Using Live Sequence Chart Specifications for Formal Verification

Rahul Kumar
Brigham Young University - Provo

Follow this and additional works at: https://scholarsarchive.byu.edu/etd
Part of the Computer Sciences Commons

BYU ScholarsArchive Citation
https://scholarsarchive.byu.edu/etd/1500

This Dissertation is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in All Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.
USING LIVE SEQUENCE CHART SPECIFICATIONS FOR
FORMAL VERIFICATION OF SYSTEMS

by
Rahul Kumar

A dissertation submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science
Brigham Young University
December 2008
This dissertation has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

<table>
<thead>
<tr>
<th>Date</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Eric G Mercer, Chair</td>
</tr>
<tr>
<td></td>
<td>Michael D. Jones</td>
</tr>
<tr>
<td></td>
<td>Quinn O. Snell</td>
</tr>
<tr>
<td></td>
<td>Bryan S. Morse</td>
</tr>
<tr>
<td></td>
<td>David Embley</td>
</tr>
</tbody>
</table>
As chair of the candidate’s graduate committee, I have read the dissertation of Rahul Kumar in its final form and have found that 1. its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; 2. its illustrative materials including figures, tables, and charts are in place; and 3. the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

________________________
Eric G Mercer
Chair, Graduate Committee

Accepted for the Department

Date

________________________
Kent E. Seamons
Graduate Coordinator

Accepted for the College

Date

________________________
Thomas W. Sederberg
Associate Dean, College of Physical and Mathematical Sciences
ABSTRACT

USING LIVE SEQUENCE CHART SPECIFICATIONS FOR
FORMAL VERIFICATION OF SYSTEMS

Rahul Kumar
Department of Computer Science
Doctor of Philosophy

Formal methods play an important part in the development as well as testing stages of software and hardware systems. A significant and often overlooked part of the process is the development of specifications and correctness requirements for the system under test. Traditionally, English has been used as the specification language, which has resulted in verbose and difficult to use specification documents that are usually abandoned during product development. This research focuses on investigating the use of Live Sequence Charts (LSCs), a graphical and intuitive language directly suited for expressing communication behaviors of a system as the specification language for a system under test. The research presents two methods for using LSCs as a specification language: first, by translating LSCs to temporal logic, and second, by translating LSCs to an automaton structure that is directly suited for formal verification of systems. The research first presents the translation for each method and further, identifies the pros and cons for each verification method.
I would like to thank Eric Mercer for his constant support, faith, and encouragement in every aspect of this entire process. This work would not have been possible without his infinite patience and guidance. Our long discussions on and off the track will always inspire and remind me of this glorious time. Annette Bunker has played an important role in inspiring this work with her own PhD work. Her willingness to help, provide information, and her support have been invaluable to the development of this work. I would also like to thank Mike Jones, who has been my crystal ball, always providing me with the right direction and guidance for all my technical proofs and presentations. His keen sense of story telling has improved my work by leaps and bounds.

The love and support displayed by my wife Jennifer has exceeded what notions of limits I had in my mind. She has been a pillar of strength for me during this entire process. My parents, Satish and Sunita, and my sister Arushi, have been my greatest source of inspiration and persistence. Without their decisions, hard work, and vision, this would have been impossible.

A special thanks goes to Neha Rungta, Joseph Edelman, and Tonglaga Bao for indulging me in the lab, and not killing me for my wonderful reactions.

I would like to thank the entire staff at the CS office. Their kindness and helpfulness can not be expressed by me in words. Thanks to them, I am not burning my way through a mountain of forms!
# Contents

Contents vii

List of Figures xiii

List of Tables xvii

1 Introduction 1

1.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.2 Structure of Document . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

1.2.1 Publications . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2 An Introduction to Live Sequence Charts 7

2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.2 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

2.3 Live Sequence Charts Constructs . . . . . . . . . . . . . . . . . . . . . . . 10

2.3.1 Processes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

2.3.2 Locations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

2.3.3 Messages . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

2.3.4 Coregions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

2.3.5 Simultaneous regions . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

2.3.6 Conditions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

2.3.7 Kleene star . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

2.3.8 Actions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

2.3.9 Prechart . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.10 Main chart</td>
<td>15</td>
</tr>
<tr>
<td>2.3.11 Temperatures</td>
<td>16</td>
</tr>
<tr>
<td>2.3.12 Subcharts</td>
<td>17</td>
</tr>
<tr>
<td>2.3.13 Hierarchical Charts</td>
<td>17</td>
</tr>
<tr>
<td>2.4 Specifying Systems Using LSCs</td>
<td>18</td>
</tr>
<tr>
<td>2.5 Trace Semantics of Live Sequence Charts</td>
<td>21</td>
</tr>
<tr>
<td>2.5.1 LSC to Automaton Translation</td>
<td>26</td>
</tr>
<tr>
<td>2.6 Conclusions</td>
<td>28</td>
</tr>
<tr>
<td><strong>3 Improving Translation of Live Sequence Charts to Temporal Logic</strong></td>
<td>31</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>32</td>
</tr>
<tr>
<td>3.2 Related Work</td>
<td>34</td>
</tr>
<tr>
<td>3.3 Live Sequence Charts</td>
<td>36</td>
</tr>
<tr>
<td>3.4 Quadratic Translation of LSCs to LTL</td>
<td>40</td>
</tr>
<tr>
<td>3.5 Temporal Logic Reductions</td>
<td>44</td>
</tr>
<tr>
<td>3.5.1 Reducing Ordering Properties</td>
<td>44</td>
</tr>
<tr>
<td>3.5.2 Reducing Uniqueness Properties</td>
<td>46</td>
</tr>
<tr>
<td>3.6 Improved Translation of LSCs to LTL</td>
<td>46</td>
</tr>
<tr>
<td>3.7 Translating Additional Constructs</td>
<td>48</td>
</tr>
<tr>
<td>3.8 Analysis</td>
<td>49</td>
</tr>
<tr>
<td>3.9 Results</td>
<td>50</td>
</tr>
<tr>
<td>3.10 Conclusions and Future Work</td>
<td>54</td>
</tr>
<tr>
<td><strong>4 Specifying Multi-Agent Systems Using Live Sequence Charts</strong></td>
<td>55</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>56</td>
</tr>
<tr>
<td>4.2 Related Work</td>
<td>58</td>
</tr>
<tr>
<td>4.3 Live Sequence Charts</td>
<td>59</td>
</tr>
<tr>
<td>4.4 Live Sequence Charts and Multi-Agent Systems</td>
<td>62</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Translating Live Sequence Charts to Temporal Logic</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Knowledge Properties in Live Sequence Charts</td>
</tr>
<tr>
<td>4.5</td>
<td>Case Study: Auction Protocol</td>
</tr>
<tr>
<td>4.6</td>
<td>Embedding Arbitrary Temporal Logic Properties in LSCs</td>
</tr>
<tr>
<td>4.7</td>
<td>Conclusions and Future Work</td>
</tr>
<tr>
<td>5</td>
<td>Improving Live Sequence Chart to Automata Transformation for Verification</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>5.2</td>
<td>LSC Overview</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Transforming Live Sequence Charts to Automata</td>
</tr>
<tr>
<td>5.3</td>
<td>Transformation and Verification Example</td>
</tr>
<tr>
<td>5.4</td>
<td>Transformation and Verification Details</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Verification Approach</td>
</tr>
<tr>
<td>5.5</td>
<td>Analysis</td>
</tr>
<tr>
<td>5.6</td>
<td>Results</td>
</tr>
<tr>
<td>5.7</td>
<td>Conclusions and Future Work</td>
</tr>
<tr>
<td>6</td>
<td>Verifying Communication Protocols Using Live Sequence Chart Specifications</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>6.2</td>
<td>Related Work</td>
</tr>
<tr>
<td>6.3</td>
<td>Live Sequence Charts</td>
</tr>
<tr>
<td>6.4</td>
<td>LSC to Automaton Translation</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Kleene Star</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Subcharts</td>
</tr>
<tr>
<td>6.4.3</td>
<td>Hierarchical Charts</td>
</tr>
<tr>
<td>6.5</td>
<td>Case Study: BVCI Protocol Verification</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 An overview of the model checking process. .......................... 2
1.2 An overview of the research performed in this work. ................. 4

2.1 Example LSC showing a subset of the available constructs. ........... 11
2.2 Example of an LSC process, life-line and locations. .................. 11
2.3 Example of synchronous and asynchronous messages. .................. 12
2.4 Example of a coregion in LSCs. ...................................... 13
2.5 Example of a simultaneous region in an LSC. .......................... 13
2.6 Example of conditions in LSCs. ........................................ 14
2.7 Example of a message with a Kleene star and plus operator. ........... 14
2.8 LSC with action. .................................................................. 15
2.9 Example of universal and existential charts in LSCs. .................... 15
2.10 Temperatures on messages, conditions, and locations in LSCs. ....... 16
2.11 Example subchart in an LSC. ............................................. 18
2.12 Example subchart in an LSC. ............................................. 18
2.13 Hierarchical charts showing sequential composition and choice. ...... 19
2.14 An example LSC containing a universal chart. ......................... 19
2.15 An example LSC containing a universal chart. ......................... 20
2.16 Function that returns letter given a location. .......................... 23
2.17 Algorithm for building a basic automaton from an LSC. ............... 27
2.18 Example of automaton generated by unwinding algorithm. .......... 28

3.1 The process of using scenario-based specifications to verify systems. .. 33
3.2 A LSC illustrating most of the language features supported in this work with a lattice defining the partial order on messages in the chart. 37

3.3 Function that returns letter given a location. 39

4.1 Example LSC describing the auction protocol. 63

4.2 Embedding the knowledge operator $K$ in LSCs. 71

4.3 LSC describing the auction protocol with an alliance. 73

4.4 LSC describing the alliance creation in the auction protocol. 74

5.1 An example specification describing the interaction between a cluster node, database, and a job scheduler. 82

5.2 Initial and transformed automaton using transformation algorithm. 88

5.3 A generic state in the transformed automaton. 90

5.4 Algorithm for building a negated automaton from an input LSC automaton. 98

6.1 Example LSC showing the Normal Request scenario. 104

6.2 Generic state of the LSC to automaton translation. 111

6.3 Example of translating a message with Kleene star. 114

6.4 Translation of the plus operator on a message. 115

6.5 Example of translating an LSC with multiple subcharts. 116

6.6 Example LSC and corresponding automaton for an LSC containing a subchart with a precondition and Kleene star. 118

6.7 Translating hierarchical charts to automaton. 119

A.1 Expressibility between between CTL*, CTL, and LTL. 134

B.1 Definition of the $\models$ relations for testing if a system implements an LSC. 136

B.2 An LSC illustrating asynchronous messages and the corresponding lattice depicting the partial order of events. 141
B.3 An LSC illustrating a coregion and the corresponding lattice depicting the partial order of events. 142

B.4 An LSC illustrating a bonded condition in a simultaneous region and the corresponding lattice depicting the partial order of events. 143
List of Tables

3.1 Properties generated for the left hand side of the implication ........................................... 42
3.2 Properties generated for the right hand side of the implication ........................................... 43
3.3 Verification results using SPIN. .................................................................................................. 52
3.4 Verification results using NuSMV. ............................................................................................ 53

5.1 Results for the traditional and improved verification approaches using NuSMV. ......................... 96
5.2 Results for the improved verification approach using SPIN. .................................................... 96

6.1 Verification results for Promela models against the BVCI protocol LSC specification. ........................................ 121
Chapter 1

Introduction

1.1 Overview

Due to the rapid growth in the complexity of software and hardware systems, the need for effective and more importantly complete testing methodologies has gained much interest. This need is even greater when dealing with systems such as space shuttles, rockets, cars, etc., where safety of the system in all possible situations is critical. Given such a need for effective and complete testing methods, significant research has been performed in the area of formal verification.

Formal verification methods, such as model checking, provide a methodology for exploring all possible behaviors of a system and comparing them to a specification that describes the expected or allowed behaviors of the system [18]. Due to the exhaustive and formal nature of model checking, model checking applications have shown great promise for analyzing and debugging systems, especially in situations where traditional testing and analysis techniques have failed to provide conclusive results [18,15]. The system (software or hardware) is usually described in a modeling language and can be used as input to a model checker. Given a system and a specification, the model checker enumerates and compares every possible behavior of the system against the specification. If the system exhibits any behaviors that violate the specification, an error and counter examples are reported as the verification result.
On the other hand, if the system satisfies the specification, the model checker reports no errors. Figure 1.1 shows an overview of the model checking process.

The specification of a system is usually extracted from a correctness requirements document. Commonly used languages for the specification are temporal logic, automata, and formulas in first order logic. One important and often overlooked part of the development process is the use of an effective language for describing the correctness requirements of a system. Often, the correctness requirements created during the early stages of development are in English. These specification documents tend to be very verbose, cumbersome, and informal. Consequently, two problems often arise. First, the correctness requirements developed during the initial stages of product development are abandoned during verification or second, extracting formal specifications from the correctness requirements becomes a manual and resource intense with significant room for error. In either outcome the formal verification of the system under test remains a challenge for the verification engineers and system developers.

Graphical languages solve the problem of unusable requirements and specifications in an elegant and formal manner. Graphical languages are intuitive, precise, and lend themselves to describing complex systems in a concise manner. Additionally, the formal nature of graphical languages greatly simplifies formal verification tasks.
such as temporal logic property generation and test vector generation, which greatly aids verifiers and system developers in all stages of the product development cycle.

Live Sequence Charts (LSCs) are a scenario-based graphical language that have been used to specify a variety of systems such as protocols, radio communications, and user interfaces [19,20,5,14]. They are extremely well suited to describing message interactions between multiple processes/agents in a system. The LSC grammar also supports the specification of control structures (if then statements and while loops) and provides a distinction between provisional/mandatory behaviors of the system.

Past research in the area of using LSCs in a formal verification environment has identified three major problems related to the performance, expressiveness, and formal semantics of LSC verification that directly affect the applicability and effectiveness of LSCs in formal verification. First, much of the work has focused on using only a subset of the LSC grammar, thus limiting the applicability of the verification approach to a limited set of systems [30]. Second, several of the verification strategies incorporating LSCs have failed to provide a mathematically rigorous framework to establish a definitive and formal relationship between the system and the specification [1,14,11]. Finally, the techniques that do provide a conclusive relationship between a system and a specification do so at a large performance cost [31].

The focus of this research is to use LSCs in a formal verification environment more efficiently by solving problems encountered in past research. We present two methods for using LSCs as a specification language in a formal development environment. Each technique establishes a formal and definitive relationship of the system relative to the specification and improves on previous approaches by reducing the complexity of the LSC verification task. The first method translates LSCs to temporal logic formulas that can be used as input to a model checker for verification of the system, and the second method presents an LSC to automaton translation that
results in an automaton structure directly suited for verification of a system using standard model checking algorithms.

Figure 1.2 is an overview of the verification strategy using LSCs as the specification language in a formal verification environment. An LSC specification is translated to either temporal logic (part one) or automata (part two). The respective temporal logic formula or automata is provided as the specification input to a model checker along with the system under test. The model checker either reports an error along with a counterexample trace or reports no errors. The shaded part in the figure highlights the research presented in this thesis.

1.2 Structure of Document

The structure of this document is based around five papers directly related to the research area. Chapter 2 presents an overview of the LSC language and semantics. Appendix A provides a brief overview of temporal logic. Chapter 3 presents the LSC to temporal logic translation. Proofs and details for the LSC to tempo-
ral logic translation can be found in Appendix B. Chapter 4 extends the translation of LSCs to multi-agent systems temporal logics and presents expressibility results of LSCs and temporal logic. Corresponding proofs can be found in Appendix C. Chapter 5 presents the LSC to automaton translation and Appendix D provides the supporting results and proofs for the LSC to automata translation [38]. Chapter 6 extends the translation of LSCs to automata for additional constructs of LSCs and Appendix E presents the supporting proofs and algorithms. Finally, Chapter 7 discusses the conclusions and future work for this research.

1.2.1 Publications

The following work has been published from this research:


Chapter 2

An Introduction to Live Sequence Charts

Abstract

This paper provides an overview of Live Sequence Charts (LSCs) and their usage as a specification language. LSCs are a scenario-based graphical language that build on previous scenario-based languages such as Message Sequence Charts (MSCs). LSCs, due to their intuitive and graphical nature have found significant applications in protocol and communication systems development. Their use has been investigated not only as a modeling language during initial stages of protocol development but also as a specification language for a system under test. We first present an introduction to the various constructs in the LSC grammar and then present a case analysis of using LSCs as a specification language. Further, the paper also presents details of the trace semantics of LSCs and presents the LSC to automaton translation algorithm.
2.1 Introduction

Formal methods play an important role in the development and testing stages of a product. A significant part of the process is the development of intuitive and usable specifications, or correctness requirements, that define the expected behavior of the system under test. English is the primary language used to describe specifications and requirements. The specification is further translated (manually) to temporal logics, or automata based structures that can be used in a formal verification environment to test the respective system. A major drawback of this development strategy is the significant amount of human resources and effort required to create a specification that is usable for formal verification.

Graphical languages are gaining popularity in specifying systems. The primary reason for their growing popularity is the intuitive and easy-to-use paradigm provided by the visual aspect of the language, which significantly reduces the time required for developing and testing specifications. Additionally, the formal nature of graphical languages greatly simplifies common design tasks such as model synthesis, temporal logic property generation, and test vector generation, which enables their use throughout the development cycle of a system. Finally, the precise nature of the languages along with the relatively short learning curve is a significant advantage over traditional specification languages that require a significant level of comfort and expertise to specify systems.

In this paper we focus on Live Sequence Charts (LSCs), a scenario based graphical language [30, 10, 19]. LSCs are an extension of Message Sequence Charts (MSCs) [29], which are a variant of the more popular UML sequence diagrams [21]. LSCs extend the MSC language by primarily adding the ability to express provisional behavior that distinguishes between mandatory and optional behaviors of a system. LSCs also extend the MSCs language by providing activation, multiplicity, and hierarchical constructs that allow the creation of specifications that are much larger and
more complex. We choose LSCs as our graphical language because they can be used to specify and express a wide range of behaviors.

The structure of the paper is as follows. Section 2.2 presents an overview of graphical languages and related work in the area of LSCs. Section 2.3 presents an example of an LSC and discusses the various constructs available in the LSC grammar in. Section 2.4 presents an example of using LSCs as a specification language for a handshake protocol, and Section 2.5 provides the formal trace semantics of LSCs along with the unwinding algorithm to translate LSCs to automaton. Finally, Section 2.6 presents the conclusions.

2.2 Related Work

MSCs and LSCs are examples of scenario-based languages that describe the interaction between different agents of a system [29,10,19]. The behaviors of the agents are partitioned into separate scenarios that are unique with respect to each other. Other graphical languages such as Interface Diagrams (IDs) and Sequence Diagrams (SDs) in UML are used to specify the interaction between agents in a system, but are not as powerful as LSCs [21]. Timing Diagrams (TDs) have also been used to describe the behavior of multiple agents in a system but are highly specialized for describing hardware systems [2,16]. Multiple variations of TDs, such as Constraint Diagrams and Symbolic Timing Diagrams build on TDs to increase their expressiveness. Statecharts are extended state machines that allow annotations for expressing concurrency, hierarchy and communication of a system and have been successfully integrated into UML for designing large systems with multiple hierarchical levels [23].

Significant research has been done in the area of using LSCs as a graphical language for the development and specification of systems. Work presented in [47, 1] uses the LSC specification as a model for verifying requirements of the system. Another approach to using LSCs is the automatic synthesis of systems from the
LSC specification [24, 25]. This approach has proven difficult due to the inherent limitations of specifying data aspects of a system using LSCs. A large portion of the research has investigated the use of LSCs in verifying systems by translating them to temporal logics and exploring the expressibility of LSCs relative to temporal logics [14, 49, 35]. The primary drawback of these approaches is the high complexity of verification as well as the lack of support for the complete LSC grammar. Other work has also investigated the translation of LSCs to automata for automatic verification of systems [31, 30]. In this paper we summarize the translation of LSCs to automata.

2.3 Live Sequence Charts Constructs

We now present an overview of the various constructs of LSCs along with examples of each construct. Following the presentation of the individual constructs, we present examples of LSCs that are used to specify a communication handshake protocol and the translation of LSCs to automata. The following description of LSC constructs and LSCs is adapted from [10, 19, 31]. Figure 2.1 shows an example LSC that describes the interaction between a User, a Controller, a Engine, and a Display. Whenever the User provides input to the system by pressing button X, the controller requests an update from the Engine, which in turn updates the Display and sends an acknowledgment back to the Controller. The behaviors enclosed in the dashed hexagon specify the activating scenario of the chart, and the behaviors described in the solid rectangle describe the main chart of the LSC. We use this LSC as our running example during the description of the various constructs of LSCs.

2.3.1 Processes

Processes are drawn with rectangular instance heads that denote the start of the processes. A vertical line originating from the instance head signifies the life-line of the process and ends in a filled rectangle, which terminates the respective process. Fig-
Figure 2.1: Example LSC showing a subset of the available constructs.

Figure 2.2: Example of an LSC process, life-line and locations.

Figure 2.2 shows an example of an instance head with a life-line for the process A. In the example LSC of Figure 2.1 there are four processes: User, Controller, Display, and Engine.

2.3.2 Locations

The life-line of each process is marked with locations that are points where events and other constructs may be described. Locations are unique to each process and start at number 0. For each new event or construct placed on the process life-line, the location number is incremented for the respective process. Figure 2.2 shows the two locations of interest marked by a filled circle. At each location a message is being sent to another instance in the LSC (not shown).
2.3.3 Messages

Messages are the only form of communication between processes in the LSC. Each message has a sender and receiver process attached to it. Messages are annotated with labels that identify the message. Messages can be *simultaneous* or *asynchronous*. Simultaneous messages are drawn with a solid arrow head and occur instantaneously when both the sender and receiver are ready for the communication. Each process is blocked until both the sender and receiver are ready for the communication to occur. Asynchronous messages are drawn with an open arrow head and can be received any time after the send event has occurred. Only the receiver of the message blocks for the communication to occur. It should be noted that the send event is forced to occur before the receive event. Additionally, all messages are guaranteed to be delivered unless specified otherwise using temperatures (discussed in Section 2.3.11). Figure 2.3 shows an example of a synchronous (*syn*) and asynchronous (*asyn*) message between two processes in an LSC. All messages in the LSC shown in Figure 2.1 are simultaneous messages.

2.3.4 Coregions

Coregions are drawn with a dashed vertical line next to the life-line of a process. They describe behavior that can occur in any order. Figure 2.4 shows an example of a coregion with two messages *A* and *B*. The messages can be observed in any order.

Figure 2.3: Example of synchronous and asynchronous messages.
2.3.5 Simultaneous regions

Simultaneous regions describe events that have to occur at the exact same time. Dots are drawn on locations of the simultaneous. Figure 2.5 shows an example where messages $A$ and $B$ should occur simultaneously.

2.3.6 Conditions

Conditions are placed in the chart by drawing hexagons around the life-lines of processes evaluating the condition. The condition label describes a predicate that must be satisfied at the current location(s) of the process(es). Conditions spanning multiple process life-lines act as synchronizing points for the involved processes, and the condition is not evaluated unless all the processes are at the respective condition locations. Conditions attached to a message (using simultaneous regions) are called *bonded conditions*. Bonded conditions are evaluated at the same time as the occurrence of the message they are attached to. Conditions placed on their own location and not attached to a message in the chart are called *non-bonded* conditions [30]. Non-bonded conditions are evaluated continuously until they are satisfied. If the con-
Figure 2.6: Example of conditions in LSCs.

Figure 2.7: Example of a message with a Kleene star and plus operator.

dition is never satisfied, an error should be reported. Figure 2.6(a) shows an example of a bonded condition and Figure 2.6(b) shows an example of a non-bonded condition.

2.3.7 Kleene star

The Kleene star construct, ‘∗’, can be placed on charts or messages and is used to represent multiplicity where the associated construct can occur zero or more times (countably infinite). A variation of the Kleene star is the ‘+’ symbol that forces at least one (and allows more than one) occurrence of the associated construct. Figure 2.7 shows an example LSC with messages attached with a Kleene star and the plus operator.

2.3.8 Actions

The LSC language also provides the action construct that allows a process to perform an action on its local or global variables. For example, variables may be incremented, decremented or assigned a value at certain points on the life-line of a process. In the example LSC of Figure 2.8, the count++ action is performed by the Target process.
2.3.9 Prechart

The prechart is drawn with a dashed hexagon encompassing the instance heads and connects to the main body of the chart that is described in the solid rectangle following the prechart. The prechart describes the behavior of the system under which the main body of the chart is to be observed. A single condition or message in the prechart is equivalent to having an activation condition for the main chart.

2.3.10 Main chart

The main chart of the LSC is the behaviors described in the rectangle following by the prechart. The main chart can be either existential or universal. Universal charts, drawn with a solid rectangle, specify behavior that must be satisfied by the system every time the prechart is satisfied. Existential charts are drawn with a dashed rectangle and specify behavior that the system must exhibit at least once when the prechart is satisfied. Figure 2.9 shows examples of both universal and existential main charts. The main chart in the example LSC of Figure 2.1 is a universal chart.
Temperatures in LSCs can be assigned to messages, conditions, and locations. A *hot* temperature is drawn by using a solid line to draw the construct and specifies behavior that must be satisfied by the system. A *cold* temperature is drawn using a dashed line for the construct and specifies behavior that may be satisfied.

Figure 2.10(a) shows a hot and a cold message exchanged between the *Initiator* and *Target* processes. Hot messages must be observed in the system to satisfy the LSC specification, whereas a cold message may or may not be observed in the system. In the case of a cold message, the LSC waits indefinitely for the cold message to occur. Progress is only possible if the cold message is observed or the construct directly after the cold message is observed. In the latter case the LSC skips the cold message entirely and progresses to the immediately succeeding locations.

Locations can be assigned a hot or cold temperature as well. If the location of a process is hot, the process must progress off the hot location to the next location on its lifeline. A cold location on the other hand does not enforce progress to the next location on the lifeline of a process and is a legal termination/exit point for the respective process. For example, the LSC shown in Figure 2.1 has two cold locations of interest. The first corresponds to the *Engine* sending the *screenUpdated* message and the second corresponds to the *Controller* receiving the *screenUpdated* message; thus, it is not necessary that the *screenUpdated* message be observed in the system.
Temperatures on conditions are treated in conjunction with scope (discussed in Section 2.3.12). Bonded cold conditions do not affect the LSC execution. If the bonded cold condition is not satisfied, an error is not reported and the LSC exits the current scope to a higher scope. If no higher scope exists, the LSC exits completely. In the case of a non-bonded cold condition, the LSC waits indefinitely at the current location for the condition to be satisfied and can only exit the current scope if a construct at a higher scope is observed. It is not possible for the LSC to move to a location after the non-bonded cold condition within the same chart until the non-bonded cold condition is satisfied. If no higher scope exists, the LSC waits indefinitely for the non-bonded cold condition to be satisfied.

2.3.12 Subcharts

Subcharts are LSC charts that can be included within the body of a larger main chart. They are usually not preceded by a prechart. When a subchart $X$ is included within the main chart of $A$, chart $A$ is at a higher scope than subchart $X$. Figure 2.11 shows an example LSC that contains two subcharts $X$ and $Y$. The charts $X$ and $Y$ are at the same scope and the main chart is at a higher scope.

Subcharts in conjunction with conditions and Kleene stars can be used to create control and looping structures such as if-then and while blocks. Figure 2.12 shows example subcharts that represent (a) an if-then construct and (b) a while loop.

2.3.13 Hierarchical Charts

Hierarchical charts join LSCs together to create LSC specifications with control flow. Using hierarchical charts, multiple LSCs can be joined together using sequential composition or choice. Sequential composition of LSCs force the LSCs to execute in order. Choice is used to select one possible LSC from multiple future LSCs. Figure 2.13 shows examples of each type of combination. Figure 2.13(a) shows two LSCs
Figure 2.11: Example subchart in an LSC.

Figure 2.12: Example subchart in an LSC.

A and B joined together using sequential composition. Figure 2.13(b) shows three LSCs A, B, and C joined together by choice. After LSC A has completed execution successfully, either LSC B or C may be activated and executed.

2.4 Specifying Systems Using LSCs

An LSC specification can be comprised of multiple charts. As mentioned earlier, each chart in the LSC contains a prechart and a main chart, which can be either existential or universal. Messages, conditions, Kleene stars, coregions and simultaneous regions are included within the the main chart to describe the behaviors of the system. The
Figure 2.13: Hierarchical charts showing sequential composition and choice.

(a) (b)

Figure 2.14: An example LSC containing a universal chart.

main chart can also contain multiple subcharts that describe different aspects of the system within a scenario. For example, it may be the case that one of four possible behaviors is observed when a request is by process A to process B. In such a case, the prechart would contain the request message and the main chart would be comprised of four subcharts that describe the four possible behaviors possible after a request message. Additionally, multiple charts can be combined together using hierarchical charts as described earlier. We now show an example of specifying a communication handshake protocol using LSCs.

Figure 2.14 shows an LSC with two processes: Active and Passive. The LSC describes one scenario of the Three-Way Handshake Establishment Protocol (TWHEP) as described in [43]. The area enclosed by the dashed hexagon is the prechart. In this example, the prechart is satisfied when the Active process sends a SYN
message to Passive. Once a successful request has been made, the Passive process should send a SYNACK message back to Active to acknowledge the SYN message. At this point, the Active process acknowledges the SYNACK process by sending the ACK message to the Passive process. Finally, each process updates its local state, which is specified using actions. The Active process updates the state of the aEst variable to true and the Passive process updates the value of the pEst variable to true. Each local variable is used to detect the current status of the connection.

To account for data corruption and possible unreliabilities in the communication medium, the TWHEP protocol is extended to make use of sequence numbers with each message. This interaction is described in the chart shown in Figure 2.14. When the Active process sends a SYN message to the Passive process, it chooses a sequence number to do so. Passive records the sequence number of the SYN message and expects every subsequent message to have the correctly incremented sequence number. If Passive receives a message with an incorrect sequence number (inSeq
variable is true for correct sequence number) in an established state (aEst is true), Passive responds to Active by sending a RESET message. The prechart is satisfied when Passive receives an incorrect sequence number in an established state, which is checked using the local state variables (written as aEst ∧ ¬inSeq). If the condition is satisfied, Passive sends a RESET message to the Active process, which after receiving the RESET message restarts the handshake protocol by generating a new sequence number and sending the SYN message.

In addition to the two scenarios shown here, three other scenarios are required to completely describe the TWHEP protocol. These scenarios arise for the Active process receiving an out of order message in an established state, and one scenario each for the Active and Passive processes receiving an out of order message in an unestablished state. To conserve space, these scenarios have not been included.

2.5 Trace Semantics of Live Sequence Charts

We now present a formal semantics of LSCs. For simplicity and space purposes, the semantics only deal with a restricted set of the full LSC grammar.

We use the symbol c to denote an individual chart, such as the chart shown in Figure 2.14. The set of instance lines for a chart is given by $\text{inst}(c)$. An instance line has locations to track the state of the associated agent. For an instance line i and chart c, \( \text{dom}(c, i) = \{l_0, l_1, \ldots, l_{\text{max}}(i)\} \) is the set of locations for i in c with the first location (top most location for a process line) given by \( l_0 \) and the last location given by \( l_{\text{max}}(i) \) when moving from the top to the bottom of the instance line. As locations are not uniquely labeled in the chart across instance lines, the set of pairs \( \text{dom}(c) = \{(i, l) \mid i \in \text{inst}(c) \land l \in \text{dom}(c, i)\} \) represents all instance and location pairs in a chart c. The complete state of a chart is the state of the individual instances as denoted by their current location. The initial state has every instance in its first
location. The state of the chart evolves as instances move from one location to another down the chart.

The symbol $AP$ denotes the set of messages in the system. Communication is specified as a triple of the form $(\langle i, l \rangle, e, \langle i', l' \rangle)$, where $e \in AP$ is the message communicated from $\langle i, l \rangle$ to $\langle i', l' \rangle$. The set of all message communications for a chart is given by the relation $R(c)$. Well-formed charts are charts where the relation $R(c)$ is acyclic. This work, like [35], only considers well-formed charts.

The pre-chart begins at the first location, $l_0$, for each instance line. The main chart is the box below the pre-chart. A special location is reserved to denote the start of the main chart (end of the pre-chart). For each instance line, $i$, the last location, $l_{\text{max}}(i)$, marks the end of the main chart. As such, we require three unique locations in a chart that are not used by messages: the start of the pre-chart, the start of the main chart (end of the pre-chart), and the end of the main chart; all other locations must be used by messages (in this restricted case). We use $p$ to represent messages in the pre-chart, $m$ to represent messages in the main chart, and $e$ to represent any message in the chart regardless of its position.

A chart defines a partial order on its messages as its state changes on location transitions. To be specific, we introduce the symbols $\top$ (top), $\dashv$ (middle), and $\bot$ (bottom) to synchronize the agents when the chart starts, completes its pre-chart, and completes its main chart, respectively. The function shown in Figure 2.16 returns the letter that is communicated in a message or the symbol $\top$, $\dashv$, or $\bot$. For convenience, we define $msg(c)$, $msg_p(c)$ and $msg_m(c)$ to be the messages of the entire chart, the pre-chart, and the main chart respectively.

We define an order relation, $\triangleleft$, to capture the sequences of messages and symbols as specified in the instance lines when starting at the first locations and
\[
msg(c)(⟨i, l⟩) = \begin{cases} 
e & \text{if } \exists e, i', l' : (⟨i, l⟩, e, ⟨i', l'⟩) \in R(c) \lor (⟨i', l'⟩, e, ⟨i, l⟩) \in R(c) \\
\top & \text{if } l = l_0 \\
\bot & \text{if } l = l_{\text{max}}(i) \\
\vdash & \text{otherwise} 
\end{cases}
\]

Figure 2.16: Function that returns letter given a location.

traversing the lines to their last locations:

\[
\forall ⟨i, l_i⟩, ⟨i, l_{i+1}⟩ \in \text{dom}(c), \ msg(c)(⟨i, l_i⟩) \prec msg(c)(⟨i, l_{i+1}⟩).
\]

We do not explicitly describe the rules of moving from one state to another. Rather, we relate the message sequences as observed along each instance line in the order relation. In other words, the order relation describes the sequence of messages observed in the scope of the individual instance lines. The locations that communicate each message in the chart connect the sequences observed in one agent to those of the other agents. A partial order, \(<\), on messages and symbols is created from the order relation (\(\preceq\)) by adding to it the reflexive terms and then computing its transitive closure.

The \textbf{preset} (\(\bullet\)) of a location \(⟨i, l⟩\) is the set of locations of chart \(m\) that is smaller than \(⟨i, l⟩\):

\[
\bullet < i, l >= \{ < i', l' > \in \text{dom}(m) | < i', l' > \preceq_m < i, l > \}
\]

A \textbf{cut} through a chart, \(c\), is a set of locations \(c\), one for each instance in the chart, such that for every location in the chart, the preset \(\bullet < i, l >\) does not contain a location \(⟨i', l'⟩\) such that \(⟨j, l_j⟩ \prec ⟨i', l'⟩\) for some location \(⟨j, l_j⟩\) in \(c\).

\[
c = \{ < i_1, l_1 >, < i_2, l_2 >, \ldots, < i_n, l_n > \}
\]

Intuitively a \textbf{cut} can be acquired from an LSC by placing a string across the LSC such that it touches each process life-line in exactly one location. The limitation
on the cut set ensures that the cut set is legal and does not violate the natural partial
order induced by the instances in the chart. The initial cut $c_0$ is the cut where all
processes are at location 0:

$$c_0 = (<i_0, 0>, <i_1, 0>, \ldots, <i_n, 0>)$$

The set $\text{cuts}(m)$ for a chart $m$ is the set of all possible cuts. A cut $c' = (<i_0, l_1>, \ldots, <i_n, l_n>)$ is said to succeed a cut $c = (<i_0, l_i'>, \ldots, <i_n, l_n'>)$, if

$$\exists j, 0 \leq j \leq n, l_j' = l_j + 1 \land \forall i \neq j, l_i' = l_i.$$ 

A cut $c'$ succeeds a cut $c$, if the location of one instance progresses to its immediately successive location and the remaining instances in the chart maintain their location. A run of a chart, $c$, is a sequence of cuts $c_0, c_1, \ldots, c_k$ such that

- $c_0$ is an initial cut
- $\forall 0 \leq i < k, c_{i+1}$ succeeds $c_i$
- in the final cut, all locations are maximal.

The collective set of runs for chart $m$ is written as $\text{Runs}(m)$. We also define the $\text{getLabel}(m)$ function that maps two cuts to the alphabet $AP$. Given two cuts, the $\text{getLabel}(m)$ function returns the corresponding event that caused progress in the chart. It should be noted that in the case of a simultaneous message, the $\text{getLabel}(m)$ function only returns a letter for the send instance and not for the receive instance. Using the functions defined above, we can now define the function $f$ to map between $\text{cut} \times \text{cut}$ and the alphabet $AP$, as follows:

$$f(m)(c_i, c_j) = \begin{cases} 
\text{msg}(m)(\text{getLabel}(m)(c_i, c_j)) & \text{if } \text{succ}(m)(c_i, c_j) \\
\epsilon & \text{otherwise}
\end{cases}$$
The $f$ function allows us to map between two cuts of the chart to a letter in the alphabet of the chart, thus giving us the ability to determine the letter of interest between two cuts.

**Definition** The **trace** for a run $c = c_0, \ldots, c_k$ of a chart $m$, written as $w = trace(c)$, is the word $w = w_1 \cdot w_2 \cdots w_k$ over the alphabet $AP$, where each $w_i = f(m)(c_i, c_{i+1})$.

Using the definition of a trace, we can now define the trace language of a chart over all the possible traces in the system.

**Definition** The **trace language** generated by the chart $m$, $\mathcal{L}_{m}^{\text{trc}} \subseteq AP^*$ is the set of traces permitted by the chart, $\mathcal{L}_{m}^{\text{trc}} = \{w | w = trace(\text{Runs}(m))\}$.

In the case of a universal chart, an activation message $amsg$ can be sent to designate the start of the sequence of events as described by the chart. For a universal chart, the observation of an activation message requires that the next set of events follow the events as defined by the chart every time. For an existential chart, the sequence of events specified by the chart should occur in at least one trace of the system. The language $\mathcal{L}_m \subseteq AP^* \cup AP^w$ of a chart $m$ is defined as follows. For existential charts:

$$\mathcal{L}_m = \{w = w_1 \cdot w_2 \cdots | \exists i_0, i_1, \ldots, i_k \text{ and } \exists v = v_1 \cdot v_2 \cdots v_k \in \mathcal{L}_{m}^{\text{trc}}, \text{ s.t.}
\begin{align*}
&(i_0 < i_1 < \ldots < i_k) \land (w_{i_0} = amsg) \land \\
&(\forall j, 1 \leq j \leq k, w_{i_j} = v_j) \land \\
&(\forall j', i_0, \leq j' \leq i_k; j' \not\in \{i_0, i_1, \ldots, i_k\} \Rightarrow w_{j'} \not\in Messages(m))\}\}$$

The formula above requires that there should exist at least one run in the system which contains an instance of the activation message, and following the activation message the behaviors described in the existential chart are observed successfully. For universal charts, each time the activation message $amsg$ is observed, the trace should satisfy the behaviors specified in the main chart. This is written as follows:
\[ L_m = \{ w = w_1 \cdot w_2 \cdots : \forall i, w_i = amsg \Rightarrow \exists i_1, \ldots, i_k \text{ and} \] \\
\exists v = v_1 \cdot v_2 \cdots v_k \in L_{m}^{trc}, \text{s.t.} (i < i_1 < \ldots < i_k) \land \\
(\forall j, 1 \leq j \leq k, w_{i_j} = v_j) \land \\
(\forall j', i, i \leq j' \leq i_k, j' \not\in \{i_0, i_1, \ldots, i_k\} \Rightarrow w_{j'} \not\in Messages(m)) \} \]

At this point the relationship between a system and an LSC can be formalized as follows [35]:

**Definition** A system \( S \) **satisfies** the LSC specification \( LS = \langle M, amsg, mode \rangle \), written \( S \models LS \), if

1. \( \forall m \in M, mode(m) = universal \Rightarrow L_S \subseteq L_m \)

2. \( \forall m \in M, mode(m) = existential \Rightarrow L_S \cap L_m \neq \emptyset \).

### 2.5.1 LSC to Automaton Translation

We now discuss the LSC to automaton unwinding algorithm that creates an equivalent automaton structure for a given LSC. To simplify our discussion of the unwinding algorithm, we only present the translation of a basic chart (does not contain temperatures, prechart, and subcharts). We also present an example of the unwinding algorithm after the algorithm description.

Intuitively, the unwinding algorithm generates every possible cut for a given chart (using a basic depth first search and the \( \text{succ}(m) \) function) and creates the automaton structure by relating each unique cut to a unique state in the automaton. The transitions between states are labeled with the relevant letter that gives rise to the successor cut (\( \text{getLabel}(c) \) function).

Figure 2.5.1 shows the unwinding algorithm. The first step creates the initial cut for a given chart and the corresponding state from the initial cut. For any given cut, the \( createState \) method creates the corresponding state. The initial cut is then enqueued for further processing and added to the state set \( \Xi \). Next, lines 5-16 process
Algorithm: \textit{LSC2AUTOMATON}(c)

1: /* Create the initial cut and state. */
2: \( c_0 = \text{getInitialCut}(c), s_0 = \text{createState}(c_0) \)
3: /* Add initial state to state set and enqueue. */
4: \( \text{addState}(\Xi, s_0), \text{enqueue}(q, c_0) \)
5: /* Process cuts in queue. */
6: \textbf{while} notEmpty\((q)\) \textbf{do}
7: \( c = \text{dequeue}(q) \)
8: \( s = \text{getState}(c) \)
9: /* Get all successor cuts for current cut. */
10: \( \Theta = \text{getSuccs}(c) \)
11: \textbf{for all} \( c' \in \Theta \) \textbf{do}
12: \( s' = \text{createState}(c') \)
13: /* Enqueue new cuts/states in queue. */
14: \textbf{if} notSeenBefore\((\Xi, s')\) \textbf{then}
15: \( \text{enqueue}(q, c') \)
16: \( \text{addState}(\Xi, s') \)
17: /* Add a transition in the automaton for successor cut. */
18: \( \text{addTransition}(s, s', \text{getLabel}(c, c')) \)
19: \textbf{return}(s_0, \Xi)

Figure 2.17: Algorithm for building a basic automaton from an LSC.

Every cut in the queue. For each cut, the successor cuts are generated using the \textit{getSuccs} method. Each unique cut is enqueued in the queue and the corresponding state is added to the state set. Additionally, the transition from the parent state \( s \) to the child state \( s' \) is added in the automaton. The \textit{addTransition} method adds the transition from state \( s \) to \( s' \) with the \textit{getLabel}(c, c') transition label. Additionally, the \textit{addTransition} method also adds a self-loop for the parent state \( s \) that enables state \( s \) to wait in the current state until a relevant letter is observed. It should be noted that the state may already exist in the automaton (if generated earlier). After all cuts have been processed, the initial state of the automaton is returned as the pointer to
the entire automaton of the chart. The final states of the automaton are marked as accept states to signify the end of the chart. Each final state of the automaton also has a self-loop with the $true$ annotation.

Figure 2.18 shows the automaton for the main chart of the LSC shown in Figure 2.14. We ignore the actions at the end of the chart. State $q_0$ corresponds to the initial cut for the main chart where the next message to be received is the $SYNACK$ message. After the $SYNACK$ message has been received, the next cut of the chart corresponds to the $ACK$ message (state $q_1$). After the $ACK$ message has been observed, the automaton reaches state $q_2$, which corresponds to the final state of the automaton and is marked as an accept state.

### 2.6 Conclusions

We have presented an overview of the various constructs of the LSC grammar. Using the constructs, we also present an example of using LSCs for specifying different aspects of the TWHEP communication handshake protocol. The TWHEP communication protocol describes multiple situations to tackle cases relating to exceptions and error handling. Additionally, we also present an overview of the trace semantics.
of LSCs and the translation of LSCs to automaton using an unwinding algorithm that generates all possible cuts for a given chart.

LSCs lend themselves naturally to the specification of protocols that are primarily composed of message communication between multiple agents. Constructs such as messages, multiplicity, and temperatures provide sufficient power to effectively describe behaviors of a protocol and distinguish between provisional and mandatory behaviors. Additionally, multiple scenarios and special cases of a protocol are easily handled by LSC specifications that provide the framework to integrate subcharts, multiple charts, and hierarchical charts.

Past research related to LSC verification has explored the advantages of LSCs as a specification language but has failed to provide an effective and mathematically rigorous technique to relate systems with LSC specifications [31, 14]. The focus of our research is to investigate methods that allow the effective and mathematically rigorous verification of systems against LSC specifications.
Chapter 3

Improving Translation of Live Sequence Charts to Temporal Logic

Abstract

An efficient and mathematically rigorous translation from Live Sequence Charts (LSCs) to temporal logic is essential to providing an end-to-end specification and verification method for System on Chip (SoC) protocols. Without mathematical rigor, no translation can be trusted to completely represent the LSC specification, while inefficiency renders even provably sound translations useless in verifying the correctness of industrial-strength protocols. Previous work shows that the LSC-to-temporal logic and LSC-to-automata translations can be automated and formalized for the LSC language. In the LSC-to-temporal logic translation, the extraordinary size of the resulting formula limits the scalability of the charts that can be translated and verified. Our work, on the other hand, leverages intuitive temporal logic reductions to generate a formula that is at most quadratic in the size of the chart and demonstrates the benefits of the improved translation on several examples.
3.1 Introduction

Recently, development trends have shifted towards building systems by integrating numerous heterogeneous Intellectual Property (IP) cores on a single chip. Such System on Chip (SoC) designs implement multiple communication protocols that are necessary for the IP cores to interface and interact with each other. Given this trend, it becomes important to not only verify the individual IP cores, but the interface design and implementation as well. Verification of the interface provides the IP integrators the peace-of-mind guarantee of the IP core’s communication protocol as well as ease of integration in the SoC; thus, providing benefits to the IP core developers and integrators.

Pivotal to any verification effort is the ability to develop specifications against which the system in question is to be verified. Traditionally, English has been the default language choice for specifying and describing communication protocols. In our experience, English is not an ideal medium for expressing protocol specifications because of its context sensitive, imprecise, and cumbersome nature.

Protocol Live Sequence Charts (PLSCs) are a scenario-based language especially targeted for protocol design and verification of SoC systems implementing protocols [12]. Scenario-based languages, like PLSCs, provide a more intuitive and mathematically precise language for describing system interactions. Strictly speaking, PLSCs are a restricted form of Live Sequence Charts (LSCs) [10, 19]. Additionally, they provide syntactic constructs for easily specifying certain protocol-specific behaviors such as clocks and invariants. As an example of the conciseness and expressive power of PLSCs, the Virtual Component Interface (VCI) SoC communication protocol has been translated from its 60 page English format to a one page PLSC chart describing all possible interactions [12]. As PLSCs are a subset of LSCs, we concern ourselves with LSCs for the remainder of the paper.
LSCs can be used at various stages of the development and verification process. Initially, LSCs can be used to verify properties of the communication protocol to guarantee behaviors of the protocol. Further in the development and testing stages of the system, LSCs can provide a specification against which implementations are verified. LSCs can also be published as the supported interface of the IP core.

We focus on the problem of formally verifying systems against LSC specifications. Fig. 3.1 shows a high level overview of the process of verifying a system against an LSC specification as developed and presented in \[12,35,31\]. An LSC specification and the system under test are given as input. The scenario-based specification is translated to automata or temporal logic that is used to verify the system with the help of a model checker (symbolic or explicit), as shown in the shaded portion of the figure.

Inefficient translations to automata or temporal logic directly affect the verification task complexity, thus motivating the need to improve methods to translate LSCs to temporal logic or automata. We present a translation of LSCs to temporal logic that generates a temporal logic formula which is of at most quadratic size with respect to the number of maximal messages of the LSC. Earlier translations produce
formulas that are of quadratic size with respect to the size of the chart [35, 49], which in the average case, is much larger compared to the number of maximal messages in the chart. The translation uses logic minimization over the until (U) operator to improve upon earlier translations. We prove that our improved translation covers the same set of behaviors as those specified by the quadratic translation in [35] and is more than competitive with the work presented in [31], which translates LSCs directly to automata. Further, we extend the translation to other constructs of the LSC grammar that have not been directly translated to temporal logic in earlier research. Finally, we present results in explicit and symbolic model checking that show the benefits of using a smaller formula during verification as generated by our improved translation. Specifically, the verification time and state space are greatly reduced, especially in cases where counterexamples need to be generated.

3.2 Related Work

LSCs extend the semantics of MSCs to be able to specify provisional behavior [46, 19, 10]. Additionally, LSCs have also been used in the past to model and verify systems. Work in [7] describes the modeling of an air traffic control system and the verification performed on the system. Work in [34] describes the modeling of an automotive system using LSCs. Other such examples utilize the expressive power of LSCs to concisely describe and verify systems and relate them to other specification languages [5, 32, 20].

There is ongoing interest in the model checking of LSCs and translation of LSCs to automata for the automatic synthesis of systems [33, 1, 25, 6]. Additionally there has been significant work done in translating LSCs to temporal logic for verification of systems. The major limitation of translating LSCs to temporal logic is the sheer size of the resulting temporal logic formula [35, 49]. In [12], small and efficient-to-verify ordering properties are generated from LSCs but no formal relationship is
established between the system and the specification. Although the system satisfies the ordering properties, the properties do not imply that the system implements the LSC specification. In [35], an LSC is translated to an LTL formula that is quadratic in the size of the LSC. The size of the LTL formula is large enough to hinder the verification of anything but small LSCs. The work in [35] forms the basis of our work.

The work in [31, 48], and [49] give an alternate verification approach in an effort to avoid the explosion encountered during the LSC-to-automata translation. The approach unwinds the LSC to create an automaton with size proportional to the number of reachable states in the LSC, and it uses the automaton in a multi-tiered verification effort comprised of four different model checking procedures ordered by least-to-worst algorithmic complexity: reachability analysis with safety observer, ACTL model checking with and without observer, and finally LTL model checking. If a less powerful technique is not able to complete the verification, then the approach moves to the next procedure until it arrives at full LTL model checking. Although the approach deals with the full semantic model of LSCs, except Kleene stars, reachability is never powerful enough in systems that are non-timed as seen in the results from [31] where at least three procedures are run before obtaining an actual verification result in the non-timed charts; thus, calling upon full LTL verification consistently. The need for the more powerful model checking procedures described in [31] is critical to the work presented in this paper because the procedures require a temporal logic formula. The size of the formula produced from the automaton-to-temporal logic translation in [31] results in such a large formula that ACTL and LTL verification are only feasible on small non-concurrent charts.

The LSC-to-temporal logic translation in this paper produces an LTL formula that is small enough to be practical for a single step direct verification of systems
against even highly concurrent charts, and we show this by presenting results for LTL verification on charts larger and more concurrent than those presented in [31].

### 3.3 Live Sequence Charts

In this section we formalize our presentation of LSCs before relating them to LTL and verification. For simplicity and space purposes, the semantics only deal with a restricted set of the full LSC grammar; although, the translation extends to most of the LSC semantic constructs as discussed in Section 3.7.

LSCs are a graphical language for specifying communication between agents in a system [46, 19, 10]. Figure 3.2(a) is an example LSC that contains the relevant language features discussed in this section. The boxes at the top of the figure are agents in the system. Each agent has a vertical instance line descending from the agent. The horizontal lines with filled arrowheads are synchronous messages. Synchronous messages force the send and receive events of the message to occur in the same state; thus, the sender and receiver have to be at their respective send and receive locations in the chart.

Messages in the LSC are divided into a pre-chart and a main chart. The pre-chart is the activation condition for the main chart and indicates the scenario in which the main chart operates. It is represented by the partially dashed hexagon in the figure. The main chart (directly below the pre-chart) is represented by the rectangle. A universal chart that specifies mandatory behavior is drawn with a solid line and an existential chart specifying provisional behavior is drawn with a dashed line. Universal charts force the occurrence of the main chart events after every occurrence of the pre-chart. Existential charts only assert the presence of one instance of the pre-chart and main chart events, and do not force the main chart to occur after every occurrence of the pre-chart. For the LSC shown in Fig. 3.2(a), the pre-chart is satisfied by a trace that contains the messages \{p_1, p_2\} in any order, followed by p_3,
Figure 3.2: A LSC illustrating most of the language features supported in this work with a lattice defining the partial order on messages in the chart. (a) The LSC. (b) The partial order defined on the messages in the LSC.

and finally \( \{p_4,p_5\} \) in any order. Note that the messages \( p_1 \) and \( p_2 \) are not ordered with respect to each other but are ordered with respect to \( p_3 \). The same observation holds for the messages \( \{p_4,p_5\} \) and \( p_3 \). Once the pre-chart has been satisfied, the mandatory behavior of the universal chart forces the main chart to occur after the pre-chart. The main chart is satisfied if the messages \( \{m_1,m_2\} \) occur in any order, followed by \( m_3, m_4 \), and finally the messages \( \{m_5,m_6\} \) in any order.

We use the symbol \( c \) to denote an individual chart like that in Fig. 3.2(a). The set of instance lines for a chart is given by \( \text{inst}(c) \). An instance line has locations to track the state of the associated agent. For an instance line \( i \) and chart \( c \), \( \text{dom}(c,i) = \{l_0,l_1,\ldots,l_{\text{max}}(i)\} \) is the set of locations for \( i \) in \( c \) with the first location (top most location for a instance line) given by \( l_0 \) and the last location given by \( l_{\text{max}}(i) \) when moving from the top to the bottom of the instance line. As locations are not uniquely labeled in the chart across instance lines, the set of pairs \( \text{dom}(c) = \{\langle i,l \rangle \mid i \in \text{inst}(c) \land l \in \text{dom}(c,i)\} \) represents all instance and location pairs in a chart \( c \). The complete state of a chart is the state of the individual instances as denoted by their
current location. The initial state has every instance in its first location. The state of the chart evolves as instances move from one location to another down the chart.

The symbol $AP$ denotes the set of messages in the system. Communication is specified as a triple of the form $(\langle i, l \rangle, e, \langle i', l' \rangle)$, where $e \in AP$ is the message communicated from $\langle i, l \rangle$ to $\langle i', l' \rangle$. The set of all message communications for a chart is given by the relation $R(c)$. Well-formed charts are charts where the relation $R(c)$ is acyclic. This work, like [35], only considers well-formed charts.

The pre-chart begins at the first location, $l_0$, for each instance line. The main chart is the box below the pre-chart. A special location is reserved to denote the start of the main chart (end of the pre-chart). For each instance line, $i$, the last location, $l_{\text{max}}(i)$, marks the end of the main chart. As such, we require three unique locations in a chart that are not used by messages: the start of the pre-chart, the start of the main chart (end of the pre-chart), and the end of the main chart; all other locations must be used by messages. We use $p$ to represent messages in the pre-chart, $m$ to represent messages in the main chart, and $e$ to represent any message in the chart regardless of position in the chart.

A chart defines a partial order on its messages as its state changes on location transitions. To be specific, we introduce the symbols $\top$ (top), $\downarrow$ (middle), and $\bot$ (bottom) to synchronize the agents when the chart starts, completes its pre-chart, and completes its main chart, respectively. The function shown in Figure 3.3 returns the letter that is communicated in a message or the symbol $\top$, $\downarrow$, or $\bot$. For convenience, we define $msg(c)$, $msg_p(c)$ and $msg_m(c)$ to be the messages of the entire chart, the pre-chart, and the main chart respectively.

We define an order relation, $\prec$, to capture the sequences of messages and symbols as specified in the instance lines when starting at the first locations and
traversing the lines to their last locations:

\[
\forall\langle i, l_i \rangle, \langle i, l_{i+1} \rangle \in \text{dom}(c), \quad msg(c)((i, l_i)) \hookrightarrow msg(c)((i, l_{i+1})).
\]

We do not explicitly describe the rules of moving from one state to another. Rather, we relate the message sequences as observed along each instance line in the order relation. In other words, the order relation describes the sequence of messages observed in the scope of the individual instance lines. The locations that communicate each message in the chart connect the sequences observed in one agent to those of the other agents. A partial order, \( \prec \), on messages and symbols is created from the order relation \( \sqsubseteq \) by adding to it the reflexive terms and then computing its transitive closure. The partial order induced by the chart in Figure 3.2(a) is shown in Figure 3.2(b).

The lattice formed by the partial order shows those messages that are unordered with respect to other messages in the chart. The observation is key to our improved translation because these events must be ordered before synchronizing events but not relative to each other. For convenience in writing the necessary ordering properties, we also define \( \text{max}_m(c) = \{ e \mid e \sqsubseteq \perp \} \) and \( \text{max}_p(c) = \{ e \mid e \sqsubseteq \top \} \) to be the maximal messages of the main chart and the pre-chart respectively. Intuitively, \( \text{max}_m(c) \) is the set of messages that occurs last in the main chart and \( \text{max}_p(c) \) is the set of messages that occurs last in the pre-chart and immediately before the start of the main chart.
An LSC specification is a set of scenarios, \( C \), defined in individual charts, \( c \). For a given LSC specification, the equivalent temporal logic specification is given as \( \psi = \bigwedge_{c \in C} \psi_c \) where \( \psi_c \) is the temporal logic formula for chart \( c \) using one of the translation approaches described in this work. A proof for the relationship between a system, an LSC specification, and the generated temporal logic formula is given in [35]. We summarize the proposition here and assert that since our improved translation is equivalent to the translation in [35], the proposition holds for our improved translation as well.

**Proposition 3.3.1** Given a set of LSCs, \( C \), for a specification, let \( \psi \) be the temporal logic formula \( \bigwedge_{c \in C} \psi_c \), \( S \) be a system implementing the scenarios in \( C \), and \( s_0 \) be the initial state of the system. Then

\[
S, s_0 \models \psi \iff \forall_{c \in C} S, s_0 \models c.
\]

The translation, as presented in [35], writes a property for every entry in the partial order induced by the chart; thus, producing a formula that is at least quadratic in the size of the chart.

### 3.4 Quadratic Translation of LSCs to LTL

Two kinds of properties are generated in the quadratic translation of [35]: properties that establish the order between any two messages in the chart (\( \phi \) properties) and properties that guarantee the uniqueness of each message instance in the chart (\( \chi \) properties). The work presented in [35] and in this paper restricts all charts to contain only one instance of a message. For any two messages \( x_i \) and \( x_j \), such that \( x_i \prec x_j \) in the partial order imposed by the chart, the property \( \phi_{x_i, x_j} = \neg x_j \mathbin{U} x_i \) is generated stating that the message \( x_j \) does not occur until message \( x_i \) has occurred. For any two messages \( x_i \) and \( x_j \) such that \( x_i \not\prec x_j \), the \( \chi_{x_i, x_j} = (\neg x_i \land \neg x_j) \mathbin{U} (x_i \land X (\neg x_i \land \neg x_j) \mathbin{U} x_i) \) property states that the message \( x_i \) occurs twice before \( x_j \).
The negation of this property states that message $x_i$ does not occur twice before message $x_j$.

The quadratic translation for a universal chart as defined in [35] is given in Equation 3.1. Again, we use the letters $e, p$ and $m$ to denote the events in the chart in general, pre-chart, and main chart respectively. Equation 3.1 is divided into two parts by the implication; if the pre-chart is correctly satisfied (left), then the main chart has to follow (right). The top terms on each side of Equation 3.1 describe the order of the pre-chart and main chart messages. The $\phi$ properties are generated for all pairs belonging to the $\prec$-relation as restricted by the $p, m$ and $e$ notation. The middle term on the left side guarantees that no main chart events occur in the pre-chart, thus, correctly framing the pre-chart. The middle term on the right guarantees the occurrence of the maximal messages (since they do not occur on the right side of any $\phi$ formula). The framing is important because it is responsible for correctly triggering the verification of the main chart events. In other words, we do not check for main chart events until we have correctly observed the pre-chart events. Finally, the bottom terms on each side of the implication of Equation 3.1 guarantee that each message instance only occurs once in the chart. For the example LSC shown in Figure 3.2(a), Equation 3.1 generates 50 pre-chart properties and 80 main chart properties. In this context, a property refers to a single $\phi$ or $\chi$ term generated by the translation.

$$\psi_e = \text{G} \left( \left( \bigwedge_{p_i \prec p_j} \phi_{p_i, p_j} \right) \land \left( \bigwedge_{\forall p_i, m_j} \phi_{p_i, m_j} \right) \land \left( \bigwedge_{p_i \neq p_j} \neg \chi_{p_i, p_j} \right) \right) \Rightarrow \left( \left( \bigwedge_{m_i \prec m_j} \phi_{m_i, m_j} \right) \land \left( \bigwedge_{m_j \text{ is max}} F m_j \right) \land \left( \bigwedge_{\forall e_i, m_j} \neg \chi_{e_i, m_j} \right) \right)$$

(3.1)
Table 3.1: Properties generated for the left hand side of the implication

Table 3.4 shows the properties generated for the left side of the implication in Equation 3.1 and Table 3.2 shows the properties generated by the right side of the implication.

Since the model checking process (explicitly and symbolic) depends directly on the length of the formula being verified against the model, we measure the complexity of the produced formula in terms of the individual properties generated by the translation to temporal logic. The complexity analysis of the translation is divided into two parts: analyzing terms that establish order (top left, middle left, top right, and middle right terms) and uniqueness (bottom right and bottom left) in Equation 3.1. The total number of individual properties required to establish order is bounded by $|msg_p(c)|^2 + (|msg_p(c)| \times |msg_m(c)|) + |msg_m(c)|^2 + |max_m(c)|$. For establishing uniqueness, the number of properties is bounded by $|msg_p(c)|^2 + (|msg_p(c)| + |msg_m(c)|) \times |msg_m(c)|$. Combining these bounds gives us a complexity that is at least quadratic in the size of the chart.

Despite the quadratic bound of the translation presented in [35], which improves on the classical exponential translation, the resulting formula is too large to be practical. The large size is due to the use of the partial order in Equation 3.1.
Building properties from the \(-\)-relation includes redundant ordering formulas in the pre-chart and the main chart. For example, if \(\neg p_3 \cup p_2\) and \(\neg p_2 \cup p_1\) hold, then by transitivity, we know that \(\neg p_3 \cup p_1\) holds, and do not need to explicitly establish the relation. Additionally, checking for uniqueness of an event with respect to every other event includes multiple redundant checks. For example, if message \(m_5\) does not occur until after \(m_3\), and message \(m_1\) occurs once before \(m_5\) and \(m_3\), then it is implied that message \(m_1\) occurs only once before \(m_3\). These reductions are formalized in the next section.
3.5 Temporal Logic Reductions

The improved translation makes use of transitivity to reduce the size of the quadratic formula presented in [35]. It eliminates both ordering ($\phi$) and uniqueness ($\chi$) properties from the formula. These reductions are discussed in the following sub-sections.\(^1\)

3.5.1 Reducing Ordering Properties

The following reduction result for the until (U) LTL operator is a general modal logic calculation which has been restated for this domain from [4]; it forms the basis of our reduction in the number of $\phi$ formulas in the final translation.

**Lemma 3.5.1** For any three messages $x_t$, $x_u$, and $x_v$ and a trace $\pi = s_0, s_1, ...$

\[
M, \pi \models (\neg x_v \ U x_u) \land (\neg x_u \ U x_t) \Rightarrow M, \pi \models (\neg x_v \ U x_t).
\]

Intuitively, the U-operator forces the existence of the right hand predicate, and ensures that the left hand predicate holds until the state where the right side predicate occurs. The above result takes advantage of the transitivity of the U-operator to eliminate properties that do not need to be explicitly verified but are part of the partial order on messages induced by the chart.

Furthermore, since there are events that may be succeeded by multiple unordered events, it is useful to be able to collapse multiple ordering properties relative to a single event into a single ordering property. An example of the reduction is seen in Figure 3.2(a) where we would not expect to see $p_5$ or $p_4$ until $p_3$; and $p_5$ and $p_4$ are unordered relative to each other. In essence, the reduction collapses expressions that share the right hand term of the U-operator into a single property.

\(^1\)All proofs are available in the full version of the paper available at http://vv.cs.byu.edu/~rahul/LSCTOLTLL.pdf
**Lemma 3.5.2** For a given set of messages $N$ and a message $x_i$ such that $\forall x_j \in N, x_i \prec x_j$

$$M, \pi \models \bigwedge_{x_j \in N} (\neg x_j \cup x_i) \iff M, \pi \models (\bigwedge_{x_j \in N} \neg x_j) \cup x_i.$$ 

For our example, Lemma 3.5.2 results in a single property, rather than two properties, that does not allow either $p_5$ or $p_4$ until it sees $p_3$.

We now show via application of Lemma 3.5.1 and Lemma 3.5.2 how the total number of ordering properties can be reduced in the quadratic translation of [35]. First, we define the function $\text{next}(c)(e_i) = \{e_j \mid (e_i \triangleleft e_j) \land (e_j \not\in \{\top, \bot\})\}$. The $\text{next}(c)$ function, given a message $e_i$, returns the set of immediate successors according to $\triangleleft$-relation induced by the instance lines involved in the message communication. Using this function, we directly reference Lemma 3.5.2 to coalesce $\phi$ properties. We also define the $\phi'$ helper function as follows:

$$\phi'_{x_i,N} = (\bigwedge_{o \in N} \neg o) \cup x_i.$$ 

The $\phi'$ function is a modified version of the $\phi$ function presented earlier. The second argument of the $\phi'$ function is a set rather than a single event and relies on Lemma 3.5.2 to produce a single ordering property when $N$ meets the necessary conditions. We now present the main reduction for this section.

**Corollary 3.5.3** Given a chart, $c$, the ordering imposed by the formulas including the transitive and not including the transitive properties in the partial order is exactly the same.

$$M, \pi \models \bigwedge_{p_i \prec p_j} \phi_{p_i,p_j} \iff M, \pi \models \bigwedge_{p \in \text{msg}(c)} \phi'_{p,\text{next}(c)(p)}.$$ 

**Proof** This result follows by successive application of Lemma 3.5.1 and then Lemma 3.5.2 to the set of formulas in $\bigwedge_{p_i \prec p_j} \phi_{p_i,p_j}$. 

45
The corollary reduces the set of properties needed to specify the order of events by omitting transitive properties which are implied by Lemma 3.5.1, and it proves the equivalence to the set of properties that explicitly writes the order between every event. Intuitively, the right formula is written from the $\triangleright$-relation while the left formula is written from the $\prec$-relation. The $U$-operator implies by transitivity the presence of the extra orderings included in the $\prec$-relation.

### 3.5.2 Reducing Uniqueness Properties

The $\chi$ formula in the previous section states that message $x_i$ occurs twice before message $x_j$, and the negative form of the $\chi$ formula states that message $x_i$ does not occur twice before message $x_j$. The next reduction result eliminates redundant properties when establishing uniqueness of a message in a chart (main or pre-chart). The reduction result is stated as follows:

**Lemma 3.5.4** Given $k$ messages in a set $N$ that occur in the order $e_1 \prec \ldots \prec e_k$, if a message $e_i$ does not re-occur between its occurrence and the final message $e_k$, then it does not re-occur between its occurrence and any intermediate message $e_j$.

$$M, \pi \models \bigwedge_{e_i \neq e_j} \neg \chi_{e_j,e_i} \iff M, \pi \models \bigwedge_{e_i \in N} \neg \chi_{e_i,e_k}.$$  

Intuitively, Lemma 3.5.4 states that given a validated message ordering, we only need to establish the uniqueness of a message with respect to the last message in the ordering (maximal message of chart), as opposed to establishing uniqueness with respect to every message in the ordering.

### 3.6 Improved Translation of LSCs to LTL

The improved translation is produced by using the reduction results of Lemmas 3.5.1, 3.5.2, 3.5.4, and Corollary 3.5.3. The improved formula, $\psi'_c$, for universal
charts is shown in Equation 3.2. The improved translation has a structure similar to
the structure of the quadratic translation shown in Equation 3.1. The key differences
are in the use of the $\prec$-relation (used by the $\text{next}(c)$ function) for specifying order,
and the use of the $\text{max}_m(c)$ function for specifying fewer uniqueness properties. For
explicit state model checking, the formula in its negated form is synthesized to au-
tomata for the verification of systems. Any violation provides a counterexample of
the chart, which represents a flaw in the system implementing the chart. For symbolic
model checking, the formula is not negated but directly verified on the system.

\[
\psi'_c = G \left( \begin{align*}
\bigwedge_{e \in \text{msg}_{p}(c)} & \phi'_{e, \text{next}(c)(e)} \\
\bigwedge_{e \in \text{max}_{p}(c)} & \phi'_{e, \text{msg}_{m}(c)} \\
\bigwedge_{(e,p) \in \text{msg}_{p}(c) \times \text{max}_{p}(c)} & \neg \chi_{e,p}
\end{align*} \right) \Rightarrow \left( \begin{align*}
\bigwedge_{e \in \text{msg}_{m}(c)} & \phi'_{e, \text{next}(c)(e)} \\
\bigwedge_{(e,m) \in \text{msg}(c) \times \text{max}_{m}(c)} & \neg \chi_{e,m}
\end{align*} \right)
\] (3.2)

Since existential charts (provisional behavior) have to be satisfied by some trace of the
system, as opposed to universal charts, the distinction between the pre-chart and main
chart is no longer necessary. A witness of the events described in the existential chart
is sufficient to prove the correctness of a system/model. To establish the existence
of this witness the $\textbf{EF}$ operator is used [31]. Equation 3.3 shows the translation of
existential charts. The existential formula states that there exists a trace in the future
that satisfies the sequence of events as described in the existential chart. Similar to
the earlier formulas, the $\phi'$ properties establish the order of the messages of the
pre-chart and the main chart, and the $\chi$ properties establish the uniqueness of the
messages with respect to the maximal messages in the chart. Equation 3.3 is a $\text{CTL}^*$
formula. Negating this $\text{CTL}^*$ formula gives us an $\text{LTL}$ formula that has the form
AG (¬(θ_{order} ∧ θ_{uniqueness})) where θ_{order} are the properties used to specify the order of events and θ_{uniqueness} are the properties that specify uniqueness of messages in the chart. This LTL formula is used for explicit or symbolic state verification. A violation of the formula provides a witness to the existential chart, which is the desired result, since an existential chart should be satisfied by at least one run of the system. If, on the other hand, no witness is generated, then the implementation violates the specification. Again, it should be noted that we start with a CTL* formula but actually perform LTL verification since the negated formula is in LTL.

\[
\text{EF} \left( \bigwedge_{e \in \text{msg}(c)} \phi'_{e, \text{next}(c)(e)} \land \bigwedge_{(e,m) \in \text{msg}(c) \times \text{max}_m(c)} \neg \chi_{e,m} \right)
\]  

(3.3)

**Theorem 3.6.1** The improved translation as presented in Equation 3.2 and Equation 3.3 is equivalent to the quadratic translation presented in [35].

\[ M, \pi \models \psi_c \iff M, \pi \models \psi'_c. \]

For the example LSC in Fig. 3.2, the improved translation generates 13 properties for the pre-chart and 28 properties for the main chart. This provides a dramatic reduction over the 50 and 80 properties generated using the translation presented in [35].

### 3.7 Translating Additional Constructs

The work presented in this paper can translate all constructs except non-bonded conditions (conditions not tied to a specific message), temperatures (progress is forced or unforced), tolerant behavior (allowing multiple instances of an event within a chart), and multiplicities (Kleene star). In our experience, these constructs are rare
in practice, and charts omitting these constructs are more than expressive enough to specify IP core interactions. We briefly present extensions to remaining constructs. Further details and examples can be found in Appendix B.

**Asynchronous Messages**: Asynchronous messages drawn in charts with an open ended arrow head are used to specify messages where the receiver may not be ready to receive the message. We build a new $\prec$-relation that ranges over the individual send and receive events and forces the send event to always occur before the receive event. Properties are then generated from this new $\prec$-relation.

**Co-regions**: Co-regions, specified by dashed lines parallel to the agent instance line, are used to express unordered events. To translate co-regions into temporal logic, we do not explicitly specify the order of all the events. Instead, we specify the order of the events that are to occur before and after the co-region and force the existence of the co-region events in the correct order.

**Bonded Conditions**: Bonded conditions are boolean predicates that are checked with a message in a simultaneous region (all events occur in the same state); thus, they are bonded to the message. Since a message always occurs with the condition (simultaneous region), the temporal logic translation only needs to consider the conjunction of the boolean predicate of the condition and the message.

**Invariants**: Invariants are boolean predicates that can be specified for a certain region of the LSC. We treat invariants as a set of bonded conditions where each event in the invariant region is conjuncted with the invariant condition. Using this approach, the invariant is only checked when an event occurs.

### 3.8 Analysis

The formulas presented in Equation 3.2, Equation 3.3 and their extensions to additional constructs are quadratic in the number of maximal messages in the main chart, with all except one term being linear. In this context, linear implies a one-
to-one relation between the LSC event and a $\phi'$ or $\chi$ property. As before, we break up the analysis into two parts: one for the number of individual properties needed to establish event order and the second for the number of individual properties needed to establish uniqueness of events. The terms used to specify order are bounded by $|msg_p(c)| + |max_p(c)| + |msg_m(c)|$ properties, which is linear as compared to the quadratic number of properties generated by the quadratic translation presented in [35]. Terms used for specifying uniqueness are bounded by $(|msg_p(c)| \times |max_p(c)|) + (|msg(c)| \times |max_m(c)|)$ properties. For the pre-chart and main chart the number of uniqueness properties depends on the size of the chart and the number of maximal messages in the chart as opposed to the original translation where the number of uniqueness properties depends on the square of the number of messages in the chart, which in the typical case is much greater than the number of maximal messages in the chart (which can be greater than one only if the chart ends in a set of concurrent messages). Equation 3.3 has similar bounds.

3.9 Results

We are interested in understanding the performance of our improved LSC-to-temporal logic translation in both explicit and symbolic state model checking in terms of total verification time. Ideally, we would like to directly compare verification time using our translation to that in [35] and verification using the iterative approach in [31]. Such a direct comparison to [31] is not possible, however, because we do not have access to the implementation of the approach (which builds on VIS). Since the specification descriptions of [31] are the worst-case specifications for the translation process, we design an experiment using the specification descriptions from the empirical study in [31], and only indirectly compare the results.

The independent variables (inputs) in our experiment are the two translating approaches, the model checkers, the specifications, and the implementations. The
model checkers are SPIN for explicit state model checking and NuSMV for symbolic state model checking. We generate specifications \( \text{SpecB} \) and \( \text{SpecC} \) containing five to seven messages, and we use the \( A2 \), \( A3 \), and \( A4 \) specifications from [31]. The \( Ax \) specifications are highly concurrent specifications consisting of 3 non-timed sequential co-regions with \( x \) messages in each co-region. For example, the \( A4 \) specification has twelve total messages that appear in three groups of four concurrent messages.

The implementations of the specifications in our experiment are broken into two categories: those that correctly implement the specification and those that do not as indicated by the _e annotation on the model name (created by introducing errors in the initial stages of the main chart). All of the \( \text{Spec*} \) systems directly implement the message sequences in the specification using inter-process communication as provided by Promela constructs. The \( Ax \) implementations, however, follow the pattern described in [31] in that they first perform arbitrary computation by solving a puzzle and then implement the message sequence described in the specification. The pattern is similar to one observed in SoC designs using IP cores that perform some local computation followed by an exchange of information as specified by the IP core’s interface and communication protocol. Since the puzzles nor the sizes of the implementations in [31] could be obtained, we use the sokoban block sliding puzzle (soko), the bridge crossing puzzle (bridge), and the Alternating Bit Protocol (abp4) to represent arbitrary computation before implementing chart behavior. All models were written manually.  

The dependent variables (outputs) in all our experiments are the number of states explored and the verification time, which we refer to as wall clock time. In the case of explicit state model checking, wall clock time does not include the time for automata generation and compilation. All of our experiments are run on an Intel Pentium 4 3.0 GHz machine with 2 GB of main memory. LTL-to-automata synthesis

\[1\] All of our models, and specification formulas can be downloaded at http://vv.cs.byu.edu/~rahul/experiments.tar.gz
is handled using LTL2BA. The results of our experiments are presented in Table 3.3 (SPIN) and Table 3.4 (NuSMV). A ‘–’ entry in a table indicates that LTL-to-automata synthesis times-out (> 2 hours). The quadratic translation is the translation from [35] while the improved translation is the translation as presented in this paper.

The explicit state model checking results in Table 3.3 show the verification time and number of states explored is much smaller for the improved translation compared to the quadratic translation. For specification A2, the improved translation completes in less than half the time of the quadratic translation and produces half as many states, thus, supporting our claim that verification cost is directly proportional to the size of the formula. Note that the LTL-to-automata synthesis times-out for the quadratic translation for two of the specifications. LTL2BA can be a limiting factor for explicit state model checking since LTL synthesis also times-out for both the quadratic and the improved translations for models and specifications not included in the table.

Table 3.4 shows that the improved translation results in a much smaller verification time in symbolic model checking. For specification A4, the improved translation completes the verification in less than a fourth of the time required by the quadratic translation. The performance improvement is noticeably larger when a counterex-
Table 3.4: Verification results using NuSMV.

<table>
<thead>
<tr>
<th>Specification</th>
<th>Model</th>
<th>States</th>
<th>Quadratic Time (seconds)</th>
<th>Improved Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2</td>
<td>bridge</td>
<td>76992</td>
<td>14</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>abp4</td>
<td>2236420</td>
<td>30</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>bridge.e</td>
<td>76992</td>
<td>29</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>abp4.e</td>
<td>2236420</td>
<td>76</td>
<td>27</td>
</tr>
<tr>
<td>A3</td>
<td>bridge</td>
<td>76992</td>
<td>22</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>abp4</td>
<td>2236420</td>
<td>59</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>bridge.e</td>
<td>76992</td>
<td>56</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>abp.e</td>
<td>2236420</td>
<td>146</td>
<td>50</td>
</tr>
<tr>
<td>A4</td>
<td>bridge</td>
<td>76992</td>
<td>49</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>abp4</td>
<td>2236420</td>
<td>174</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>bridge.e</td>
<td>76992</td>
<td>132</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>abp.e</td>
<td>2236420</td>
<td>337</td>
<td>67</td>
</tr>
<tr>
<td>A5</td>
<td>bridge</td>
<td>76992</td>
<td>175</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>abp4</td>
<td>2236420</td>
<td>555</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>bridge.e</td>
<td>76992</td>
<td>509</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>abp.e</td>
<td>2236420</td>
<td>1271</td>
<td>131</td>
</tr>
</tbody>
</table>

ample trace is generated for the model. More importantly, the improved translation readily verifies A5 implementations which are not verified in [31] due to the large formula size from the translation. In fact, the improved translation allows direct LTL verification of the Ax implementations which in the multi-tiered approach of [31] require three separate verification runs before obtaining a meaningful result.

There are three key threats to validity in this empirical study. First, the omitted times for LTL-to-automata synthesis in explicit state model checking. The LTL synthesis time is a one time cost that either completes or does not complete. We assume that once synthesized, the formula is reused over several verification runs as the system is implemented. In our experiment, all LTL synthesis occurred in less than three minutes of wall clock time. The second threat to validity is the indirect comparison to the multi-tiered approach of [31]. Our experiments try to duplicate the models in [31] as closely as possible. Anecdotally, our results scale to the A4 and A5 specifications in very little time and with great ease in NuSMV, which is of prime importance, since the specifications represent our worst-case for this translation. Real world specifications tend to be much less concurrent and smaller as compared to the
Ax specifications. The third and final threat is the model checker implementations. It is not known if a different model checker such as VIS would give different results to invalidate those presented here. We suspect such a result to be highly unlikely since we know that formula size (as well as nesting depth) directly affects the verification cost.

3.10 Conclusions and Future Work

We present an improved translation of LSCs to temporal logic for a subset of LSCs by capitalizing on temporal logic reductions to reduce the number of individual properties needed for specifying event order and uniqueness. The known quadratic translation is directly proportional to the number of messages in the main chart as opposed to the improved translation that is proportional to the number of maximal messages in the main chart, which in the common case, is much smaller than the total number of messages in the main chart. We present results that show the benefits of using the improved translation during automata synthesis and verification in both the explicit and symbolic domains. Verification time, number of states, and automata synthesis benefit from the improved translation because of the smaller formula size. Future work in this area involves extending the translation to the LSC constructs of non-bonded conditions and temperatures. We are also investigating methods to decompose the formula into smaller pieces, as well as creating specialized algorithms to take advantage of the formula structure for optimized LTL synthesis. By doing so, we plan to complete the verification chain for SoC interface design and validation.
Chapter 4

Specifying Multi-Agent Systems Using Live Sequence Charts

Abstract

Multi-agent systems are an important branch of computer science that provide solutions to complex tasks and problems. Their growing popularity and applicability in different domains has led to the investigation of techniques and tools to formally analyze and prove properties about multi-agent systems. In this paper we present a methodology to specify properties about multi-agent systems using Live Sequence Charts (LSCs): a graphical and intuitive language that has been used for specifying and modeling systems. We present techniques to embed not only temporal but knowledge properties of multi-agent systems in LSCs and further show how model checking can be performed using the LSC specifications. Additionally, we present a case study to demonstrate the effectiveness of our technique for specifying and verifying complex temporal events and knowledge properties of a multi-agent system.
4.1 Introduction

Multi-agent systems research is arguably one of the most exciting and investigated research fields today. There are multi-agent systems finding uses in a variety of software applications such as safety critical systems implemented for aircraft/spaceship controllers, sensor management systems, and everyday software applications used in various settings ranging from games to financial portfolio management applications [26]. Their inherent ability to provide solutions to complex problems is cited as a major reason for their popularity and applicability. As with all complex and specialized systems, there comes the burden of ensuring the correctness and safety of the multi-agent systems. Past research has shown that formal methods and model checking are very effective techniques in tracking down errors and creating systems that are resilient.

Model checking is an exhaustive technique to enumerate all possible behaviors of the system and verify them against a specification, usually provided in temporal logic. Traditional model checking only supports the verification of properties that are based on temporal events in the system. Temporal events and their verification, although an important aspect of the system to be analyzed, prove to be incomplete for multi-agent systems. Multi-agent systems require not only the verification of temporal properties but also the verification of knowledge properties – properties based on knowledge of the agents in a given state. Research in the past has focused significantly on creating temporal logics and verification algorithms that support knowledge operators to specify and verify knowledge properties of multi-agent systems.

Temporal logics, although very useful for writing system specifications, are not user friendly and intuitive in nature. Their use in model checking is absolutely essential but limited to experts of the field who are well versed in the use of temporal logic. Even then, there may be specifications and properties that are difficult for experts to express correctly in temporal logic. Alternatives such as sequence dia-
grams, timing diagrams, interaction diagrams etc. have been explored in the past to
describe system properties and behaviors. Live Sequence Charts (LSCs) is one such
specification format developed to describe the behaviors of the individual processes
of a system and their interactions with each other [19, 29]. Their fully developed
grammar and semantics provide a strong mathematical framework for relating them
to model checking and formal methods. They have been successfully used to model
and specify a variety of applications such as train and automotive systems [5, 34] as
well as System on Chip designs [14].

In this paper we show how LSCs can be effectively used to specify complex
temporal as well as knowledge properties of a multi-agent system. We first provide
a translation of LSCs to $\text{CTL}$ that is at most quadratic in the size of the LSC. Using the $\text{CTL}$ translation of LSCs, we show how knowledge formulas can be visually
embedded in the LSC using the knowledge operator as described in the $\text{CTLK}$ temporal logic [45]. Further, we extend the translation of LSCs to $\text{CTLK}$ and present a
case study in which we specify and verify properties of an auction protocol. The case
study effectively demonstrates the relative ease with which non-experts of temporal
logic can specify series of complex temporal events and knowledge properties of a
multi-agent system using only LSCs as opposed to temporal logic. Finally, we also
generalize our result of embedding temporal logic formulas in LSCs to other temporal
logics such as LTL and $\text{CTL}$.

The paper is structured as follows. Section 4.2 discusses the related work in
this area. Next we present an overview of LSCs in Section 4.3 and their application
to multi-agent model checking in Section 4.4. In Section 4.5 we present a case study
of specifying an auction protocol using the techniques presented in this paper. Section 4.6 presents our result for embedding arbitrary temporal logic formulas in LSCs.
Finally, we present conclusions and future work in Section 4.7.
4.2 Related Work

Wooldridge et. al. in [50] present a methodology to verify multi-agent systems based on the MABLE language. Multi-agent systems are modeled using MABLE and then translated to Promela and verified using the SPIN model checker [28]. Claims or properties (specifications) to be verified on the systems are specified using the CKL temporal logic that extends CTL with knowledge operators as presented in [50]. These claims are further translated to LTL using the concept of local propositions. Another approach similar to the approach of [50] is the model checking techniques for formulas written in Alternating Temporal Logic (ATL) [44]. The claims/properties written in ATL are verified on the model by translating the model from the multi-agent specification language to the native input language of a model checker such as SPIN. The work presented in [8] presents a technique to translate models specified in AgentSpeak to Promela and Java which can then be verified using SPIN and JPF2 respectively. The primary drawback of the methods listed above is the performance bottleneck that is created when verifying large models translated from multi-agent modeling languages to Promela and SPIN. Another drawback is the lack of automation provided by the techniques to specify claims of knowledge and time using less expressive logics such as LTL and CTL. Additionally, the presented temporal logics and translations between temporal logics prove to be non-trivial and non-intuitive, which is a serious limiting factor in their adoption and applicability.

Other solutions for model checking multi-agent systems have chosen to create specialized and dedicated tools, algorithms and logics for the verification and specification of multi-agent systems. Initial work in this area was presented in [3], which presented Multi-agent Temporal Logic (MATL), a language based on CTL and Hierarchical Meta-Logic (HML) for specifying the beliefs, desires and intentions of agents in temporal logic. Model checking procedures based on symbolic model checking as well as verification examples were provided for verifying MATL properties of the sys-
tem in question. Model Checking Multi-Agent Systems (MCMAS) is a multi-agent model checker that can verify CTLK properties, which are expressive enough to specify temporal as well as knowledge properties. MCMAS provides a framework for specifying Interpreted Systems (IS) [45] and uses symbolic model checking with labeling algorithms to perform CTLK verification. Model Checking Knowledge (MCK) is a tool that implements OBDD based symbolic model checking techniques to verify properties on systems [22] specified using the LTL and CTL logics extended with knowledge operators. The specialized model checkers that are listed here are very effective in performing verification of the multi-agent systems in question, but do not provide very intuitive or easy ways for specifying the properties that are to be verified on the system; thus, they lead to verification of simple properties that do not span over larger more complex behaviors of the system in question. Our research presents a solution for this exact problem by providing a visual specification language that can be automatically translated to CTLK.

4.3 Live Sequence Charts

Live Sequence Charts (LSCs) are a graphical language that have been used to model as well as specify systems [34,5,19]. They extend the semantics of Message Sequence Charts (MSCs) to include the notion of liveness that allows the distinction between behaviors that must happen as opposed to behaviors that may happen. In this section we briefly introduce LSCs by discussing in detail the constructs of the LSC grammar and using an example to describe the semantics of the respective constructs. We refer the reader to [19] for a detailed presentation of the semantics of LSCs.

Figure 4.1 shows an example LSC for the Auction Protocol Without Alliance. The LSC information is described in the LSC header found at the very beginning of the LSC. The LSC grammar contains the following constructs to describe agent interactions and behaviors:
Agents or Instances: Agents are drawn with a rectangular *instance head* that denotes the start of the agent. A vertical line originating from the instance head signifies the *life-line* of the agent and ends in a filled rectangle, which terminates the respective agent. The example LSC describes the interaction between three bidder agents $A_1$, $A_2$, and $A_3$ and the environment agent, Environment.

Messages: Messages are the only form of communication between agents in the LSC. Each message has a sender and receiver agent attached to it. Messages are annotated with a message label that identifies the message. Messages can be simultaneous or asynchronous. Simultaneous messages are drawn with a solid arrow head and occur when both the sender and receiver are ready for the communication. Asynchronous messages are drawn with an open arrow head and can be sent/received at any time (we force the send event to occur before the receive event). In the example LSC all messages are simultaneous messages.

Locations: The life-line of each agent is marked with locations that are points where events and other constructs may be described. Locations are unique to each agent in the LSC. For example, the message Bid6 is sent from agent $A_1$ at location Loc2 to agent Environment at location Loc4. Locations for each agent start at location Loc1 and increment whenever a new event or construct is placed on the life-line of the respective agent.

Coregions: Coregions are drawn with a dashed vertical line next to the life-line of an agent. They describe behavior that has no particular order. For example, the messages Auc1, Auc2, and Auc3 occur in a coregion for the Environment agent and can be sent in any order. So it is possible for agent $A_2$ to receive the auction announcement before agent $A_1$.

Conditions: Conditions are placed in the chart by drawing hexagons around the life-lines of agents evaluating the condition. The condition label describes a predicate that must be satisfied at the current location(s) of the agent(s). Conditions
spanning multiple agent life-lines act as synchronizing points for the involved agents and the condition is not evaluated unless all the agents are at the respective condition locations. Conditions attached to a message are called *bonded conditions*. Conditions placed on their own location that are not attached to a message in the chart are called *non-bonded* conditions [30]. In our example LSC, the hexagon attached to the \texttt{Auc3} message is a bonded condition that evaluates the predicate \texttt{noAlliance}.

**Prechart:** The prechart is drawn with a dashed hexagon encompassing the instance heads and connects to the *main* body of the chart that is described in the solid rectangle following the prechart. The prechart describes the behaviors of the system under which the main body of the chart is to be activated/observed. For the example LSC shown in Figure 4.1, whenever the messages \texttt{Auc1}, \texttt{Auc2}, and \texttt{Auc3} are observed, the bidding and winner announcement must be observed in the future to satisfy the main chart.

**Main chart:** The main chart of the LSC is the behaviors described in the rectangle followed by the prechart. The main chart can be either *existential* or *universal*. Universal charts, drawn with a solid rectangle, specify behavior that must be satisfied by the system every time the prechart is satisfied. Existential charts are drawn with a dashed rectangle and specify behavior that the system must exhibit at least once when the prechart is satisfied, but not every time the prechart is satisfied. In the example LSC of Figure 4.1, the main chart is a universal chart.

**Temperatures:** Temperatures in LSCs can be assigned to messages, conditions, and locations. A *hot* temperature is depicted by using a solid line to draw the construct and specifies behavior that must be satisfied by the system. A *cold* temperature is drawn by using a dashed line for the construct and specifies behavior that may be satisfied. All constructs in the example LSC are hot constructs. Our research is currently restricted to hot constructs only. Since most specifications primarily deal
with hot constructs, this limitation does not impact the general applicability and usability of the results presented in this paper.

The chart also induces a natural partial order along each instance line from top to bottom. Instances evolve in the downward direction and are blocked if an event on their life-line does not occur. For the example chart shown in Figure 4.1, the prechart is satisfied (an instance is observed) when the Environment agent sends an auction announcement to each other agent (Auc1, Auc2, and Auc3). These announcements can occur in any order. Once the announcements have been received, it is necessarily the case that each agent places a bid by sending a message to the Environment agent (Bid6, Bid3, and Bid4). The Environment can receive the bids from A1 (bid amount 6), A2 (bid amount 3), and A3 (bid amount 4) in any order. Finally, after receiving all bids, the Environment agent announces the winner by sending the Winner1 message to agent A1 (highest bid amount of 6). We also define the maximal messages in a chart as the messages that occur on the last locations of the chart after which no other messages are defined in the chart. It is possible to have a chart end with multiple maximal messages (concurrency).

In our research, we deal with all the described constructs of LSCs with the following restrictions: (a) we impose the strict semantics of LSCs, which only allow a message to be sent/received once in a given chart (b) we only allow bonded conditions in the chart and (c) we only allow hot temperatures on any constructs.

4.4 Live Sequence Charts and Multi-Agent Systems

Multi-agent specifications critically depend on expressing temporal as well as knowledge based properties of the system. Temporal properties are important in order to express the interaction between the different agents in the system and knowledge properties are important in order to reason about the results/consequences of the
temporal events and interactions of the agents. We show how LSCs can be used to
describe temporal as well knowledge properties of multi-agent systems. We do this
by providing a complete translation of LSCs to the $\text{CTLK}$ temporal logic. First, we
show how LSCs can be translated to $\text{CTL}$, and then we show how the knowledge
operator of $\text{CTLK}$, $K_i$, can be embedded in LSCs to specify knowledge properties of
the agents.

4.4.1 Translating Live Sequence Charts to Temporal Logic

In this section we present an overview of the translation of LSCs to LTL as presented
in [37] and then prove that this formula can also be expressed in $\text{CTL}$. We adopt the
same set of restrictions as described in the previous section and in [37]. To conserve
space, we only present the translation of universal charts since the translation to
existential charts is similar in nature. Given a chart \( c \) (for example Figure 4.1), we define \( \text{msg}_p(c) \) to be the messages that are observed in the prechart (\{Auc1, Auc2, Auc3\}), \( \text{max}_p(c) \) to be the maximal messages in the prechart (in this case the same as \( \text{msg}_p(c) \)), \( \text{msg}_m(c) \) to be the messages observed in the main chart (\{Bid6, Bid4, Bid3\}) and \( \text{max}_m(c) \) to be the maximal messages in the main chart (\{Winner1\}).

The function \( \text{next}(c)(e) \) returns the set of events that may follow a given event \( e \) in a chart \( c \). The formula \( \phi_{x_i,x_j} = \neg x_j \cup x_i \) is used to specify that message \( x_j \) does not occur until message \( x_i \) and the formula \( \chi_{x_i,x_j} = (\neg x_i \land \neg x_j) \cup (x_i \land X ((\neg x_i \land \neg x_j) \cup x_i)) \) is used to specify that message \( x_i \) occurs twice before \( x_j \). The negation of \( \chi \) specifies that message \( x_i \) does not occur twice before \( x_j \). Additionally, we use the letters \( e \), \( p \) and \( m \) to denote the events of the entire chart, prechart and main chart respectively. For a universal chart \( c \), the equivalent LTL formula \( \psi_c^{\text{LTL}} \) has the following form [37]:

\[
\psi_c^{\text{LTL}} = G \left( \bigwedge_{e \in \text{msg}_p(c)} \phi'_{e,\text{next}(c)(e)} \land \bigwedge_{e \in \text{max}_p(c)} \phi'_{e,\text{msg}_m(c)} \land \bigwedge_{(e,p) \in \text{msg}_p(c) \times \text{max}_p(c)} \neg \chi_{e,p} \right) \Rightarrow \left( \bigwedge_{e \in \text{msg}_m(c)} \phi'_{e,\text{next}(c)(e)} \land \bigwedge_{(e,m) \in \text{msg}(c) \times \text{max}_m(c)} \neg \chi_{e,m} \right)
\]

Intuitively, the formula in Equation 4.1 expresses the total order that is induced by the chart while not allowing duplicate occurrences of the messages within a single instance of the chart. The left side of the implication describes the behaviors of the prechart, which if satisfied force the behaviors of the main chart to be satisfied as well (right side of implication). The framing is important since the prechart is the activation condition for the verification of the main chart events. If the prechart is never observed successfully, we do not need to verify events of the main chart.
The prechart formulas are satisfied if the events occur in the specified order (the \( \phi \) formulas) and do not occur more than once (the \( \chi \) formulas). The \( G \) operator surrounding the implication is used to verify all possible instances of the chart in the system. For the example LSC in Figure 4.1 the auction announcement to agents \( A_1 \), \( A_2 \) and \( A_3 \) may occur in any order for the prechart to be satisfied. The equivalent LTL for the prechart from the left side of the implication in Equation 4.1 is as follows:

\[
((\neg \text{Bid}6 \land \neg \text{Bid}4 \land \neg \text{Bid}3 \land \neg \text{Winner}1) \lor \text{Auc}1) \land ((\neg \text{Bid}6 \land \neg \text{Bid}4 \land \neg \text{Bid}3 \land \neg \text{Winner}1) \lor \text{Auc}2) \land ((\neg \text{Bid}6 \land \neg \text{Bid}4 \land \neg \text{Bid}3 \land \neg \text{Winner}1) \lor \text{Auc}3 \land \text{noAlliance}) \land ((\neg \text{Auc}1 \land \neg \text{Auc}2) \lor \text{Auc}1 \land X((\neg \text{Auc}1 \land \neg \text{Auc}2) \lor \text{Auc}1)) \land \ldots
\]

Similarly, an LTL expression for the right side of the implication is created to express the behaviors of the main chart. It should be noted that the predicates occurring in bonded conditions only appear on the right side of the until operator to check for their validity and are never included on the left side of the until operator when checking the absence of the message to which the condition is bonded. This is because condition predicates may be satisfied at other points in the system (they are open) and forcing them to be not satisfied (left side of until operator) may cause an unwanted violation and lead to a false positive error.

The size of the LTL formula generated from the chart using the translation presented in [37] results in at most a quadratic number of individual \( \phi \) and \( \chi \) formulas relative to the number of events specified in the chart. The total size is dictated by the number of maximal messages (\( \text{max}_m(e) \)) in the chart and directly affects the cost/complexity of the verification.

Since the CTLK logic [45] is CTL based, we show how an equivalent CTL formula can be obtained for the LTL translation of LSCs presented in Equation 4.1. By doing so, we can then extend the translation to CTLK. We show this
result in two stages. First, we construct an $\text{ACTL}^{\text{det}}$ (a subset of $\text{CTL}$) formula and then reduce the $\text{ACTL}^{\text{det}}$ formula to an LTL formula that is logically equivalent to the formula presented in Equation 4.1. To conserve space we omit the proof details but provide an intuitive description instead. Additionally, because of the similar nature of the universal and existential cases, we only provide the proof description for translating universal charts to $\text{CTL}$.

We first define $\text{ACTL}^{\text{det}}$ as follows:

**Definition ACTL$^{\text{det}}$**
- $p$ is a predicate
- For $\text{ACTL}^{\text{det}}$ formulas $\phi_1$ and $\phi_2$ and a predicate $p$:
  \[
  \phi_1 \land \phi_2, \ AX \phi_1, (p \land \phi_1) \lor (\neg p \land \phi_2), A(p \land \phi_1) U (\neg p \land \phi_2) \]
  are $\text{ACTL}^{\text{det}}$ formulas.

$\text{ACTL}^{\text{det}}$ is a subset of $\text{CTL}$ in which only the universal path quantifier is allowed and disjunctions can only be created using propositions that are in opposite forms (negative/positive) in each clause of the disjunction. Using this definition of $\text{ACTL}^{\text{det}}$, we construct the simpler $\phi$ and $\chi$ formulas of Equation 4.1. For example, we create the $\text{ACTL}^{\text{det}}$ formula $A(\neg \text{Winner1} U \text{Bid6})$ to specify the order of the $\text{Winner1}$ and $\text{Bid6}$ messages and $A((\neg \text{Bid6} \land \text{Winner1}) U (\text{Bid6} \land AX(A(\text{Bid6} \land \text{Winner1}) U \text{Bid6})))$ to specify that message $\text{Bid6}$ occurs twice before message $\text{Winner1}$. As mentioned earlier, the $\chi$ formula only provides us with a specification that a message $u$ occurs twice before a second message $v$. In order to specify that the message $u$ does not occur twice, we have to negate the $\chi$ formula. Since the $\text{ACTL}^{\text{det}}$ grammar only allows the negation of propositions and not formulas, to negate the $\chi$ formula, we first state a result that allows us to treat any $\text{ACTL}^{\text{det}}$ sub-formula as a proposition during the construction of the larger $\text{ACTL}^{\text{det}}$ formula.

**Lemma 4.4.1** Given a correct $\text{ACTL}^{\text{det}}$ labeling algorithm, any $\text{ACTL}^{\text{det}}$ sub-formula $\zeta^{\text{ACTL}^{\text{det}}}$ can be treated as an atomic proposition $\zeta^{\text{ACTL}^{\text{det}}}_{\text{prop}}$ after labeling.
**Proof** Proof by definition.

The proof of Lemma 4.4.1 follows by definition of the $\text{ACTL}^{\text{det}}$ labeling algorithm. Given a correct $\text{ACTL}^{\text{det}}$ labeling algorithm, a formula will only be labeled in a state if the formula is satisfied. The label is then used as a proposition itself for further verification of the rest of the formula. Using the result of Lemma 4.4.1 we can now treat individual $\chi$ formulas as propositions and negate the respective proposition to specify $\neg \chi$ formulas. Using the created $\phi$ and $\neg \chi$ formulas, we can now create the formulas for the left ($\psi^{\text{ACTL}^{\text{det}}}_{\text{pre}}$) and right side ($\psi^{\text{ACTL}^{\text{det}}}_{\text{main}}$) of the implication in Equation 4.1. Since the left and right sides of the implication in Equation 4.1 are a series of conjunctions of $\phi$ and $\neg \chi$ formulas, no special rules have to be applied to create the individual $\psi^{\text{ACTL}^{\text{det}}}_{\text{pre}}$ and $\psi^{\text{ACTL}^{\text{det}}}_{\text{main}}$ formulas.

Next, we create the larger implication by applying the result of Lemma 4.4.1 to negate the left side of the implication (by treating $\psi^{\text{ACTL}^{\text{det}}}_{\text{pre}}$ as a proposition) and combine it in a disjunction with the right side as follows:
\[\psi^\text{ACTL}_{det/c} \equiv \lnot \psi^\text{ACTL}_{pre} \\land \] 
\[\psi^\text{ACTL}_{det/c} \equiv (\lnot \psi^\text{ACTL}_{pre} \land true) \lor (\psi^\text{ACTL}_{main} \land \psi^\text{ACTL}_{pre}) \] 
\[\psi^\text{ACTL}_{det/c} \equiv (\psi^\text{ACTL}_{pre}) \to (\psi^\text{ACTL}_{main} \land \psi^\text{ACTL}_{pre}) \] 
\[\psi^\text{ACTL}_{det/c} \equiv \psi^\text{ACTL}_{pre} \to \psi^\text{ACTL}_{main} \]

We use the disjunction rules of the \textit{ACTL}^{det} grammar to create the disjunction between the left and right sides of the formula, which results in the implication. The larger \textit{ACTL}^{det} formula, \(\psi^\text{ACTL}_{det/c}\), is created with the same form as the LTL formula of Equation 4.1, the only difference being in the addition of universal quantifiers added to each temporal operator. The proof is presented for three messages and using induction is extended to an arbitrary number of messages (arbitrary size chart).

After constructing the \textit{ACTL}^{det} formula \(\psi^\text{ACTL}_{det/c}\), we utilize a result presented by Maidl [40] that establishes the existence of an LTL formula for any formula that is expressible in \textit{ACTL}^{det}:

\textbf{Theorem 4.4.2} For an \textit{ACTL} formula \(\phi\) there exists an LTL formula \(\psi\) which is equivalent to \(\phi\) iff \(\phi\) can be expressed in \textit{ACTL}^{det}.

The above result proves the existence of an LTL formula (\(\psi^\text{LTL}_{c}\) in this case) for any formula that can be expressed in \textit{ACTL}^{det} (\(\psi^\text{ACTL}_{det/c}\) in this case). To obtain the equivalent LTL formula from the \textit{ACTL}^{det} formula, we use a result presented by Clarke and Draghicesku [17] that allows us to create the LTL formula (\(\psi^\text{LTL}_{c}\)) from the existing \textit{ACTL}^{det} formula (\(\psi^\text{ACTL}_{det/c}\)). We summarize the result as follows:

\textbf{Theorem 4.4.3} For an \textit{ACTL}^{det} formula \(\phi\), the equivalent LTL formula \(\phi^d\) is obtained by removing all path quantifiers from \(\phi\).

The result of Theorem 4.4.3 allows us to remove all path quantifiers from the \textit{ACTL}^{det} formula \(\psi^\text{ACTL}_{det/c}\) to create the logically equivalent LTL formula \(\psi^\text{LTL}_{c}\).
Since the $\text{ACTL}^{\text{det}}$ formula $\psi^\text{ACTL}^{\text{det}}_c$ has the same form as the LTL formula of Equation 4.1, removing the path quantifiers gives us exactly the same formula as presented in Equation 4.1. The $\psi^\text{ACTL}^{\text{det}}_c$ formula can now be used as our $\text{CTL}$ formula for all verification purposes. We would like to point out that the use of Lemma 4.4.1 has no affect on the verification process. It is only used in the proof process to prove the translation of LSCs to $\text{CTL}$. For verification, the LSC specification is directly translated to $\text{CTL}^K$, and verification is performed by using the equivalent formula generated from the LSC specification. We now state one of the primary results of our paper that allows us to express the translation of LSCs to LTL in $\text{CTL}$ by adding a universal path quantifier to each temporal operator that is used in the LTL translation.

**Theorem 4.4.4** The LTL translation of LSCs ($\psi^{\text{LTL}}_c$) lies in the common fragment of $\text{CTL}$ and LTL and the equivalent formula can be obtained by adding the universal path quantifier to each temporal operator in the LTL formula of Equation 4.1.

Using this result, any chart can be translated to a $\text{CTL}$ formula that is at most quadratic in the size of the chart [37] as opposed to previous translations of LSCs to $\text{CTL}$ that were exponential in the size of the chart [35]. The resulting $\text{CTL}$ formula of LSCs forms the basis for verifying temporal events in multi-agent systems. The process for translating existential charts to $\text{CTL}$ is similar in nature but is not shown here to conserve space.

### 4.4.2 Knowledge Properties in Live Sequence Charts

Multi-agent specifications should provide the ability to specify temporal events as well as knowledge formulas of agents to completely specify system behaviors. Although LSCs are naturally suited to describing temporal events of systems, they do not provide any methods to specify knowledge properties. We show how knowledge properties can be visually specified in LSCs and translated to $\text{CTL}^K$ for verification.
We first provide an intuitive description of the knowledge operator as presented in the CTLK temporal logic [45]. The knowledge operator, written as $K_i \phi$, is specified in conjunction with a state $w$ and is interpreted as follows: for the specified agent $i$, all global states with local states of agent $i$ that look exactly the same as the specified state $w$ (including state $w$) should satisfy the formula $\phi$. An epistemic relation $\sim_i$ is used to check for states that are equivalent to each other for the specified agent $i$. If the knowledge formula is true, each state that is epistemically related to state $w$ receives the label.

Before we discuss how to embed knowledge formulas in LSCs, we state a result that allows us to treat any knowledge formula specified in CTLK as an atomic proposition and further allows us to embed knowledge formulas in LSCs.

**Lemma 4.4.5** Given a correct CTLK labeling algorithm, any CTLK knowledge sub-formula $\eta_{CTLK}$ can be treated as an atomic proposition $\eta^{prop}$ after labeling.

**Proof** Proof by definition.

The proof of Lemma 4.4.5 follows by definition of the CTLK labeling algorithm. The primary difference between Lemma 4.4.1 and Lemma 4.4.5 is the temporal logic for which the result is stated. Using the result above, we now show how knowledge formulas can be embedded in LSCs.

Figure 4.4.2 shows two methods for embedding knowledge properties in LSCs. In Figure 4.4.2(a), a knowledge property is specified for each agent. Agent $A_1$ is specified to know $\phi_1$ when message $a$ is sent to agent $A_2$ and forces the predicate $pred$ to be satisfied as well. Agent $A_2$ knows $\phi_2$ when message $b$ is sent to agent $A_1$. The condition attached to each message is used to specify the knowledge properties. By utilizing the result of Lemma 4.4.5, we can now treat the knowledge formula embedded in the condition as a proposition. The translation to CTLK is performed by treating the knowledge operator as a predicate and including it in the translation.
of the bonded condition as described in the previous section. As mentioned earlier, the predicates in bonded conditions do not occur on the left side of an until operator in negative form. They only appear on the right side of the until operator when their existence is enforced along with the event they are bonded to. This is to ensure that the absence of knowledge is not enforced in states where it could possibly hold and avoid false positive error reports.

Figure 4.4.2(b) shows how knowledge can be embedded for a set of agents $\Gamma$. In this example, the agents $A_1$, $A_2$ and $A_3$ are part of the set $\Gamma$ and believe some knowledge $\phi_\Gamma$ when message $x$ is sent from agent $A_1$ to agent $A_2$. Again, we use the result of Lemma 4.4.5 and treat the knowledge formula as a proposition that is evaluated by the $\text{CTLK}$ labeling algorithm during verification, which facilitates verification and embedding of knowledge formulas in LSCs.

Knowledge formulas specified in the bonded conditions can also be nested knowledge formulas which specify the knowledge of agent $i$ w.r.t. the knowledge of agent $j$. For example, agent $i$ may specify $K_i(K_j\phi)$ as the knowledge formula in a condition as well. The nesting of the knowledge operators within the formula is not visually captured in the LSC.

**4.5 Case Study: Auction Protocol**

We now present a case study of applying LSCs to specify multi-agent systems. We first present a description of the auction protocol as described in [9, 8]. After the
description, we present LSCs that specify behaviors of the system and the equivalent
CTLK formulas that are generated from the LSC.

Auction Protocol: A simple auction environment Environment announces ten
bids to the three agents A1, A2, and A3. Agent A1 is a simple agent that always bids a
value of 6. Agent A2 initially bids a value of 3 and agent A3 bids a value of 4. In this
case agent A1 always wins the auction. After the first T auctions (T independently
set by A3), agent A3 creates an alliance with agent A2. Once the alliance is created,
agent A3 bids on behalf of agent A2 and A3 by placing a bid value of 7. A2 places a
bid value of 0 when an alliance has been created. After an alliance has been created,
agent A3 always wins the remaining auctions.

We specify the following properties on our system:

1. The Environment knows that agent A1 is the winner in all auctions where each
agent places an individual bid (Property 1)

2. If an alliance has been created between agents A2 and A3, agent A3 knows the
winner is A3 (Property 2)

3. An alliance will be created when the current bid number exceeds the threshold
of agent A3 (Property 3).

The LSC shown in Figure 4.1 describes the interaction and knowledge prop-
erties of Property 1. Whenever an auction is announced, it is necessarily true that
agent A1 will bid a value of 6, agent A2 will bid a value of 3 and agent A3 will bid a
value of 4. After the bidding is complete, the Environment will inform A1 that it has
won the auction. To embed the knowledge of the Environment when the winner is
announced, we have to introduce a second bonded condition with the Winner1 mes-
sage that checks for the formula $K_{\text{Environment}}(\text{Winner}_{A1})$ (not shown in figure). We
use a universal chart to specify the property because this behavior must always be
satisfied when no alliance exists. . The corresponding CTLK formula is as follows:
Figure 4.3: LSC describing the auction protocol with an alliance.

\[\begin{align*}
AG(A(\neg Bid_6 \land \neg Bid_3 \land \neg Bid_4 \land \neg Winner_1) U Auc_1) \\
A(\neg Bid_6 \land \neg Bid_3 \land \neg Bid_4 \land \neg Winner_1) U Auc_2 \\
A(\neg Bid_6 \land \neg Bid_3 \land \neg Bid_4 \land \neg Winner_1) U Auc_3 \\
\Rightarrow A(\neg Winner_1 U Bid_6) \land A(\neg Winner_1 U Bid_3) \land A(\neg Winner_1 U Bid_4) \\
A(true U (Winner_1 \land K_{Env} Winner_{A1})) \land \neg \chi(Auc_2, Winner_1) \\
\neg \chi(Bid_6, Winner_1) \land \neg \chi(Bid_4, Winner_1) \land \neg \chi(Bid_3, Winner_1) \\
\neg \chi(Auc_3, Winner_1)
\end{align*}\]
Figure 4.3 shows the LSC for **Property 2**. In this case when the **Environment** sends the auction announcement to each agent and if agents **A3** and **A2** know that an alliance has been created, the bidding values of agents **A2** and **A3** change according to their alliance agreement. Agent **A3** bids a value of 7 which is the sum of the individual bid values of agent **A2** and **A3**. Agent **A2** bids a value of 0 in this case. Also, it must necessarily be true that agent **A3** knows itself to be the winner after the bids have completed and the winner has been announced. We use a universal chart to expresses the behavior in this case since this behavior must be observed every time an auction is announced and agent **A3** knows an alliance has been created.

Figure 4.4 shows a chart that specifies **Property 3**: the creation of an alliance between agent **A2** and **A3**. After sending its normal bid (Bid4) to the **Environment** if agent **A3** knows that the bid number has exceeded its threshold, agent **A3** approaches **A2** and requests an alliance with the **Alliance**$_{req}$ message. Agent **A2** must respond with an acknowledgement **Alliance**$_{ack}$ to **A3** and at this point both agents know that an alliance has been created. We use an existential chart to specify this property because it is the case that this behavior must hold at least once in the chart but not
every time agent \(A3\) believes the threshold has been exceeded. To conserve space, we do not show the translation to \(\text{CTLK}\). Verification of this property on our system resulted in true as well.

To test the formulas generated from our LSC specifications, we create a model of the auction protocol using the MCMAS input language \([45]\). We verify each formula generated for the three properties listed above. The model initially contained errors that resulted in the failure of properties 2 and 3. After fixing the errors in the model, all properties (equivalent formulas) passed verification. The total time to verify all three formulas was 0.48s on a 2.16 GHz Intel Core 2 Duo processor with 1 GB memory.

### 4.6 Embedding Arbitrary Temporal Logic Properties in LSCs

We have presented a technique that enables us to embed knowledge formulas from the \(\text{CTLK}\) temporal logic in bonded conditions of LSCs. We now extend this result to arbitrary temporal logics. For example, given a correct \(\text{CTL}\) labeling algorithm, we can extend the result of Lemma 4.4.1 and Lemma 4.4.5 to treat any \(\text{CTL}\) formula as a proposition. By doing so, an arbitrary \(\text{CTL}\) formula can be embedded in a bonded condition in an LSC and used for verification of a system under test. The generalized result can be stated as follows:

**Theorem 4.6.1** Given a correct labeling algorithm for a temporal logic \(\Upsilon\), any arbitrary formula \(\psi^\Upsilon\) of the \(\Upsilon\) logic can be treated as a proposition \(\psi^{\Upsilon\text{prop}}\) and can be embedded in an LSC using bonded conditions.

**Proof** Proof follows by definition.

The proof of this result follows by definition of the temporal logic and the labeling algorithm for the temporal logic. It should be noted that the embedded
formula must always be in the same logic as the logic the chart is translated to. Additionally, this result relies critically on the labeling algorithm of the temporal logic. If a labeling algorithm is not used (automata theoretic algorithm etc.), the result is inapplicable.

4.7 Conclusions and Future Work

We have demonstrated how knowledge formulas can be embedded in LSCs and further show how knowledge embedded LSCs can be translated to the \( \text{CTLK} \) logic for verification of multi-agent systems. We do this by providing an LSC to \( \text{CTL} \) translation, which we extend to include the knowledge operator of \( \text{CTLK} \). We then perform a case study of the auction protocol to specify different properties of the system and verify an implementation using the formulas translated from LSCs. The case study effectively demonstrates how complex temporal interactions of multi-agent systems as well as epistemic properties can be specified with relative ease using LSCs. Additionally, we generalize our approach of embedding knowledge formulas in LSCs to arbitrary temporal logics that enable us to embed arbitrary formulas in the LSC. We are now investigating the possibility of creating an automata structure from the LSC that can be used for automatic verification of multi-agent systems. Our primary motivation for creating an automata from the LSC is to support a wider range of constructs in the LSC grammar as well as improve performance. We are also currently investigating specification of data aspects of a system within the LSC.
Chapter 5

Improving Live Sequence Chart to Automata Transformation for Verification

Abstract

This paper presents a Live Sequence Chart (LSC) to automata transformation algorithm that enables the verification of communication protocol implementations. Using this LSC to automata transformation a communication protocol implementation can be verified using a single verification run as opposed to previous techniques that rely on a three stage verification approach. The novelty and simplicity of the transformation algorithm lies in its placement of accept states in the automata generated from the LSC. We present in detail an example of the transformation as well as the transformation algorithm. Further, we present a detailed analysis and an empirical study comparing the verification strategy to earlier work to show the benefits of the improved transformation algorithm.
5.1 Introduction

Current trends in system development are shifting towards creating and developing larger systems using several smaller communicating sub-systems. With the increasing popularity of such modular designs comes the burden of creating, implementing, and testing the implemented communication protocols. Specification of communication protocols has been explored significantly in the past. English, which has been traditionally used as the most common language for specifying protocols, lacks the formal rigor and preciseness needed for clarity. Viable alternatives are formal specification languages such as UML, Message Sequence Charts (MSCs) and Live Sequence Charts (LSCs) [29, 19, 10]. The evolution of these graphical languages has led to their application to modeling and specifying communication behaviors in a variety of different domains [7, 34, 20]. Other research has also investigated the automatic synthesis of systems from LSCs as well as the verification and validation of requirements on the LSCs themselves [24, 1, 47]. Efficient methodologies for using these graphical languages in a formal verification environment provide the support in the development process to completely certify, test and develop a system. Since LSCs are a more expressive and semantically rich visual specification language compared to MSCs, Timing Diagrams and Sequence Diagrams in UML, we focus on techniques related to LSCs. Due to the encompassing nature of LSCs, the techniques and algorithms presented in this paper are also applicable to the afore mentioned specification languages.

Previous work in [31, 30] presents a strategy to verify systems against LSC specifications by transforming the LSC to a positive automaton. We use the term positive automaton to denote automaton that witness chart completions. With the positive automaton, a system is verified against the LSC in three stages: reachability analysis for detecting safety violations, ACTL verification for detecting liveness errors, and finally, if the first two steps fail to provide a significant result, full LTL verification is required to completely verify the system. The authors argue that the
verification algorithms are applied in increasing order of cost and for certain subclasses of LSCs not all algorithms need to be applied, which can eventually save on the total verification cost. Although the approach presented in [31] is sound, it has several drawbacks. For any arbitrary LSC, the approach at a minimum has to apply reachability analysis as well as ACTL model checking for verifying the safety and liveness properties of the system against the LSC. In the worst case, LTL verification is required to completely verify the system, which was shown to be impractical for LSC verification [37]. Another drawback of the verification approach is the specialized algorithms and tools that have to be created to perform the verification, which limit the general applicability and acceptance of the approach. The approach presented in this paper only requires one verification algorithm of the same cost as reachability analysis to completely verify a system against any arbitrary LSC.

We present a direct and obvious transformation of the LSC to a negative automaton by changing the placement of accept states. We use the term negative automaton to denote automaton that witness chart violations as opposed to chart completions. Using this improved LSC to automaton transformation a system can be formally verified against the LSC specification by performing only language containment on the parallel composition of the system automaton and the negative automaton of the LSC. Additionally, this approach does not require the use of customized algorithms and tools to verify a system against a specification. Using our new LSC to automaton transformation, we verify systems against larger more concurrent LSCs that were previously not verifiable with direct LSC to LTL or LSC to positive automaton transformations.

The structure of the paper is as follows. Section 5.2 presents a brief introduction to LSCs and an overview of the basic LSC to automaton transformation algorithm as described in [30]. Section 5.3 discusses in detail an example of using our approach for verifying a system against an LSC. This example will be used for the
remainder of the paper as well. Section 5.4 discusses the details of the transformation algorithm and presents the theoretical results to prove the correctness of the transformation algorithm. Section 5.5 presents an analysis of the improved transformation compared to the old transformation presented in [30]. Section 5.6 presents a subset of the results using the improved verification approach in both symbolic and explicit state model checkers. Finally, Section 5.7 discusses the conclusions and future work. Proofs, details and additional results can be found in the long version of the paper at http://vv.cs.byu.edu/~raham/lsc2automata.pdf.

5.2 LSC Overview

We briefly introduce some constructs of the LSC grammar. Figure 5.1(a) shows an example LSC where an idle node in a compute cluster requests and processes a job from the scheduler’s queue with a possible implementation of the Node and DB process in Figure 5.1(b). There are three processes in the example LSC: Scheduler, Node and DB. Each process is drawn with a rectangular instance head and a vertical life-line originating from each instance head. The life-line represents the time dimension in the LSC with time progressing in the downward direction. Communication between processes occurs via messages with the arrows representing the direction of communication. The idle message is an example of a synchronous message (filled arrowhead) where both the sender and receiver have to be ready for the message to be observed. The actual message communication occurs instantaneously for the sender and receiver. The result message is an example of an asynchronous message (unfilled arrowhead) where the sender does not have to block for the receiver to be ready to receive the message. The send event is written as result! and the receive event is written as result?. The example LSC also contains a cold non-bonded condition (second dashed hexagon) which enforces the validID predicate after a jobID

\footnote{See [19, 10] for details.}
has been received from the *Scheduler*. If the condition is violated, the *Node* process exits the chart. On each life-line any point where a condition or an event occurs is referred to as a *location*. Locations are unique to each life-line and in our research are represented by numbers next to the instance life-line. By default all locations are *hot* or *mandatory* locations unless specified otherwise using a dashed line for the life-line. The location for receiving the *result* message in the *Scheduler* life-line is the only cold location in the example chart. The behavior specified on a cold location is not mandatory, which implies that the *result* message may or may not be received by the *Scheduler*. Finally, behaviors described by the LSC are partitioned into the *pre-chart* (dashed hexagon before solid rectangle) and the *main chart* (rectangle after pre-chart). The pre-chart specifies the activation condition of the LSC and the main chart describes the behavior which must follow the pre-chart. In the example LSC, the main chart is a *universal* main chart (solid line), which represents behaviors that have to be observed every time the pre-chart is satisfied.

In addition to the constructs shown in the example LSC, several other constructs are also available. The main chart can be specified as an *existential* chart (drawn with a dashed rectangle) that specifies behavior the system must satisfy at least once when the pre-chart is satisfied (as opposed to every time the pre-chart is satisfied). Conditions if attached to another event are *bonded* otherwise *non-bonded*. By attaching conditions to other events, the condition is evaluated at the exact moment the bonded event occurs, as opposed to non-bonded conditions where the condition is continuously evaluated until satisfied. LSCs also allow the specification of *invariants* which are conditions spanning over multiple events in the LSC. *Co-regions* specified with a dashed line parallel to a life-line allow events to occur in any order. For example, if the messages *getData* and *data* are specified in a co-region, either message *data* or *getData* may occur first. It is only necessary for all events in a co-region to occur. Finally, conditions, messages, and locations may be specified as *hot* or *cold*.
Our method supports all the mentioned constructs of LSCs with the following commonly accepted restrictions. First, we adopt the strict interpretation of LSCs (i.e., no duplicate message instances are allowed within a chart). Second, the LSC and all charts within the LSC are to be acyclic. Third, we also do not consider overlapping LSCs or iterative LSCs (Kleene stars) where multiple instances of the chart may be executed simultaneously. Since most scenario-based specifications in general do not deal with the constructs omitted from this research, the restrictions do not affect the general applicability of our results.
5.2.1 Transforming Live Sequence Charts to Automata

Past research in the area of transforming LSCs to automaton has primarily revolved around the generation of positive automaton that detect chart completions [30, 24, 6, 33]. Work in [30] gives a detailed presentation of the algorithm to transform an LSC to positive automaton. We present an overview of this algorithm followed by a discussion of some key aspects of the algorithm.

The LSC to automaton unwinding algorithm explores all possible inter-leavings of the events defined in the LSC starting from the top and ending at the bottom of each life-line in the chart. The possible event inter-leavings are explored by considering the partial order induced by the semantics of the LSC. The partial order of the chart dictates that the locations in each instance are totally ordered unless part of a co-region; thus, implying that each instance has to progress linearly from top to bottom. For example, in the chart shown in Figure 5.1(a), instance Node cannot move from location 1 to location 4. From location 1, Node has to move to the next logical location: location 2. To maintain the current state of the LSC, we define a cut as a set of locations in the chart with exactly one location for each instance. The cut is used to record the current state of the chart and create successor cuts. The reachable set of cuts from the initial cut is the automaton for the chart. Each state of the automaton corresponds to a reachable cut of the chart. Successor cuts are generated using the set of enabled transitions for a given cut. The initial cut for all charts is created by placing each instance at its first location, (0, 0, 0), where the first, second and third locations correspond to the locations for the Scheduler, Node and DB instances.

The enabled set of transitions for a cut is created using the chart semantics. For example, a synchronous message is enabled if both the sender and receiver of the message are at their respective send and receive locations. In our chart, the message idle is observed if the Scheduler and Node instances are each at locations
At the initial cut, (0, 0, 0), the *idle* message is enabled. On the other hand, since the *Node* is not at location 3, the *getData* message is not enabled in the initial cut, even though the *DB* is at location 0. When the *idle* message is explored from the enabled set, a successor cut is generated where the locations for the involved instances have been updated. In this case, the locations for the *Node* and *Scheduler* instances are updated to their next logical location giving us the successor cut (1, 1, 0). At the cut (1, 1, 0), the *jobID* message is enabled, which leads to the cut (2, 2, 0).

Asynchronous sends are enabled by default when the corresponding instance is at the send location and asynchronous receives are enabled only if the corresponding send event has occurred and the receiving instance is at the receive location. Conditions act as a synchronization point where each participating instance should be at its respective condition location for the condition to be evaluated. A full description of these semantics can be found in [30]. Multiple enabled transitions lead to multiple successor cuts from the given cut representing the concurrency in the chart.

Using the chart semantics, successor cuts are generated from the initial cut and each unique cut is processed until the final cut is reached where each instance is at the bottom of its life-line. Each unique cut of the chart corresponds to a state in the final automaton. The initial cut (0, 0, 0) corresponds to state $q_0$ in Figure 5.2(a). The successor cut (1, 1, 0) corresponds to the state $q_1$ where the *idle* message has already been observed and the next message to be observed is *jobID*. Cut (2, 2, 0) corresponds to state $q_2$ and the final cut corresponds to state $q_7$ where no further events are to be observed. Notice that transitions taken to generate successor cuts correspond to the transition labels in the automaton.

Finally, to create the positive automaton from the LSC, states corresponding to legal exits of the chart are marked as accept states. For example, state $q_7$ in Figure 5.2(a) is marked as an accept state because it corresponds to the final cut of the LSC which represents a legal completion of the chart. Additionally, state $q_6$ is also
marked as an accept state since it corresponds to the cut where the cold message

result does not have to be received.

From the automaton in Figure 5.2(a) we also notice that state $q_2$, where cold
condition validID occurs is non-deterministic. This non-determinism is a result of the
adopted semantics of cold conditions in [30]. If validID is not satisfied, the automaton
can either stay in state $q_2$ and wait for the condition to be satisfied or move to the
exit state $q_{\text{exit}}$ to signify that the cold condition was not satisfied and the chart
has exited successfully. This non-determinism resulting from non-bonded conditions
forces the approach of [30, 31] to translate the LSC automaton to an LTL property
and re-perform the verification using the LTL property, which has been shown to
be ineffective for even moderate size charts due to the size of the resulting LTL
formula [37].

5.3 Transformation and Verification Example

We use the automaton produced by the unwinding algorithm discussed earlier as our
initial automaton. The initial automaton from the unwinding algorithm is shown
in Figure 5.2(a). We transform this positive automaton to a negative automaton that
can be used in our single pass verification approach. Figure 5.2(b) and (c) show the
transformed negative automaton.

Our approach transforms the LSC chart to a negative automaton capable of
detecting chart violations (as opposed to chart completions) that is naturally suited
for verifying systems using language containment. The first step in the transformation
process is to remove all the accept labels from the automaton. Next the exit state $q_{\text{exit}}$
and any transitions leading to the exit state are removed from the initial automaton.
In our example of Figure 5.2(a) we remove the transition from state $q_2$ to state $q_{\text{exit}}$,
which also removes the non-determinism from the automaton arising from the non-
bonded condition. The algorithm then introduces safety transitions (dashed edges
in Figure 5.2(b)) from all states that contain a transition belonging to the main chart to the safety state $q_{safety}$. The safety state is an accept state introduced in the automaton to capture all safety violations in the system. It has a single outgoing transition to itself predicated on $true$. The safety transitions enable the detection of safety violations which consist of duplicate messages (messages that have been observed before) and out of order messages in states that correspond to main chart states. For example, in state $q_1$ of Figure 5.2(b), the only legal transition is if the $jobID$ message is observed. Since $jobID$ is a main chart transition, state $q_1$ corresponds to a main chart state and a safety transition is introduced. The safety transition $idle \lor getData \lor data \lor result! \lor result?$ from state $q_1$ to $q_{safety}$ is taken if any message except $jobID$ is observed.

After the introduction of safety transitions, the algorithm updates the self-loops on each state (dotted edges in Figure 5.2(b)). The self-loops enable the automaton to remain in a given state until an event forcing progress is observed. For example, in the automaton shown in Figure 5.2(b), state $q_4$ has a self-loop, $¬idle \land ¬jobID \land ¬getData \land ¬data \land ¬result! \land ¬result?$, that is taken until the $data$ message is observed, which moves the automaton to the next state $q_5$. The only exception is the self-loop for the first state and the final state. The first state $q_0$ contains a self-loop with the $true$ annotation to capture all possible future instances (and possible errors) of the chart in a reactive system. The final state does not have any self-loops. This is because the final state represents the successful completion of the chart and no further errors are possible unless a new chart instance is observed, which is detected in the first state.

Finally, the algorithm marks illegal end points of the main chart as accept states to facilitate detection of chart violations. For example, state $q_1$ in Figure 5.2(b) is at the beginning of the main chart where the message $jobID$ is yet to be received. If the $jobID$ message is never observed, the automaton remains in state $q_1$ indefinitely,
which should be reported as an error. To report this error, state $q_1$ is marked as an accept state. States containing no transitions corresponding to hot constructs in the main chart are not marked as accept states. For example, in Figure 5.2(b), state $q_2$ is not marked as an accept state because the $validID$ condition is a cold condition, and its absence does not result in an error. State $q_0$ is not marked as an accept state either because it does not contain any outgoing transitions corresponding to a hot construct in the main chart. If the $idle$ message is never observed, the pre-chart is not satisfied, which is not a violation of the specification. State $q_6$ is not marked as an accept state since the location of the $result?$ event is cold implying that the $result?$ event does not have to be observed. Finally, state $q_7$ in Figure 5.2(b) is also not marked as an accept state since it is the final state where the behavior as described in the universal chart has been satisfied without errors.

Verification of the system is performed by first creating the system automaton in the usual manner. We verify the parallel composition of the system automaton and the negative automaton of the LSC by searching the behavior space of the intersection for accepting cycles. Any cycles detected correspond to errors in the system. Figure 5.1(b) shows a possible implementation of the $Node$ and $DB$ processes in a cluster. The $Scheduler$ process has not been shown in the implementation but is assumed to be correctly implemented. When idle, the $Node$ process requests a job from the scheduler (line 2). The $Node$ process then waits to receive the $jobID$ and validates the $jobID$ using the predicate $validID$ (lines 3 - 5). Next, the $Node$ process requests data from the $DB$ (line 6), processes the data and sends the result to the $Scheduler$ (lines 7 - 8). The $DB$ process receives and processes messages as they arrive (lines 12 - 19). In this particular implementation, the $DB$ process is erroneous because it never receives/processes the $getData$ message from the $Node$. Since the $getData$ message is a synchronous message and the $DB$ process is never ready to receive the $getData$ message, the $Node$ and $DB$ processes never progress even though they should. Verification
Figure 5.2: The initial and transformed automaton for the example LSC shown in Figure 5.1(a). (a) the initial automaton (b) the transformed automaton and (c) list of transition labels.

of the parallel composition of the system automaton (not shown) with the property automaton in Figure 5.2(b) produces the word \((idle, jobID, validID, (\neg getData)\omega)\), with the corresponding trace: \((q_0, q_1, q_2, (q_3)^\omega)\), where \(\omega\) indicates infinite repetition.

Since \(q_3\) is marked as an accept state, the trace is reported as an accepting cycle and the violation has been discovered. Using the positive automaton in the verification approach of [31] requires two verification runs of comparable complexity to detect the same violation.
5.4 Transformation and Verification Details

The transformation presented in this work is based on language containment and automata theory. We use Symbolic automata, an extension of Büchi automata, that allows observing any of a possible set of inputs on an edge. Formally Symbolic automata are given by $A = \langle \Sigma, Q, \Delta, q^0, F \rangle$ where, $\Sigma$ is the finite alphabet of input symbols (variables), $Q$ is the finite set of states, $q^0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final/accepting states, and $\Delta \subseteq Q \times \phi \times Q$ is the transition relation. A transition $(q, \phi, q') \in \Delta$ represents the change from state $q$ to state $q'$ when the formula $\phi$ is satisfied.

We partition the set of Boolean variables $\Sigma$ into three distinct sets $\Sigma_{msgs}$, $\Sigma_{invariants}$, and $\Sigma_{conditions}$, that contain the Boolean variables that are used for messages, invariants and conditions in the chart respectively. For the chart shown in Figure 5.1(a), $\Sigma_{msgs} = \{idle, jobID, getData, data, result?, result!\}$ and $\Sigma_{conditions} = \{validID\}$. The set $\Sigma_{main} = \{jobID, validID, data, getData, result?, result!\}$ is the set of Boolean variables that are used in the main chart only. We also have a set $\Delta_{hot} \subseteq \Delta$ which only contains transitions that correspond to hot constructs in the chart (hot messages, hot conditions etc.).

For a set of Boolean functions $\Gamma = \{\phi_0, \phi_1, ..., \phi_n\}$ we define the function $\text{disjunct}(\Gamma)$ which returns the disjunct of the individual formulas in $\Gamma$ and the function $\text{conjunct}(\Gamma)$ which returns the conjunction of the individual formulas in $\Gamma$. The function $f(\Sigma, \phi) = \{\sigma | \sigma \in \Sigma \text{ and } \sigma \text{ or } \neg \sigma \text{ appears in } \phi\}$ returns the set of Boolean variables from $\Sigma$ that appear in $\phi$ in either a positive or negative form. For example, if $\phi = idle \land validID$, $f(\Sigma_{msgs}, \phi) = \{idle\}$ and $f(\Sigma_{condition}, \phi) = \{validID\}$.

We take as input the automaton structure for a chart in the form of a symbolic automata structure, $A$, with an empty final state set. Intuitively, to capture the bad behaviors of a chart, we transform the basic automaton structure to the negative automaton that is capable of detecting safety and liveness errors by yielding accepting
Figure 5.3: A generic state in the transformed automaton with complete annotations for all types of outgoing transitions. 1. \( \phi_{\text{self}} \): self-loop for non-progress, 2. \( \phi_{\text{safety}} \): transition to state \( q_{\text{safety}} \) for detecting safety errors, and 3. \( \phi_{\text{child}} \) transitions to the successor states.

cycles in the verification. We do so by adding accept states to the automaton and adding/updating all transitions.

Figure 5.3 shows an intuitive description of the outgoing transitions of a state in the transformed automaton. The sets \( \psi_c, \psi_m \) and \( \psi_i \) (initialized by the algorithm in Figure 5.4) are sets of condition, message, and invariant letters used in the outgoing transitions of a given state. There are three types of transitions that are introduced/upated for each state in the automaton. The \( \phi_{\text{safety}} \) transition (dashed edge) leads to the safety state and is responsible for detecting any safety errors. The self-loop (dotted edge), \( \phi_{\text{self}} \), enables the automaton to remain in the current state until an event or condition progresses the automaton to a successor state. The \( \phi_{\text{child}} \) transitions (solid edges) lead to the successor states. The dash-dot edge is only added to the first state of the automaton to enable verification of multiple chart instances in a reactive system.

States are marked as accept states in the automaton based on two criteria. First, the safety state is marked as an accept state for detecting safety violations such as duplicate message instances and out of order messages. Second, any state
that is not a legal exit point of the chart is marked as an accept state. We now discuss in detail the creation of the transitions and the marking of accept states.

Figure 5.4 shows the algorithm for transforming the input automaton. We only present an overview of the algorithm in this version of the paper and refer the reader to the long version for more details. The algorithm has a general Depth First Search (DFS) structure with line 4 enumerating the successors and line 11 making a recursive call for each successor. The algorithm is always invoked for the one initial state of the input automaton to be transformed. Lines 1 - 2 remove any transitions to the exit state $q_{\text{exit}}$. In the automaton shown in Figure 5.2(a), the transition from state $q_2$ to the exit state $q_{\text{exit}}$ is removed. Lines 5 - 7 of the algorithm build the sets of variables that are used for messages, invariants, and conditions in the transitions from the current state to the successor states. Lines 8 - 10 update the transitions to the successor states by first removing the transition and adding a new transition with the updated label. The updated child transition ensures that only the enabled messages, invariants and conditions at a given state can enforce progress in the automaton. For example, the algorithm transforms the transition from state $q_1$ to state $q_2$ in Figure 5.2(a) from $\phi = \text{jobID}$ to $\phi_{\text{child}} = \text{jobID} \land \neg \text{idle} \land \neg \text{getData} \land \neg \text{data} \land \neg \text{result}! \land \neg \text{result}?$. Lines 12 - 15 update the self-loop for the current state to ensure that the automaton remains in the current state if no relevant messages are observed. For example, in state $q_1$ of Figure 5.2(b), the self-loop $\neg \text{idle} \land \neg \text{jobID} \land \neg \text{getData} \land \neg \text{data} \land \neg \text{result}? \land \neg \text{result}!$ is enabled if no message is observed. As mentioned earlier, the first state of the automaton has a self-loop with the true label and the final state of the automaton has no self-loops. These special cases are not shown in the transformation algorithm in Figure 5.4.

If the current state $q$ contains a main chart transition (labels of transitions to successor states are members of the main chart alphabet $\Sigma_{\text{main}}$), then lines 16
- 18 of the algorithm add a safety transition to the safety state $q_{\text{safety}}$. The safety transition enables the automaton to detect message order violations or duplicate messages. For the automaton shown in Figure 5.2(a), state $q_1$ contains a single transition for the $jobID$ message. Since $jobID$ is a member of the main chart alphabet ($jobID \in \Sigma_{\text{main}}$) a safety transition needs to be added. The safety transition for state $q_1$, $idle \lor getData \lor data \lor result? \lor result!$, detects the presence of any message except the one allowed message $jobID$. Because states with no main chart transitions can not violate the chart, no safety transitions are added to them.

Lines 19 - 20 of the algorithm label the current state as an accept state if it belongs to the main chart and contains a hot outgoing transition. The check for main chart transitions is performed on line 16. To check for hot outgoing transitions, each outgoing transition is checked for membership in the $\Delta_{\text{hot}}$ set (line 19). If all outgoing transitions from a state are cold, the state is not marked as an accept state. In our example, for state $q_2$, the only outgoing transition corresponds to a cold condition and is not part of the $\Delta_{\text{hot}}$ set; thus, state $q_2$ is not marked as an accept state. On the other hand state $q_1$ is marked as an accept state because it has one successor transition that corresponds to the hot message $jobID$.

We now state the theoretical results of the presented transformation. We first show that for any main chart state in the automaton at least one transition is enabled for any arbitrary input (i.e. the transition relation for main chart states is total). Having enabled transitions guarantees that the automaton does not ignore any inputs which could cause violations or progress in the chart. To conserve space, all proofs have been omitted from this version of the paper but are available in the long version of the paper.

**Lemma 5.4.1** For all states containing outgoing main chart transitions, the transition relation is total. Formally, given a state $q$ with a main chart transition

$$\left( \forall \phi_i, q_1 : (q, \phi_i, q_1) \in \Delta \phi_i \right) = \text{true}.$$
Lemma D.1.1 is only applicable to states containing main chart transitions. Regarding states that do not contain main chart transitions, the safety transition $\phi_{safety}$ is not added, resulting in an incomplete transition relation. Since these states are responsible for detecting the completion of the pre-chart and not for detecting violations or errors, the incompleteness of the transition relation does not affect the correctness of observing the pre-chart. Our next result states that for all states except the first state of the automaton, the transition relation is deterministic. The transformed automaton is non-deterministic only in the first state (self-loop annotated with $true$) to accommodate for the global verification of every possible instance of the chart in the system. Non-deterministic automata as used in [31] result in error traces that have to be validated using full LTL verification, which has been shown to be impractical for LSCs [37]. Using deterministic automata guarantees that any reported errors are in fact valid errors in the system.

**Lemma 5.4.2** For states $q$ in the transformed automaton (except the initial state), the transition relation is deterministic. Formally, $\forall q \in Q, \forall \phi_i, \phi_j : (q, \phi_i, q_i) \in \Delta \land (q, \phi_j, q_j) \in \Delta, (\phi_i \land \phi_j) = false$.

The above result guarantees that for any given input to the transformed automaton (except the first and last state) exactly one transition is ever enabled. We now state our primary result for the transformed automaton. Intuitively, we show by application of Lemma D.1.1 and Lemma D.1.2 that the transformed automaton accepts only those words that are not accepted by the LSC and is capable of detecting all behaviors in a system that violate the LSC. We assume that the automaton created detects all pre-chart instances correctly.

**Theorem 5.4.3** The automaton, $A$, generated by the transformation algorithm in Figure 5.4 for a given LSC, SPEC, defined over an alphabet $\Sigma_{SPEC} \subseteq \Sigma$, reads
exactly the complement of the language of the \textit{SPEC}. Formally, \( \forall \theta = \theta_0\theta_1\theta_2 \ldots \)

\[
[\theta \in L(\textit{SPEC}) \implies \theta \not\in L(\textit{A})] \land [\theta \not\in L(\textit{SPEC}) \implies \theta \in L(\textit{A})].
\]

where \( L(\textit{A}) \) and \( L(\textit{SPEC}) \) are the languages of the transformed automaton and the \textit{SPEC}.

5.4.1 Verification Approach

For explicit state model checking, verification of a system against the specification is performed in the usual manner. The composition of the system and transformed LSC automata is computed on-the-fly and checked for accepting cycles using the Double Depth First Search (DDFS) algorithm. If the DDFS algorithm does not discover any accepting cycles, the system implements the safety and liveness behaviors as described in the chart. For symbolic model checking, we first label accept states as fair states in the composition of the system and transformed LSC automata. This automaton is then verified against the ACTL property \( \text{EG}(\text{true}) \), which searches for fair Strongly Connected Components (SCCs) reachable from the initial state. Any reported SCCs are violations of the specification.

5.5 Analysis

The verification approach presented in [31] utilizes at least two and in the worst case three algorithms to completely verify a system against an LSC. If reachability analysis followed by ACTL verification fails to produce a significant result (proof of correctness or a violation) the system is verified against an LTL formula generated from the LSC specification [49]. Compared to the verification approach of [31], the new verification approach presented in this paper only performs one verification run of comparable complexity as the reachability analysis and ACTL verification in the
approach of [31]. In the average case the total verification cost is reduced by a factor of two and in the best case (worst case in old approach) by a factor of three or more.

One side effect of using the negative automaton is the inability to verify multiple instances of a chart with cold construct violations. For example, if in our example system the Node receives jobID but is unable to validate jobID, the cold condition validID is never observed and the chart automaton will remain in state $q_2$. This is not an error since state $q_2$ is a non-accepting state waiting to observe the cold condition validID. If Node restarts the job acquisition by sending the idle message to the Scheduler, the safety transition from state $q_2$ to $q_{safety}$ is taken. Consequently, a false error will be reported (duplicate message). Generally speaking, if in one instance of the chart a cold construct is never observed, no future instances of the chart can be observed in a given trace. This drawback can be limiting for highly reactive and iterative systems with multiple instances in a single trace. A solution is being investigated as future work.

5.6 Results

We briefly discuss our experiments and results in this section. For a detailed presentation we refer the reader to the long version of the paper. We create models with multiple communicating processes and test them against highly concurrent worst case specifications as described in [31]. All specifications are named $A_c \times m$ where $c$ and $m$ are the number of co-regions and messages in each co-region respectively.

We first test the scalability of our approach in the symbolic model checking domain and compare it to the results presented in [31]. Table 5.1 shows a subset of the results for verifying the abp model using the NuSMV model checker. In general, our verification approach performs twice as fast as the approach presented in [31] and we scale to specification sizes that were unobtainable using the verification approach in [31]. We also test the scalability of our approach in explicit state model
Table 5.1: Results for the traditional and improved verification approaches using NuSMV.

<table>
<thead>
<tr>
<th>Specification</th>
<th>Traditional Verification</th>
<th>Improved Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reachability</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ACTL</td>
<td>Total</td>
</tr>
<tr>
<td></td>
<td>States</td>
<td>Time (s)</td>
</tr>
<tr>
<td>A3x5</td>
<td>1.01616e+06</td>
<td>34</td>
</tr>
<tr>
<td>A3x6</td>
<td>1.01616e+06</td>
<td>237</td>
</tr>
<tr>
<td>A3x7</td>
<td>879408</td>
<td>1568</td>
</tr>
</tbody>
</table>

Table 5.2: Results for the improved verification approach using SPIN.

<table>
<thead>
<tr>
<th>Specification</th>
<th>Model</th>
<th>Without Errors</th>
<th>With Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>States</td>
<td>Memory (MB)</td>
<td>Time (s)</td>
</tr>
<tr>
<td>A7x6</td>
<td>soko</td>
<td>97500</td>
<td>17.216</td>
</tr>
<tr>
<td></td>
<td>plain</td>
<td>406</td>
<td>7.385</td>
</tr>
<tr>
<td>A8x6</td>
<td>soko</td>
<td>97500</td>
<td>18.491</td>
</tr>
<tr>
<td></td>
<td>plain</td>
<td>406</td>
<td>8.661</td>
</tr>
<tr>
<td>A9x6</td>
<td>soko</td>
<td>97500</td>
<td>20.104</td>
</tr>
<tr>
<td></td>
<td>plain</td>
<td>406</td>
<td>10.274</td>
</tr>
</tbody>
</table>

checking using the SPIN model checker. Table 5.2 shows a subset of the results for verifying the *plain* and *soko* models. Our approach performs better and scales to larger specifications when compared to the approach of [37].

5.7 Conclusions and Future Work

The presented LSC to automaton transformation algorithm allows us to verify a system against an LSC using only language containment with readily available tools. Compared to past approaches, this approach only requires one verification run of comparable complexity as opposed to three verification runs for any arbitrary LSC. Further, we prove that the generated automaton can detect all safety and liveness violations in a system and empirically show the effectiveness of the approach. For future work we are investigating the use of LSCs for automated environment generation to test individual interfaces in a system. We are also investigating the possibility of extending the transformation algorithm to constructs such as overlapping chart
instances, Kleene star and multiple instance detection with the presence of cold constructs (as discussed earlier).
Algorithm: $TRANSFORM(q)$

1: for $\forall \delta : (q, \phi, q_{exit}) \in \Delta$ do
2: \[ \Delta \leftarrow \Delta \setminus \{\delta\} \]
3: $\psi_m \leftarrow \emptyset$, $\psi_i \leftarrow \emptyset$, $\psi_c \leftarrow \emptyset$

4: for $\forall \phi, q' : (q, \phi, q') \in \Delta$ do
5: \[ \psi_m \leftarrow \psi_m \cup f(\Sigma\text{msgs}, \phi) \]
6: \[ \psi_i \leftarrow \psi_i \cup f(\Sigma\text{invariant}, \phi) \]
7: \[ \psi_c \leftarrow \psi_c \cup f(\Sigma\text{conditions}, \phi) \]
8: \[ \Delta \leftarrow \Delta \setminus \{(q, \phi, q')\} \]
9: $\phi_{\text{child}} \leftarrow \phi \land \neg\text{disjunct}(\Sigma\text{msgs} \setminus f(\Sigma\text{msgs}, \phi))$
10: \[ \Delta \leftarrow \Delta \cup \{(q, \phi_{\text{child}}, q')\} \]
11: $TRANSFORM(q')$

12: for $\phi : (q, \phi, q) \in \Delta$ do
13: \[ \Delta \leftarrow \Delta \setminus \{(q, \phi, q)\} \]
14: $\phi_{\text{self}} \leftarrow \neg\text{disjunct}(\Sigma\text{msgs} \land \neg\text{disjunct}(\psi_c) \land \text{conjunct}(\psi_i))$
15: \[ \Delta \leftarrow \Delta \cup \{(q, \phi_{\text{self}}, q)\} \]
16: if $\exists \phi, q' : (q, \phi, q') \in \Delta$ and $f(\Sigma_{\text{main}}, \phi) \neq \emptyset$ then
17: $\phi_{\text{safety}} \leftarrow \text{disjunct}(\Sigma\text{msgs} \setminus \psi_m) \lor \neg\text{conjunct}(\psi_i)$
18: \[ \Delta \leftarrow \Delta \cup \{(q, \phi_{\text{safety}}, q_{\text{safety}})\} \]
19: if $(q, \phi, q') \in \Delta_{\text{hot}}$ then
20: \[ F \leftarrow F \cup q \]
21: return($A$)

Figure 5.4: Algorithm for building a negated automaton from an input LSC automaton.
Chapter 6

Verifying Communication Protocols Using Live Sequence Chart Specifications

Abstract

The need for a formal verification process in System on Chip (SoC) design and Intellectual Property (IP) integration has been recognized and investigated significantly in the past. A major drawback is the use of a suitable specification language against which definitive and efficient verification of inter-core communication can be performed to prove compliance of an IP block against the protocol specification. Previous research has yielded positive results of verifying systems against the graphical language of Live Sequence Charts (LSCs) but has identified key limitations of the process that arise from the lack of support for important constructs of LSCs such as Kleene stars, subcharts, and hierarchical charts. In this paper we further investigate the use of LSCs as a specification language and show how it can be formally translated to automata suitable for input to a model checker for automatic verification of the system under test. We present the translation for subcharts, Kleene stars, and hierarchical charts that are essential for protocol specification and have not been translated to automata before. Further, we successfully translate the BVCI protocol (point to point communication protocol) specification from LSC to an automaton and present a case study of verifying models using the resulting automaton.
6.1 Introduction

System on Chip (SoC) designs are fast moving towards a development environment that incorporates third party Intellectual Property (IP) cores and blocks. Due to the use of such heterogeneous IP cores, multiple communication protocols are required to achieve the desired interactions, behavior and functionality. With this diverse development environment comes the burden of verifying the system under development as well as the externally developed system being incorporated to ensure correctness and compliance to the specification. This need is especially important for vendors looking to market and promote their products in new markets and development environments. To reduce the verification costs and redundancy of verification (verified twice: once by the IP core developer and once again by the integrator), IP cores are often verified against commonly accepted standards and specifications to provide compliance results that can be easily utilized in an integration environment. A significant issue that hinders the process is the lack of an accepted specification language that can be formally integrated into the verification environment.

Traditionally, English has been used as the specification language for describing communication protocols. Due to the ambiguous and informal nature of English, in our experience, it has proven to be an inefficient specification language for use in a formal verification environment. Other specification languages based on temporal logic have also been used to specify correctness requirements of systems. Due to the complex nature of the temporal logics and the lack of support in all verification tools, these specifications tend to be limited in their use and applicability.

Other research has also investigated the use of graphical languages such as Live Sequence Charts (LSCs) for specifying communication protocols and have reported positive and encouraging results [13,14]. We choose LSCs as our specification language because of their direct applicability to specifying communication protocols that primarily describe inter-process communication. Additionally, their graphical and in-
intuitive nature makes them extremely usable for everyone involved in the development process and not only experts of formal verification.

In the past, LSCs have been used both as a specification and a modeling language. Because of their inherent ability to specify communication patterns without data information, we choose to use LSCs as a specification language describing the correctness requirements of a system. Previous work in the area of using LSCs as a specification language in a formal verification environment has been effective in exploring a verification approach but has failed to provide a comprehensive solution that supports the entire LSC grammar, which includes constructs such as Kleene stars, subcharts and hierarchical charts. We show how a communication protocol implementation can be formally verified against an LSC specification by providing translations of the entire LSC grammar to an automaton that is similar in nature to a never claim generated by SPIN [27]. This automaton can then be used directly as input to a model checker for verification of the system under test. Further, we provide a case analysis where the entire BVCI protocol is translated to an automaton and Promela models are verified against the translated automata [14].

Using a graphical specification language targeted towards communication protocols provides the inherent advantage of rapid development of specifications that are intuitive and useful throughout the development cycle of the product. Since LSCs can be used both as a modeling and a specification language, they provide a common medium for verifying requirements as well as systems. Using the translation to automaton as presented in this paper, the specification can now be applied more directly in a formal verification approach. Additionally, this approach does not require specialized tools and algorithms to be applied for formal verification of a system. It relies only on the synchronous composition of the model with the specification for detection of accepting cycles using the Double Depth First Search (DDFS) algorithm or a simple ACTL formula (for labeling algorithms) [39]. Doing so, allows the technique
to retain the advantage of any custom model abstractions or state space reduction techniques supplied by the model checker.

The paper is organized as follows. Section 6.2 presents an overview of related work in the field of LSCs and verification using LSCs. Section 6.3 gives an overview of the LSC constructs and provides an example of an LSC that is explained in detail. Section 6.4 presents an overview and examples of the LSC to automaton translation method. Section 6.5 discusses the case study and presents results of verifying Promela models against the BVCI LSC specification followed by conclusions in Section 6.6.

6.2 Related Work

LSCs are an extension to Message Sequence Charts (MSCs) [29]. The most significant addition to the MSC language is the introduction of liveness or provisional behavior that distinguishes between mandatory and optional behavior [10]. Additionally, the LSC language also provides constructs such as temperatures, subcharts, and precharts that enable the user to describe behaviors that could not have been described in MSCs. Protocol Live Sequence Charts (PLSCs) are an extension to LSCs that are targeted to describing protocols [13, 14].

LSCs have been used to model and specify a variety of systems such as air traffic control systems [7], radio based communication systems [20], and train systems [5]. Their use in these case studies has shown their effectiveness in specifying and verifying complex behaviors of a system. LSCs and PLSCs have also been used in the past to specify SoC communication protocols and formally verify aspects of the protocol on the system [14, 13]. Additionally, they have also been used for automatic synthesis of systems as well [25].

Recently, LSC based verification techniques have been gaining significant attention. One aspect of LSC related verification deals with verifying properties on the LSC specification itself [1, 47]. In this case, the LSC is used as the model.
Another aspect of LSC based verification deals with the verification of systems against the LSC specifications. Two primary methods have been proposed to perform verification of systems against LSCs. The first deals with temporal logic. One approach converts the LSC specification to multiple small temporal logic properties that are verified on the system [14,13]. These individual properties are easily verified on a system but are insufficient to establish a formal relationship between the specification and implementation itself. Other approaches translate the complete chart to temporal logic, which is then used as the specification input to a model checker such as SPIN or NuSMV [49,37,35]. The primary limitation of these approaches is the exponential explosion encountered in the generated temporal logic formula, which severely reduces the scalability of the approach. Additionally, the lack of support for translating the complete grammar of LSCs to temporal logic is a great limiting factor in the applicability of the approaches.

The second method for verifying systems against LSCs does so by converting the LSC to an automaton and using the automaton in language containment based verification techniques [49,39]. This method supports a greater subset of the LSC grammar and scales to much larger specification sizes. Although the verification results and performance using the automaton approach for verifying systems are very promising, the research does not deal with constructs such as subcharts, hierarchical charts, and Kleene stars, which are essential to the specification and verification of SoC interface and communication protocols. The work presented in this paper innovates upon previous work by extending the translation of LSCs to the complete grammar of LSCs.

6.3 Live Sequence Charts

The LSC language provides constructs to express behavior of systems and individual processes with relative ease and intuitiveness. The primary advantage lies in its
graphical yet formal nature. Figure 6.1 shows an example LSC for the Normal Request subchart in the BVCI protocol [13]. We use this example as our basis for introducing the constructs and semantics of the LSC language.

Processes or Instances: Processes are drawn with rectangular instance heads that denote the start of the processes. A vertical line originating from the instance head signifies the life-line of the process and ends in a filled rectangle, which terminates the respective process. The example LSC describes the interaction between the Initiator and Target processes.
**Locations:** The life-line of each process is marked with locations that are points where events and other constructs may be described. Locations are unique to each process and start at location L1. For each new event or construct placed on the process life-line, the location number is incremented for the respective process. For example, the *address* message is sent from the *Initiator* process at L3 and *Target* evaluates the *cmdack == 0* post condition at L17.

**Messages:** Messages are a form of communication between processes in the LSC. Each message has a sender and receiver process attached to it. Messages are annotated with a message label that identifies the message. Messages can be *simultaneous* or *asynchronous*. Simultaneous messages are drawn with a solid arrow head and occur instantaneously when both the sender and receiver are ready for the communication. Asynchronous messages are drawn with an open arrow head and can be received any time after sending (we force the send event to occur before the receive event). In the example LSC, the *address* message is an asynchronous message and the *cmdack:High* message is a synchronous message.

**Conditions:** Conditions are placed in the chart by drawing hexagons around the life-lines of processes evaluating the condition. The condition label describes a predicate that must be satisfied at the current location(s) of the process(es). Conditions spanning multiple process life-lines act as synchronizing points for the involved processes and the condition is not evaluated unless all the processes are at the respective condition locations. Conditions attached to a message are called *bonded conditions*. Conditions placed on their own location and not attached to a message in the chart are called *non-bonded* conditions [30]. Non-bonded conditions are evaluated continuously until they are satisfied. In our example LSC, all conditions (the *cmdack == 0* precondition and the *cmdack == 0* postcondition) are non-bonded. *Invariants* are conditions spanning over multiple locations in the chart.
**Coregions:** Coregions are drawn with a dashed vertical line next to the life-line of a process. They describe behavior that can occur in any order. All messages in the dashed vertical line (address, be, clen, etc.) next to the Initiator and Target processes are in a coregion.

**Simultaneous regions:** Simultaneous regions describe events that have to occur at the exact same time. Dots are drawn on locations to indicate simultaneity of events. Examples of simultaneous events include the clock signal (drawn with a straight line across all the processes in the system) and bonded conditions (condition with message).

**Actions:** The LSC language also provides the action construct that allows a process to perform an action on its local or global variables. For example, variables may be incremented, decremented or assigned a value at certain points on the life-line of a process. In the example LSC of Figure 6.1, the count++ action is performed by the Target process before the postcondition is evaluated. Currently, actions do not translate to the automaton generated from the LSC. Although, it is possible to check the effect of an action by automatically generating a condition in the LSC to ensure that the action has been performed successfully.

**Prechart:** The prechart is drawn with a dashed hexagon encompassing the instance heads and connects to the main body of the chart that is described in the solid rectangle following the prechart. The prechart describes the behavior of the system under which the main body of the chart is to be observed. The prechart can also be substituted with a single activation condition.

**Main chart:** The main chart of the LSC is the behaviors described in the rectangle following by the prechart. The main chart can be either existential or universal. Universal charts, drawn with a solid rectangle, specify behavior that must be satisfied by the system every time the prechart is satisfied. Existential charts are drawn with a dashed rectangle and specify behavior that the system must exhibit at
least once when the prechart is satisfied. In the example LSC of Figure 6.1, the main chart is a universal chart.

**Subcharts:** Subcharts are LSC charts that can be included within the body of a larger main chart. They are usually not preceded by a prechart. When a subchart $B$ is included within the main chart of $A$, chart $A$ is at a higher scope than subchart $B$. Subcharts in conjunction with conditions and Kleene stars can be used to create control and looping structures such as if-then and while blocks. The example chart shown in Figure 6.6(a) shows one main chart `SubchartExample` that contains a smaller subchart. The semantics of subcharts are discussed in greater detail in Section 6.4.2.

**Temperatures:** Temperatures in LSCs can be assigned to messages, conditions, and locations. A *hot* temperature is depicted by using a solid line to draw the construct and specifies behavior that must be satisfied by the system. A *cold* temperature is drawn using a dashed line for the construct and specifies behavior that may be satisfied. If a cold message is never observed, the LSC waits at the current location for the message. If on the other hand, the construct after the cold message in the chart is observed, the LSC progresses to the location after the cold message. Bonded cold conditions do not affect the LSC execution. If the condition is not satisfied, an error is *not* reported and the LSC exits the current scope to a higher scope. If no higher scope exists, the LSC exits completely. In the case of a non-bonded cold condition, the LSC waits indefinitely at the current location for the condition to be satisfied and can only exit the current scope if a construct at a higher scope is observed. It is not possible for the LSC to move to a location after the non-bonded cold condition within the same chart until the non-bonded cold condition is satisfied. If no higher scope exists, the LSC waits indefinitely for the non-bonded cold condition to be satisfied. For the example chart shown in Figure 6.6(a), if the non-bonded cold condition $p$ is not satisfied, then the LSC waits at the current location until either $p$
is satisfied or a $b$ is observed. All constructs in the example LSC in Figure 6.1 are hot except for the activation condition $cmdack == 0$.

**Kleene star:** The Kleene star construct, '$\ast$', is used to represent multiplicity where the associated chart/message can occur zero or more times (countably infinite). In our example LSC, the clock signal following the $cmdack == 0$ condition can occur as many times as required before the coregion messages are observed. A variation of the Kleene star is the '+’ symbol that forces at least one (and allows more than one) occurrence of the associated construct.

**Hierarchical charts:** Hierarchical charts are constructed using individual LSCs and are useful for creating specifications that require control flow. Hierarchical LSCs are similar to LSC subcharts and high level MSCs as described in [41]. Figure 6.7(a) shows an example of hierarchical LSCs where $A$, $B$, and $C$ are individual LSCs joined together to form a hierarchical LSC.

We have presented the entire set of LSC constructs that are currently supported by our LSC to automaton translation. Apart from the listed constructs, the chart also induces a natural partial order for all constructs along each instance line. Intuitively, instances evolve in the downward direction and are blocked until an event on their life-line occurs. For the example chart shown in Figure 6.1, the chart is entered when the $cmdack == 0$ condition is satisfied. After the precondition is satisfied, multiple clock signals may occur before the $cmdvalHigh$ message should be observed followed by the coregion. After all messages as described in the LSC are observed, the Target process increments the $count$ variable and waits for the $cmdack == 0$ condition to be satisfied.

Additionally, we incorporate the **delayed choice** semantics when dealing with subcharts, hierarchical charts, and cold constructs. The delayed choice semantics allow the chart to resolve a choice by waiting for relevant input before committing to a certain path in the LSC. Since we are using the chart as a specification language rather
than a modeling language, delayed choice semantics help avoid non-determinism (reduce false positives). If the LSC were to be used as a model rather than a specification, delayed choice semantics would be removed to allow non-determinism in the model.

In our research, we deal with all the described constructs of LSCs with the following restrictions: (a) overlapping instances of charts are not permitted, (b) the prechart and the main chart should have a disjoint set of messages and conditions (to avoid overlapping instances of charts), and (c) only one subchart can be enabled at a given time. These limitations have been introduced to simplify the LSC to automaton translation process and remove any non-determinism. Since most specifications in general do not require overlapping charts and instances, the limitations do not affect the general applicability of the results.

### 6.4 LSC to Automaton Translation

Our verification approach using LSCs is based on detecting accepting cycles on the synchronous composition of the system automaton and the negative automaton of the LSC. The negative automaton of the LSC is the automaton that enables detection of unwanted behaviors in the system (using accept cycles recognized by the LSC automaton). The automaton is similar in nature to the never claim used in SPIN and has been shown to be an effective method of using LSCs for verification [39]. We first present an overview of the LSC to automaton translation for basic constructs as discussed in [39] and then present the translation for extended constructs that have not been explored in previous work: the Kleene star operator, subcharts, and hierarchical charts. To conserve space, we restrict our discussion to universal main charts only.

Before we discuss the LSC to automaton translation, we introduce some necessary formalism. We use Symbolic automata, an extension of Büchi automata, that allow observing any of a possible set of inputs on an edge. Formally, Symbolic au-
tomata are given by $A = \langle \Sigma, Q, \Delta, q^0, F \rangle$ where, $\Sigma$ is the finite alphabet of input symbols (variables), $Q$ is the finite set of states, $q^0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final/accepting states, and $\Delta \subseteq Q \times \phi \times Q$ is the transition relation. A transition $(q, \phi, q') \in \Delta$ represents the change from state $q$ to state $q'$ when the formula $\phi$ is satisfied.

We partition the set of Boolean variables $\Sigma$ into three distinct sets $\Sigma_{msgs}$, $\Sigma_{invariants}$, and $\Sigma_{conditions}$, that contain the Boolean variables that are used for messages, invariants and conditions in the chart respectively. For the chart shown in Figure 6.1, $\Sigma_{msgs} = \{\text{address, opcode, clen,} \ldots\}$ and $\Sigma_{conditions} = \{\text{cmdack == 0}\}$. The set $\Sigma_{main} = \{\text{address, opcode, clen,} \ldots\}$ is the set of Boolean variables that are used in the main chart only. We also have a set $\Delta_{hot} \subseteq \Delta$ which only contains transitions that correspond to hot constructs in the chart (hot messages, hot conditions etc.).

For a set of Boolean formulas $\Gamma = \{\phi_0, \phi_1, \ldots, \phi_n\}$ we define the function $\text{disjunct}(\Gamma)$ to return the disjunct of the individual formulas in $\Gamma$ and the function $\text{conjunct}(\Gamma)$ to return the conjunction of the individual formulas in $\Gamma$. The function $f(\Sigma, \phi) = \{\sigma | \sigma \in \Sigma \text{ and } \sigma \text{ or } \neg \sigma \text{ appears in } \phi\}$ returns the set of Boolean variables from $\Sigma$ that appear in $\phi$ in either a positive or negative form. For example, if $\phi = a \land b$, and $b$ is a condition predicate, $f(\Sigma_{msgs}, \phi) = \{a\}$ and $f(\Sigma_{condition}, \phi) = \{b\}$. Additionally, the $\psi_m$, $\psi_c$ and $\psi_i$ sets contain the message, condition and invariant predicates used in the current state of the automaton.

The automaton of the chart is obtained by exploring every possible cut through the chart. A cut through a chart represents the current state of the chart as specified by the location of each process in the chart and the state of the variables of the chart. For the example LSC shown in Figure 6.1, the X marks on each instance line represent a cut through the chart. At this cut the Initiator and Target processes are
\[ \phi_{\text{self}} = \neg \text{disjunct}(\Sigma \text{msgs}) \wedge \ldots \neg \text{disjunct}(\psi_i) \wedge \text{conjunct}(\psi_i) \]

\[ \phi_{\text{safety}} = \text{disjunct}(\Sigma \text{msgs} \setminus \psi_m) \lor \neg \text{conjunct}(\psi_i) \]

\[ \phi_{\text{child}} = \phi \land \neg \text{disjunct}(\Sigma \text{msgs} \setminus f(\Sigma \text{msgs}, \phi)) \]

Figure 6.2: Generic state of the LSC to automaton translation.

at the beginning of their coregion ready to send/receive any of the messages in the coregion and count is zero (initialized).

From a given cut, enabled transitions lead to successor cuts. The enabled transitions correspond to the set of events that can occur from a given cut without violating the partial order induced on the events by the instances in the chart. Each unique cut of the LSC corresponds to a unique state in the automaton. The unwinding algorithm as presented in [30] provides a method to unroll the LSC and all possible cuts of the LSC; thus, it gives the basic structure of the LSC automaton. This basic structure of the automaton is then transformed to a negative automaton using the transformation algorithm presented in [39]. It should be noted that the unwinding algorithm presented in [30] does not support Kleene stars, subcharts, and hierarchical charts. Additionally, the basic structure generated from the LSC is not as efficient as the transformed automaton presented in [39].

The general structure of the LSC automaton can be split into two parts: the prechart automaton and the main chart automaton. Additionally, a special state in the automaton is the safety state, \( q_s \): an accepting state that contains only one outgoing transition to itself labeled with true. The prechart automaton contains only non-accepting states since the prechart is responsible for the detection of the activation condition of the main chart. Additionally, the prechart states do not contain transitions to the safety state since the prechart does not detect errors or incorrect
behavior. The first state of the prechart contains a special outgoing transition to itself that is labeled *true* to ensure that all possible instances of the charts in the system are checked for errors (corresponds to globally).

Figure 6.2 shows a generic state of the automaton with all possible outgoing transitions. The dotted self-loop in each state is to detect non-progress when no relevant letters are observed (also liveness errors). The solid transitions to child states detect the progress through the LSC when relevant letters are observed. Multiple child states occur when concurrency is present in the chart. The dashed transition from main chart states to the safety state allows detection of safety errors (out of order messages, invariants, etc.). Main chart states are marked accept states if at least one progress transition corresponds to a hot construct from the main chart. The final state of the automaton is non-accepting and contains no outgoing transitions. Using the transition labels as shown in the generic state of Figure 6.2, the transition relation for each main chart state is proven to be deterministic and total, which is further utilized in the proof of correctness for the translation.

### 6.4.1 Kleene Star

Kleene stars can be placed on messages or subcharts to indicate repetition. When a Kleene star is placed on a construct (message or subchart), the construct may be observed in the system zero or more times (countably infinite). We first show how Kleene stars attached to messages are translated to automaton and discuss the translation of subcharts with Kleene stars in Section 6.4.2.

Figure 6.3(a) shows an LSC where message *b* may occur zero or more times. The corresponding automaton is shown in Figure 6.3(b). The first state, *q₀*, corresponds to the locations in the LSC where message *a* is yet to occur. The safety transition is enabled if any out of order messages (*b∨c*) are observed. The second state *q₁* corresponds to the state where the message *b* can occur a countably infinite number
of times. To accommodate for this repetition, a new disjunctive clause \( b \land \neg a \land \neg c \) is added to the self-loop (expressions reduce to \( \neg a \land \neg c \)). The modification allows the automaton to remain in state \( q_1 \) as long as no relevant messages or the message \( b \) is observed repeatedly. The safety transition is also modified to detect multiple letters and out of order messages. Finally, if \( c \) is observed, the automaton moves to state \( q_2 \), which is the end of the LSC and the automaton.

We now formalize our translation of Kleene stars to automaton. The transition labels as shown in Figure 6.2 are modified to incorporate translation of the Kleene star. We introduce the sets \( \psi_k \) and \( \psi_x \) corresponding to the messages that are attached with a Kleene star and messages not attached with a Kleene star for a given state. The self-loop is modified to allow the automaton to remain in the current state if no relevant messages are observed, or if the Kleene star messages are observed: \( \phi_{\text{self}} = \neg \text{disjunct}(\Sigma_{\text{msgs}} \setminus \psi_k) \land \neg \text{disjunct}(\psi_c) \land \text{conjunct}(\psi_i) \). Secondly, the safety transition is modified to disable detection of multiple instances of a message and detect all other possible safety errors as follows: \( \phi_{\text{safety}} = \text{disjunct}(\Sigma_{\text{msgs}} \setminus (\psi_k \cup \psi_x)) \lor \text{conjunct}(\{\neg \text{disjunct}(\Sigma_{\text{msgs}} \setminus (\psi_k \cup \psi_x)), \psi_x\}) \). Using these modified transition labels, we can now prove that the transition relation for each main chart state is deterministic and total (to prove correctness of translation), as shown in [39].

A variation of the Kleene star construct is the ‘+’ operator that is used to specify that a message should appear at least once. If message \( b \) in Figure 6.3(a) is changed to have a ‘+’ rather than a ‘*’ operator, the corresponding automaton is shown in Figure 6.4. The major difference between the Kleene star and ‘+’ operator translation lies in the introduction of an extra state \( q_1 \) to ensure at least one instance of the message \( b \) is observed. The state \( q_1 \) waits for the first \( b \) to be observed and the state \( q_2 \) allows an infinite number of \( b \)'s to be observed.
Figure 6.3: Example of translating a message with Kleene star.

6.4.2 Subcharts

We now focus on translating subcharts to automaton. Figure 6.5(a) shows an example of an LSC with three subcharts that start with the letters \( x \), \( y \), and \( z \) respectively. To conserve space we do not show the entire contents of the subcharts, but focus on the start letter of each subchart. Given the example LSC of Figure 6.5(a), the corresponding automaton translation is presented in Figure 6.5(b).

To translate a chart that contains subcharts, each subchart is first individually translated to an automaton. After each subchart has been translated to its respective automaton, the automata are combined into one large automaton using the scheme presented in Figure 6.5(b). The automaton for the LSC moves from the initial state \( q_0 \) to the automaton of chart \( A \) when an \( x \) is observed (automata combined using stan-
Figure 6.4: Translation of the plus operator on a message.

dard sequential composition) [42]. After subchart A has been observed successfully, the automaton moves to subchart B and so on.

The dashed transitions in the automaton are introduced to incorporate delayed choice semantics. Such dashed transitions are introduced from every possible legal exit (last state and states corresponding to cold constructs) of a subchart to the entry points of other subcharts or higher scopes. For example, if at a legal exit of subchart A, the letter y is observed, progress is made by exiting chart A and entering chart B. Similarly, from a legal exit of chart A, progress can be made to the beginning of chart C by observing a z. The dash-dot transitions from the legal exits of B to the beginning of B are introduced to incorporate delayed choice for the Kleene star attached to subchart B. They ensure that a new instance of B can be observed from an legal exit of B.

Subcharts that occur concurrently with other constructs in the chart are combined using standard parallel composition of automata (all possible inter-leavings are explicitly expressed) as opposed to sequential composition described before [42]. For example, if subcharts A and B are concurrent with each other, the automata for the
subcharts are composed in parallel and the last state of the automaton is used for performing sequential composition with the automaton of subchart $C$. Performing the parallel composition increases the complexity of the translated automaton, which in the worst case can be exponential. Again, the transition labels are modified in Figure 6.2 to account for transitions that incorporate delayed choice. To conserve space, the details have been omitted from this version of the paper.

Figure 6.6(a) shows the SubchartExample chart that contains a subchart with a Kleene star. The corresponding automaton is shown in Figure 6.6(b). At state $q_0$
the automaton expects to see the message $a$. Once this message has been observed, the automaton moves to state $q_1$ where it either expects to enter the subchart (waiting for condition $p$ to be satisfied) or to move to the location after the subchart when message $b$ is observed (using the dashed transition from $q_1$ to $q_4$). If the subchart is entered, message $x$ is observed in the normal manner. At state $q_3$, message $y$ is observed and the outgoing transition depends on the Kleene star. As a Kleene star is attached to the subchart, observing message $y$ leads the automaton back to state $q_1$ using the dotted transition. If the Kleene star did not exist, the automaton would move to state $q_4$ after observing $y$. After $y$ has been observed, the automaton waits for $b$.

### 6.4.3 Hierarchical Charts

Hierarchical charts allow individual LSCs to be joined together sequentially. The construct is particularly useful in combining large individual LSCs into a visually concise LSC incorporating control flow and choice. Figure 6.7(a) shows an example of a hierarchical chart. $A$, $B$, and $C$ are individual charts and either chart $B$ or $C$ can be executed after chart $A$ has completed execution. After chart $C$ has completed execution, the hierarchical LSC moves back to chart $A$. Message $x$ is the final message of chart $A$ and messages $y$ and $z$ are the first messages of charts $B$ and $C$ respectively (activation). The corresponding automaton of the hierarchical chart is shown in Figure 6.7(b). To conserve space, we only show the general structure of the automaton and the hierarchical chart.

From the last state of chart $A$, the automaton has the option of moving to the first state of chart $B$ or the first state of $C$. If a $y$ is observed, the automaton moves to the automaton of chart $B$ and if a $z$ is observed it moves to the automaton of $C$. The final state of chart $A$ is not accepting, since no behavior must be observed at this point. The automata are joined together using standard sequential composition.
Figure 6.6: Example LSC showing a subchart with a precondition and Kleene star and the corresponding automaton (a) example LSC with subchart and (b) corresponding automaton.
Since chart $C$ always moves back to chart $A$, a dash-dot transition is introduced from the last state of chart $C$’s automaton to move back to the first state of chart $A$. Additionally, the dashed transitions from $A$ to $B$ and $A$ to $C$ are introduced to handle cold constructs. A similar set of transitions is introduced from $C$ to $A$ for cold constructs in $C$. These transitions can only be introduced to the successor charts of a given chart. For example, such transitions are not introduced from chart $B$ to chart $A$ or $C$ since $B$ has no successors.

It should be noted that in accordance with delayed choice semantics the minimal common prefix is chosen to identify the next chart that is to be executed (one message in the example, but could be more than one when complex precharts are specified)\(^1\). In the case of complex precharts (more than one message/condition), each legal exit of a chart leads to a new instance of prechart detection where it is possible to detect progress in the prechart of a successor chart or in the current chart itself.

\(^1\)We assume by virtue of delayed choice semantics that we can observe enough letters to resolve to a single chart.
(by observing the cold construct). For multiple successor charts, multiple prechart detections are introduced from each legal exit of a chart.

6.5 Case Study: BVCI Protocol Verification

The Basic Virtual Component Interface (BVCI) protocol is part of the Virtual Component Interface (VCI) standards family that was developed to specify point to point communication protocols. We use the BVCI protocol as our specification for case analysis because of the complex nature of the specification as well as the past research that has been performed on verifying systems against the BVCI protocol [13, 14]. We now describe our modeling and verification approach, and present results of verifying our models against the LSC specification.

**Specification:** The specification consists of one LSC that contains four subcharts, each with a unique activation condition. Figure 6.1 shows the first subchart in the LSC specification. The remaining subcharts are not included in the paper to conserve space. Each subchart contains Kleene stars and plus operators that are translated using the schemes presented in Section 6.4. Each individual subchart is translated and combined into one large automaton using the subchart translation scheme presented in Figure 6.5. The total size of the resulting automaton is 291 states and it describes all possible behaviors the Target and Initiator processes may exhibit.

**Modeling:** Four different models that implement the behaviors of the BVCI model as described in the individual subcharts of the BVCI specification are created in Promela. The clock is abstracted away in the modeling process and replaced with a synchronous message that is exchanged between the Initiator and Target processes.

**Verification:** To verify the models against the specification, each model is combined with the automaton translated from the BVCI LSC specification using SPIN. A synchronous composition of the two is then checked for accepting cycles.
using the built in Double Depth First Search (DDFS) algorithm of SPIN. Results are shown in Table 6.1. For each model, we list the number of states that were explored to completely verify the model against the entire BVCI specification along with the memory and time resources that were utilized. The results presented here are for models that did not contain any errors. In a separate verification exercise, safety and liveness errors were introduced in the models and verified against the specification. Each error introduced in the model was successfully discovered by the model checker. To conserve space, results for the erroneous verification runs are not included.

### 6.6 Conclusions

We have shown how additional constructs of LSCs such as subcharts, Kleene stars, and hierarchical charts can be translated to a negative automaton. We have also presented a case study of using our translation technique to create an automaton from the BVCI protocol and perform verification of Promela models against the resulting specification, which has not been done before. Our results and experiences indicate that using the LSC language as a specification language is extremely useful for writing and developing specifications that can be used during formal verification. Additionally, their use as a modeling language further strengthens their applicability in the initial stages of the protocol development process.

For future work in this area we are developing a technique that allows us to create individual negative automata for each process/instance described in the
LSC to perform modular verification of a system and reduce the verification state space. We are also investigating the possibility of creating observers and monitors from LSC specifications that can be used in a runtime verification approach to assist in the discovery and avoidance of errors in implementations. Finally, we are working towards incorporating the current approach in a tool chain for protocol development/verification.
Chapter 7

Conclusions and Future Work

7.1 Conclusions

This research presents methods to automatically extract mathematical structures from LSCs suitable for formal verification of a system under test. Using the techniques presented in this research, the complexity of the LSC verification task has been reduced significantly, which has a direct impact on the performance and scalability of the LSC verification task. The performance of LSC verification has been increased by at least a factor of two and scalability of the LSC verification task has been increased by a factor of fifteen. Additionally, LSC verification has been extended to a wider range of systems by extending the translation techniques to support a larger subset of the LSC grammar, which are essential for protocol and system development/verification. Finally, the expressibility of LSC relative to temporal logic has also been enhanced greatly using the results of this work. A summary of the contributions of this work is presented as follows.

- **Improving LSC to LTL translation**: traditional translations of LSC to temporal logics are primarily based on expressing the partial order of the LSC or on expressing every possible path through the chart, which in the worst case results in a temporal logic formula that is exponential in the size of the LSC. The improved translation presented in this work expresses the total order of the LSC, which in the worst case is quadratic in the size of the LSC. Compared
to past LSC to temporal logic translations, the improved translation presented in this research has shown great benefits in performance and scalability during verification of systems against the temporal logic formula derived from the LSC specification (Chapter 3).

- **Extending LSC to LTL translation**: translation of additional constructs of LSCs such as asynchronous messages, coregions, simultaneous regions and bonded conditions was added to the improved LSC to LTL translation scheme, which enabled the verification of a larger set of protocols implementations that could not have been verified before (Chapter 3).

- **Expressing improved LSC to LTL translation in CTL**: the improved translation of LSCs was restricted to LTL. The work was further extended by showing that the improved translation of LSCs to LTL lies in the common fragment of LTL and CTL and the equivalent CTL formula can be generated by adding the universal path quantifier to each temporal operator in the LTL translation of LSCs (Chapter 4).

- **Specifying arbitrary CTL properties in an LSC**: the expressibility of LSCs in temporal logic is limited to two types of specification patterns. The result presented in this work allows arbitrary CTL properties to be embedded in an LSC; thus, enabling a verifier to specify and verify properties that are not inherently expressible by LSCs. Additionally, this allows specification of knowledge based properties of multi-agent systems, which were not supported before (Chapter 4).

- **Improving LSC to automaton translation**: previous translations of LSC to automaton focused on creating automaton structures that are directed towards proving the correctness of systems using a multi-stage verification approach. The improved translation of LSCs to automaton is directly suited for discovery
of errors in a system and requires only one standard model checking algorithm to perform the verification. Both performance and scalability is increased significantly using the improved translation as compared to past LSC to automaton approaches (Chapter 5).

- **Extending LSC to automaton translation**: the improved LSC to automaton translation is further extended to constructs such as Kleene stars, plus operators, subcharts, and hierarchical charts that are necessary for verifying communication protocols. Using this extended translation, verification of communication protocol implementations was performed, which could not have been performed before (Chapter 6).

- **Modal logic results**: two important modal logic results are presented in this thesis that are related to the transitivity and coverage of the until operator in temporal logic. Both results are essential in the translation of LSCs to temporal logic. (Chapter 3).

Apart from the theoretical contributions listed above examples, benchmark suites and translation algorithms were created to empirically evaluate the techniques developed in this work against past methods and research in the field of LSC verification.

### 7.2 Future Work

The work presented in this thesis has demonstrated the use of LSCs in a formal verification environment by translating LSCs to temporal logic and automaton that can be directly used as a specification input to a model checker. Many interesting avenues of research have also opened up by during the course of this research. A brief list is presented as follows.
• **Modular verification**: the current translation of LSCs to automaton provide a final automaton structure that enables the verification of the entire system. In many cases, especially with third party development and integration, it is desirable to verify only one particular interface of a system. In such a situation the approach presented in this thesis as well as past research would prove to be too complex and cumbersome because a complex environment would have to be generated for the system under test. Gaining the ability to perform modular verification by extracting only the automaton for a single interface from the LSC would prove to be useful in such a situation. Additionally, performing modular verification of individual interfaces of a system also provides the advantage of increasing the scalability of the verification approach. This is because the individual automaton structures generated for each interface would be much smaller and will reduce the state space of each verification run significantly. In some cases it may also be possible to perform the verification using modular verification where it was not possible to perform verification using one large automaton structure as the specification.

• **Introducing data in LSCs**: currently, LSCs are only capable of specifying communication patterns of a system with extremely limited data aspects associated with each communication construct within the LSC. To be able to completely verify a system, it is desirable to specify not only the communication patterns but also the content of the messages exchanged between interfaces within a system. Extending LSCs to enable data specification would be extremely useful for communication protocol specification and verification.

• **Runtime verification using observers**: in many situations, performing the complete verification of a system may not be possible due to lack of resources. In such cases, it would be useful to be able to automatically extract observers from the LSC specifications that would enable runtime verification of an interface
providing graceful exit capabilities such as restart, watchdog timers, and safety error detection.

• **Tool chain integration for protocol development**: a next logical step in this research is to integrate the techniques and algorithms presented in this paper within a tool chain that is specifically targeted towards protocol development and verification. Doing so would provide a platform for analyzing the effectiveness and drawbacks of the current techniques and eventually open up further avenues of research.
Appendices
Appendix A

Temporal Logics

A.1 Introduction

Temporal logic is a formalism used to specify properties of a state transition system over time. It builds on the already available boolean connectives and atomic propositions of a system to describe behaviors of a sequence of states in a given system. Time is not explicitly specified in the temporal logic formulas, rather temporal operators are used to express properties such as never, always, and eventually. We first introduce the operators that allow us to write temporal logic formulas describing certain behaviors of systems. After the description of the operators, we give an overview of the CTL*, CTL, and LTL temporal logics as described in [18].

Path Quantifiers are used to describe the branching time structure of a system. Two path quantifiers are available in temporal logics. The universal path quantifier, $A$, is used to write properties over all the possible paths from a given state in the system. The existential path quantifier, $E$, specifies that there exists a path with a specific property.

Temporal Operators are used to describe the properties of a path through a system. The five temporal operators are as follows:

1. $X$: The next time operator forces a property to hold on the second state of a path.
2. **F**: The *future* operator asserts that a property holds at some state (after the current state) in the path.

3. **G**: The *globally* operator specifies that a property always holds on every state of a path.

4. **U**: The *Until* operator combines two properties to specify that there should be a state on the path where the second property holds and all states before that state should satisfy property one.

5. **R**: The *release* operator is the logical dual of the **U** operator. It requires that the second property hold on all states of a path until the first property holds. It also enforces that the second property hold in the state where the first property is satisfied.

We now discuss how the path quantifiers and temporal operators can be combined together to create formulas (specifications).

### A.2 CTL*

Two types of formulas exist in temporal logic. State formulas can be true for a given state and are described by the following syntax:

- If \( p \in AP \), \( p \) is a state formula.
- If \( f \) and \( g \) are state formulas then \( \neg f, f \lor g, f \land g \) are state formulas.
- If \( f \) is a path formula, then \( Ef \) and \( Af \) are state formulas.

Path formulas are true for a specific path and are specified using the following syntax:

- If \( f \) is a state formula, then \( f \) is also a path formula.
• If \( f \) and \( g \) are path formulas, then \( \neg f, f \lor g, f \land g, Xf, Ff, Gf, fUg, fRg \) are all path formulas.

CTL* formulas are created using the rules described above.

A.3 CTL: Computational Tree Logic

Computation Tree Logic (CTL) is a restricted subset of CTL* in which all the temporal operators must be immediately preceded by a path quantifier. The following rule is used to create path formulas:

• If \( f \) and \( g \) are state formulas, then \( Xf, Ff, Gf, fUg, fRg \) are path formulas.

Compared to CTL*, we see that not all formulas are path formulas, which restricts the expressibility of CTL compared to CTL*. Some examples of CTL formulas are:

• \( \text{EF}(a \land \neg b) \): There exists a path where in the future a state satisfies \( a \) and does not satisfy \( b \).

• \( \text{AG}(\text{request} \rightarrow \text{AFack}) \): It is always globally true that a request will eventually be acknowledged.

A.4 LTL: Linear Temporal Logic

Linear Temporal Logic (LTL) consists of formulas that are of the form \( Af \) where \( f \) is a path formula in which all state sub-formulas are atomic propositions. The syntax for creating path formulas for LTL is as follows:

• If \( p \in AP \), then \( p \) is a path formula.

• If \( f \) and \( g \) are path formulas, then \( \neg f, f \lor g, f \land g, Xf, Ff, Gf, fUg, fRg \) are all path formulas.
Some examples of LTL formulas are:

- **AF(a ∧ ¬b):** For all paths, in the future a state satisfies $a$ and not $b$.

- **A(openUclose):** For all paths, the door is open until it is closed.

The three logics described above have different expressive powers. Figure A.1 shows the relationship between all three logics. CTL* is the most expressive and unconstrained of all three logics, but CTL and LTL are different w.r.t. each other with a common fragment. There are formulas that can be written in CTL, but not in LTL and vice-versa. For example, the LTL formula $A(FGp)$ has no equivalent CTL formula and the CTL formula $AG(EFp)$ has no equivalent LTL formula. The disjunction of the two formulas, $A(FGp) ∨ AG(EFp)$ is a CTL* formula that has no equivalent CTL or LTL formula.
Appendix B

Improving Translation of Live Sequence Charts to Temporal Logic

This chapter presents the proof details and additional results for the paper presented in Chapter 3.

B.1 Inductive Definitions

Using the definitions and notation presented in Chapter 3, we define what it means for a system to satisfy a chart in a manner similar to that used for computation tree logic semantics in [18]. The $\models$ relation connects the sequences of states generated by a system implementing a chart to the chart itself. A path in a system, $\pi = s_0, s_1, s_2, \ldots$, is an infinite sequence of system states. The notation $\pi^i$ denotes the suffix of $\pi$ starting at $s_i$. The notation $S, s \models c$ indicates that a system $S$ at state $s$ is a model of the chart $c$. We define the notation for $S, \pi \models c$ similarly. In the definition of $\models$, we assume the presence of a labeling function $L(s)$ to map system states to messages in $AP$. We now give the definition which is shown in Figure B.1.

Rules 1 and 3 are satisfied if a state is labeled with $e$ or the first state of a path is labeled with $e$. Rule 2 checks the type of the chart and either forces every path to satisfy the chart when the pre-chart is activated (provisional behavior) or some path to activate the pre-chart and satisfy the chart (mandatory behavior) [10, 19, 46, 35]. This is further clarified in rule 4 that checks for instances of the pre-chart using rule 5.
Figure B.1: Definition of the $\models$ relations for testing if a system implements an LSC.

Rule 6 is a recursive definition that checks the order, absence, and uniqueness of messages. It operates on two sets and a message. The set $V$ contains messages that must be seen on the path while the set $N$ contains messages that must not appear on the path. The message $e$ is the most recently observed message in the sequence and is used to validate message order. The recursion terminates when there are no more messages that need to be observed ($V = \emptyset$) or a term violates any of the four properties. The required properties are, first, a required message, $e'$, appears on the path ($S, \pi^i \models e'$); second, $e'$ follows the defined partial order in the chart ($e' \not\prec e$); third, nothing between $e'$ is in the set of messages that must not appear in the chart ($\forall j < i, \forall e'' \in V \cup N, S, \pi^j \not\models e''$); and fourth, the recursive call with updated sets and $e'$ is satisfied ($S, \pi^{i+1} \models V \setminus \{e', N \cup \{e', e''\}$). Notice that $e'$ in the recursive call becomes the last seen message on the path, and as such, it is removed from the required set and added to the set of message that must not be seen since duplicate messages are not allowed in a chart.
B.2 Proofs

Lemma B.2.1 For any three messages \(x_t, x_u,\) and \(x_v\) and a trace \(\pi = s_0, s_1, \ldots\)

\[M, \pi \models (\neg x_v \cup x_u) \land (\neg x_u \cup x_t) \Rightarrow M, \pi \models (\neg x_v \cup x_t).\]

Proof From the inductive definition of the \(\models\) relation for \(\cup\), if the left hand side of
the implication is to hold, then \(\exists k_2, k_1 \geq 0\) such that \(M, \pi^{k_1} \models x_u\) and \(M, \pi^{k_2} \models x_t\).
Consider the case where \(k_1 < k_2\). This implies that \(\pi\) has the form \(\ldots, x_u, \ldots, x_t, \ldots\) which by definition gives \(M, \pi \not\models \neg x_u \cup x_t\). This causes the implication to hold since
the left hand side does not satisfy the relation.

Now consider the case where \(k_1 > k_2\). In this scenario, \(\pi\) orders \(x_u\) after
\(x_t\) which does not immediately invalidate the \(\models\) relation on the left-hand side of
the implication. Again from the inductive definition, if the left hand side of
the implication is to hold, then \(\forall 0 \leq j_1 \leq k_1\), \(M, \pi^{j_1} \models \neg x_v\) and \(\forall 0 \leq j_2 \leq k_2\), \(M, \pi^{j_2} \models \neg x_u\) giving \(\pi\) the form \(\ldots, x_t, \ldots, x_u, \ldots\) which by definition of the \(\models\) relation for \(\cup\)
gives \(M, \pi \models (\neg x_v \cup x_t)\).

It should be noted that the existence of \(x_v\) is not forced but the existence of
\(x_t\) and \(x_u\) is forced because of the \(\cup\) operator. If \(x_v\) does occur, it will have to occur
after \(x_t\) and \(x_u\); thus satisfying the implication.

Lemma B.2.2 For a given set of messages \(N\) and a message \(x_i\) such that \(\forall x_j \in \)
\(N, x_i < x_j\)

\[M, \pi \models \bigwedge_{x_j \in N} (\neg x_j \cup x_i) \Leftrightarrow M, \pi \models (\bigwedge_{x_j \in N} \neg x_j) \cup x_i.\]

Proof Given the ordering of the messages \(x_t, x_u, x_v\), we know that \(\exists k_1, k_2 \geq 0\)
such that, \(M, \pi^{k_1} \models x_t, \forall i \leq k_1, \pi^i \models \neg x_u\) and \(M, \pi^{k_2} \models x_t, \forall j \leq k_2, \pi^j \models \neg x_v\).
Since we are given the ordering of messages \(x_t, x_u, x_v\) and we know that message \(x_t\)
only occurs once, it follows that \(k_1 = k_2\) and \(\forall m \leq k_1, \pi^m \models (\neg x_u \land \neg x_v)\); thus
\((\neg x_v \land \neg x_u) \cup x_t\). Similarly the reverse side of the double implication can be proven by definition as well.

**Lemma B.2.3** Given \(k\) messages in a set \(N\) that occur in the order \(e_1 \prec \ldots \prec e_k\), if a message \(e_i\) does not re-occur between its occurrence and the final message \(e_k\), then it does not re-occur between its occurrence and any intermediate message \(e_j\).

\[
M, \pi \models \bigwedge_{e_i \neq e_j} \neg \chi_{e_j, e_i} \iff M, \pi \models \bigwedge_{e_i \in N} \neg \chi_{e_i, e_k}.
\]

**Proof** Given the order of the messages \(e_1, \ldots, e_k\), we know that the message \(e_i\) does not occur between its first occurrence and the last message \(e_k\). So for any message \(e_j\) that occurs between the message \([e_i, e_k]\), we know that \(e_i\) occurs only once before the message \(e_j\) and never again; thus, we know that for all messages occurring between \(e_i\) and \(e_k\), the message \(e_i\) does not occur again.

It should be noted that this result is only valid for a single unique final message in the chart. In the case of multiple possible final messages in the chart (unordered final messages), the uniqueness of each message \(e\) will have to be established with respect to each possible maximal message; thus giving us the formula

\[
\bigwedge_{(e, p) \in \text{msg}(c) \times \text{max}_m(c)} \neg \chi_{e, m}.
\]

**Theorem B.2.4** The improved translation as presented in Equation 3.2 and Equation 3.3 is equivalent to the quadratic translation presented in [35].

\[
M, \pi \models \psi_c \iff M, \pi \models \psi'_c.
\]
**Proof** The top term on the left hand side for each translation is required to establish the order of events in the pre-chart. Using the results from Lemma 3.5.1 and Lemma 3.5.2 we know that the following holds for the pre-chart messages:

\[
M, \pi \models \bigwedge_{p_i \prec p_j} \phi_{p_i, p_j} \iff \bigwedge_{p \in \text{msg}(c)} \phi'_{p, \text{next}(c)(p)} \]  

(B.1)

Once the order of the pre-chart messages has been established, we have to guarantee the non-existence of all the main chart events. This is required to frame the pre-chart correctly. From the result of Lemma 3.5.1 and Lemma 3.5.2 we know that if the ordering of the pre-chart events hold, by guaranteeing the order of all the main chart messages with the maximal messages of the pre-chart, we guarantee the non-existence of all the messages of the main chart with all the messages of the pre-chart; thus, we get a conjunction of two terms on the right hand side (one term to establish pre-chart order, same as in Equation B.1 and one to establish order between maximal pre-chart events and main chart events):

\[
M, \pi \models \bigwedge_{p_i, m_j} \phi_{p_i, m_j} \iff \\
M, \pi \models \left( \bigwedge_{p \in \text{msg}(c)} \phi'_{p, \text{msg}(c)} \bigwedge_{p \in \text{msg}(c)} \phi'_{p, \text{next}(c)(p)} \right) \]  

(B.2)

Now we have to establish the uniqueness of each element in the pre-chart. We establish the uniqueness of each element with respect to every maximal message in the pre-chart. This guarantees that each message occurs only once in the pre-chart before the maximal message of the pre-chart. Again, we are depending on the order that is established in the pre-chart messages and the messages of the main chart. If the order does not hold, then the uniqueness cannot be established with the reduced set of properties, but if the order holds, the reduced set of properties is sufficient. We
recall the result of Lemma 3.5.4 to achieve a reduction in the number of terms and get the following:

\[ M, \pi \models \bigwedge_{p_i \neq p_j} \neg \chi_{p_j, p_i} \iff \left( \bigwedge_{(e, p) \in \text{msg}_c \times \text{max}_c} \neg \chi_{e, p} \right) \wedge \left( \bigwedge_{p \in \text{max}_c} \phi'_{p, \text{msg}_m(c)} \right) \wedge \left( \bigwedge_{p \in \text{msg}_c} \phi'_{p, \text{next}(c)(p)} \right) \]

(B.3)

In the next step we establish the order of the messages of the main chart by only explicitly ordering the non-transitive messages, as done so for the messages in the pre-chart. Again, we use the results of Lemma 3.5.1 and Lemma 3.5.2 to get the following equivalence:

\[ M, \pi \models \bigwedge_{m_i \prec m_j} \phi_{m_i, m_j} \iff M, \pi \models \bigwedge_{m \in \text{msg}_m(c)} \phi'_{m, \text{next}(c)(m)} \]

(B.4)

Now we guarantee the uniqueness of each element in the main chart and the pre-chart with respect to all the maximal messages of the main chart. We know that this is equivalent to establishing the uniqueness of every term with every other term by the result of Lemma 3.5.4 and the ordering of the messages; thus, the following:

\[ M, \pi \models \bigwedge_{e_i \neq e_j} \neg \chi_{e_i, e_j} \iff \left( \bigwedge_{e \in \text{msg}_m(c)} \phi'_{e, \text{next}(c)(e)} \right) \wedge \left( \bigwedge_{(e, p) \in \text{msg}(c) \times \text{max}_c} \neg \chi_{e, p} \right) \]

(B.5)
In the quadratic translation, we have to guarantee that the final messages of the chart occur. In Equation B.4 we produce a $\phi'$ formula for every message in the main chart. Since the final message is ordered relative to $\bot$, the ordering formula will produce messages of the nature ($true \ U e$) where $e$ is the final message. This formula implicitly forces the existence of the message $e$; thus, we do not have to explicitly force the existence of the final messages as is the case in the quadratic translation. A conjunction of the Equation B.1–Equation B.5 above yields $M, \pi \models \psi_c \iff M, \pi \models \psi'_c$.

\section*{B.3 Translation of Additional Constructs to Temporal Logic}

\subsection*{B.3.1 Asynchronous Messages}

Fig. B.2(a) shows two asynchronous messages $a$ and $b$ that are sent from the \textit{initiator} process to the \textit{target} process. The corresponding lattice depicting the partial order of the events of the LSC is shown in Fig. B.2(b). In the lattice, all send events are written as the letter of the message followed by a “!” and all receive events are written by a “?”.

For the example chart shown above, the equivalent temporal logic is as follows:
Figure B.3: An LSC illustrating a coregion and the corresponding lattice depicting the partial order of events (a) The LSC. (b) The partial order defined on the messages in the LSC.

\[ G (\neg a? U b!) \land (\neg b? U a!) \land (F a?) \land (F b?) ). \]

The above temporal logic formula specifies that \( a \) cannot be received until \( b \) has been sent, \( b \) cannot be received until \( a \) has been sent and eventually both \( a \) and \( b \) are received.

### B.3.2 Coregions

Fig. B.3(a) shows an example LSC where the messages \( b \) and \( c \) are unordered.

For the coregion shown in Fig. B.3(a), the generate temporal logic is as follows:

\[ G (\neg b U a) \land (\neg c U a) \land (\neg d U b) \land (\neg d U c) \land (F d)). \]

### B.3.3 Bonded Conditions

Fig. B.4(a) shows an example of a bonded condition. The condition \( c_1 \) occurs in a simultaneous class along with message \( b \). Before the simultaneous class, message \( a \) occurs and message \( c \) occurs after the simultaneous class. The translation to temporal logic is as follows:

\[ G ((\neg(b \land c_1) U a) \land (\neg c) U (b \land c_1) \land F c) \]
Figure B.4: An LSC illustrating a bonded condition in a simultaneous class and the corresponding lattice for the LSC (a) The example LSC with a bonded condition and (b) the lattice for the example LSC.

The above formula describes the lattice as shown in Fig. B.4(b). The condition and the message in the simultaneous class are treated as a single event. Since they are treated as a single event, they can be easily expressed in temporal logic. Non-bonded conditions on the other hand, do not occur along with a synchronized message in a simultaneous region; thus, the condition cannot be treated as a unique event, since it can be satisfied at any point of the execution of the system. Non-bonded conditions are not very common in practice and have unwanted side effects that make them less appealing than bonded conditions [30, 20]. Also, it should be noted that invariants can be translated a set of bonded conditions along with the events that occur in the interval as defined by the invariant; thus, translating invariants to temporal logic is equivalent to attaching a condition to each event in the interval of the invariant.
Appendix C

Specifying Multi-Agent Systems Using Live Sequence Charts

C.1 Proofs

Theorem C.1.1 The \( LTL \) translation of LSCs (\( \psi^{LTL}_c \)) lies in the common fragment of \( CTL \) and \( LTL \) and the equivalent formula can be obtained by adding the universal path quantifier to each temporal operator in the \( LTL \) formula of Equation 4.1.

Proof Using the definitions and results listed above, we can construct an \( ACTL^{det} \) formula \( \phi_m \) as follows:

\[
\begin{align*}
\phi_0 &= p_2, \phi_2 = p_1 \\
\phi_1 &= A((\neg p_2 \land true) \cup (p_2 \land true)) = A(true \cup p_2) \\
\phi_3 &= A((\neg p_1 \land \neg p_2) \cup (p_1 \land true)) = A(\neg p_2 \cup p_1) \\
\phi_4 &= A((\neg p_1 \land \neg p_2) \cup (p_1 \land true)) = A((\neg p_1 \land \neg p_2) \cup p_1) \\
\phi_5 &= AX(\phi_4) = AX \\
\phi_6 &= p_1 \land \phi_5 = p_1 \land AX((\neg p_1 \land \neg p_2) \cup p_1) \\
\phi_7 &= A((\neg p_1 \land \neg p_2) \cup \phi_6) = \\
&\quad A((\neg p_1 \land \neg p_2) \cup (p_1 \land AX(A((\neg p_1 \land \neg p_2) \cup p_1)))).
\end{align*}
\]

The formula constructed above describes the order of the messages \( p_1 \) and \( p_2 \) as well as the uniqueness of message \( p_1 \) with respect to \( p_2 \). We only show the construction of the ordering and uniqueness formulas for two messages, but the formula can be constructively created for an arbitrary number of messages. Using the constructive
method shown above, we can now create two formulas $\theta_p$ and $\theta_m$ that completely describe the behaviors of the prechart and the main chart. We also introduce a new label $l_{\theta_p}$ that is true in the states where the prechart formula $\theta_p$ is satisfied (by using a correct CTL labeling algorithm). The $\text{ACTL}^{\text{det}}$ formula for the entire chart can now be created using the following construction:

\[
\begin{align*}
\phi_0 &= \theta_m \\
\phi_1 &= \neg l_{\theta_p} \\
\phi_2 &= (\phi_1 \land \text{true}) \lor (\neg \phi_1 \land \phi_0) \\
\phi_3 &= \neg \phi_1 \rightarrow \neg \phi_1 \land \phi_0 \\
\phi_4 &= \neg \phi_1 \rightarrow \phi_0 \\
\phi_5 &= l_{\theta_p} \rightarrow \theta_m \\
\psi_{\text{ACTL}^{\text{det}}}^{\text{c}} &= AG(l_{\theta_p} \rightarrow \theta_m) = AG(\theta_p \rightarrow \theta_m)
\end{align*}
\]

The final formula $\psi'$ is an $\text{ACTL}^{\text{det}}$ formula that is only true if the prechart is never satisfied or if the prechart and the main chart is satisfied. Using the label $l_{\theta_p}$ is used as the right side of the implication, the LTL formula in Equation 4.1 can be re-written as $G(l_{\theta_p}) \rightarrow \theta_m$. Using the result of Theorem 4.4.3 we can remove all path quantifiers from the constructed formula and obtain an LTL formula that has the exact same form as

Since the resulting formula will be an $\text{ACTL}^{\text{det}}$ formula, we can now remove all the path quantifiers and create the equivalent LTL formula using the results of Theorem 4.4.2 and Theorem 4.4.3. The LTL formula that results is exactly the same formula as shown in Equation 4.1.
Appendix D

Improving Live Sequence Chart to Automata Transformation for Verification

D.1 Proofs

Lemma D.1.1 For all states containing outgoing main chart transitions, the transition relation is total. Formally, given a state \( q \) with a main chart transition

\[
\left( \bigvee_{\forall \phi_i, q_i : (q, \phi_i, q_i) \in \Delta} \phi_i \right) = \text{true}.
\]

Proof Since there are three types of transitions for a state containing an outgoing main chart transition, \( \phi_{\text{self}} \lor \phi_{\text{safety}} \lor \bigvee_{\forall \phi', q' : (q, \phi', q') \in \Delta} \phi' \) must hold for the transition set to be total. From Figure 5.3 and our algorithm shown in Figure 5.4 the condition holds by definition.

Lemma D.1.2 For states \( q \) in the transformed automaton (except the initial state), the transition relation is deterministic. Formally, \( \forall q \in Q, \forall \phi_i, \phi_j : (q, \phi_i, q_i) \in \Delta \land (q, \phi_j, q_j) \in \Delta, (\phi_i \land \phi_j) = \text{false}. \)

Proof For a state \( q \) containing an outgoing main chart transition, the pairwise conjunction of all outgoing transitions from state \( q \) results in \text{false} by definition of the
algorithm in Figure 5.4.

\[ \phi_{\text{safety}} \land \phi_{\text{self}} = \text{false} \]
\[ \forall q' : \{q, \phi, q'\}, \phi_{\text{safety}} \land \phi_{\text{child}_{q'}} = \text{false} \]
\[ \forall q' : \{q, \phi, q'\}, \phi_{\text{self}} \land \phi_{\text{child}_{q'}} = \text{false} \]

For a state \( q \) that does not contain any outgoing main chart transitions, the \( \phi_{\text{safety}} \) transition is not added to the outgoing transition set; thus, it is sufficient to prove
\[ \forall q' : \{q, \phi, q'\}, \phi_{\text{self}} \land \phi_{\text{child}_{q'}} = \text{false} \]

**Theorem D.1.3** The automaton, \( A \), generated by the transformation algorithm in Figure 5.4 for a given LSC, \( \text{SPEC} \), defined over an alphabet \( \Sigma_{\text{SPEC}} \subseteq \Sigma \), reads exactly the complement of the language of the \( \text{SPEC} \). Formally, \( \forall \theta = \theta_0 \theta_1 \theta_2 \ldots \)

\[ [\theta \in L(\text{SPEC}) \implies \theta \notin L(A)] \land [\theta \notin L(\text{SPEC}) \implies \theta \in L(A)] \]

where \( L(A) \) and \( L(\text{SPEC}) \) are the languages of the transformed automaton and the \( \text{SPEC} \).

**Proof** We need to consider both cases where words are read and not read by the automaton \( A \). We proceed using proof by contradiction on the first case. Assume that \( \theta \) is read by \( \text{SPEC} \) and \( A \). \( \text{SPEC} \) only reads \( \theta \) if it contains at least one instance of the chart and every instance terminates in a valid end state. Let \( (v_0, v_1, \ldots, v_n) \) be a sequence of monotonically increasing indexes such that \( (\theta_{v_0}, \theta_{v_1}, \ldots, \theta_{v_n}) \) is an instance of \( \text{SPEC} \) in \( \theta \) and all intermediate states between any \( \theta_{v_i} \) and \( \theta_{v_{i+1}} \) are not relevant to \( \Sigma_{\text{SPEC}} \). The automaton can only read the instance of \( \text{SPEC} \) if it results in a safety or liveness violation by definition. For safety violations, there must exist an \( i < n \) such that reading \( (\theta_{v_i}, \theta_{v_1}, \ldots, \theta_{v_n}) \) places the automaton in state \( q_i \), and state \( q_i \) has an edge \( (q_i, \phi, q_{\text{safety}}) \in \Delta \) where \( \phi(\theta_{v_{i+1}}) \) evaluates to true—meaning the next state in the chart instance moves the automaton to the safety accept state.
By Lemma D.1.1 and Lemma D.1.2 only one transition is enabled from state $q_i$ on
$\theta_{v_{i+1}}$ for any $i$. By the algorithm in Figure 5.4 the one enabled transition must be a
$\phi_{\text{child}}$ because any other transition would not move the $SPEC$ towards a valid end
state to contradict the original assumption that $\theta \in L(A)$.

For liveness violations, there must exist an $i < n$ such that reading $(\theta_{v_0}, \theta_{v_1},$
$\ldots, \theta_{v_i})$ places the automaton in state $q_i$ (marked as an accept state), and $\forall j >$
$i, (q_i, \phi, q_i) \in \Delta \land \phi(\theta_{v_j})$ is true—meaning for all letters in the word $\theta_j, \theta_{j+1}, \ldots$, no
relevant message/condition is observed and progress is never made. By Lemma D.1.1
and Lemma D.1.2, only one transition is enabled from state $q_i$ for all $\theta_{v_j}$. Additionally,
by the algorithm in Figure 5.4 the one enabled transition must be $\phi_{\text{child}}$ because we
know the letter is relevant to move the $SPEC$ towards a valid end state to contradict
the original assumption that $\theta \in L(A)$. The other direction to show a word not read
by $SPEC$ but read by $A$ is proven in a similar fashion but omitted due to space.
Appendix E

Verifying Communication Protocols Using Live Sequence Chart Specifications

E.1 Proofs

Lemma E.1.1 For all states containing outgoing main chart transitions, the transition relation is total. Formally, given a state $q$ with a main chart transition
\[
\left(\bigvee_{\forall \phi_i, q_i : (q, \phi_i, q_i) \in \Delta} \phi_i\right) = \text{true}.
\]

Proof Since there are three types of transitions for a state containing an outgoing main chart transition, $\phi_{\text{self}} \lor \phi_{\text{safety}} \lor \bigvee_{\forall \phi_i', q_i' : (q, \phi_i', q_i') \in \Delta} \phi_i'$ must hold for the transition set to be total. Using the definitions of the transitions from Section 6.4, we see that the condition holds by definition.

Lemma E.1.2 For states $q$ in the transformed automaton (except the initial state), the transition relation is deterministic. Formally, $\forall q \in Q, \forall \phi_i, \phi_j : (q, \phi_i, q_j) \in \Delta \land (q, \phi_j, q_j) \in \Delta, (\phi_i \land \phi_j) = \text{false}$.

Proof For a state $q$ containing an outgoing main chart transition, the pairwise conjunction of all outgoing transitions from state $q$ results in $false$ by definition of the transitions in Section 6.4.

\[
\begin{align*}
\phi_{\text{safety}} \land \phi_{\text{self}} &= false \\
\forall q' : \{q, \phi, q'\}, \phi_{\text{safety}} \land \phi_{\text{child}_{q'}} &= false \\
\forall q' : \{q, \phi, q'\}, \phi_{\text{self}} \land \phi_{\text{child}_{q'}} &= false.
\end{align*}
\]
For a state $q$ that does not contain any outgoing main chart transitions, the $\phi_{\text{safety}}$ transition is not added to the outgoing transition set; thus, it is sufficient to prove
\[ \forall q' : \{q, \phi, q'\}, \phi_{\text{self}} \land \phi_{\text{child}}_{q'} = false. \]

**Theorem E.1.3** The automaton, $A$, generated by the transformation algorithm for a given LSC (including Kleene star and subcharts), SPEC, defined over an alphabet $\Sigma_{\text{SPEC}} \subseteq \Sigma$, reads exactly the complement of the language of the SPEC. Formally,
\[ \forall \theta = \theta_0 \theta_1 \theta_2 \ldots \]
\[ [\theta \in L(\text{SPEC}) \implies \theta \notin L(A)] \land [\theta \notin L(\text{SPEC}) \implies \theta \in L(A)]. \]

where $L(A)$ and $L(\text{SPEC})$ are the languages of the transformed automaton and the SPEC.

**Proof** The proof follows the same format as presented in Theorem D.1.3.
Bibliography


