Figure 8.8: Black edges represent the "standard" transitions that would exist in a non-noisy version (i.e. $\phi = a\,\mathbf{U}\,b$). Red edges represent transitions between noisy state operators of different parameter value c. Green edges represent transitions leading to the accepting state.

and finally $\phi_3 = a\,\mathbf{U}\,b$:

$$\begin{aligned}\phi_3 &\equiv b \vee [a \wedge \mathbf{X}(a\,\mathbf{U}\,b)] \\ &\equiv [b \wedge \mathbf{X}(\texttt{tt})] \vee [a \wedge \mathbf{X}(\phi_3)]\end{aligned}$$

again with the standard tt state:

$$\texttt{tt} \equiv \mathbf{X}(\texttt{tt})$$

Putting this all together we can construct the 4-state Büchi automaton for $\phi = a\,\mathbf{U}_\mathrm{N}[c = 2]b$, as shown graphically in Figure 8.8.

Example 3: $\phi = (a\,\mathbf{U}\,b)\,\mathbf{U}\,c$

We now consider the more complex scenario of $\phi = (a\,\mathbf{U}\,b)\,\mathbf{U}\,c$, where we have nested (standard) LTL operators. We start with $\phi_1 = \phi$:

$$\begin{aligned}
\phi_1 &\equiv [c] \vee [(a\,\mathbf{U}\,b) \wedge \mathbf{X}((a\,\mathbf{U}\,b)\,\mathbf{U}\,c)] \\
&\equiv [c] \vee [(a\,\mathbf{U}\,b) \wedge \mathbf{X}(\phi_1)]
\end{aligned}$$

Here, the subformula $\phi_2 = (a\,\mathbf{U}\,b)$ is not covered by an immediately preceding $\mathbf{X}$, so we must now compute its own unrolling, and return to the final unrolling of ϕ_1:

$$\begin{aligned}
\phi_2 &\equiv b \vee [a \wedge \mathbf{X}(a\,\mathbf{U}\,b)] \\
&\equiv [b \wedge \mathbf{X}(\texttt{tt})] \vee [a \wedge \mathbf{X}(\phi_2)]
\end{aligned}$$

We can now continue the unrolling for ϕ_1:

$$\begin{aligned}
\phi_1 &\equiv [c] \vee [\phi_2 \wedge \mathbf{X}(\phi_1)] \\
&\equiv [c] \vee [(b \vee [a \wedge \mathbf{X}(\phi_2)]) \wedge \mathbf{X}(\phi_1)] \\
&\equiv [c] \vee [b \wedge \mathbf{X}(\phi_1)] \vee [a \wedge \mathbf{X}(\phi_2) \wedge \mathbf{X}(\phi_1)] \\
&\equiv [c \wedge \mathbf{X}(\texttt{tt})] \vee [b \wedge \mathbf{X}(\phi_1)] \vee [a \wedge \mathbf{X}(\phi_1 \wedge \phi_2)]
\end{aligned}$$

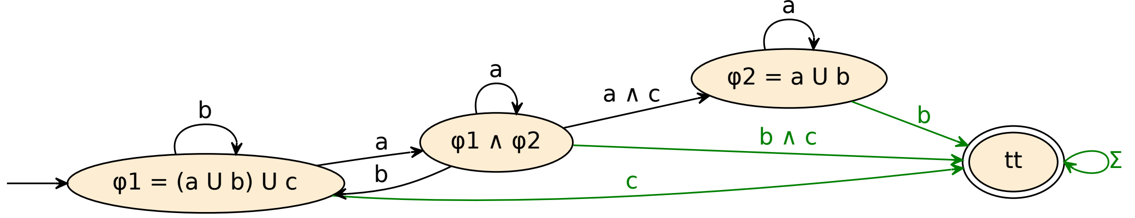


Figure 8.9: $\phi = (a\,\mathbf{U}\,b)\,\mathbf{U}\,c$. Green edges as before are transitions that lead to the accepting state.

which leads us to the state for $\phi_1 \wedge \phi_2$:

$$\begin{aligned}
\phi_1 \wedge \phi_2 &\equiv \Big([c \wedge \mathbf{X}(\mathtt{tt})] \vee [b \wedge \mathbf{X}(\phi_1)] \vee [a \wedge \mathbf{X}(\phi_1 \wedge \phi_2)]\Big) \wedge \Big([b \wedge \mathbf{X}(\mathtt{tt})] \vee [a \wedge \mathbf{X}(\phi_2)]\Big) \\
&\equiv \Big([c] \vee [b \wedge \mathbf{X}(\phi_1)] \vee [a \wedge \mathbf{X}(\phi_1 \wedge \phi_2)]\Big) \wedge \Big([b] \vee [a \wedge \mathbf{X}(\phi_2)]\Big) \\
&\equiv [b \wedge c] \vee [a \wedge c \wedge \mathbf{X}(\phi_2)] \vee [b \wedge \mathbf{X}(\phi_1)] \vee \\
&\quad\ [a \wedge b \wedge \mathbf{X}(\phi_1 \wedge \phi_2)] \vee [a \wedge b \wedge \mathbf{X}(\phi_1 \wedge \phi_2)] \vee [a \wedge \mathbf{X}(\phi_1 \wedge \phi_2)] \\
&\equiv [b \wedge c] \vee [a \wedge c \wedge \mathbf{X}(\phi_2)] \vee [b \wedge \mathbf{X}(\phi_1)] \vee [a \wedge \mathbf{X}(\phi_1 \wedge \phi_2)]
\end{aligned}$$

Figure 8.9 shows the graphical representation of this.

Example 4: $\phi = (a\,\mathbf{U}\,b)\,\mathbf{U}_{\mathrm{N}}[c = 2]c$

We now look at adding noise to the previous scenario, specifically on the outer-level modality, namely $\phi = (a\,\mathbf{U}\,b)\,\mathbf{U}_{\mathrm{N}}[c = 2]c$. We start with $\phi_1 = \phi$:

$$\begin{aligned}
\phi_1 &\equiv [c] \vee [(a\,\mathbf{U}\,b) \wedge \mathbf{X}((a\,\mathbf{U}\,b)\,\mathbf{U}_{\mathrm{N}}[c = 2]c)] \vee [\mathbf{X}((a\,\mathbf{U}\,b)\,\mathbf{U}_{\mathrm{N}}[c = 1]c] \\
&\equiv [c] \vee [(a\,\mathbf{U}\,b) \wedge \mathbf{X}(\phi_1)] \vee [\mathbf{X}((a\,\mathbf{U}\,b)\,\mathbf{U}_{\mathrm{N}}[c = 1]c]
\end{aligned}$$

We pause to unroll $\phi_2 = a\,\mathbf{U}\,b$:

$$\phi_2 \equiv [b] \vee [a \wedge \mathbf{X}(a\,\mathbf{U}\,b)]$$

as well as $\phi_3 = (a\,\mathbf{U}\,b)\,\mathbf{U}_{\mathrm{N}}[c=1]c = \phi_2\,\mathbf{U}_{\mathrm{N}}[c=1]c$:

$$\begin{aligned}\phi_3 &\equiv [c] \vee [\phi_2 \wedge \mathbf{X}(\phi_2\,\mathbf{U}_{\mathrm{N}}[c=1]c)] \vee [\mathbf{X}(\phi_2\,\mathbf{U}_{\mathrm{N}}[c=0]c)] \\ &\equiv [c] \vee [\phi_2 \wedge \mathbf{X}(\phi_3)] \vee [\mathbf{X}(\phi_2\,\mathbf{U}\,c)]\end{aligned}$$

We then recurse on unrolling $\phi_4 = (a\,\mathbf{U}\,b)\,\mathbf{U}\,c$. From the previous example, we have that this unrolling is

$$\phi_4 \equiv [c] \vee [b \wedge \mathbf{X}(\phi_4)] \vee [a \wedge \mathbf{X}(\phi_2 \wedge \phi_4)]$$

Also from the previous example, we have the unrolling for $\phi_2 \wedge \phi_4$:

$$\phi_2 \wedge \phi_4 \equiv [b \wedge c] \vee [b \wedge \mathbf{X}(\phi_4)] \vee [a \wedge c \wedge \mathbf{X}(\phi_2)] \vee [a \wedge \mathbf{X}(\phi_2 \wedge \phi_4)]$$

We can now resume the unrolling for ϕ_3 by inserting the unrolling for ϕ_2:

$$\begin{aligned}\phi_3 &\equiv [c] \vee [(b \vee (a \wedge \mathbf{X}(\phi_2))) \wedge \mathbf{X}(\phi_3)] \vee [\mathbf{X}(\phi_4)] \\ &\equiv [c] \vee [b \wedge \mathbf{X}(\phi_3)] \vee [a \wedge \mathbf{X}(\phi_2 \wedge \phi_3)] \vee [\mathbf{X}(\phi_4)]\end{aligned}$$

And we can return to finish the unrolling for ϕ_1:

$$\begin{aligned}\phi_1 &\equiv [c] \vee [([b] \vee [a \wedge \mathbf{X}(\phi_2]) \wedge \mathbf{X}(\phi_1)] \vee [\mathbf{X}(\phi_3)] \\ &\equiv [c] \vee [b \wedge \mathbf{X}(\phi_1)] \vee [a \wedge \mathbf{X}(\phi_1 \wedge \phi_2)] \vee [\mathbf{X}(\phi_3)]\end{aligned}$$

We now can compute the unrolling for the state $\phi_2 \wedge \phi_3$:

$$\begin{aligned}\phi_2 \wedge \phi_3 \equiv &\Big[b \vee (a \wedge \mathbf{X}(\phi_2))\Big] \wedge \Big[[c] \vee [b \wedge \mathbf{X}(\phi_3)] \vee [a \wedge \mathbf{X}(\phi_2 \wedge \phi_3)] \vee [\mathbf{X}(\phi_4)]\Big] \\ &\equiv [b \wedge c] \vee [b \wedge \mathbf{X}(\phi_3)] \vee [a \wedge b \wedge \mathbf{X}(\phi_2 \wedge \phi_3)] \vee [b \wedge \mathbf{X}(\phi_4)] \vee \\ &\quad [a \wedge c \wedge \mathbf{X}(\phi_2)] \vee [a \wedge b \wedge \mathbf{X}(\phi_2 \wedge \phi_3)] \vee [a \wedge \mathbf{X}(\phi_2 \wedge \phi_3)] \vee [a \wedge \mathbf{X}(\phi_2 \wedge \phi_4)] \\ &\equiv [b \wedge c] \vee [b \wedge \mathbf{X}(\phi_3)] \vee [b \wedge \mathbf{X}(\phi_4)] \vee \\ &\quad [a \wedge c \wedge \mathbf{X}(\phi_2)] \vee [a \wedge \mathbf{X}(\phi_2 \wedge \phi_3)] \vee [a \wedge \mathbf{X}(\phi_2 \wedge \phi_4)]\end{aligned}$$

Finally, we can unroll $\phi_1 \wedge \phi_2$:

$$\begin{aligned}\phi_1 \wedge \phi_2 \equiv &\Big[[c] \vee [b \wedge \mathbf{X}(\phi_1)] \vee [a \wedge \mathbf{X}(\phi_1 \wedge \phi_2)] \vee [\mathbf{X}(\phi_3)]\Big] \wedge \Big[[b] \vee [a \wedge \mathbf{X}(\phi_2)]\Big] \\ &\equiv [b \wedge c] \vee [a \wedge c \wedge \mathbf{X}(\phi_2)] \vee [b \wedge \mathbf{X}(\phi_1)] \vee [a \wedge b \wedge \mathbf{X}(\phi_1 \wedge \phi_2)] \vee \\ &\quad [a \wedge b \wedge \mathbf{X}(\phi_1 \wedge \phi_2)] \vee [a \wedge \mathbf{X}(\phi_1 \wedge \phi_2)] \vee [b \wedge \mathbf{X}(\phi_3)] \vee [a \wedge \mathbf{X}(\phi_2 \wedge \phi_3)] \\ &\equiv [b \wedge c] \vee [a \wedge c \wedge \mathbf{X}(\phi_2)] \vee [b \wedge \mathbf{X}(\phi_1)] \vee \\ &\quad [a \wedge \mathbf{X}(\phi_1 \wedge \phi_2)] \vee [b \wedge \mathbf{X}(\phi_3)] \vee [a \wedge \mathbf{X}(\phi_2 \wedge \phi_3)]\end{aligned}$$

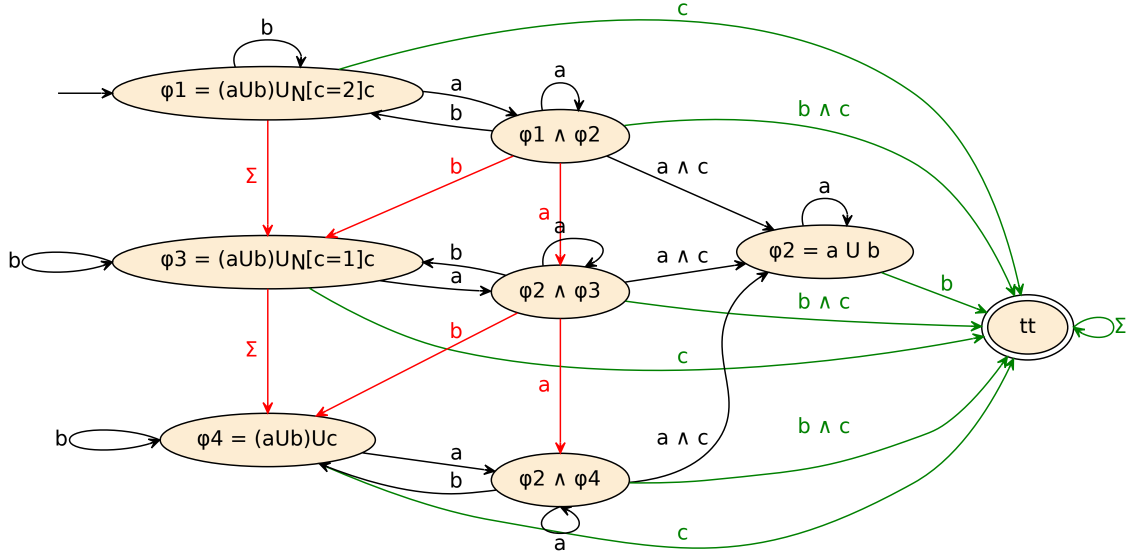


Figure 8.10: $\phi = (a\,\mathbf{U}\,b)\,\mathbf{U}_{\mathrm{N}}[c = 2]c$. Edge colors are as in the past few figures. Note that when compared to Figure 8.9, we have effectively produced parameterized copies of states referring to the outer modality of ϕ.

Which concludes the final unrolling with all temporal modalities captured by a $\mathbf{X}$ operator. Figure 8.10 shows the graphical representation of this.

Example 5: $\phi = a\,\mathbf{R}\,b$

Let $\phi_1 = \phi$:

$$\phi_1 \equiv b \wedge [a \vee \mathbf{X}(a\,\mathbf{R}\,b)]$$

$$\equiv [a \wedge b] \vee [b \wedge \mathbf{X}(a\,\mathbf{R}\,b)]$$

$$\equiv [a \wedge b \wedge \mathbf{X}(\texttt{tt})] \vee [b \wedge \mathbf{X}(\phi_1)]$$

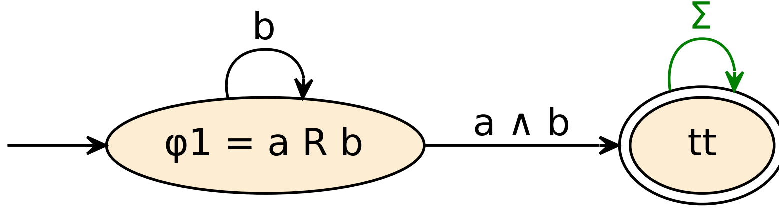


Figure 8.11: $\phi = a\,\mathbf{R}\,b$.

Graphically, this is shown in Figure 8.11

Example 6: $\phi = a\,\mathbf{R}_{\mathrm{N}}[c = 2]b$

Let $\phi_1 = \phi$:

$$\begin{aligned}\phi_1 &\equiv [a \wedge b \wedge \mathbf{X}(a\,\mathbf{R}_{\mathrm{N}}[c = 1]b)] \vee [b \wedge \mathbf{X}(a\,\mathbf{R}_{\mathrm{N}}[c = 2]b)] \\ &\equiv [a \wedge b \wedge \mathbf{X}(a\,\mathbf{R}_{\mathrm{N}}[c = 1]b)] \vee [b \wedge \mathbf{X}(\phi_1)]\end{aligned}$$

Unrolling $\phi_2 = a\,\mathbf{R}_{\mathrm{N}}[c = 1]b$:

$$\begin{aligned}\phi_2 &\equiv [a \wedge b \wedge \mathbf{X}(a\,\mathbf{R}_{\mathrm{N}}[c = 0]b)] \vee [b \wedge \mathbf{X}(a\,\mathbf{R}_{\mathrm{N}}[c = 1]b)] \\ &\equiv [a \wedge b \wedge \mathbf{X}(a\,\mathbf{R}\,b)] \vee [b \wedge \mathbf{X}(\phi_2)]\end{aligned}$$

Unrolling $\phi_3 = a\,\mathbf{R}\,b$:

$$\phi_3 \equiv [a \wedge b] \vee [b \wedge \mathbf{X}(\phi_3)]$$

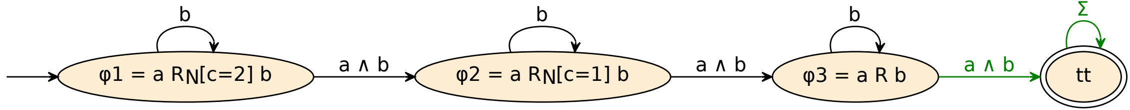


Figure 8.12: $\phi = a\,\mathbf{R}_{\mathrm{N}}[c = 2]b$.

Returning to "anonymize" ϕ_1 and ϕ_2:

$$\phi_1 \equiv [a \wedge b \wedge \mathbf{X}(\phi_2)] \vee [b \wedge \mathbf{X}(\phi_1)]$$

$$\phi_2 \equiv [a \wedge b \wedge \mathbf{X}(\phi_3)] \vee [b \wedge \mathbf{X}(\phi_2)]$$

Figure 8.12 shows the graphical representation.

Example 7: $\phi = (a\,\mathbf{R}\,b)\,\mathbf{R}\,c$

Let $\phi_1 = \phi$:

$$\phi_1 \equiv [(a\,\mathbf{R}\,b) \wedge c] \vee [c \wedge \mathbf{X}((a\,\mathbf{U}\,b)\,\mathbf{U}\,c)] \equiv [(a\,\mathbf{R}\,b) \wedge c] \vee [c \wedge \mathbf{X}(\phi_1]$$

Recurse by unrolling $\phi_2 = a\,\mathbf{R}\,b$:

$$\phi_2 \equiv [a \wedge b] \vee [b \wedge \mathbf{X}(\phi_2)]$$

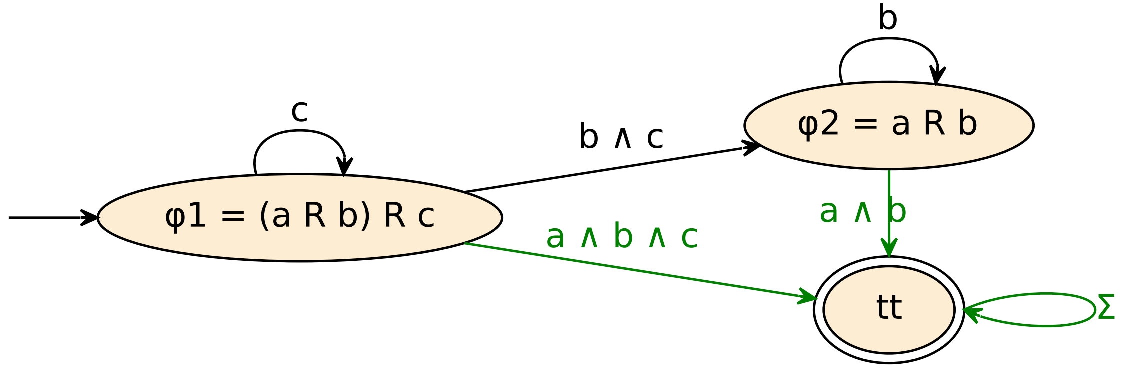


Figure 8.13: $\phi = (a \,\mathbf{R}\, b) \,\mathbf{R}\, c$.

Going back to ϕ_1:

$$\phi_1 \equiv [([a \wedge b] \vee [b \wedge \mathbf{X}(\phi_2)]) \wedge c] \vee [c \wedge \mathbf{X}(\phi_1]$$

$$\equiv [a \wedge b \wedge c] \vee [b \wedge c \wedge \mathbf{X}(\phi_2)] \vee [c \wedge \mathbf{X}(\phi_1]$$

Figure 8.13 shows the graphical representation.

Example 8: $\phi = (a \,\mathbf{R}\, b) \,\mathbf{R}_{\mathrm{N}}[c = 2]c$

Let $\phi_1 = \phi$:

$$\phi_1 \equiv [(a \,\mathbf{R}\, b) \wedge c \wedge \mathbf{X}((a \,\mathbf{R}\, b) \,\mathbf{R}_{\mathrm{N}}[c = 1]c)] \vee [c \wedge \mathbf{X}(\phi_1)]$$

Unrolling $\phi_2 = (a\,\mathbf{R}\,b)\,\mathbf{R}_{\mathrm{N}}[c=1]c$:

$$
\begin{aligned}
\phi_2 &\equiv [(a\,\mathbf{R}\,b) \wedge c \wedge \mathbf{X}((a\,\mathbf{R}\,b)\,\mathbf{R}_{\mathrm{N}}[c=0]c)] \vee [c \wedge \mathbf{X}(\phi_2)] \\
&\equiv [(a\,\mathbf{R}\,b) \wedge c \wedge \mathbf{X}((a\,\mathbf{R}\,b)\,\mathbf{R}\,c)] \vee [c \wedge \mathbf{X}(\phi_2)]
\end{aligned}
$$

Unrolling $\phi_3 = a\,\mathbf{R}\,b$:

$$
\phi_3 \equiv [a \wedge b] \vee [b \wedge \mathbf{X}(\phi_3)]
$$

Unrolling $\phi_4 = (a\,\mathbf{R}\,b)\,\mathbf{R}\,c$:

$$
\phi_4 \equiv [a \wedge b \wedge c] \vee [b \wedge c \wedge \mathbf{X}(\phi_3)] \vee [c \wedge \mathbf{X}(\phi_4)]
$$

Returning to ϕ_2:

$$
\begin{aligned}
\phi_2 &\equiv [\phi_3 \wedge c \wedge \mathbf{X}(\phi_4)] \vee [c \wedge \mathbf{X}(\phi_2)] \\
&\equiv [\Big([a \wedge b] \vee [b \wedge \mathbf{X}(\phi_3)]\Big) \wedge c \wedge \mathbf{X}(\phi_4)] \vee [c \wedge \mathbf{X}(\phi_2)] \\
&\equiv [a \wedge b \wedge c \wedge \mathbf{X}(\phi_4)] \vee [b \wedge c \wedge \mathbf{X}(\phi_3 \wedge \phi_4)] \vee [c \wedge \mathbf{X}(\phi_2)]
\end{aligned}
$$

And returning to ϕ_1:

$$
\begin{aligned}
\phi_1 &\equiv [\phi_3 \wedge c \wedge \mathbf{X}(\phi_2)] \vee [c \wedge \mathbf{X}(\phi_1)] \\
&\equiv [\Big([a \wedge b] \vee [b \wedge \mathbf{X}(\phi_3)]\Big) \wedge c \wedge \mathbf{X}(\phi_2)] \vee [c \wedge \mathbf{X}(\phi_1)] \\
&\equiv [a \wedge b \wedge c \wedge \mathbf{X}(\phi_2)] \vee [b \wedge c \wedge \mathbf{X}(\phi_2 \wedge \phi_3)] \vee [c \wedge \mathbf{X}(\phi_1)]
\end{aligned}
$$

Unrolling $\phi_2 \wedge \phi_3$:

$$
\begin{aligned}
\phi_2 \wedge \phi_3 &\equiv \Big([a \wedge b \wedge c \wedge \mathbf{X}(\phi_4)] \vee [b \wedge c \wedge \mathbf{X}(\phi_3 \wedge \phi_4)] \vee [c \wedge \mathbf{X}(\phi_2)]\Big) \wedge \Big([a \wedge b] \vee [b \wedge \mathbf{X}(\phi_3)]\Big) \\
&\equiv [a \wedge b \wedge c \wedge \mathbf{X}(\phi_4)] \vee [a \wedge b \wedge c \wedge \mathbf{X}(\phi_3 \wedge \phi_4)] \vee [a \wedge b \wedge c \wedge \mathbf{X}(\phi_2)] \vee \\
&\quad [a \wedge b \wedge c \wedge \mathbf{X}(\phi_3 \wedge \phi_4)] \vee [b \wedge c \wedge \mathbf{X}(\phi_3 \wedge \phi_4)] \vee [b \wedge c \wedge \mathbf{X}(\phi_2 \wedge \phi_3)] \\
&\equiv [a \wedge b \wedge c \wedge \mathbf{X}(\phi_4)] \vee [a \wedge b \wedge c \wedge \mathbf{X}(\phi_2)] \vee \\
&\quad [b \wedge c \wedge \mathbf{X}(\phi_3 \wedge \phi_4)] \vee [b \wedge c \wedge \mathbf{X}(\phi_2 \wedge \phi_3)]
\end{aligned}
$$

Unrolling $\phi_3 \wedge \phi_4$:

$$
\begin{aligned}
\phi_3 \wedge \phi_4 &\equiv \Big([a \wedge b] \vee [b \wedge \mathbf{X}(\phi_3)]\Big) \wedge \Big([a \wedge b \wedge c] \vee [b \wedge c \wedge \mathbf{X}(\phi_3)] \vee [c \wedge \mathbf{X}(\phi_4)]\Big) \\
&\equiv [a \wedge b \wedge c] \vee [a \wedge b \wedge c \wedge \mathbf{X}(\phi_3)] \vee [a \wedge b \wedge c \wedge \mathbf{X}(\phi_4)] \vee \\
&\quad [a \wedge b \wedge c \wedge \mathbf{X}(\phi_3)] \vee [b \wedge c \wedge \mathbf{X}(\phi_3)] \vee [b \wedge c \wedge \mathbf{X}(\phi_3 \wedge \phi_4)] \\
&\equiv [a \wedge b \wedge c] \vee [a \wedge b \wedge c \wedge \mathbf{X}(\phi_4)] \vee [b \wedge c \wedge \mathbf{X}(\phi_3)] \vee [b \wedge c \wedge \mathbf{X}(\phi_3 \wedge \phi_4)]
\end{aligned}
$$

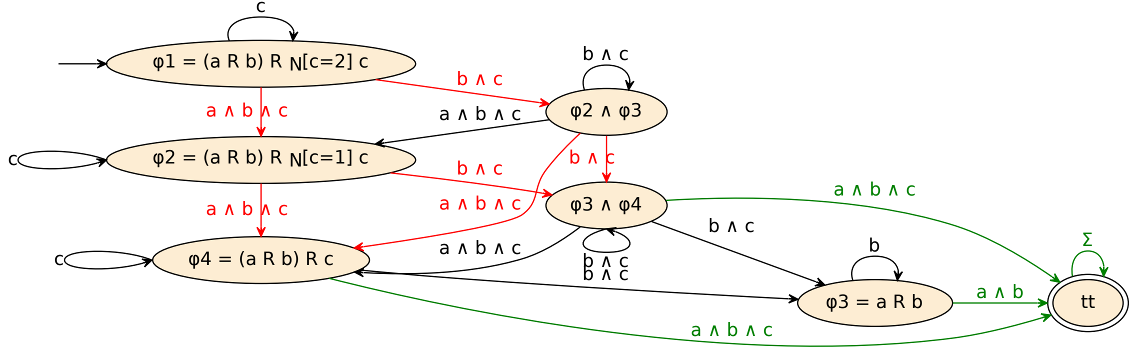


Figure 8.14: $\phi = (a\,\mathbf{R}\,b)\,\mathbf{R}_{\mathrm{N}}[c = 2]c$

Figure 8.14 shows the graphical representation.

8.5 Window-based Noisy LTL

The above syntax and semantics for noisy LTL employ violation sets bounded in terms of *raw counts* of time points where noise can occur. In some instances this may not be the most desirable way of expressing deviation. For example, the formula $a\,\mathbf{U}_{\mathrm{N}}[3]b$ requires an unspecified, finite range of time where property a is satisfied at each point (allowing up to 3 time points where a may be omitted), up until property b is satisfied once. As no bounds is placed on the length of the a range, the choice of $c = 3$ as the noisy parameter may have distorted implications when considered over different sequences where the lengths of the a ranges differ drastically. We consider an alternative manner to represent noise, using a notion of a sliding window over the data stream π. Such approaches have been researched in the past decade and a half (at least) in the database community [95, 96] and have found increasing importance with the growing size of data sets.

The intuition behind using a sliding window is to create fixed-width, overlapping frames along π on top of normative ranges from LTL. In each such frame the noisy version of the requirement is locally enforced. Noisy operators using a sliding window have an additional noise parameter w, which is the frame width (e.g. $\mathbf{U}_{\mathrm{N}}[c, w]$). For example, $a\, \mathbf{R}'_{\mathrm{N}}[c = 3, w = 100] b$ would require that for every 100-width frame, until a is observed, we would allow at most 3 missing values of b in the frame. We first introduce some notation to make expressing the semantics of window-based noisy LTL easier.

Definition 37 (Sliding Window Notation)**.** *For the purposes of a sliding window, We define the following terms:*

- *For* $i, w \in \mathbb{N}$*, let* $\mathrm{window}(i, w) = \{i, \ldots, i + w - 1\}$*, i.e. the indices of the* w*-width window starting at time point* i*.*
- *For* $w, f \in \mathbb{N}$*, let* $\mathrm{before}(w, f) = \{0, \ldots, f - w\}$ *be the set of all starting indices for* w*-width windows that do not include indices past* f*.*

Definition 38 (Sliding Window Syntax and Semantics)**.** *For a fixed set of atomic propositions* $\mathcal{AP}$*,* sliding-window noisy LTL *has the following syntax and semantics:*

Syntax *Syntax is the same as in Definition 33, with noise parameters* $[c, w]$

replace noise parameter $[c]$*:*

$$\phi ::= a \mid \neg\phi \mid \phi \wedge \phi \mid \mathbf{X}\,\phi \mid \phi\,\mathbf{U}\,\phi \qquad \textit{LTL (base)}$$

$$\mid \boldsymbol{ff} \mid \boldsymbol{tt} \mid \phi \vee \phi \mid \phi\,\mathbf{R}\,\phi \mid \mathbf{F}\,\phi \mid \mathbf{G}\,\phi \qquad \textit{LTL (derived)}$$

$$\mid \phi\,\mathbf{U}_{\mathrm{N}}[c,w]\phi \mid \phi\,\mathbf{R}_{\mathrm{N}}[c,w]\phi \qquad \textit{left-noisy operators}$$

$$\mid \phi\,\mathbf{U}'_{\mathrm{N}}[c,w]\phi \mid \phi\,\mathbf{R}'_{\mathrm{N}}[c,w]\phi \mid \mathbf{F}'_{\mathrm{N}}[c]\phi \mid \mathbf{G}'_{\mathrm{N}}[c]\phi \qquad \textit{right-noisy operators}$$

Semantics *The semantics for the traditional LTL fragment are the same as presented in the raw count case. The semantics for the noisy window operators are:*

$$\begin{aligned}
\pi \models \phi\,\mathbf{U}_{\mathrm{N}}[c,w]\psi \ \textit{iff}\ \exists\,\mathcal{V} \subseteq \mathbb{N}, \exists i \in \mathbb{N} :\\
\pi(i) \models \psi \wedge \forall j \in \mathrm{before}(w,i) :\\
(|\,\mathcal{V} \cap \mathrm{window}(j,w)| \leq c) \wedge \forall k \in \mathbb{N} - \mathcal{V} :\\
(k \in \mathrm{window}(j,w)) \implies \pi(k) \models \phi
\end{aligned}$$

$$\begin{aligned}
\pi \models \phi\,\mathbf{R}'_{\mathrm{N}}[c,w]\psi \ \textit{iff}\ \exists\,\mathcal{V} \subseteq \mathbb{N}, \forall i \in \mathbb{N} :\\
(|\,\mathcal{V} \cap \mathrm{window}(i,w)| \leq c) \wedge \forall j \in \mathrm{window}(i,w) - \mathcal{V} :\\
\pi(j) \models \psi \vee \exists k \in \mathbb{N} : (0 \leq k < j) \wedge \pi(k) \models \phi
\end{aligned}$$

The laxness is enforced per window frame. The duals $\mathbf{R}_{\mathrm{N}}[c, w]$ and $\mathbf{U}'_{\mathrm{N}}[c, w]$ can be computed as normal (as well as $\mathbf{F}'_{\mathrm{N}}[c, w]$ and $\mathbf{G}'_{\mathrm{N}}[c, w]$), which requires redundancy per window frame. One could also equivalently employ a ratio-based noisy operator here (denoted with r). We observe that here the choice between raw-count c or ratio-based r as a noisy parameter is inconsequential; for the above semantics one can switch between the conventions $[c, w]$ and $[r, w]$ using $r = \frac{c}{w}$.

8.5.1 Decidability for Sliding Window Semantics

We pause here to consider possible methods of computing decidability for noisy LTL under sliding window semantics. As suggested before, De Bruijn graphs have the natural support for a sliding window. One thought would be to encode the semantics of a sliding window frame as a De Bruijn graph represented as a Büchi automaton and then compose it with a de-noised version of the input formula to produce a joint automata representing semantics of both the sliding window AND the temporal logic formula. In general we would have a representation of multiple De Bruijn windows, one for each noisy operator with window semantics.

Definition 39 (De Bruijn Graph)**.** *For a set of symbols Σ with $|\Sigma| = m$ and $n \in \mathbb{N}$, a* De Bruijn graph *$DB(m, n) = (V, E)$ is a directed graph of m^n vertices, defined explicitly as follows:*

- *The vertex set $V = \Sigma^n$ is the set of all n-length sequences of symbols from Σ.*
- *An edge (u, v) belongs to the edge set $E \subseteq V \times V$ if and only if there exists*

some word $w \in \Sigma^{n-1}$ *and two symbols* $a, b \in \Sigma$ *such that* $u = aw$ *and* $v = wb$. *We can consider the edge labeled with* b.

Some possible structures that could be of use as as follows. For a noisy LTL formula $\phi \in \Phi^{\mathcal{AP}}$ and a path sequence $\pi \in (2^{\mathcal{AP}})^\omega$, consider:

- Let $\phi(\pi) = \begin{cases} 1 & \text{if } \pi \models \phi \\ 0 & \text{otherwise} \end{cases}$ be an indicator variable for whether or not π models ϕ,

- Let $\pi_\phi = \phi(\pi(0))\phi(\pi(1))\dots\phi(\pi(n))\dots$ be the infinite sequence of bits such that the bit at time index i indicates whether or not the suffix of π starting at index i (itself an infinite sequences) satisfies ϕ or not.

- For noise parameter c with window size w, let $G = DB(2, w)$. Let G' be the subgraph of G formed by deleting all vertices containing $c + 1$ or more 0's in its sequence, as well as all incident edges. $L(G')$, the set of all infinite sequences generated by G', is exactly the set of all infinite sequences that satisfy the sliding window semantics. Note that we can use $L(\cdot)$ here as defined for Büchi automaton by first converting G' into a Büchi $B_{DB(\phi)}$ by using the implied transition label as mentioned in Definition 39 and marking all states as accepting. For a particular choice of c and w, we know the exact number of states and edges of $G' = (V, E)$:

  - $|V| = \sum_{i=0}^{c} \binom{w}{i}$

– $|E| = 2(\sum_{i=0}^{c-1} \binom{w}{i}) + 1.5\binom{w}{c} = 2|V| - 0.5\binom{w}{c} = O(|V|)$

Observe that for any node u containing at most $(c-1)$ 0's, there is in G' exactly two edges (u, v_1) and (u, v_2) with u as a source node. However, for any node u which has exactly c 0's in its sequence, the edge leaving labeled by a 0 would have led to a node that was deleted from G if and only if the first symbol of u's sequence is a 1. Thus by symmetry half of these nodes have one outgoing edge and the other half have two and thus on average there are 1.5 outgoing edges.

Using such a De Bruijn construct, we could compose it with with a de-noised version of a noisy LTL formula-turned Büchi to produce a joint Büchi automaton. The De Bruijn construct would capture the sliding window semantics for legal (as defined by the noisy parameters) frameshifts while the de-noised noisy LTL formula captures the LTL semantics. The joint Büchi would capture the semantics of both. For example, if $\mathcal{AP} = \{a, b\}$, and $\phi = a\,\mathbf{U}_{\mathrm{N}}[c = 2, w = 5]b$, then we would do the following:

1. Let $\phi' = a\,\mathbf{U}\,b$ be the de-noised LTL formula. Compute B_ϕ. This represents the LTL window semantics.

2. Compute $B_{DB(\phi)}$ as above from $DB(2, 5)$. This represents the window semantics.

3. Compute the compose $B_\phi \cap B_{DB(\phi)}$.

The language of this composed Büchi would be equivalent to the set of sequences modeling ϕ. This resulting Büchi would then be used in the normal fashion for model checking (see 3.2.1). This approach currently only is tractable for formulas where noisy operators have only propositional formulas as their noisy subformula, the reason being that individual indices of π_ϕ are computable in $O(1)$ time as sequences are read. This is in general not the case when an arbitrary subformula is used. Chapter 9 lists addressing this limitation as future work.

8.5.2 Window-Based Example Formulas

We provide some additional example formulase here for window-bases semantics.

- `sick_day` $\mathbf{U}_{\mathrm{N}}[c = 3, w = 30]$`end_of_school` - a student is never sick for more than 3 days on a given 30 day interval.
- `knock` $\mathbf{R}_{\mathrm{N}}[c = 2, w = 10]$`door_closed` - The door remains closed unless you knock at least 3 times within 10 time units.
- `knock` $\mathbf{R}'_{\mathrm{N}}[c = 2, w = 10]$`door_closed` - The door is closed *most of the time* until you knock.
- `hungry` $\mathbf{U}'_{\mathrm{N}}[r = 0.5, w = 2]$`eat` - I will be hungry until I eat enough within a 2 hour window.
- $\mathbf{F}'_{\mathrm{N}}[c = 3, w = 20]$`light` - At some future time point, the light will be on at least 4 times within the next 20 time steps.

- $\mathbf{G}'_{\mathrm{N}}[r = 0.99, w = 1000]\,\mathrm{service_up}$ - for any 1000 consecutive time units, the service is up 99% of the time. [1]

8.6 Conclusion

In this chapter we have explored the notion of partial invariants as a means to characterize degrees of uncertainty in a system and developed a novel temporal logic termed "noisy Linear Temporal Logic" to aid in characterizing it. We showed how to convert formulas from the logic to equivalent Büchi automata, which allows for well-established standard LTL to be applicable in order to solve the usual formal methods problems such as language emptiness and verification. We also explored an alternative version of noisy LTL which used sliding windows in order to better capture properties of interest. have We shown decidabilty for a fragment of this variant.

[1] This models the notion of High Availability for a service.

Chapter 9: Conclusion

9.1 Summarized Results of Problems Addressed

This document describes the work done developing a framework for learning temporal properties from data streams, leveraging research conducted both in the model checking and machine learning domains. The problem domains and motivation were provided in Chapter 1. Related work was presented in Chapter 2, while more related work more foundational to our own research was presented with more in-depth detail in Chapter 3. The remaining chapters presented our research contributions scoped of this thesis. We provide here a per-chapter summary of the specific problems considered along with our resulting solutions, including some discussion on the evidence/artifacts encountered that support our findings.

9.1.1 Chapter 4 — Model-Based Invariant Extraction from Test Cases

An automated methodology for mining putative system invariants from test cases was proposed. Putative invariants were mined using an iterative approach incorporating data mining and coverage-guided test case generation. The technique was

applied in a pilot study involving a MATLAB/Simulink implementation of a vehicle cruise control, where invariants were recovered and compared against a natural language documentation specifying system behavior. Some invariants were found to exist that were not present in the document specification, and some were found to be missing from the original specification. This justified the use of the methodology as an automated technique for comparing model-based implementations to earlier representations such as design documents earlier in product development.

9.1.2 Chapter 5 — LTL Query Checking

This chapter presented an automaton-based solution to LTL query checking problem. This method adapted the well-known automaton-theoretic approach to LTL model checking by extending the standard conversion of LTL formulas into automata to accommodate a representation for LTL queries into query automata. Such a modification was shown to be done syntactically, and thus existing tools for constructing Büchi automaton from LTL formulas could be used (in our case LTL3BA). The subproblem of shattering edges in a Büchi automaton was also presented and explored. Sample models from the NuSMV model checker distribution were taken and run against an instrumented NuSMV which produced explicit Kripke structures for the models. Query checking was then performed for a number of safety properties. It was shown that the complexity of the algorithm was $O\left(|\mathcal{M}| \cdot 2^{2^{|\mathcal{AP}|}}\right)$. However, in situations where a single minimal shattering set exists, a greedy algorithm was

presented that performs in $O\left(|\mathcal{M}| \cdot |\mathcal{AP}| \cdot 2^{|\mathcal{AP}|}\right)$.

9.1.3 Chapter 6 — Finite LTL

This chapter presented a novel representation of linear temporal logic using finite-sequence semantics in order to naturally support and model properties over sequences that occur in the real world. We presented syntax and semantics for the novel logic, show a number of theoretical results concerning Finite LTL, and provide a tableau-style construction to product a finite automaton from a Finite LTL formula, analogous to the well-known tableau constructions for traditional LTL that construct a Büchi automaton from an LTL formula. We also discuss the potential for several optimizations that could be made to the construction which improve the running time of the construction as well as the size of the resulting automaton. Lastly, we provide empirical results from our implementation of the tableau construction.

9.1.4 Chapter 7 — Finite LTL Query Checking

This chapter combined some of the insights drawn from the previous two chapters, integrating the problem of LTL query checking and the problem of supporting finite sequence semantics of the model. We present work done on Finite LTL query checking, and assume we are provided not with a model to query check but instead with a set of its executions. The subproblem of shattering was revisited with extra

considerations taken due to the nature of the finite sequence semantics being used, and a number of theoretical results were provided. We also reported on a set of empirical results.

9.1.5 Chapter 8 — Formal Verification of Noisy Sequences

In the last chapter detailing our research contributions to this document, we explored the problem of representing uncertainty and deviation in data streams from some intended baseline behavior. We introduced Noisy LTL, a derivative of LTL whose robustness to deviation is parameterized for added flexibility. We provided a number of theoretical results both regarding Noisy LTL itself as well as relating it to traditional LTL. We also provided a construction from a Noisy LTL formula to Büchi automaton, which enables model checking of Noisy LTL for data streams.

9.2 Future Work by Chapter

This section discusses possible future lines of research for content presented in this thesis, broken down by chapter. While outside the scope of the thesis itself, much of the future work listed here can be considered natural extensions of the thesis material.

9.2.1 Chapter 4 Future Work

- **Alternative coverage criteria** - The iterative refinement methodology presented in Chapter 4 presented a template framework. As such, we leave as future work the exploration of alternative approaches different coverage criteria would have on the quality of invariant sets.

- **Expand expressiveness of requirements** - The language of association rules is somewhat limited compared to the full expressibility that a temporal logic offers. For the purposes of our project, we were effectively mining the temporal logic fragment $\Gamma \implies \mathbf{X}\, newState$. As future work we leave the exploration of different requirement-generation strategies which would permit the inference of a class of requirements at the level of (perhaps) a full temporal logic. As mentioned in Section 4.4.2, the choice of utilizing Magnum Opus restricts invariant expression, especially in the case where ranged invariants have a semantic meaning attached (especially for real-valued inputs and outputs). If, given a data set where such meaning is possible, we would need to make use of an alternative data mining approach, such as an adaptation of Daikon to suite our needs.

9.2.2 Chapter 5 Future Work

- **Relax query restrictions** - We have presented a graph-based technique for performing query checking on system models. Currently we have only explored formulas in which a single missing subformula is allowed. We leave as future work the extension to queries involving multiple missing subformulas, as well as queries in which the missing subformula can appear both positively and negatively in the query.

- **Exploit edge label relationships** - When currently considering sets of edge labels, we assume that they are i.i.d.. However, some preliminary empirical findings suggest that oftentimes we have distinct edge labels that are logically related. For example, in a set E of edge labels we might have two propositional queries $\phi[\texttt{var}]$ and $\psi[\texttt{var}]$ such that $\phi[\texttt{var}] \implies \psi[\texttt{var}]$. In this case, we can group $\phi[\texttt{var}]$ and $\psi[\texttt{var}]$ as a single group action to shatter or not shatter. Exploring such relationships could help to reduce search time of a shattering solution.

- **Explore edge label orderings** - Currently when building the sets of edge labels to determine a shattering condition we choose labels at random. Different orderings could be explored, such as biasing selection of edge labels based on some metrics of the graph. Raw frequency is one option, one could also consider computing some notion of edge centrality to each label and ranking

based on that metric. Solving edge shattering in general is still computationally challenging; the goal would be to explore which heuristic works best under which classes of scenarios.

9.2.3 Chapter 6 Future Work

- **Additional optimizations to tableau construction** - Our tableau construction is to our knowledge the first such construction for a LTL logic defined over finite semantics. The general tableau approach for constructing automaton from a temporal logic formula presents computational challenges (the size of the resulting machine M_{phi} is exponential in the number of subformulas of ϕ); consequently research has focused on optimizations and heuristics to the construction. While our novel approach does leverage some of the more common improvements (on-the-fly construction of Q_ϕ and symbolic representation of transitions), a number of additional heuristics could be leveraged from existing standard LTL literature to the finite-semantic tableau construction. Furthermore, we leave as future work the exploration of additional improvements that could be made due to the finite nature of our logic, such as the ability to syntactically determine acceptance (which we already have presented).

9.2.4 Chapter 7 Future Work

- **Alternative representations for** Π - Our query checking methodology first converts data streams into finite automata. These automata are linear, representing only a single execution. Solving the query checking problem is then done on a per-automaton basis and a consensus is taken before reporting an overall solution.

  Instead, one could consider exploring an alternative approach where, after computing the individual PNFA B_{π_i}, one produces a composed automaton B_c formed from all of the individual automata. Such an approach would only require solving a single query checking problem rather than one per stream. There are a number of ways to perform the composition, we leave as future work exploring the impact adopting such an approach would have.

- **Alternative satisfaction for** $\mathrm{QC}(\Pi, \phi[\mathtt{var}])$ - We have explored the query checking problem under finite semantics with both a single data stream and multiple data streams. For the latter case, there are a number of ways one could choose to define a set of streams to satisfy a formula. In Chapter 7, we have required that all streams in the set must satisfy the solution to the query checking problem. One could consider having a looser requirement, perhaps with a fractional cutoff (e.g. a simple majority) where at least some specified portion of the streams satisfy the proposed solution. Such a property

could be expressed as a partial invariant of the system using some form of noisy LTL, for example. We leave as future work the exploration of these alternative definitions for satisfying the query checking problem over multiple data streams.

9.2.5 Chapter 8 Future Work

- **Noisy LTL operators as discrimination functions** - First, the idea of partial invariants bears resemblance to decision points made during during decision trees. One could possibly explore the benefits of allowing noisy LTL formulas as discriminant functions used in building a decision tree during a learning process for improved accuracy.

- **Expand decidability for sliding window semantics** - At present, decidability for sliding window semantics of noisy LTL is known only in cases where the noised subformula is a propositional formula (no temporal modalities). The current challenge lies in the notion of composition: the Büchi representing the de-noised formula expects a stream of alphabet symbols, while the De Bruijn graph expects a stream of (in the general case) noisy LTL properties. This may be attainable with some application of fixpoint theory by applying some transformation on-the-fly to the stream of alphabet symbols. As future work, one could seek to expand this to support a larger fragment of noisy LTL (or full noisy LTL).

9.3 Final Remarks

In summary, the work presented in this document explored different aspects and approaches to reason and learn about a system when taking real world limitations such as finiteness or noise into consideration. This was done either by studying the system directly or through observations made from its executions. We believe that through formal investigation of a system's data streams, we are able to effectively transfer that knowledge to the underlying system.

Bibliography

[1] Chris Ackermann, Rance Cleaveland, Samuel Huang, Arnab Ray, Charles Shelton, and Elizabeth Latronico. Automatic requirement extraction from test cases. In Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Roşu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification*, pages 1–15, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[2] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[3] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logics of Programs, Workshop, Yorktown Heights, New York, May 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[4] Sumit Gulwani. Dimensions in program synthesis. In Roderick Bloem and Natasha Sharygina, editors, *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, page 1. IEEE, 2010.

[5] Sumit Gulwani. Synthesis from examples: Interaction models and algorithms. In Andrei Voronkov, Viorel Negru, Tetsuo Ida, Tudor Jebelean, Dana Petcu, Stephen M. Watt, and Daniela Zaharie, editors, *14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2012, Timisoara, Romania, September 26-29, 2012*, pages 8–14. IEEE Computer Society, 2012.

[6] James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983.

[7] Grigore Rosu and Saddek Bensalem. Allen linear (interval) temporal logic - translation to LTL and monitor synthesis. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2006.

[8] Christian Freksa. Temporal reasoning based on semi-intervals. *Artif. Intell.*, 54(1):199–227, 1992.

[9] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. *SIGMOD Rec.*, 22(2):207–216, June 1993.

[10] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proceedings of the Eleventh International Conference on Data Engineering*, ICDE '95, pages 3–14, Washington, DC, USA, 1995. IEEE Computer Society.

[11] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pages 1–12. ACM, 2000.

[12] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. New algorithms for fast discovery of association rules. In David Heckerman, Heikki Mannila, and Daryl Pregibon, editors, *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining (KDD-97), Newport Beach, California, USA, August 14-17, 1997*, pages 283–286. AAAI Press, 1997.

[13] Mohammed Javeed Zaki. Scalable algorithms for association mining. *IEEE Trans. Knowl. Data Eng.*, 12(3):372–390, 2000.

[14] Geoffrey I. Webb. Efficient search for association rules. In Raghu Ramakrishnan, Salvatore J. Stolfo, Roberto J. Bayardo, and Ismail Parsa, editors, *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, Boston, MA, USA, August 20-23, 2000*, pages 99–107. ACM, 2000.

[15] Geoffrey I. Webb. OPUS: an efficient admissible algorithm for unordered search. *J. Artif. Intell. Res. (JAIR)*, 3:431–465, 1995.

[16] William Chan. Temporal-logic queries. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 450–463. Springer, 2000.

[17] Arie Gurfinkel, Benet Devereux, and Marsha Chechik. Model exploration with temporal logic query checking. In *Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering 2002, Charleston, South Carolina, USA, November 18-22, 2002*, pages 139–148. ACM, 2002.

[18] Glenn Bruns and Patrice Godefroid. Temporal logic query checking. In *16th Annual IEEE Symposium on Logic in Computer Science, LICS 2001*, pages 409–417. IEEE, June 2001.

[19] Arie Gurfinkel, Marsha Chechik, and Benet Devereux. Temporal logic query checking: A tool for model exploration. *IEEE Transactions on Software Engineering*, 29(10):898–914, 2003.

[20] Dezhuang Zhang and Rance Cleaveland. Efficient temporal-logic query checking for Presburger systems. In *20th IEEE/ACM International Conference on Automated Software Engineering, ASE 2005*, pages 24–33, Long Beach, CA, USA, 2005. ACM.

[21] Hana Chockler, Arie Gurfinkel, and Ofer Strichman. Variants of LTL query checking. In Sharon Barner, Ian Harris, Daniel Kroening, and Orna Raz, editors, *6th International Haifa Verification Conference, HVC 2010*, volume 6504 of *Lecture Notes in Computer Science*, pages 76–92. Springer Verlag, October 2010.

[22] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[23] Caroline Lemieux, D Park, and I Beschastnikh. General ltl specification mining. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[24] Wil M. P. van der Aalst, H. T. de Beer, and Boudewijn F. van Dongen. Process mining and verification of properties: An approach based on temporal logic. In Robert Meersman, Zahir Tari, Mohand-Said Hacid, John Mylopoulos, Barbara Pernici, Özalp Babaoglu, Hans-Arno Jacobsen, Joseph P. Loyall, Michael Kifer, and Stefano Spaccapietra, editors, *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE, OTM Confederated International Conferences CoopIS, DOA, and ODBASE 2005, Agia Napa, Cyprus, October 31 - November 4, 2005, Proceedings, Part I*, volume 3760 of *Lecture Notes in Computer Science*, pages 130–147. Springer, 2005.

[25] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors,

Proc. 23rd International Conference on Computer Aided Verification (CAV'11), volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[26] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[27] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous time Markov chains. In R. Alur and T. Henzinger, editors, *Proc. 8th International Conference on Computer Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 269–276. Springer, 1996.

[28] Orna Raz, Philip Koopman, and Mary Shaw. Semantic anomaly detection in online data sources. In Will Tracz, Michal Young, and Jeff Magee, editors, *Proceedings of the 22rd International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pages 302–312. ACM, 2002.

[29] Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, pages 854–860. AAAI Press, 2013.

[30] Giuseppe De Giacomo, Riccardo De Masellis, and Marco Montali. Reasoning on ltl on finite traces: Insensitivity to infiniteness. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, AAAI'14, pages 1027–1033. AAAI Press, 2014.

[31] Jianwen Li, Lijun Zhang, Geguang Pu, Moshe Y. Vardi, and Jifeng He. Ltlf satisfiability checking. *CoRR*, abs/1403.1666, 2014.

[32] Grigore Roşu. Finite-trace linear temporal logic: Coinductive completeness. In Yliès Falcone and César Sánchez, editors, *Runtime Verification*, pages 333–350, Cham, 2016. Springer International Publishing.

[33] Valeria Fionda and Gianluigi Greco. The complexity of ltl on finite traces: Hard and easy fragments. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 971–977. AAAI Press, 2016.

[34] Jianwen Li, Moshe Y. Vardi, and Kristin Y. Rozier. Satisfiability checking for mission-time ltl. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 3–22, Cham, 2019. Springer International Publishing.

[35] Houssam Abbas, Georgios Fainekos, Sriram Sankaranarayanan, Franjo Ivančić, and Aarti Gupta. Probabilistic temporal logic falsification of cyber-physical systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2s):95, 2013.

[36] Paulo Tabuada and Daniel Neider. Robust linear temporal logic. *CoRR*, abs/1510.08970, 2015.

[37] Philippe Schnoebelen. The complexity of temporal logic model checking. In Philippe Balbiani, Nobu-Yuki Suzuki, Frank Wolter, and Michael Zakharyaschev, editors, *Advances in Modal Logic 4, papers from the fourth conference on "Advances in Modal logic," held in Toulouse (France) in October 2002*, pages 393–436. King's College Publications, 2002.

[38] E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, 1986.

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

[40] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In Mary S. Van Deusen, Zvi Galil, and Brian K. Reid, editors, *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, USA, January 1985*, pages 97–107. ACM Press, 1985.

[41] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986*, pages 332–344. IEEE Computer Society, 1986.

[42] Moshe Y Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for concurrency*, pages 238–266. Springer, 1996.

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

[44] Tomáš Babiak, Mojmír Křetínský, Vojtěch Řehák, and Jan Strejček. Ltl to büchi automata translation: Fast and more deterministic. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 18th International Conference, TACAS 2012*, volume 7214 of *Lecture Notes in Computer Science*, pages 95–109, Tallinn, Estonia, March 2012. Springer Verlag.

[45] Kousha Etessami and Gerard J. Holzmann. Optimizing Büchi automata. In Catuscia Palamidessi, editor, *CONCUR 2000 — Concurrency Theory: 11th International Conference*, volume 1877 of *Lecture Notes in Computer Science*, pages 153–168, University Park, PA, USA, August 2000. Springer Verlag.

[46] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification: 13th International Conference, CAV 2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65, Paris, France, July 2001. Springer Verlag.

[47] Dimitra Giannakopoulou and Flavio Lerda. From states to transitions: Improving translation of LTL formulae to Büchi automata. In Doron A. Peled and Moshe Y. Vardi, editors, *Formal Techniques for Networked and Distributed Sytems — FORTE 2002: 22nd IFIP WG 6.1 International Conference*, volume 2529 of *Lecture Notes in Computer Science*, pages 308–326, Houston, TX, USA, November 2002. Springer Verlag.

[48] E. Allen Emerson. Automata, tableaux and temporal logics (extended abstract). In Rohit Parikh, editor, *Logics of Programs, Conference, Brooklyn College, June 17-19, 1985, Proceedings*, volume 193 of *Lecture Notes in Computer Science*, pages 79–88. Springer, 1985.

[49] E. Allen Emerson and Chin-Laung Lei. Modalities for model checking: Branching time logic strikes back. *Sci. Comput. Program.*, 8(3):275–306, 1987.

[50] Orna Kupferman, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. *J. ACM*, 47(2):312–360, 2000.

[51] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[52] André Arnold and Paul Crubillé. A linear algorithm to solve fixed-point equations on transition systems. *Inf. Process. Lett.*, 29(2):57–66, 1988.

[53] D. Bhalodiya, K.M. Patel, and C. Patel. An efficient way to find frequent pattern with dynamic programming approach. In *Engineering (NUiCONE), 2013 Nirma University International Conference on*, pages 1–5, Nov 2013.

[54] Orna Raz. *Helping everyday users find anomalies in data feeds*. PhD thesis, Pittsburgh, PA, USA, 2004. Chair-Shaw, Mary.

[55] Chris Ackermann, Arnab Ray, Rance Cleaveland, Juergen Heit, Charles Shelton, and Chris Martin. Model-based design verification: A monitor based approach. In *Society of Automotive Engineers*, 2008.

[56] Robert N. Charette. This car runs on code. `http://www.spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code`, February 2009.

[57] Geoffrey I. Webb. Discovering significant patterns. *Mach. Learn.*, 68(1):1–33, 2007.

[58] L. Hamers, Y. Hemeryck, G. Herweyers, M. Janssen, H. Keters, R. Rousseau, and A. Vanhoutte. Similarity measures in scientometric research: the jaccard index versus salton's cosine formula. *Inf. Process. Manage.*, 25(3):315–318, 1989.

[59] H. Charles Romesburg. *Cluster Analysis for Researchers*. Lulu Press, North Carolina, USA, 2004.

[60] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. *SIGPLAN Not.*, 37(1):4–16, 2002.

[61] A. W. Biermann and J. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Computers*, pages 592–597, 1972.

[62] Mark Gabel and Zhendong Su. Symbolic mining of temporal specifications. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 51–60, New York, NY, USA, 2008. ACM.

[63] Claire Goues and Westley Weimer. Specification mining with few false positives. In *TACAS '09: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 292–306, Berlin, Heidelberg, 2009. Springer-Verlag.

[64] Murali K. Ramanathan, Ananth Grama, and Suresh Jagannathan. Static specification inference using predicate mining. *SIGPLAN Not.*, 42(6):123–134, June 2007.

[65] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 174–184, New York, NY, USA, 2007. ACM.

[66] Westley Weimer and George C Necula. Mining temporal specifications for error detection. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 461–476. Springer, 2005.

[67] C. C. Michael and Anup Ghosh. Using finite automata to mine execution data for intrusion detection: a preliminary report. In *In Recent Advances in Intrusion Detection (RAID)*, pages 66–79, 2000.

[68] Anand Raman, Jon Patrick, and Palmerston North. The sk-strings method for inferring pfsa. In *Proceedings of the Workshop on Automata Induction, Grammatical Inference and Language Acquisition at the 14th International Conference on Machine Learning (ICML97)*, 1997.

[69] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[70] Ian H. Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques.* Morgan Kaufmann, San Francisco, 2nd edition edition, 2005.

[71] Sudheendra Hangal, Naveen Chandra, Sridhar Narayanan, and Sandeep Chakravorty. Iodine: a tool to automatically infer dynamic invariants for hardware designs. In *DAC '05: Proceedings of the 42nd annual Design Automation Conference*, pages 775–778, New York, NY, USA, 2005. ACM.

[72] Ali Mesbah and Arie van Deursen. Invariant-based automatic testing of ajax user interfaces. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 210–220, Washington, DC, USA, 2009. IEEE Computer Society.

[73] Xueqi Cheng and Michael S. Hsiao. Simulation-directed invariant mining for software verification. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 682–687, New York, NY, USA, 2008. ACM.

[74] E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B)*, pages 995–1072. MIT Press, 1990.

[75] Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT Press, 2008.

[76] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking.* MIT Press, 1999.

[77] Chris Ackermann, Rance Cleaveland, Samuel Huang, Arnab Ray, Charles Shelton, and Elizabeth Latronico. Automatic requirement extraction from test cases. In Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Roşu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification: First International Conference, RV 2010*, volume 6418 of *Lecture Notes in Computer Science*, pages 1–15, St. Julians, Malta, November 2010. Springer Verlag.

[78] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007. Special issue on Experimental Software and Toolkits.

[79] Wenchao Li, Alessandro Forin, and Sanjit A. Seshia. Scalable specification mining for verification and diagnosis. In *47th Design Automation Conference, DAC 2010*, pages 755–760, Anaheim, CA, USA, June 2010. ACM.

[80] Sharon Shoham, Eran Yahav, Stephen J. Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. *IEEE Transactions on Software Engineering*, 34(5):651–666, 2008.

[81] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "not never" revisited: On branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.

[82] Roy Armoni, Limor Fix, Alon Flaisher, Rob Gerth, Boris Ginsburg, Tomer Kanza, Avner Landver, Sela Mador-Haim, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi, and Yael Zbar. The forspec temporal logic: A new temporal property-specification language. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 8th International Conference, TACAS 2002*, volume 2280 of *Lecture Notes in Computer Science*, pages 296–311, Grenoble, France, April 2002. Springer Verlag.

[83] Kirsten Winter. Model checking for abstract state machines. *Journal of Universal Computer Science*, 3(5):689–701, 1997.

[84] Samuel Huang and Rance Cleaveland. Query checking for linear temporal logic. In Laure Petrucci, Cristina Seceleanu, and Ana Cavalcanti, editors, *Critical Systems: Formal Methods and Automated Verification*, pages 34–48, Cham, 2017. Springer International Publishing.

[85] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, Oct 1977.

[86] Jean-Michel Couvreur. On-the-fly verification of linear temporal logic. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 — Formal Methods*, pages 253–271, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[87] Moshe Y Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331. IEEE Computer Society, 1986.

[88] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, volume 9938 of *Lecture Notes in Computer Science*, pages 122–129. Springer, October 2016.

[89] Alexandre Duret-Lutz. LTL translation improvements in Spot 1.0. *International Journal on Critical Computer-Based Systems*, 5(1/2):31–54, March 2014.

[90] Dmitriy Fradkin and Fabian Mörchen. Mining sequential patterns for classification. *Knowledge and Information Systems*, 45(3):731–749, Dec 2015.

[91] Kleanthi Georgala, Mohamed Ahmed Sherif, and Axel-Cyrille Ngonga Ngomo. An efficient approach for the generation of allen relations. In *Proceedings of the Twenty-second European Conference on Artificial Intelligence*, ECAI'16, pages 948–956, Amsterdam, The Netherlands, The Netherlands, 2016. IOS Press.

[92] Rudolf Ahlswede and Ning Cai. Incomparability and intersection properties of boolean interval lattices and chain posets. *European Journal of Combinatorics*, 17(8):677 – 687, 1996.

[93] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017.

[94] J. Cichon, A. Czubak, and A. Jasinski. Minimal büchi automata for certain classes of ltl formulas. In *2009 Fourth International Conference on Dependability of Computer Systems*, pages 17–24, June 2009.

[95] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 311–322, New York, NY, USA, 2005. ACM.

[96] Guangxuan Song, Wenwen Qu, Xiaojie Liu, and Xiaoling Wang. Approximate calculation of window aggregate functions via global random sample. *Data Science and Engineering*, 3(1):40–51, Mar 2018.