**Trace-Based Network Proof Systems:
Expressiveness and Completeness**

Jennifer Widom
Ph. D. Thesis

87-833
May 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

TRACE-BASED NETWORK PROOF SYSTEMS: EXPRESSIVENESS AND COMPLETENESS

A Thesis

Presented to the Faculty of the Graduate School
of Cornell University

in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by

Jennifer Widom

May, 1987

© Jennifer Widom 1987
ALL RIGHTS RESERVED

TRACE-BASED NETWORK PROOF SYSTEMS: EXPRESSIVENESS AND COMPLETENESS

Jennifer Widom, Ph.D.
Cornell University 1987

Most trace-based proof systems for networks of processes are known to be incomplete. Extensions to achieve completeness are generally complicated and cumbersome. In this thesis, we isolate the components of a trace-based network proof system that are necessary and sufficient to achieve relative completeness. We then consider the expressiveness required of any trace logic that encodes these components.

A simple trace-based proof system is defined and several examples are presented to show its inherent incompleteness. Surprisingly, two examples consist of only one process, indicating that network composition is not required for incompleteness. Two computational properties are then identified that must be axiomatized within a relatively complete proof system. We show that axiomatization of these properties is also sufficient for achieving relative completeness.

We consider the expressiveness required of a trace logic if it is to axiomatize those properties necessary for relative completeness. We prove that first-order trace logic is not strong enough for this. The necessary properties can be expressed in temporal logic, but temporal logic is more powerful and complex than needed. A hierarchy of temporal logic subsets is defined; a subset consisting of first-order trace logic with a version of the temporal *Always* operator is shown to be necessary and sufficient for expressing the required properties.

Biographical Sketch

Jennifer Widom was born in Ithaca, New York on October 20, 1960. She graduated from Cornell Nursery School in 1965 and moved to California shortly thereafter. Jennifer graduated from Santa Cruz High School in 1977 and set off to study trumpet at the Indiana University School of Music. In 1982, she received an interesting degree called “Bachelor of Science in Music with Outside Fields in Mathematics and Computer Science”. Ms. Widom remained at Indiana University to earn a Master of Science in Computer Science, which was conferred in 1983. She then returned to the place of her nursery schooling and was, two years later, awarded another Master of Science in Computer Science, this time by Cornell University. Jennifer’s final academic degree, a Ph.D. in Computer Science, was conferred by Cornell University in Ithaca, New York on May 31, 1987.

Acknowledgements

Sincere thanks go to David Gries, for being a terrific advisor, to Fred Schneider, for acting as a second (but hardly secondary) advisor, to Prakash Panangaden, for a whole bunch of truly valuable help, and to Alex Aiken, for too much to list. I am also grateful to Willem de Roever and Dexter Kozen, for useful discussions, and to committee members Ed Murray and Tim Teitelbaum.

Table of Contents

1	Introduction	1
1.1	Organization and Summary of the Thesis	2
1.2	Proof Systems for Sequential Programs	5
1.3	Proof Systems for Concurrent Programs	6
1.4	Proof Systems for Networks	7
1.4.1	Levin and Gries's Proof System for CSP	8
1.4.2	Hoare's Formalism and Proof System for CSP	9
1.4.3	Misra and Chandy's Proof System for Networks of Processes	11
1.4.4	Hehner and Hoare's Model of Communicating Processes . .	12
1.4.5	Nguyen's Model and Temporal Proof System for Networks of Processes	14
1.4.6	Milner's Calculus of Communicating Systems	16
1.5	Expressiveness and Completeness	17
2	A Model for Networks of Processes	19
2.1	Process Networks	19
2.2	A Model of Computation	21
2.2.1	Computation Trees for Processes	22
2.2.2	Computation Trees for Networks	22
2.2.3	Paths and Computations	25
2.3	Modification for Asynchronous Networks	28

2.4	Hierarchical Networks	31
3	Simple Trace Logic	33
3.1	Specification Language	33
3.1.1	Syntax	35
3.1.2	Semantics	35
3.2	Axioms and Inference Rules	39
3.3	Modifications for Asynchronous and Hierarchical Networks	40
3.3.1	Asynchronous Networks	40
3.3.2	Hierarchical Networks	41
3.4	Comparison with Other Trace Logics	42
3.5	Soundness of STL	44
3.6	The Completeness Question	45
3.6.1	Precise Specifications	46
3.6.2	Relative Completeness	48
3.7	Incompleteness of STL	48
3.7.1	Brock-Ackerman Example	49
3.7.2	Every-Other Example	51
3.7.3	Single Process Example #1	52
3.7.4	Single Process Example #2	53
3.8	Completeness and Incompleteness of Other Trace Logics	54
4	Completeness in Trace-Based Proof Systems	59
4.1	The Need for Computation Characterization	59
4.2	The Temporal Ordering Property	61
4.3	The Prefix Property	63
4.4	Necessity and Sufficiency of Temporal Ordering and Prefix Axioms	65
4.4.1	Sufficiency	66
4.4.2	Necessity	69

4.5	Example Encodings of the Axioms	70
4.6	Expressing the Axioms	72
5	A Model-Based Generalization	74
5.1	Models of First-Order Formulas	75
5.2	Computational Models of First-Order Formulas	77
5.3	Computational Models and Proof Systems	79
5.3.1	Valid and Precise Specifications	80
5.3.2	Computational Models in STL	82
5.3.3	Computational Models in Other Trace Logics	85
5.4	Recognizing Computational Models	87
6	Temporal Logic and Computational Models	91
6.1	Temporal Logic	93
6.1.1	Syntax	93
6.1.2	Semantics	94
6.2	Mapping Temporal Logic to Extended Predicate Logic	97
6.2.1	Definition of the Mapping	98
6.2.2	Example of the Mapping	98
6.2.3	Correctness of the Mapping	100
6.3	Revisions to Formula $CM(f)$	105
7	Expressiveness Bounds for Relative Completeness	110
7.1	Temporal Operators over Non-Temporal Formulas	111
7.2	Subsets of Temporal Logic	113
7.3	Necessity and Sufficiency of the <i>Unrestricted Always</i> Operator	114
7.3.1	Sufficiency	115
7.3.2	Necessity	118
7.4	Refining the Subset Hierarchy	125
7.5	A Minimally Complete Proof System	127

8	Conclusions	131
8.1	Summary	131
8.2	Implications	132
8.3	Extensions	133
8.4	Changing the Assumptions	134
8.5	Future Work	136
	Bibliography	138

List of Tables

3.1	Syntax of STL	36
3.2	Syntactic Abbreviations	37
6.1	Syntax of Temporal Logic	95
6.2	Syntactic Abbreviations	95
6.3	Transformation Function $\mathcal{T}$	99

List of Figures

2.1	A network of processes	20
2.2	Example network	22
2.3	Computation tree for process <i>MERGE</i>	23
2.4	Computation tree for process <i>BUFFER</i>	23
2.5	Computation tree for <i>MERGE</i> <i>BUFFER</i>	25
2.6	Computation tree for asynchronous <i>MERGE</i> <i>BUFFER</i>	30
3.1	Process <i>P</i>	34
3.2	Modified Brock-Ackerman network	49
3.3	Every-other network	51
3.4	Single-process network	52
3.5	Simplest network of all	54
5.1	Models and computational models for specifications <i>S1</i> and <i>S2</i> . .	83
6.1	Semantics-preservation	92
7.1	Subsets of temporal logic	114
7.2	Expressiveness bounds	115

Chapter 1

Introduction

A *program proof system* is a formalism for the specification and verification of computer programs. A program specification is typically a formula or set of formulas in some *specification language*. To verify that a program meets its specification, a *logic*—axioms and inference rules for reasoning about programs and specifications—is defined. A programming logic provides its user with a formal technique for developing correct programs.

Proof systems have been developed for sequential programs [11,12,16,19,40], for concurrent programs [14,27,28,29,32,34,35,36,43,44,45], and for distributed programs, or process networks [2,8,10,20,22,25,30,39,42,48,50,56]. We consider the class of *compositional* proof systems for process networks. A proof system is compositional if the specification for a network can be completely deduced from specifications for its component processes. Most compositional network proof systems are *trace-based*: in them, one specifies and reasons about traces (histories) of the values transmitted along the communication channels of the network.

We are interested in the completeness or incompleteness of trace-based network proof systems. A proof system is *complete* if every valid specification for every network can be verified using the given logic. Trace-based proof systems

are defined in [10,20,22,39], but they are incomplete [7,41]. Simple trace logics are modified to increase expressiveness in [25,46] and to obtain completeness in [7,17,42,56]. The modifications tend to be extensive and cumbersome; the simplicity of the underlying logic is lost.

Our goal is to isolate the components of a trace-based network proof system that are necessary and sufficient to achieve relative completeness¹. Incompleteness in simple trace-based proof systems is explored, and behavioral properties that must be axiomatized in any relatively complete trace logic are identified. These properties are not expressible in a simple trace-based logic; we therefore examine the expressiveness required of a logic if it is to be used for a relatively complete proof system.

1.1 Organization and Summary of the Thesis

In this introductory chapter, we give a short history and description of proof systems for sequential programs. Proof systems for concurrent (shared memory) programs are then described, followed by the presentation of several existing proof systems for networks of processes. Expressiveness and completeness in existing trace-based network proof systems is briefly discussed, as an introduction to the remainder of the thesis.

Chapter 2 defines the class of synchronous process networks used in our work and presents a formal model of process and network computation. The model is an abstraction of computation used to describe process and network behavior and to reason about properties of network proof systems. We base our model on the *computation tree*, which captures all possible behaviors of a given process or

¹Program proof systems are usually based on first-order predicate logic, which itself is incomplete [13,49]. A proof system is *relatively complete* if, under the assumption that predicate logic is complete, every valid specification is provable.

network. We show that the computation-tree model can, if desired, be extended to accommodate asynchronous communication and hierarchically structured networks.

In Chapter 3, we describe *Simple Trace Logic* (*STL*), a formalism and proof system for network specification and verification that captures the essence of most trace-based systems. The syntax and semantics of STL's specification language are defined, and a set of axioms and inference rules are given. We show that STL can also be extended for asynchronous communication and hierarchical networks.

The incompleteness of STL is illustrated by several example networks. Two examples consist of only a single process; these networks demonstrate that, while compositionality is an important feature of trace-based logics, incompleteness is caused not by network composition but by the inability to reason about certain inherent properties of network computation.

In Chapter 4, the single-process incompleteness examples of Chapter 3 are used to identify two extensions to simple trace logics that are necessary and sufficient to achieve relative completeness. The first source of incompleteness is the inability to state and reason about constraints on the temporal ordering of network events. The second source is the inability to assert that the sequence of values transmitted along a communication channel is always a prefix of that channel's sequence at some later point. We formally prove that these two properties—the temporal ordering and prefix properties—must be available as reasoning tools in any relatively complete proof system. We also prove that adding temporal ordering and prefix axioms to a trace logic suffices for achieving relative completeness.

The temporal ordering and prefix axioms of Chapter 4 are not expressible in STL. Consequently, in Chapter 5, we consider the expressiveness required of a trace logic if it is to axiomatize properties that must be encoded in a complete proof system. This expressiveness requirement is formally characterized using

standard models of first-order formulas—the models correspond to single states of network computation. We show that a logic must be just expressive enough to distinguish those states that are reachable by a legal network computation from those that are not. Using an extended predicate logic (*EPL*), a formula is defined that recognizes all and only states reachable by legal computation. This formula, therefore, exactly characterizes the expressiveness required of a language if it is to be used for a relatively complete trace logic.

Formal reasoning using the extended predicate logic of Chapter 5 can involve the enumeration of an exponential number of sequences of network states, corresponding to all possible network computations. Therefore, although sufficiently expressive, EPL is unsuitable for use in an actual proof system. *Temporal Logic* (*TL*) is an existing logic for reasoning over sequences of states [37,47] and is appropriate for expressing properties of network computation. Chapter 6 describes TL and develops the necessary tools for applying TL to the expressiveness requirement characterized in Chapter 5.

TL turns out to be far more powerful—and thus more complex—than is needed for the computational properties we are interested in encoding. In Chapter 7, we introduce a hierarchy of weaker versions of TL; we then use the tools of Chapter 6 to isolate a subset of TL that is necessary and sufficient to express a formula equivalent to the EPL formula of Chapter 5. This subset thus has the exact expressive power required of a relatively complete trace logic. The result is applied in the description of a trace-based proof system that is minimally complete with respect to our hierarchy of TL subsets—any weaker logic results in an incomplete proof system.

In Chapter 8, we summarize our results and discuss implications of the work. Potential extensions are considered, and we describe how changes in the assumptions can affect the methods and results. We conclude by proposing future directions of research.

1.2 Proof Systems for Sequential Programs

The importance of formal program correctness proofs was initially discussed by Naur in 1966 [40], although only informal specification techniques were introduced. In 1967, Floyd suggested associating logical assertions with the edges of a program flow-chart and showing that whenever program execution reaches an edge, the corresponding assertion is true [15]. In 1969, Hoare expanded and formalized Floyd's idea by defining a logical system of axioms and inference rules for proving correctness of sequential programs [19].

Hoare's logical system provided both an axiomatic definition of a programming language and a proof system for program correctness; the work founded a school of research in this area. A number of subsequent achievements in the field are listed in [16]; further historical notes, extensive axiomatic proof systems, and the use of this approach as a programming methodology can be found in [12,16].

Axiomatic program proof systems like those in [12,16,19] are based on formulas of the form $\{P\} S \{Q\}$, called *triples* (or *Hoare triples*). S is a simple or compound program statement; P and Q are assertions—formulas of first-order predicate logic [13,16] in which the free variables correspond to program variables in S . Formula $\{P\} S \{Q\}$ asserts that if statement S begins execution in a state satisfying P ,² and if S terminates, then S terminates in a state satisfying Q .³

Since S can be any compound statement—including an entire program—a triple $\{P\} Pgm \{Q\}$ is used to specify and verify program Pgm . Axioms are provided for proving correctness of triples for simple statements; inference rules are applied to build proofs of entire programs. Such correctness proofs usually

²A program state satisfies an assertion if replacing the assertion's free (program) variables by the corresponding values in the program state renders the assertion true.

³We have defined *partial correctness*. For *total correctness*, $\{P\} S \{Q\}$ requires the termination of S .

take the form of *proof outlines*, in which the program is given with (provable) assertions inserted at appropriate points [12,16,19].

1.3 Proof Systems for Concurrent Programs

Axiomatic proof systems for shared-memory concurrent programming languages emerged several years after the introduction of sequential program verification. Initial work on concurrent program verification was done by Owicki and Gries [43,44]. In [43], a proof system is defined in which the individual processes of a concurrent program are first considered in isolation—existing methods for sequential program verification can be used for this. The process proofs are then shown to be *interference-free*: execution of any process cannot invalidate assertions in the proof of another process. Finally, the interference-free process proofs are combined into a proof for the entire concurrent program. In [44], the non-interference proof is eliminated by guaranteeing that only one process at a time has access to shared resources. As a result, however, the language is considerably more restrictive.

A different approach is taken by Flon and Suzuki in [14], where a proof system is devised based on the fact that shared-memory concurrent programs executing on conventional machines have equivalent non-deterministic sequential counterparts. Thus, to prove correctness of a concurrent program, the program is first expressed in an equivalent non-deterministic form. The desired properties can then be verified using a proof system for non-deterministic sequential programs. Unfortunately, the authors do not demonstrate their system on any real examples.

The proof systems of [43,44] turn out to be inadequate for many simple programs: to obtain correctness proofs, *auxiliary variables* must be added to the programs. Auxiliary variables do not affect the flow of control or the values of program variables, but are necessary for assertional reasoning about control

locations of concurrently executing processes. Lamport eliminates the need for auxiliary variables by introducing a logic in which assertions use special control location predicates (*at*, *in*, and *after*) as well as program variables [29,28,27]. The proof systems consist of Hoare-style axioms and inference rules, along with axiom schemata to describe the effect of language constructs on control predicates.

In [45], Owicki and Lamport present a method for proving liveness properties of concurrent programs using temporal logic [47]. The application of temporal logic to the specification and verification of concurrent programs is also discussed in a series of reports by Manna and Pnueli [32,33,34,35,36,37]. Both approaches incorporate control predicates as in [27], using temporal logic for formal reasoning about the behavior of the program state—control points as well as variable values—over time.

A thorough history and survey of verification techniques for concurrent and distributed programs is given by Barringer in [3].

1.4 Proof Systems for Networks

A *network* (or *distributed system*) is a group of concurrently executing processes that communicate and synchronize not through shared memory, but by transmitting messages along communication channels. Theoretical work on process networks is often based on Hoare's *Communicating Sequential Processes* (*CSP*), a simple language for message-based concurrent programming [21,22,23]. When proof systems for concurrent programs were initially being developed, CSP was an established language capturing the important features of distributed programming. Consequently, a number of proof systems for CSP have been developed [2,8,10,17,20,22,30,48,50]; a representative sample are discussed below.

Misra and Chandy present a network proof system in [39] based on a language-independent view of process networks: processes are unspecified entities that read

and write values on their incoming and outgoing communication channels. Process specifications are assumed to be given in the form of assertions over histories of the values transmitted on the process's communication channels. Specifications for networks are built compositionally from specifications for the network's component processes.

An extension to this simple trace-based approach is discussed by Jonsson in [25]. In [42], Nguyen et. al. use temporal logic to develop a proof system similar to but stronger than [39]. The CSP proof systems described in [8,10,20,22], although language-dependent, also are based on specifications over channel traces. Most recently, an axiomatic trace-based proof system for a CSP-like language has been presented by Zwiers et. al. in [56].

We now describe, in somewhat more depth, several existing network proof systems. These formalisms are subsequently used as examples and in comparison with our Simple Trace Logic, defined in Chapter 3.

1.4.1 Levin and Gries's Proof System for CSP

An axiomatic proof system for Hoare's CSP is described by Levin and Gries in [30]; similar CSP proof systems are presented in [2,50]. The approach used in these systems is close to the shared-variable approach of Owicki/Gries [43] (recall Section 1.3). Sequential proofs for each process are first considered in isolation; to obtain the sequential proofs, however, assumptions about communication behavior must be made. This is done by allowing any assertion to appear after a *send* or a *receive* command. Axioms for the communication statements are thus $\{P\} \text{ send}(exp) \{Q\}$ and $\{P\} \text{ receive}(v) \{Q\}$.⁴ A *satisfaction proof* must be performed to ensure that the postulated assertions are valid when the processes run concurrently and communication actually takes place. For every potentially

⁴Rather than introduce the notation of CSP, we use a language that is comparable and self-explanatory.

matching pair of communication statements

$$\{P1\} \text{ send}(exp) \{Q1\} \quad \text{and} \quad \{P2\} \text{ receive}(v) \{Q2\},$$

the satisfaction proof requires showing

$$\{P1 \wedge P2\} \quad v := exp \quad \{Q1 \wedge Q2\}.$$

CSP processes do not share program variables, so it appears that a proof of non-interference like that in [43] is not required. As in [43], however, obtaining correctness proofs often requires (shared) auxiliary variables to be added to a program. In such cases, a non-interference proof must then be performed. The inference rule for deducing network specifications from sequential process specifications is thus:

$$\frac{\begin{array}{c} \{P_1\} S_1 \{Q_1\}, \{P_2\} S_2 \{Q_2\}, \dots, \{P_n\} S_n \{Q_n\} \\ \text{satisfied and interference-free} \end{array}}{\{\bigwedge_i P_i\} S_1 \| S_2 \| \dots \| S_n \{\bigwedge_i Q_i\}}$$

where $S_1 \| S_2 \| \dots \| S_n$ denotes concurrent execution of processes $S_1, \dots, S_n$.

Recall that compositionality in a network proof system requires that the specification for a network is completely deduced from specifications for its component processes. The above inference rule for deducing network specifications is not compositional: proving $\{\bigwedge_i P_i\} S_1 \| S_2 \| \dots \| S_n \{\bigwedge_i Q_i\}$ does involve reasoning over process specifications, but it also requires considering the process programs themselves, since satisfaction and non-interference proofs use individual program statements. Next, we present a compositional proof system for CSP.

1.4.2 Hoare's Formalism and Proof System for CSP

After introducing CSP, Hoare (and his students) developed models and proof systems for the language [8,10,17,20,22,23]. The proof system presented in [10] captures the essence of the work. In this formalism, communication is assumed

to take place along named channels; the channel names are then used to represent the sequences of values that have been transmitted along the corresponding communication links. A process specification is an assertion over these channel traces, with the assertion's free variables corresponding to the process's incident channels. An assertion is *satisfied* by a process if it is invariant—that is, if every possible computation up to any point in time yields channel traces satisfying the assertion. Formula $P \text{ sat } A$ is used to denote the fact that process P satisfies assertion A .

Axioms and inference rules are given to describe formally how values of channel trace variables in specifications are affected by execution of CSP language constructs. As an example, the rule for the *receive* statement is:

$$\frac{P[v/x] \text{ sat } A[c \cdot v/c], \text{ for all } v}{(c.\text{receive}(x) \rightarrow P) \text{ sat } A}$$

where

- $P[v/x]$ denotes program P with all (non-defining) occurrences of variable x replaced by value v ;
- $A[c \cdot v/c]$ denotes assertion A with value v appended to channel trace c ;
- $S \rightarrow P$ denotes the program consisting of statement S followed by program P .

The inference rule for deducing network specifications from process specifications is straightforward—the specification for a network is simply the conjunction of specifications for its component processes:

$$\frac{P_1 \text{ sat } A_1, P_2 \text{ sat } A_2, \dots, P_n \text{ sat } A_n}{P_1 \parallel P_2 \parallel \dots \parallel P_n \text{ sat } \bigwedge_i A_i}$$

This rule clearly results in a compositional proof system. Unfortunately, however, the logic is incomplete: there exist networks with valid specifications that cannot be verified using the given axioms and inference rules [7]. (We do not provide examples here; several incompleteness examples for a similar logic are given in Chapter 3.)

1.4.3 Misra and Chandy's Proof System for Networks of Processes

Misra and Chandy define a trace-based proof system for networks of processes similar to [10], but the formalism is concerned only with proofs of network specifications; process specifications are assumed to be given. As a result, the proof system is both compositional and language-independent. Unlike [10], in this formalism a trace is taken to be a single sequence describing all communication events up to a given point in time—an interleaving of the single-channel traces of [10]. Formal reasoning does not take place over this compound sequence, however. Process and network specifications are built from assertions in which only projections of the trace onto single channels—equivalent to Hoare's single-channel traces—actually appear. (A similar approach is used in some of Hoare's work, e.g. [22].)

A process h is specified by a pair of assertions r and s on the trace of h 's incident channels.⁵ This specification is denoted by $r|h|s$, indicating

1. s holds initially in h , and
2. if r holds up to point k in any trace of h , then s holds up to point $k + 1$ in

⁵To facilitate specification and verification of hierarchical networks, Misra and Chandy actually distinguish between *external* channels—those visible to the environment of the process or network—and *internal* channels—those not. For simplicity and brevity, we omit this distinction in our description of the proof system.

that trace, for all $k \geq 0$.

The rule for deducing network specifications is based on conjunctions of process specifications:

$$\frac{r_1|h_1|s_1, r_2|h_2|s_2, \dots, r_n|h_n|s_n}{(\bigwedge_i r_i) \mid (h_1 \parallel \dots \parallel h_n) \mid (\bigwedge_i s_i)}$$

This inference rule clearly results in a compositional proof system.

The proof system also includes a pair of consequence rules for verifying weak specifications from stronger ones:

$$\frac{s \wedge r \Rightarrow r', r'|h|s}{r|h|s} \qquad \frac{r|h|s', s' \Rightarrow s}{r|h|s}$$

Like [10,22], this proof system is incomplete [41]. We next describe a trace-based proof system that achieves relative completeness.

1.4.4 Hehner and Hoare's Model of Communicating Processes

A relatively complete trace-based formalism is presented by Hehner and Hoare in [17]. In this system, primitive processes are directly defined as predicates (corresponding to the process specifications of Misra/Chandy [39]), and these are assumed to be given. A process predicate has two free variables: *past* and *present*. At any given time, *past* is the sequence representing all communications that have taken place so far, while *present* is the set of messages that can be communicated in the next step. Variable *present* is used for reasoning about liveness, an issue not addressed in [10,39]. Variable *past* corresponds to the communication-event trace of [39]. Here, however, the compound trace can be explicitly used in formal reasoning. Projections of the trace onto individual channels are allowed as well.

Every process predicate P is required to satisfy

1. $(\forall past, present, msg:$

$$P \wedge msg \in present \Rightarrow$$

$$(\exists newpresent: P[past \cdot msg / past][newpresent / present]))^6, \text{ and}$$

2. $(\forall past, present, msg:$

$$P[past \cdot msg / past] \Rightarrow$$

$$(\exists oldpresent: P[oldpresent / present] \wedge msg \in oldpresent)).$$

Item 1 asserts that P is always satisfied when $past$ is extended by a message in $present$. Item 2 asserts that a nonempty $past$ can be shortened by removing its final message, i.e. predicate P is prefix-closed with respect to sequence variable $past$. Since a given process P can be described by many predicates satisfying the above criteria (for example, all processes can be described by the predicate *true*), P must be given an *explicit definition*: a strongest describing predicate, equivalent to the conjunction of all possible descriptions of P .

Several rules are provided for constructing compound processes from simpler ones. (These rules correspond closely to the language constructs of CSP as defined in [10,20,22].) In particular, a network is considered to be the compound process that concurrently executes some given set of processes. The given processes are assumed to have disjoint channel names; when concurrent composition takes place, channels are renamed to establish communication links. The formula for composing two processes $P1$ and $P2$ into a network N , linking channels $c1$ and $c2$ into a shared channel c , is

$$N \equiv P1 \wedge P2 \wedge$$

$$present(c) = present(c1) \cap present(c2) \wedge$$

$$past(c) = past(c1) = past(c2),$$

⁶Recall, from Section 1.4.2, the meaning of this notation: $X[a/b]$ denotes entity X with all occurrences of identifier b replaced by item a .

where $present(x)$ and $past(x)$ denote the restriction of $present$ and $past$ to communication events on channel x . This rule can be generalized (or used iteratively) for any number of concurrent processes and shared channels.

Although this system is compositional, deducing network specifications can be considerably more complicated than in [10] and [39]. The complexity results from allowing explicit reasoning over a single trace of all communication events. However, it is exactly this reasoning power that renders the logic relatively complete. These and related issues are discussed further in Section 1.5.

1.4.5 Nguyen's Model and Temporal Proof System for Networks of Processes

In [42], Nguyen et. al. define a relatively complete trace-based proof system based on temporal logic [37,47]. (Temporal logic is not defined here; a detailed description is given in Chapter 6.) The model of network computation is based on *observations* and *behaviors*. An observation on a set C of channels is defined as a quadruple (t, In, Out, Rd) where

- t is a single trace of all communication events on the channels in C ;
- In is the subset of C corresponding to the channels that are ready to be read from;
- Out is the subset of C corresponding to the channels that are ready to be written to;
- Rd is a function from the channels in C to natural numbers, indicating the number of messages read from each channel.

A behavior is defined as an infinite sequence of observations. Behaviors correspond to the interleaved traces of Misra/Chandy [39] and to the *past* and *present* variables of Hehner/Hoare [17].

In this formalism, a process or network is specified by an assertion of temporal logic in which the free variables are channel names. As in [39], a channel name in a specification refers to the projection of an interleaved trace onto that channel. Specifications refer to the behavior sequence of the specified process or network only through function variables In , Out , and Rd . (Note that references to trace t are not allowed.) Because temporal logic is designed for reasoning over sequences, the behavior sequence is implicitly present in the semantics of the language. A specification S is satisfied by a process or network N , denoted by $\langle N \rangle S$, if every possible computation of N , up to any point in time, is modeled by a behavior satisfying S .

Several *axioms for behaviors* are given. These are assertions satisfied by every behavior of every process or network. The axioms can be used as reasoning tools and to strengthen given specifications. We provide informal descriptions of the axioms here;⁷ in [42] they are formally defined as assertions of temporal logic.

1. The initial trace is empty.
2. At all points in time, the next trace extends the current trace by at most one element.
3. At all points in time, the event that extends a trace occurs after all existing events in that trace.
4. The ordering among the elements of a trace is preserved as the trace is extended.

As in [17,39], specifications for primitive processes are assumed to be given. Three inference rules are provided: a rule for channel renaming (used for verification of parameterized and hierarchical networks), a simple consequence rule:

⁷Axioms irrelevant to this discussion have been omitted.

$$\frac{\langle N \rangle R, R \Rightarrow S}{\langle N \rangle S}$$

and a network composition rule identical to that in [10]: the specification for a network is the conjunction of specifications for its component processes.

1.4.6 Milner's Calculus of Communicating Systems

Milner's *Calculus of Communicating Systems* (CCS) [38] is intended for use as a formal model for describing and reasoning about concurrent systems, rather than as a practical proof system. We include a brief description of CCS here, however, due to its importance as a breakthrough in the mathematical modeling of concurrency [22] and because it is commonly used as a reasoning tool in theoretical work on concurrent systems. A network in CCS is defined as a single process executing some number of subordinate processes in parallel (as in Hehner/Hoare [17]). Therefore, throughout this discussion, references to processes indicate simple sequential processes as well as concurrent networks.

In CCS, a syntax of expressions is inductively defined and used to denote processes; the notation is close to that of [8,17,20,22]. A set of equational laws for these expressions is then given; the laws form the basis of a formal system for proving equivalence of process behavior. An important contribution of CCS is that three distinct types of equivalence are formalized: *strong equivalence*, *observational equivalence*, and *observational congruence*. (We do not define these here.) The ability to reason over different types of behavioral equivalence permits CCS to formalize a variety of models of concurrent computation.

Milner also defines *synchronization trees*, which are used to illustrate the computational behavior of processes described by CCS expressions. Inductive operations on synchronization trees are defined, corresponding to the inductive definition of process expressions. (For example, a compositional operation to

combine process trees into a tree describing concurrent behavior corresponds to the syntax for concurrent composition of process expressions.) Synchronization trees can be used for proving properties of process behavior, but, in general, exhaustive case analysis is required for this.

1.5 Expressiveness and Completeness

We have described several trace-based formalisms and proof systems for networks of processes. The proof systems of Hoare [10,22] and Misra/Chandy [39] are simple and easily applicable. Unfortunately, they are also incomplete.

Any logic involving explicit reasoning over every possible interleaving of communication events is relatively complete, and this is the approach taken by Hehner/Hoare [17] and Milner [38]. Naturally, the difficulty with such systems is the exponential number of possible computations—verifying the specification of any but very simple networks can be a formidable task. In both [17] and [38], the authors suggest that their formalisms are more appropriate as theoretical models than as a practical proof systems.

An improvement is made in the relatively complete proof system of Nguyen et. al. [42]: temporal logic is used for implicit reasoning over communication event interleavings. Although the proof system, as presented, appears quite simple, verifying network specifications often requires the verification of temporal logic formulas. Proving the correctness of temporal logic formulas can be a complex task. Numerous axioms and inference rules are needed and proofs are generally long [34]. This formalism, therefore, is considerably more complicated than the similar but incomplete trace-based systems of [10,22,39].

The issue thus arises as to the exact complexity required of a relatively complete trace logic. Determining the components of a trace-based network proof

system that are necessary and sufficient for relative completeness, and exploring the expressiveness required of a logic to encode these components, comprises the remaining chapters of this thesis.

Chapter 2

A Model for Networks of Processes

To investigate properties of network proof systems it is first necessary to define exactly our notion of a network and to provide a suitable abstraction, or *model*, of network behavior.

2.1 Process Networks

We consider networks of processes that communicate and synchronize solely by message passing. Processes and communication channels are uniquely named. Each channel is either *internal* or *external* with respect to a network. An internal channel connects two processes of the network; an external channel is connected to only one. Channels are unidirectional, and communication along them is synchronous, so both processes incident to an internal channel must be prepared to communicate before a value is actually transmitted. External channels permit communication with the environment of the network; input or output on an external channel can occur whenever the process is ready. Without loss of generality, we assume:

Figure 2.1: A network of processes

- Message transmission occurs instantaneously.
- Two message transmissions cannot occur simultaneously. Thus, there is a total order on the communication events of a given computation.
- There is a fixed domain of values that can be transmitted on communication channels. Processes send and receive values in this domain only. The domain can be finite or infinite.

In Section 2.3 we show how our model can be modified to accommodate networks based on asynchronous rather than synchronous communication. In Section 2.4 we discuss hierarchically structured networks.

Processes are generally named as P , P_1 , P_2 , $P_3, \dots$; communication channels are named as c , d , $e, \dots$ or c_1 , c_2 , $c_3, \dots$. A network made up of processes $P_1, P_2, \dots, P_n$ is denoted by $P_1 \parallel P_2 \parallel \dots \parallel P_n$, which indicates the concurrent execution of the component processes. Figure 2.1 illustrates a network of three processes and six communication channels.

2.2 A Model of Computation

To reason about the behavior of process networks we introduce a formal model of computation. Our model is based on the *computation tree*. Every process or network is represented by one computation tree. The structure of the tree describes exactly the potential execution sequences of the process or network. Vertices represent the state of the computation—the sequences of values that have been transmitted on the communication channels so far—while edges represent a single step of execution.

If c is a communication channel, then the *channel trace* of c is the sequence $\langle c.0, c.1, \dots, c.k \rangle$ of values that have been transmitted on c , with $c.0$ the first, $c.1$ the second, etc. Each vertex of a computation tree, called a *trace-set*, is a set of communication channel traces. In all computation trees:

- The root of the tree is the trace-set in which all channel traces are empty, corresponding to the initial state of a computation.
- The children of a trace-set TS within the computation tree are exactly those trace-sets that extend one channel trace of TS by one element, where the extension corresponds to a communication event that can be performed by the represented process or network when in a state corresponding to trace-set TS .

Since we are interested in reasoning about network behavior, internal computations of a process are irrelevant except as they affect the values sent and received. Thus, our computation trees do not include such changes of process state. We allow trees to be of finite or infinite depth, corresponding to finite or infinite computations. The domain of communicable values corresponds to the breadth of a tree; it too can be finite or infinite. (There is some similarity here with the synchronization tree of CCS [38].)

Figure 2.2: Example network

We first describe computation trees for primitive processes and then show how a computation tree for a network is built from trees for its component processes.

2.2.1 Computation Trees for Processes

Let P be a process with incident incoming and outgoing channels $c_1, c_2, \dots, c_k$. The behavior of P is modeled by a computation tree in which all trace-sets contain traces of $c_1, c_2, \dots, c_k$ (and no other traces). As an example, consider the network of Figure 2.2. Process *MERGE* repeatedly and nondeterministically reads a value from either c or d and then writes it on e . Process *BUFFER* simply copies values from e to d , with an arbitrary amount of internal buffering. Let the data domain for the network be $\{a\}$. The initial portions of the infinite computation trees for *MERGE* and *BUFFER* are shown in Figures 2.3 and 2.4.

2.2.2 Computation Trees for Networks

The computation tree for a network is defined in terms of the computation trees for the network's component processes.¹ First, we define compatibility of trace-sets—the criteria for determining when a group of trace-sets from process computation trees can coexist and hence can be combined into a single trace-set of a network computation tree. Let $TS_1, TS_2, \dots, TS_n$ be trace-sets, one each from

¹We could alternatively—and equivalently—have chosen to define network trees independently of the component process trees, but the constructive definition given here is both illustrative of the model and useful in subsequent proofs.

Figure 2.3: Computation tree for process *MERGE*

Figure 2.4: Computation tree for process *BUFFER*

the computation trees for processes $P_1, P_2, \dots, P_n$ of a network. This group of trace-sets is *compatible* if and only if, for all channels c such that a trace of c appears in both TS_i and TS_j , the trace of c in TS_i is identical to the trace of c in TS_j . Thus, trace-sets are compatible when the exact same transmissions have occurred on any channels they have in common.² When an appropriate set of trace-sets is identified (the identification procedure is described shortly), they are merged into a single trace-set of the network tree being constructed. Merging compatible trace-sets consists simply of forming their union.

Let $T_1, T_2, \dots, T_n$ be the computation trees for processes $P_1, P_2, \dots, P_n$, respectively, and let $N = P_1 \parallel P_2 \parallel \dots \parallel P_n$ be the network composed of these processes. The tree T for network N is defined by the following construction:

Combine($T_1, T_2, \dots, T_n$) $\equiv$

the root of T is the result of merging the roots of $T_1, T_2, \dots, T_n$;

for each $T_i, 1 \leq i \leq n$:

let G_i be the group of trace-sets consisting of the root of T_i and all the root's children;

consider every possible group of trace-sets, G , where G is constructed by choosing one trace-set from each G_i . G is *usable* if:

1. the trace sets in G are compatible, and
2. merging the trace-sets in G results in a new trace-set that extends exactly one trace of T 's root by exactly one element;

for each usable G :

add a child to the root of T , letting this trace-set be the root of the tree defined by *Combine*(set of subtrees whose roots are the trace-sets in G).

In each invocation of *Combine*, one set of process-tree trace-sets is merged

²In [50], Soundararajan defines a similar notion of compatibility in the context of a proof system.

Figure 2.5: Computation tree for $MERGE||BUFFER$

into a single network-tree trace-set, followed by the identification of all possible trace-sets the network can achieve in some “next step”. The recursive definition then results in the complete network tree, even if some or all of the process trees are infinite. (The resulting network tree need not also be infinite.) Figure 2.5 shows the initial part of the network tree for $MERGE||BUFFER$, obtained by combining the process trees pictured in Figures 2.3 and 2.4.

2.2.3 Paths and Computations

We now define the correspondence between the computation-tree model and actual network computation. Define a *path* in a computation tree to be any se-

quence of trace-sets beginning with the root and descending through the tree until a trace-set with no children is reached. (If no terminal trace-set is reached then the path is an infinite sequence.) A path corresponds to a computation of the process or network being modeled by the computation tree. For any process or network PN , define $Comps(PN)$ to be the set of all paths in the computation tree for PN . $Comps(PN)$ represents all possible computations of process or network PN .

Since computation-tree paths are sequences of trace-sets, paths can be indexed in the same way as channel traces: if ρ is a path in a computation tree, then $\rho = \langle \rho.0, \rho.1, \rho.2, \dots \rangle$. Note that, unlike channel traces, these sequences can be of finite or infinite length.

The following definitions and theorems regarding the computation-tree model will be useful in obtaining subsequent results involving network proof systems.

Definition 2.2.1 Let TS be a trace-set in the computation tree for a network N and P a process of N . $ProjectTraceSet(TS, P)$ is the trace-set consisting of the traces in TS that are traces of channels incident to process P .

Theorem 2.2.2 For any network N , process P of N , and trace-set TS in the computation tree for N , $ProjectTraceSet(TS, P)$ is a trace-set in the computation tree for P .

Proof: Any trace-set in the computation tree for N can be obtained by merging trace-sets from the trees for N 's component processes (by the method of constructing network trees from component process trees). Let TS_P be the trace-set from P 's computation tree that can be used to form TS . By definition, $TS_P = ProjectTraceSet(TS, P)$, and TS_P is a trace-set in the computation tree for P . $\otimes$

Definition 2.2.3 Let ρ be a path in the computation tree for a network N and P a process of N . $ProjectPath(\rho, P)$ is the sequence of trace-sets obtained by

1. replacing each trace-set TS in ρ with $ProjectTraceSet(TS, P)$, then
2. eliminating each trace-set that duplicates its immediate predecessor in the resulting sequence.

Theorem 2.2.4 For any network N , process P of N , and path ρ in the computation tree for N , $ProjectPath(\rho, P)$ is a path in the computation tree for P .

Proof: It suffices to show that for each i , $0 \leq i < |\rho|$,³

$$ProjectPath(\langle \rho.0, \rho.1, \dots, \rho.i \rangle, P)$$

is a prefix of a path in the computation tree for P . The proof proceeds by induction on i .

Base Case: $i = 0$. $ProjectPath(\langle \rho.0 \rangle, P)$ contains a single trace-set—the trace-set in which all traces are empty. This is also the first trace-set of any path in the computation tree for P .

Induction: $i = n$, $0 < n < |\rho|$. We must show that

$$ProjectPath(\langle \rho.0, \rho.1, \dots, \rho.n \rangle, P)$$

is a prefix of a path in the computation tree for P . By the induction hypothesis,

$$ProjectPath(\langle \rho.0, \rho.1, \dots, \rho.(n-1) \rangle, P)$$

is a prefix of such a path. If

$$ProjectTraceSet(\rho.n, P) = ProjectTraceSet(\rho.(n-1), P)$$

then, by the elimination of trace-sets that duplicate their predecessors,

$$\begin{aligned} ProjectPath(\langle \rho.0, \rho.1, \dots, \rho.n \rangle, P) = \\ ProjectPath(\langle \rho.0, \rho.1, \dots, \rho.(n-1) \rangle, P), \end{aligned}$$

³ $|\rho|$ denotes the length of sequence ρ . If ρ is infinite then $|\rho| = \omega$.

and we are done. Suppose that

$$ProjectTraceSet(\rho.n, P) \neq ProjectTraceSet(\rho.(n-1), P).$$

Then

$$\begin{aligned} ProjectPath(\langle \rho.0, \rho.1, \dots, \rho.n \rangle, P) = \\ ProjectPath(\langle \rho.0, \rho.1, \dots, \rho.(n-1) \rangle, P) \cdot \langle ProjectTraceSet(\rho.n, P) \rangle, \end{aligned}$$

where $\cdot$ denotes sequence catenation. By the method of constructing network trees from component process trees, $ProjectTraceSet(\rho.n, P)$ is a child of $ProjectTraceSet(\rho.(n-1), P)$ in the computation tree for process P . By this fact and the induction hypothesis,

$$ProjectPath(\langle \rho.0, \rho.1, \dots, \rho.n \rangle, P)$$

is a prefix of a path in the computation tree for P . (*End Induction*)

Therefore, given any finite or infinite path ρ in the computation tree for N , $ProjectPath(\rho, P)$ is a path in the computation tree for P . $\otimes$

Informally, Theorem 2.2.4 tells us that $ProjectPath(\rho, P)$ extracts from a path representing a computation of a network the trace-set sequence that shows how a single process P behaves during this computation.

2.3 Modification for Asynchronous Networks

If an output channel is asynchronous, a process may send a value along it at any time. The values are assumed to be held in a queue of arbitrary length, so a process wishing to receive input either reads a value that has been sent but not received (the item at the head of the queue) or, if the queue is empty, waits until a value has been sent. We no longer assume that message transmission occurs instantaneously. The assumption that there is a total order on communication

events remains, but sending and receiving a given message is considered to be two distinct events. Only a few modifications to the computation-tree model are necessary in order to provide a representation of asynchronous network behavior.

With synchronous communication it is not necessary to distinguish between input channel traces and output channel traces, since the traces are always identical. In the asynchronous case, however, the sequence of values sent along a channel c is not necessarily the same as the sequence of values received on c . We know only that the input trace of c is a prefix of the output trace of c . Thus, we need a new notation that will allow us to represent separately the input and output traces of a given channel; we adopt the convention used by Milner in [38]. Let d be an input channel for some process. The channel trace of d , i.e. $\langle d.0, d.1, \dots, d.k \rangle$, denotes the sequence of values that have actually been read from channel d . To denote the values sent on an output channel e (say), we write $\langle \bar{e}.0, \bar{e}.1, \dots, \bar{e}.k \rangle$, and refer to this as the channel trace of $\bar{e}$.

The computation tree for an asynchronous network has the same basic structure as that for a synchronous network, bearing in mind that for all channels c , c and $\bar{c}$ are distinct traces. Therefore, every internal channel f (say) in an asynchronous network N is represented by two separate sequences in the trace-sets of the computation tree for N —one trace $\bar{f}$ of the values sent, and one trace f of those received. If g (say) is an external channel of N , then the trace of g is represented by only one of g or $\bar{g}$, depending on whether the channel performs input or output with respect to the network.

Only one modification to the method for network computation tree construction needs to be made to accommodate this change; the basic algorithm is identical. Note that any two trace-sets from different process computation trees will have no channel-trace names in common, since any shared channel must be an output channel of one process and an input channel of the other. So, in this case, the merging of trace-sets is guaranteed to be a disjoint set union. However, a

Figure 2.6: Computation tree for asynchronous $MERGE||BUFFER$

set of states is compatible if and only if, in the union of the states, every pair of traces c and $\bar{c}$ are such that c is a prefix of $\bar{c}$, i.e. some initial subset of the values sent have been received, and in the same order. As an example, suppose the communication in network $MERGE||BUFFER$ of Figure 2.5 is asynchronous. The initial part of the computation tree for this network is then as pictured in Figure 2.6.

Our model of network behavior can be generalized further to networks in which some channels support asynchronous communication while other channels are synchronous. Without conflict, we can use both c and $\bar{c}$ for the traces of an asynchronous channel c , while using only d for the trace of a synchronous channel d in the same network.

Definitions and Theorems 2.2.1 through 2.2.4 are valid for computation trees modeling asynchronous as well as synchronous communication, since the tree

properties they rely on remain unchanged. For our purposes, asynchronous communication serves only to complicate the notation; the fundamental model of computation and subsequent results are independent of the synchrony or asynchrony of message transmission. Therefore, in general, we consider only synchronous communication, although in Chapter 3 we show how our proof system can be extended to accommodate asynchronous networks.

2.4 Hierarchical Networks

Thus far, we have made a clear distinction between a process and a network. This distinction, however, is just one of convenience and simplification—a process can itself be implemented as a subnetwork, its component processes can be sub-subnetworks, etc., as in [25,39,42]. In fact, a process can be viewed simply as any abstract object that sends and receives values on a set of external input and output channels. Our computation-tree model adapts easily to such a hierarchical approach.

Let N be a network with internal channels $c^i_1, c^i_2, \dots, c^i_k$ and external channels $c^e_1, c^e_2, \dots, c^e_l$. Each trace-set in the computation tree for N contains traces of channels $c^i_1, c^i_2, \dots, c^i_k$ and $c^e_1, c^e_2, \dots, c^e_l$. Let the *external computation tree* of N be the computation tree for N with all traces of internal channels $c^i_1, c^i_2, \dots, c^i_k$ removed from all trace-sets in the tree. The external computation tree of a network models only the behavior of the network's external channels, and thus is equivalent to the computation tree for a process. The behavior of the internal channels is hidden, similarly to the hiding of internal process state changes as discussed in Section 2.2.

Suppose we would like to use a network PN as the implementation of a process in some larger network N . The computation tree for PN is constructed in the usual fashion from the trees for PN 's component processes. Now consider

constructing the computation tree for N . We are interested only in the external channels of network PN , which may act as either internal or external channels with respect to network N . Thus, we use the external computation tree of PN in the construction of the tree for N . This approach clearly generalizes to arbitrarily many levels of nested networks, as long as there exist some base processes—and hence base computation trees—with which to begin the construction.

As with asynchronous communication, allowing hierarchically-built networks has no impact on our fundamental results. Therefore, in general, we assume a simple two-level structure of networks and processes.

Chapter 3

Simple Trace Logic

Our formalism for specifying and verifying the behavior of process networks is called *Simple Trace Logic* (*STL*). It concisely captures the essence of most trace-based network proof systems. We first describe the syntax of STL and then use the computation-tree model of Chapter 2 to define a semantics. The axioms and inference rules are given, followed by a proof of their soundness. Completeness of trace-based systems is discussed, and we show that STL and other similar proof systems are inherently incomplete.

3.1 Specification Language

Suppose, without loss of generality, that all networks under consideration transmit values from a given finite or infinite data domain V . Consider formulas of first-order predicate logic [13,16,49,52] in which all variables are of one of the following three types:

1. elements of data domain V ,
2. finite sequences of elements of V ,
3. nonnegative integers.

Figure 3.1: Process P

Variables of type 2 correspond to channel traces, variables of type 1 to elements of channel traces, and variables of type 3 to lengths of channel traces. Let a *first-order trace formula* be a formula of predicate logic—with variables only of the types above—in which all free variables are of type 2, i.e. free variables denote only channel traces.

Suppose P is a process with incident channels $c_1, c_2, \dots, c_k$. A *specification* for P is a first-order trace formula in which the set of free variables is a subset of $\{c_1, c_2, \dots, c_k\}$. (In a trace formula, variable c denotes the trace of channel c .) We say that a specification S is *valid* for process P if, at every point during any computation of P , the traces of the values transmitted on channels $c_1, c_2, \dots, c_k$ satisfy S .¹ As an example, consider process P pictured in Figure 3.1. Suppose P repeatedly reads an integer from c and writes its successor to d . A valid specification for process P is then

$$(|c| - 1 \leq |d| \leq |c|) \wedge (\forall x: 0 \leq x < |d|: d.x = c.x + 1).$$

If N is a network with (internal and external) channels $c_1, c_2, \dots, c_k$, then N is similarly specified by a first-order trace formula in which the set of free variables is a subset of $\{c_1, c_2, \dots, c_k\}$. For a process or network PN and a specification S , $PN \text{ sat } S$ is the formula of STL used to denote the fact that S is a valid specification for PN . Rigorous definitions for the syntax and semantics of STL follow.

¹This notion of validity is formalized in Section 3.1.2.

3.1.1 Syntax

Let data domain $V = \{v_1, v_2, v_3, \dots\}$, so v_j is a constant denoting the corresponding element of V . Let $Fns = \{F_1, F_2, F_3, \dots\}$ denote a set of functions on elements of V . A given function F_j takes n (say) values from V as arguments, $n \geq 0$, and returns a single value from V as the result.² Let $Prds = \{G_1, G_2, G_3, \dots\}$ denote a set of predicates on elements of V . A given predicate G_j takes n (say) values from V as arguments, $n \geq 0$, and returns either *true* or *false* as the result. This kind of structure, consisting of a domain, constants, functions, and predicates, is a typical basis for building a first-order language [13,49,52].

In addition, let $Fns^{\mathcal{N}} = \{F_1^{\mathcal{N}}, F_2^{\mathcal{N}}, F_3^{\mathcal{N}}, \dots\}$ and $Prds^{\mathcal{N}} = \{G_1^{\mathcal{N}}, G_2^{\mathcal{N}}, G_3^{\mathcal{N}}, \dots\}$ denote functions and predicates on the integers (respectively), such as $+$, $-$, $=$, and $\neq$. The length of a sequence c is denoted by $|c|$; this special function takes a sequence as an argument and returns an integer as the result. The indexing operation $c.i$ —for a sequence c and nonnegative integer i —is also a special function; it takes a sequence and an integer as arguments and returns a value from V as the result. (When $i \geq |c|$, $c.i$ is undefined.) For simplicity, and without loss of generality, we do not consider these special functions as separate cases.

Table 3.1 gives an inductive definition of the full syntax of STL; Table 3.2 lists some syntactic abbreviations.

3.1.2 Semantics

A semantics for the language of STL is given by providing a formal definition for validity of first-order trace formulas with respect to process and network behavior. To do this, a model of computation is needed; we use the computation-tree model presented in Chapter 2.

² n is called the *arity* of the function; a 0-ary function is simply a constant.

Table 3.1: Syntax of STL

<i>STL-Formula</i>	$::=$	$PN \text{ sat } S$	PN a process or network; S a <i>Trace-Formula</i>
		f	f a <i>Trace-Formula</i>
<i>Trace-Formula</i>	$::=$	$G(t_1, \dots, t_n)$	$G \in Prds$ or $G \in Prds^{\mathcal{N}}$; $t_1, \dots, t_n$ <i>Trace-Terms</i> of the appropriate type
		$f1 \vee f2$	$f1$ and $f2$ <i>Trace-Formulas</i>
		$\neg f$	f a <i>Trace-Formula</i>
		$(\exists x: f)$	f a <i>Trace-Formula</i> ; x a variable ranging over V or over nonnegative integers (i.e. not over channel names) ³
<i>Trace-Term</i>	$::=$	k	k a constant: a v_j or an integer
		x	x a variable
		$F(t_1, \dots, t_n)$	$F \in Fns$ or $F \in Fns^{\mathcal{N}}$; $t_1, \dots, t_n$ <i>Trace-Terms</i> of the appropriate type

Table 3.2: Syntactic Abbreviations

$f1 \wedge f2$	for	$\neg(\neg f1 \vee \neg f2)$	$f1$ and $f2$ <i>Trace-Formulas</i>
$f1 \Rightarrow f2$	for	$\neg f1 \vee f2$	$f1$ and $f2$ <i>Trace-Formulas</i>
$(\forall x: f)$	for	$\neg(\exists x: \neg f)$	f a <i>Trace-Formula</i> ; x a variable ranging over V or over nonnegative integers
$c1 \preceq c2$	for	$(c1 \leq c2) \wedge$ $(\forall x: 0 \leq x < c1 : c1.x = c2.x)$	$c1$ and $c2$ trace variables
$c1 = c2$	for	$(c1 \preceq c2) \wedge (c2 \preceq c1)$	$c1$ and $c2$ trace variables

Definition 3.1.1 Let TS be a trace-set and t a term of first-order trace logic (as defined in Table 3.1) in which the only variables in t are channel-trace variables corresponding to a subset of the channel traces in TS . The *value of term t in trace-set TS* , denoted by $t|_{TS}$, is defined inductively as follows:

1. $k|_{TS} = k$, for a constant k ;
2. $c|_{TS} =$ the trace of c in TS , for a channel-trace variable c ;
3. $F(t_1, \dots, t_n)|_{TS} = F(t_1|_{TS}, \dots, t_n|_{TS})$, for a function $F \in Fns$ or $F \in Fns^V$, and trace-terms $t_1, \dots, t_n$ of the appropriate type.

Definition 3.1.2 Let TS be a trace-set and f a formula of first-order trace logic in which the only free variables of f are channel-trace variables corresponding to a subset of the channel traces in TS . $TS \models f$, read as *TS satisfies f* , is defined inductively as follows:

³In general, proof systems in which specifications may include quantification over channel names are not compositional [42].

1. $TS \models G(t_1, \dots, t_n)$ iff $G(t_1|_{TS}, \dots, t_n|_{TS})$, for a predicate $G \in Prds$ or $G \in Prds^N$, and trace-terms $t_1, \dots, t_n$ of the appropriate type;
2. $TS \models f1 \vee f2$ iff $TS \models f1$ or $TS \models f2$, for trace-formulas $f1$ and $f2$;
3. $TS \models \neg f$ iff not $TS \models f$, for a trace-formula f ;
4. $TS \models (\exists x: f)$ iff there exists a constant k of the appropriate type such that $TS \models f[k/x]$, for a trace-formula f , where $f[k/x]$ denotes formula f with all occurrences of variable x replaced by constant k .

Since we are interested in sequences of trace-sets, for notational convenience we define an “always” operator, $\Box$:

Definition 3.1.3 Let ρ be a sequence of trace-sets and f a formula of first-order trace logic in which the only free variables of f are channel-trace variables corresponding to a subset of the channel traces in ρ . $\rho \models \Box f$ iff for all i , $0 \leq i < |\rho|$, $\rho.i \models f$.⁴

Definition 3.1.4 Let PN be a process or network and S an STL specification for PN . (Hence S is a formula of first-order trace logic.) S is *valid* for PN if and only if, for all computation-tree paths $\rho \in Comps(PN)$, $\rho \models \Box S$.

Thus, a specification S is valid for a process or network PN if, given any behavior of PN up to any point, the traces of values transmitted along PN 's channels satisfy S . When specification S is valid for process or network PN , $PN \text{ sat } S$ is a valid formula of STL.

⁴This version of $\Box$ is consistent with the operator $\Box$ (“henceforth”) in temporal logic, see e.g. [37,47]. The temporal logic operator is defined as $\rho \models \Box f$ iff for all i , $0 \leq i < |\rho|$, $\langle \rho.i, \rho.(i+1), \dots \rangle \models f$, but when f itself contains no temporal operators, then $(\langle \rho.i, \rho.(i+1), \dots \rangle \models f) \equiv (\rho.i \models f)$. The use of temporal operators such as $\Box$ is discussed further in Chapters 4, 6, and 7.

Since STL is defined as an extension of first-order trace logic (see Table 3.1), the formulas of STL include formulas of trace logic as well as formulas $PN \text{ sat } S$. A trace formula f is a valid formula of STL if it is a tautology; that is, if $TS \models f$ for all trace-sets TS . We are not concerned here with proof methods for trace formulas, since first-order trace logic is a version of predicate logic. (Proof systems for predicate logic are defined in, e.g., [13,16,49,52].) We assume the existence of a formalism for verifying valid trace-logic formulas; we describe axioms and inference rules for proving the correctness of formulas $PN \text{ sat } S$.

3.2 Axioms and Inference Rules

The axioms of STL consist of formulas $P \text{ sat } S$ —one for each (primitive) process P of interest—such that S is a valid specification for P . A specification for a network is to be based solely on specifications for its primitive component processes; how these primitive specifications are obtained—or even how processes are programmed—is not our concern. This puts STL at a level of abstraction that hides all details except those relevant to the questions of correctness of network specifications (based on specifications for component processes) and completeness of the proof system.

Specifications for networks can be derived from specifications for their component processes by using the following inference rule:

Definition 3.2.1 (Network Composition Rule)

$$\frac{P_1 \text{ sat } S_1, P_2 \text{ sat } S_2, \dots, P_n \text{ sat } S_n}{P_1 \parallel P_2 \parallel \dots \parallel P_n \text{ sat } \bigwedge_i S_i}$$

Conjoining specifications of processes using the Network Composition Rule results in “linking” any shared channels because in $\bigwedge_i S_i$, all c ’s (say) refer to the same channel trace.

In addition, we have the following inference rule:

Definition 3.2.2 (Consequence Rule)

$$\frac{N \text{ sat } S1, S1 \Rightarrow S2}{N \text{ sat } S2}$$

In Section 3.4 we describe how these two rules, or variants thereof, underlie all trace-based network proof systems.

3.3 Modifications for Asynchronous and Hierarchical Networks

In Chapter 2 we discussed networks that may include asynchronous communication channels, and we discussed hierarchically structured networks; in both cases we showed how our computation trees can be generalized to model such networks. Although in the remainder of the thesis we consider only synchronous non-hierarchical networks, in this section we show how STL could be modified to allow formal reasoning about asynchronous communication. We also show that STL can be used for hierarchical network specification and verification.

3.3.1 Asynchronous Networks

Let c be an asynchronous communication channel of some network N . Recall (from Chapter 2) that $\bar{c}$ denotes the sequence of values sent on channel c , while c denotes the sequence of values read from c .

Suppose P is a process with (asynchronous) input channels $c_1, \dots, c_k$, and (asynchronous) output channels $c_l, \dots, c_m$ ($l > k$). A specification S for P is a first-order trace formula in which the set of free variables is a subset of $\{c_1, \dots, c_k, \bar{c}_l, \dots, \bar{c}_m\}$. Thus, in the specification for a process, there is a necessary syntactic distinction between input and output channels. A specification S is valid for

process P if, at every point during any computation of P , the traces of values read on $c_1, \dots, c_k$ and sent on $c_l, \dots, c_m$ (denoted by $\bar{c}_l, \dots, \bar{c}_m$) satisfy S .

Network Composition Rule 3.2.1 is then replaced by the following:

Definition 3.3.1 (Asynchronous Network Composition Rule)

$$\frac{P_1 \text{ sat } S_1, P_2 \text{ sat } S_2, \dots, P_n \text{ sat } S_n}{P_1 \parallel P_2 \parallel \dots \parallel P_n \text{ sat } \bigwedge_i S_i \wedge \text{Prefix-Satisfaction}}$$

where *Prefix-Satisfaction* $\equiv$

$(\forall c: c \text{ internal and asynchronous in } P_1 \parallel P_2 \parallel \dots \parallel P_n: c \preceq \bar{c})$.

The specification for a network is still based solely on specifications for its component processes, so the system remains compositional. (Schlichting and Schneider describe an axiomatic network proof system that similarly relies on the prefix property of asynchronous communication [48].)

The remainder of STL is unaffected by synchrony or asynchrony of communication. Hence, the formalism generalizes quite easily to include asynchronous communication. The few changes are primarily syntactic and are of no consequence to general properties of the proof system; we subsequently consider only the synchronous version of STL.

3.3.2 Hierarchical Networks

Let $N1$ be a network with internal channels $c^i_1, c^i_2, \dots, c^i_k$ and external channels $c^e_1, c^e_2, \dots, c^e_l$. Suppose we would like to use $N1$ as the implementation of a process in some larger network $N2$. The external channels of $N1$ become internal or external channels of $N2$; the internal channels of $N1$ are invisible with respect to $N2$. $N1$'s specification is obtained from the specifications for its component processes using the network composition rule and contains as free variables some subset of $\{c^i_1, c^i_2, \dots, c^i_k, c^e_1, c^e_2, \dots, c^e_l\}$. Consider, however, using the

specification for $N1$ to obtain a specification for $N2$. Only the behavior of $N1$'s external channels actually affects the behavior of $N2$.

Therefore, let a specification S for $N1$ be an *external specification* if the free variables of S are a subset of $\{c^e_1, c^e_2, \dots, c^e_l\}$, the external channels of $N1$. An external specification for $N1$ can then be used in the network composition rule to obtain a specification for the larger network $N2$. Furthermore, an external specification can be derived from any specification by using the consequence rule to eliminate all references to internal channels. Thus, to use a network N as a process in some larger network, we

1. obtain a specification S for N from the specifications for N 's component processes,
2. modify this specification into an external specification S_e by using the consequence rule, and finally
3. use network N (and specification S_e) as if it were a primitive process.

The method clearly generalizes to arbitrarily many levels of network nesting, provided there exist some base processes—and hence axioms $P \text{ sat } S$ —with which to begin.

This approach to hierarchical network construction is similar to “channel hiding” in [17,20,22,56], “abstraction” in [25,39,46], and “restriction” in [42].

3.4 Comparison with Other Trace Logics

STL captures the essential components of trace-based network proof systems. Specifications for processes and networks are first-order predicates over communication channel traces. Network specifications are obtained compositionally as conjunctions of specifications for the network's component processes. A consequence rule is defined for verifying specifications that logically follow from spec-

ifications already proven to be correct. We describe how these features underlie the trace-based proof systems presented in Chapter 1.

In Hoare's formalism and proof system for CSP [10,22], the specification language is identical to that of STL. Hoare provides rules to verify specifications for simple processes as well as for process networks. In STL, we assume that correct process specifications are given, so rules for verifying these specifications are not included. The network composition and consequence rules of [10,22] are identical to our Definitions 3.2.1 and 3.2.2. Therefore, STL is equivalent to the subset of Hoare's system excluding the rules for verifying sequential processes.

STL is also a subset of Misra and Chandy's proof system for networks of processes [39]. In [39], a specification for process or network PN is denoted by $R \mid PN \mid S$, where R and S are assertions over traces of PN 's channels. Specification $R \mid PN \mid S$ asserts:

1. S holds initially in PN , and
2. if R holds after a total of k messages have been transmitted on PN 's channels, then S holds after $k + 1$ messages have been transmitted, for all $k \geq 0$.

STL formula $PN \text{ sat } S$ is equivalent to Misra and Chandy's $true \mid PN \mid S$. Thus, restricting the network composition and consequence rules of [39] to specifications of the form $true \mid PN \mid S$, we obtain:

$$\frac{true \mid P_1 \mid S_1, true \mid P_2 \mid S_2, \dots, true \mid P_n \mid S_n}{true \mid (P_1 \parallel \dots \parallel P_n) \mid (\bigwedge_i S_i)} \qquad \frac{true \mid N \mid S1, S1 \Rightarrow S2}{true \mid N \mid S2}$$

These rules are equivalent to STL Definitions 3.2.1 and 3.2.2.

Hehner and Hoare's model of communicating processes [17] is also based on assertions of first-order logic, but in this case a single trace variable *past* is used, representing an interleaving of channel traces. (Variable *present* also

appears in assertions and is used for reasoning about liveness, an issue not addressed in STL.) Processes are directly defined as predicates, and the formula for composing process predicates $P_1, P_2, \dots, P_n$ into a network predicate includes conjunct $\bigwedge_i P_i$. (Additional conjuncts are necessary for channel renaming and linking, as described in Section 1.4.4.)

Hehner and Hoare require that processes and networks are given explicit definitions—strongest describing predicates. Therefore, STL formula $N \text{ sat } S$ is equivalent, in the notation of [17], to $(\forall \text{past}, \text{present}: N_e \Rightarrow S)$, where N_e is an explicit definition of network N . Predicate logic is used to prove the following consequence theorem:

If $S1 \Rightarrow S2$, then for any network N with explicit definition N_e ,
 $(\forall \text{past}, \text{present}: N_e \Rightarrow S1) \Rightarrow (\forall \text{past}, \text{present}: N_e \Rightarrow S2)$.

This theorem directly corresponds to the STL consequence rule.

The network proof system defined by Nguyen et. al. in [42] is based on temporal logic. Process and network specifications are temporal formulas over channel-trace variables. Network composition and consequence rules identical to those in STL are defined, although temporal specifications are assumed. Since predicate logic is a subset of temporal logic [37], this proof system subsumes STL.

We conclude that the specification language of STL is equivalent to or weaker than languages used by other trace logics, and that network composition and consequence rules—as defined in Section 3.2—are the common denominator of trace-based network proof systems.

3.5 Soundness of STL

A proof system such as STL is only useful if it is *sound*; that is, if the syntactic rules of STL are used to prove $N \text{ sat } S$, then indeed specification S is valid for network N .

Theorem 3.5.1 (Soundness of STL) Let N be a network and S a specification such that $N \text{ sat } S$ is provable using STL. Then S is valid for N .

Proof: The axioms of STL consist of formulas $P \text{ sat } S$, where S is a valid specification for process P . Thus, proving this theorem consists only of showing that whenever the hypothesis of an STL inference rule is valid, so is the conclusion. We now show this for our two inference rules.

$$1. \text{ Network Composition Rule } \frac{P_1 \text{ sat } S_1, P_2 \text{ sat } S_2, \dots, P_n \text{ sat } S_n}{P_1 \parallel P_2 \parallel \dots \parallel P_n \text{ sat } \bigwedge_i S_i}$$

Assume each S_i is valid for P_i , so for all $\rho \in \text{Comps}(P_i)$, $\rho \models \Box S_i$. We must show that for all $\rho \in \text{Comps}(N)$, $\rho \models \Box \bigwedge_i S_i$, where $N = P_1 \parallel P_2 \parallel \dots \parallel P_n$. Consider an arbitrary conjunct S_i and an arbitrary $\rho \in \text{Comps}(N)$. Let $\rho_i = \text{ProjectPath}(\rho, P_i)$. By Theorem 2.2.4 we know that ρ_i is a path in the computation tree for P_i . Therefore, by assumption, $\rho_i \models \Box S_i$. Then $\rho \models \Box S_i$ as well, since the traces that were removed from ρ by ProjectPath cannot appear in S_i . Since ρ is an arbitrary sequence in $\text{Comps}(N)$, we know that for all $\rho \in \text{Comps}(N)$, $\rho \models \Box S_i$. The conjunct S_i was also chosen arbitrarily, so we conclude that for all $\rho \in \text{Comps}(N)$, $\rho \models \Box \bigwedge_i S_i$. Thus $\bigwedge_i S_i$ is valid for N .

$$2. \text{ Consequence Rule } \frac{N \text{ sat } S1, S1 \Rightarrow S2}{N \text{ sat } S2}$$

Let $S1$ be valid for N , so for all $\rho \in \text{Comps}(N)$, $\rho \models \Box S1$. By $S1 \Rightarrow S2$ and predicate logic we conclude that for all $\rho \in \text{Comps}(N)$, $\rho \models \Box S2$. Therefore $S2$ is also valid for N . $\otimes$

3.6 The Completeness Question

We would also like STL to be *complete*; that is, whenever some specification S is valid for a network N , then $N \text{ sat } S$ is provable using STL. In the next two

sections we discuss necessary modifications to the goal of completeness.

3.6.1 Precise Specifications

A specification for a network is derived using the network composition rule from specifications for its component processes. If these process specifications are valid, but too weak, then we may not be able to prove a given valid network specification. (For example, the specification *true* is valid for every process, but reveals very little about how a process actually behaves.) Thus, what we are really interested in is whether we can prove $N \text{ sat } S$ when the specifications given for the primitive processes comprising N are as “strong” as possible.

Definition 3.6.1 (Informal Definition of Preciseness) A specification S is *precise* for a process or network PN if and only if

1. S is valid for PN , and
2. any computation that satisfies S is a possible computation of PN .

A precise specification for a process or network, then, exactly characterizes its possible computations. Similar notions of precise specifications are used in [25,42], as “explicit definitions” in [17], and as “strongest invariants” in [56].

We formalize preciseness using the computation-tree model.

Definition 3.6.2 A sequence of trace-sets is *well-formed* if and only if

1. all channel traces in its initial trace-set are empty, and
2. each trace-set in the sequence, except the first, extends exactly one trace of the preceding set by exactly one element.

Thus, a sequence of trace-sets is well-formed if it could potentially appear as a path in the computation tree for some process or network.

Definition 3.6.3 (Formal Definition of Preciseness) A specification S is *precise* for a process or network PN if and only if

1. S is valid for PN , and
2. any well-formed sequence of trace-sets satisfying $\Box S$ is in $\text{Comps}(PN)$.

(In part 2 of Definition 3.6.3 we tacitly assume that the trace-sets in the sequence do not include extraneous channel traces—i.e. that all traces are histories of channels actually appearing in PN .)

It turns out that the composition of precise specifications results in a network specification that is also precise.

Theorem 3.6.4 (Preciseness-Preservation) Let S_i be a precise specification for P_i , $1 \leq i \leq n$, and let $N = P_1 \parallel P_2 \parallel \dots \parallel P_n$. Then $\bigwedge_i S_i$ is a precise specification for N .

Proof: We must show that $\bigwedge_i S_i$ satisfies both parts of Definition 3.6.3.

1. $\bigwedge_i S_i$ is valid for N :

Since the S_i are precise specifications for their respective P_i , they are valid. We must then show that $\bigwedge_i S_i$ is valid for N . This is exactly what was proven in part 1 of Theorem 3.5.1 (the soundness theorem).

2. If ρ is any well-formed sequence of trace-sets such that $\rho \models \Box \bigwedge_i S_i$, then $\rho \in \text{Comps}(N)$:

Let ρ be any well-formed sequence of trace-sets such that $\rho \models \Box \bigwedge_i S_i$. We must show that $\rho \in \text{Comps}(N)$. Let $\rho_i = \text{ProjectPath}(\rho, P_i)$, $1 \leq i \leq n$. By definition, $\rho_i \models \Box S_i$. Thus, by the preciseness of each of the S_i , $\rho_i \in \text{Comps}(P_i)$. Since ρ_i is a path in the computation tree for P_i , $1 \leq i \leq n$, by the algorithm for network-tree construction (Section 2.2.2), $\rho_1, \rho_2, \dots, \rho_n$ can be merged to form a path in the computation tree for N . This path is ρ , hence $\rho \in \text{Comps}(N)$. $\otimes$

For completeness, then, we are interested merely in the provability of $N \text{ sat } S$ when S is valid and the specifications for the processes in N are precise.

3.6.2 Relative Completeness

STL specifications can involve elements of the data domain from which messages are drawn, sequences of such elements, and lengths of sequences. Since number theory itself is incomplete [49], a valid assertion involving sequence lengths might not be provable in any system. Furthermore, STL is based on the first-order predicate calculus [13,16,49,52] which, too, is incomplete [13,49]. When designing a programming logic, one actually aims for *relative completeness* [11]: Assuming that one can prove any valid statement of predicate logic, number theory, and the data domain of the network being considered—i.e. adopting all such valid statements as axioms in one’s logic—is the proof system complete?⁵ STL is not relatively complete, as we now show.

3.7 Incompleteness of STL

It has been suggested [7,26,41] that incompleteness of simple trace-based network proof systems is due to a weak network composition rule—that a conjunction of process specifications does not yield a specification that sufficiently describes network behavior. The preciseness-preservation theorem (Theorem 3.6.4), however,

⁵Most proof systems make assumptions about both the provability of predicate logic statements and the expressiveness of the specification language involved. This is sometimes referred to as *Cook completeness* [1,11]. We, too, have made an expressiveness assumption in our supposition that precise specifications for the component processes can be written in STL. Our language is certainly powerful enough to express precise specifications for a large class of primitive processes. (Related issues of expressiveness are discussed in Chapters 4–7.)

Figure 3.2: Modified Brock-Ackerman network

tells us that information cannot be lost through network composition. A specification obtained by forming the conjunction of process specifications that are as strong as possible results in a network specification that also is as strong as possible.

In this section, we provide several examples that illustrate the incompleteness of STL. The first example is based on a network introduced by Brock and Ackerman in [7] and discussed by Nguyen et. al. in [42]; it is included for its historical importance as one of the earliest examples of trace-logic incompleteness. The second example is a mathematically more elegant variant on the first, exhibiting a similar type of incompleteness. Both examples consist of multi-process networks. To illustrate that network composition is not an ingredient of incompleteness—and to set the stage for isolating the actual causes—we introduce two simple one-process networks, both having valid specifications that are not provable in STL.

3.7.1 Brock-Ackerman Example

The network pictured in Figure 3.2 is a simplification of an example in [7]. Process $P1$ reads a value from channel c or d (the choice is nondeterministic), sends it on channel e , reads another value from c or d , sends it on e , and stops. Process $P2$

reads two values from e , then sends the first on f and the second on g . Process $PLUS$ simply reads one value v (say) from f and sends $v + 1$ on d . Let the data domain for this network be the natural numbers. Clearly if two 0's are available on channel c , then only a 0 can be output on channel g (not a 1). The value written on g must be the second value that was read from c , since no input is available on d until $P2$ has already read its two values. Thus we would like to prove

$$P1 \parallel P2 \parallel PLUS \text{ sat } g \preceq \langle c.1 \rangle.^6 \quad (3.1)$$

Precise specifications for the three component processes are:

$$\begin{aligned} P1 \text{ sat } & (e \preceq \langle c.0, c.1 \rangle \vee e \preceq \langle c.0, d.0 \rangle \vee e \preceq \langle d.0, c.0 \rangle \vee e \preceq \langle d.0, d.1 \rangle) \wedge \\ & (|c| + |d| - 1 \leq |e| \leq |c| + |d| \leq 2), \\ P2 \text{ sat } & f \preceq \langle e.0 \rangle \wedge g \preceq \langle e.1 \rangle \wedge |e| \leq 2 \wedge (|e| < 2 \Rightarrow |f| = 0 \wedge |g| = 0), \\ PLUS \text{ sat } & d \preceq \langle f.0 + 1 \rangle \wedge |f| \leq 1. \end{aligned}$$

The specification obtained by applying the network composition and consequence rules allows one to conclude only that

$$P1 \parallel P2 \parallel PLUS \text{ sat } g \preceq \langle c.1 \rangle \vee g \preceq \langle c.0 + 1 \rangle, \quad (3.2)$$

indicating, e.g., that either a 0 or 1 may be written on channel g if two 0's are read on c . Valid specification (3.1) for this network is not provable using the axioms and inference rules of STL. The problem is that we are unable to prove that the second value read by $P2$ does not come from d (via $P1$), even though

⁶This specification should actually be written as

$$(|c| < 2 \Rightarrow g = \langle \rangle) \wedge (|c| \geq 2 \Rightarrow g \preceq \langle c.1 \rangle),$$

since $c.1$ is undefined if $|c| < 2$. In this and subsequent specifications, however, we allow the simpler version by defining $c.x$ to be empty whenever $|c| < x + 1$, for any channel c and index x .

Figure 3.3: Every-other network

no values can actually be transmitted along d until $P2$ has finished reading. A similar example with a somewhat more mathematical flavor follows.

3.7.2 Every-Other Example

Consider the network of Figure 3.3. Process EO iterates forever, each time non-deterministically reading a value from c or d and then writing the value on e only if it is the 2nd, 4th, 6th, etc. value read from that input channel. (That is, every other value read from each of c and d is written on e .) Process I copies values from e to d , with an arbitrary amount of internal buffering. (If message passing were asynchronous this process would be unnecessary.) Since we will be reasoning only about numbers of values sent—and not the values themselves—we let the data domain simply be $\{a\}$. One can verify operationally that an invariant property of the network is $|c| > 0 \Rightarrow |c| > |e|$, and this is what we would like to prove.⁷ Precise specifications for the processes are:

$$EO \text{ sat } \lfloor |c|/2 \rfloor + \lfloor |d|/2 \rfloor - 1 \leq |e| \leq \lfloor |c|/2 \rfloor + \lfloor |d|/2 \rfloor,$$

$$I \text{ sat } |d| \leq |e|.$$

The conjunction of these two specifications is not enough to imply $|c| > 0 \Rightarrow$

⁷This example is due to Abha Moitra and Prakash Panangaden.

Figure 3.4: Single-process network

$|c| > |e|$. Suppose that $|c| = 2$. Then when we simultaneously solve the inequalities in the process specifications we obtain several solutions, one of which is $|c| = |d| = |e| = 2$, which clearly does not satisfy $|c| > 0 \Rightarrow |c| > |e|$.

In this and the previous example, we cannot verify the accurate specification for a network because we are unable to rule out the possibility of a process reading a value from a channel when that value is available only as a result of its already being read. Although it seems as if such incompleteness is manifested only in networks with loops and nondeterministic processes, this is actually a special case of a more general problem: network behavior that relies on multiple communication events occurring at the exact same instant—which will never happen—cannot be ruled out using STL. Our next example, consisting of a single process, is a simple illustration of this.

3.7.3 Single Process Example #1

Consider the single-process network of Figure 3.4. As an informal description of process P we are given four facts:

1. P reads at most one value from channel c .
2. P reads at most one value from channel d .
3. P reads a value from c before reading from d .
4. P reads a value from d before reading from c .

A formal specification is

$$P \text{ sat } S1: |c| \leq 1 \wedge |d| \leq 1 \wedge |d| \leq |c| \wedge |c| \leq |d|. \quad (3.3)$$

Let the data domain for this network be $\{a\}$. The following specification is valid for P and is equivalent to (3.3):

$$P \text{ sat } S2: (c = \langle \rangle \wedge d = \langle \rangle) \vee (c = \langle a \rangle \wedge d = \langle a \rangle) \quad (3.4)$$

P is always in one of two states: either no values have been read from c and d or one a has been read from each. However, P can reach a state in which $(c = \langle a \rangle \wedge d = \langle a \rangle)$ only if $c.0$ and $d.0$ are transmitted simultaneously. Since this cannot happen (by the assumptions of Section 2.1), P can never read a value from c or d . Therefore, a third valid specification for P is

$$P \text{ sat } S3: (c = \langle \rangle \wedge d = \langle \rangle). \quad (3.5)$$

All three specifications are valid and, in fact, precise. Any computation satisfying $S1$, $S2$, or $S3$ is a computation of P —no values are ever read on c or d . However (3.5) cannot be proved from precise specification $S2$ (say) of (3.4). Since there is only a single process, the network composition rule is irrelevant, and the only inference we can use is the consequence rule. But $S2 \Rightarrow S3$ does not hold. Hence (3.5) is unprovable, even though it is valid.

3.7.4 Single Process Example #2

Our last example illustrates a second property of network computation that cannot be reasoned about using STL: the trace of a channel c up to a given point in time is always a prefix of the trace of c up to any later point in time.

Consider a network with one process and one communication channel (see Figure 3.5). Suppose the network has $\{a, b\}$ as its data domain. Let a precise specification for process P be

$$P \text{ sat } S4: c = \langle \rangle \vee c = \langle a \rangle \vee c = \langle b, a \rangle. \quad (3.6)$$

Since P can send only one value at a time on channel c , $c = \langle b, a \rangle$ can never be attained—it would be reachable only from $c = \langle b \rangle$, which is prohibited by $S4$.

Figure 3.5: Simplest network of all

Therefore, (3.6) can be simplified to

$$P \text{ sat } S5: c \preceq \langle a \rangle. \quad (3.7)$$

However, $S4$ does not imply $S5$, and therefore (3.7) cannot be proved from precise specification (3.6).

3.8 Completeness and Incompleteness of Other Trace Logics

In Chapter 1, several existing trace-based proof systems, both complete and incomplete, are described. We show here that the simpler of these proof systems, like STL, are incomplete. As in STL, the logics do not permit reasoning about properties of network computation needed to verify valid specifications for the examples of Section 3.7. We also consider relatively complete trace-based proof systems, using them to describe proofs for our example networks.

The network composition and consequence rules in Hoare's proof system for CSP [10,22] are identical to the inference rules of STL. Therefore, if we use the given precise specifications for our example networks, the same incompleteness problems arise. In Hoare's system, however, rules are provided for specifying and verifying simple processes. Using these rules, any valid specification for a single-process network can be proven directly from the corresponding CSP program. Thus, example networks consisting of a single process cannot be used to show the incompleteness of Hoare's system.

Incompleteness is exhibited, however, in the Brock-Ackerman and Every-Other networks of Sections 3.7.1 and 3.7.2. In these examples, the given process specifications are as strong as any specifications obtainable from actual CSP programs. Since only the network composition and consequence rules can be used when composing process specifications into network specifications, and since these rules are identical to STL's, the logic is not strong enough to verify valid specifications for the Brock-Ackerman and Every-Other networks. Hence the proof system is incomplete.

Misra and Chandy's proof system for networks of processes [39] appears to be stronger than STL, since a specification $R \mid PN \mid S$ consists of both an assumption R and an invariant S . Unfortunately, when reasoning about the example networks of Section 3.7, such specifications are no more useful than STL specifications. As an example, consider the first single-process network (Section 3.7.3). The given precise specification for process P is

$$P \text{ sat } (c = \langle \rangle \wedge d = \langle \rangle) \vee (c = \langle a \rangle \wedge d = \langle a \rangle), \quad (3.8)$$

and we want to prove

$$P \text{ sat } (c = \langle \rangle \wedge d = \langle \rangle). \quad (3.9)$$

Using the notation of [39], (3.8) is equivalent to

$$\text{true} \mid P \mid (c = \langle \rangle \wedge d = \langle \rangle) \vee (c = \langle a \rangle \wedge d = \langle a \rangle), \quad (3.10)$$

and this specification is precise: by the model of network computation in [39], any well-formed computation satisfying (3.10) is a possible computation of P (i.e. no values are transmitted on either channel). The valid specification corresponding to (3.9) is

$$\text{true} \mid P \mid (c = \langle \rangle \wedge d = \langle \rangle). \quad (3.11)$$

By the inference rules of the proof system (recall Section 1.4.3), specification (3.11) is provable if and only if

$$((c = \langle \rangle \wedge d = \langle \rangle) \vee (c = \langle a \rangle \wedge d = \langle a \rangle)) \Rightarrow (c = \langle \rangle \wedge d = \langle \rangle). \quad (3.12)$$

As in STL, inference (3.12) does not hold, so valid specification (3.11) is not provable.

In the relatively complete proof system of Hehner and Hoare [17], processes are described by predicates. In particular, a primitive process P must be given an *explicit definition*: a predicate as strong as the conjunction of all possible definitions of P . The given precise specification for our first single-process example, specification (3.4) in Section 3.7.3, is not an explicit definition of process P . A stronger definition for P can be obtained by forming the conjunction of (3.4) with valid specification (3.5). Similarly, (3.6) is not an explicit definition for the second single-process example, since a stronger definition for P can be obtained by forming the conjunction of (3.6) with (3.7). Therefore, under the formalism of [17], our precise specifications for the single-process examples are not explicit definitions, and an incompleteness problem does not arise.

Now consider the two other examples, Brock-Ackerman in Section 3.7.1 and Every-Other in Section 3.7.2. We discuss only Every-Other; a similar proof method can be used for Brock-Ackerman. In the notation of [17], explicit definitions for processes EO and I are:

$$EO \equiv \lfloor |past(c)|/2 \rfloor + \lfloor |past(d)|/2 \rfloor - 1 \leq |past(e)| \leq \lfloor |past(c)|/2 \rfloor + \lfloor |past(d)|/2 \rfloor$$

$$I \equiv |past(d)| \leq |past(e)|.^8$$

(Recall that $past(x)$ denotes the restriction of interleaved trace $past$ to communication events on channel x .) The goal is to show that $EO \wedge I$, the concurrent composition of EO and I , implies valid specification $|past(c)| > 0 \Rightarrow$

⁸In [17], shared channel names are not actually allowed in separate process specifications since linking occurs through channel renaming in the concurrent composition rule. In this discussion, however, we do not lose generality by permitting shared channels and eliminating the renaming process.

$$|past(c)| > |past(e)|.$$

Recall the following axiom, given as part of the proof system.

For every process (or network) P :

$(\forall past', msg:$

$$P[past' \cdot msg / past] \Rightarrow$$

$$(\exists present: msg \in present \wedge P[past' / past]))$$

This axiom—call it A —is equivalent to the second of the two axioms given in Section 1.4.4. Axiom A allows $|past(c)| > 0 \Rightarrow |past(c)| > |past(e)|$ to be deduced from $EO \wedge I$.

We prove $|past(c)| > 0 \Rightarrow |past(c)| > |past(e)|$ by induction on the length of any sequence $past$ satisfying $EO \wedge I$.

Base Case: $|past| = 0$. $|past(c)| > 0 \Rightarrow |past(c)| > |past(e)|$ holds vacuously since $|past(c)| > 0 \equiv false$.

Induction: $|past| = n$, $n \geq 1$. As the induction hypothesis, suppose

$$|past'(c)| > 0 \Rightarrow |past'(c)| > |past'(e)| \tag{3.13}$$

for all $past'$ of length $n - 1$. We must show $|past(c)| > 0 \Rightarrow |past(c)| > |past(e)|$. For the sake of a contradiction, assume

$$|past(c)| > 0 \wedge |past(c)| \leq |past(e)|. \tag{3.14}$$

Since $n \geq 1$, $past = past' \cdot msg$, for some $past'$ and msg . Therefore, by (3.13) and (3.14), $|past'(e)| = |past'(c)| - 1$ and $|past(e)| = |past(c)|$. (That is, the msg extending $past'$ to $past$ must be transmitted on channel e .) From $EO \wedge I$, $|past(e)| = |past(c)|$, and arithmetic, we obtain $|past(d)| = |past(e)|$. Consequently, since $|past'(d)| = |past(d)|$ and $|past'(e)| < |past(e)|$, we know $|past'(d)| > |past'(e)|$. By axiom A (with $EO \wedge I$ for P), however, $(EO \wedge I)[past' / past]$ must be valid. Since $I[past' / past] \equiv |past'(d)| \leq |past'(e)|$,

we incur a contradiction with $|past'(d)| > |past'(e)|$. Therefore, assumption $|past(c)| > 0 \wedge |past(c)| \leq |past(e)|$ cannot hold, and $|past(c)| > 0 \Rightarrow |past(c)| > |past(e)|$. $\otimes$

From this example, we see that explicit use of variable *past* permits formal reasoning over the properties we were unable to reason about in STL—properties necessary for verifying the valid specifications of our example networks.

Finally, we consider the temporal proof system of Nguyen et. al. [42]. The axioms and inference rules of the proof system and the notion of precise specifications are identical to STL. The difference is that the specification language used in [42] is temporal logic. Because temporal logic is more expressive than the first-order trace logic of STL, *axioms for behaviors* can be defined to describe properties of network computation that cannot be described using STL. (Some of these axioms are listed in Section 1.4.5.) In particular, one axiom states that at most one communication event can occur at a time; this is the property needed to verify valid specifications for the examples of Sections 3.7.1, 3.7.2, and 3.7.3. The property needed for the example of Section 3.7.4—that the trace of a channel at a given point in time is always a prefix of that channel’s trace at any later point in time—is encoded by two other axioms. The full set of axioms allows any valid temporal logic specification for any network to be deduced from a precise temporal logic specification for that network. The proof system is thus relatively complete.

In the next chapter, we use our single-process examples to isolate two properties of network computation, showing that axiomatizations of these two properties are necessary and sufficient for achieving relative completeness in a trace-based network proof system.

Chapter 4

Completeness in Trace-Based Proof Systems

In Chapter 3, several examples were given to illustrate the fact that simpler trace-based proof systems (such as STL) are incomplete. In this chapter, the causes of this incompleteness are analyzed and formally defined. From these definitions we are able to determine exactly what properties must be axiomatized within a trace-based proof system for relative completeness.

4.1 The Need for Computation Characterization

The example networks of Section 3.7 all have valid specifications that are not provable in STL. In each case, we obtain a precise specification $S1$ (say) for the network, but there exists a valid specification $S2$ (say) such that $S1 \not\equiv S2$. Although specification $S2$ does not follow logically from specification $S1$, we are able to assert—using informal operational reasoning involving known properties of network computation—that any network satisfying specification $S1$ also satisfies $S2$. To obtain a relatively complete proof system we must incorporate into

the logic the ability to state and reason about such properties of computation.

Consider any STL proof that establishes $N \text{ sat } S$ for a network $N = P_1 \parallel P_2 \parallel \dots \parallel P_n$. As axioms, we are given $P_1 \text{ sat } S_1, P_2 \text{ sat } S_2, \dots, P_n \text{ sat } S_n$, where $S_1, S_2, \dots, S_n$ are precise. The first rule to be applied in any such proof is the network composition rule, so we immediately obtain $N \text{ sat } \bigwedge_i S_i$. All remaining steps in the proof must then be applications of the consequence rule. Since any string of consequence rules can be collapsed into one (by the transitivity of $\Rightarrow$), we see that $N \text{ sat } S$ is provable if and only if $\bigwedge_i S_i \Rightarrow S$. By Theorem 3.6.4, we know that $\bigwedge_i S_i$ is a precise specification for N . Therefore, STL would be relatively complete if $S1 \Rightarrow S2$ whenever $S1$ is a precise specification for a network N and $S2$ is a valid specification for N . The examples of Chapter 3, however, demonstrate that such an implication might not hold. By strengthening the antecedent, we can guarantee that the implication will be valid. We must find a set of axioms such that if A (say) is the conjunction of the axioms in the set, then $(S1 \wedge A) \Rightarrow S2$ is valid whenever it should be possible to deduce $S2$ from $S1$.

In the next two sections, the single-process incompleteness examples are used to isolate those properties of network computation that must be captured by the axioms in A . Unfortunately, these properties cannot be expressed in first-order trace logic. We encode the properties in an extended trace-based language, thus the axioms in A are axioms not of STL, but of some hypothetical stronger trace logic. In Section 4.6 and subsequent chapters, we discuss the expressive power required of a trace logic to encode the properties axiomatized in A and hence achieve relative completeness.

4.2 The Temporal Ordering Property

Recall the example network of Section 3.7.3, consisting of a single process P and two communication channels c and d . As a precise specification for process P we are given

$$P \text{ sat } S1: (c = \langle \rangle \wedge d = \langle \rangle) \vee (c = \langle a \rangle \wedge d = \langle a \rangle). \quad (4.1)$$

The following specification is also valid:

$$P \text{ sat } S2: (c = \langle \rangle \wedge d = \langle \rangle) \quad (4.2)$$

A state satisfying disjunct $(c = \langle a \rangle \wedge d = \langle a \rangle)$ of $S1$ can never be reached during execution since it requires communication events $c.0$ and $d.0$ to occur simultaneously.

We need to formalize the reasoning about event ordering used to obtain (4.2) from (4.1). It must assert the following:

Definition 4.2.1 (Temporal Ordering Property) Suppose that $c1$ and $c2$ are channels of a network N , that $c1.x$ and $c2.y$ are transmitted as a result of distinct communication events, and that in any computation of N

1. $c1.x$ must be transmitted before $c2.y$, and
2. $c2.y$ must be transmitted before $c1.x$.

Then $(|c1| \leq x \wedge |c2| \leq y)$ holds throughout any computation of N —neither message will be transmitted.

Property 4.2.1 allows $S2$ to be deduced from $S1$, making (4.2) provable.

Incorporating the temporal ordering property into a proof system requires a formal axiomatization. The language of STL is not strong enough, however, to express Property 4.2.1. To formally define the temporal ordering property, we call upon the “always” operator, $\Box$, introduced in Section 3.1.2.

If a communication event $c1.x$ happens before $c2.y$, then $|c2|$ cannot exceed y until $|c1|$ exceeds x . This can be expressed as $\Box(|c2| > y \Rightarrow |c1| > x)$. Note that this assertion captures temporal precedence for any channels $c1$ and $c2$ and any indices x and y , even if $x = y$ or $c1$ and $c2$ are the same channel. We are interested only in temporal ordering of distinct events, so we exclude the case in which $c1.x$ and $c2.y$ are produced by the same event (i.e. $x = y$ and $c1$ and $c2$ are the same channel). Now, if $\Box(|c1| > x \Rightarrow |c2| > y)$ as well—i.e. $c2.y$ must happen before $c1.x$ —then neither $c1.x$ nor $c2.y$ can happen, equivalently: $\Box(|c1| \leq x \wedge |c2| \leq y)$.

Our formalization differs slightly from the preceding discussion. All $>$'s are changed to $\geq$'s in the antecedent of the rule and all $\leq$'s are changed to $<$'s in the consequent. Doing so allows channel traces of length 0 in the antecedent, thereby asserting that an empty channel trace temporally precedes all communication events on that channel. Hence we state the temporal ordering axioms as:

Definition 4.2.2 (ORDERING) If $c1$ and $c2$ are channels, $x \geq 1$ and $y \geq 0$ are indices, and either $x \neq y$ or $c1$ and $c2$ are distinct, then

$$(\Box(|c1| \geq x \equiv |c2| \geq y)) \Rightarrow \Box(|c1| < x \wedge |c2| < y).$$

We require $x \geq 1$, rather than $x \geq 0$, because allowing $x = y = 0$ results in a pathological situation in which the antecedent is trivially true (since trace lengths are always at least 0), but the consequent is trivially false (since lengths cannot be less than 0).

In Chapter 3, the semantics of operator $\Box$ are formally defined with respect to sequences of trace-sets (Definition 3.1.3). Therefore, we can use the computation-tree model of Chapter 2 to prove the soundness of *ORDERING*; we show that the axiom is satisfied by every process or network computation.

Theorem 4.2.3 (Soundness of ORDERING) $\rho \models \text{ORDERING}$ for any well-formed sequence of trace-sets ρ .

Proof: Let ρ be an arbitrary well-formed sequence of trace-sets. We must show

$$\rho \models (\Box(|c1| \geq x \equiv |c2| \geq y)) \Rightarrow \Box(|c1| < x \wedge |c2| < y),$$

equivalently: if $\rho \models \Box(|c1| \geq x \equiv |c2| \geq y)$ then $\rho \models \Box(|c1| < x \wedge |c2| < y)$. Assume that $\Box(|c1| \geq x \equiv |c2| \geq y)$ holds for ρ , and suppose, for the sake of a contradiction, that $\Box(|c1| < x \wedge |c2| < y)$ does not. Thus, there is a trace-set of ρ in which $(|c1| \geq x \vee |c2| \geq y)$. Let i be the smallest index for which this is true: $(|c1| \geq x \vee |c2| \geq y)$ is true in $\rho.i$, but does not hold in any $\rho.j$ for $j < i$. Since $(|c1| \geq x \vee |c2| \geq y)$ is true in $\rho.i$, by $\rho \models \Box(|c1| \geq x \equiv |c2| \geq y)$ we know that $(|c1| \geq x \wedge |c2| \geq y)$ holds in $\rho.i$. By $x \geq 1$ (recall Definition 4.2.2), $i > 0$, since all traces in $\rho.0$ are empty. So consider trace-set $\rho.(i-1)$. By the definition of a well-formed sequence (Definition 3.6.2), $\rho.i$ extends exactly one trace of $\rho.(i-1)$ by exactly one element. Therefore, since $(|c1| \geq x \wedge |c2| \geq y)$ holds in $\rho.i$, $(|c1| \geq x \vee |c2| \geq y)$ must hold in $\rho.(i-1)$. This contradicts the assumption that i is the smallest index for which $\rho.i \models (|c1| \geq x \vee |c2| \geq y)$. Thus $\rho \models \Box(|c1| < x \wedge |c2| < y)$, and $\rho \models \text{ORDERING}$. $\otimes$

4.3 The Prefix Property

The example of Section 3.7.4 consists of one process, P , and one communication channel, c . A precise specification for process P is

$$P \text{ sat } S3: c = \langle \rangle \vee c = \langle a \rangle \vee c = \langle b, a \rangle. \quad (4.3)$$

Consequently, a valid specification for P is

$$P \text{ sat } S4: c \preceq \langle a \rangle. \quad (4.4)$$

A state satisfying disjunct $c = \langle b, a \rangle$ can never be reached during P 's execution, since when only one value has been transmitted on c , that value must be a . Here, we need:

Definition 4.3.1 (Prefix Property) For any channel c , the trace of c at any point in time is a prefix of the trace of c at any later point in time.

By applying the prefix property to $S3$, we can eliminate the disjunct $c = \langle b, a \rangle$ and obtain (4.4).

Property 4.3.1 is not expressible in STL. Operator $\Box$ is needed, and we use a second temporal operator as well: For any $i \geq 0$ and trace-set sequence ρ , let $\circ c$ (“the next value of c ”) be defined with respect to trace-set $\rho.i$ as the trace of channel c in trace-set $\rho.(i+1)$.¹ If ρ is finite, in the last trace-set let $\circ c = c$ (since there is no next trace-set). In effect, we convert finite sequences to infinite ones by repeating the final trace-set. Thus, for any sequence ρ , every channel c appearing in ρ has a corresponding and well-defined value $\circ c$ in each trace-set of the sequence. Intuitively, the value of $\circ c$ at any given time is the value that channel trace c will have after the next computation step.

We now state the prefix axiom.

Definition 4.3.2 (PREFIX) For any channel c , $\Box(c \preceq \circ c)$.²

The axiom asserts that the value of a channel trace c at any point in time is a prefix of c ’s trace at any later time. The axiom is thus equivalent to the prefix property as stated in Definition 4.3.1.

Theorem 4.3.3 (Soundness of PREFIX) $\rho \models PREFIX$ for any well-formed sequence of trace-sets ρ .

¹Operator $\circ$ corresponds to the “next” operator of temporal logic [37,47]. Do not confuse it with a second use of $\circ$ in temporal logic, where $\circ$ operates over formulas: $(\rho \models \circ f) \equiv (\langle \rho.1, \rho.2, \dots \rangle \models f)$.

²Axiom *PREFIX* can, in fact, be expressed using only a generalized version of $\Box$, but the formula is considerably more complicated. The expressibility of *ORDERING* and *PREFIX* is discussed further in Chapter 7.

Proof: Let ρ be any well-formed sequence of trace-sets. $\rho \models \text{PREFIX}$ follows directly from the definition of well-formedness: Since $\rho.(i+1)$ extends exactly one trace of $\rho.i$ by exactly one element (for all $0 \leq i < |\rho| - 1$), every channel trace of c in $\rho.i$ is a prefix of the corresponding trace in $\rho.(i+1)$. If $i = |\rho| - 1$, then by definition $c = \circ c$. Therefore PREFIX is a sound axiomatization of the prefix property. $\otimes$

4.4 Necessity and Sufficiency of Temporal Ordering and Prefix Axioms

Recall, from Section 4.1, that we are looking for a set of axioms whose conjunction A guarantees $(S1 \wedge A) \Rightarrow S2$ whenever $S1$ is a precise specification for a network N and $S2$ a valid specification for N . We prove that the temporal ordering and prefix axioms are necessary and sufficient for such an A .

There is a fundamental difference between any axiomatization of the temporal ordering or prefix property and specifications $S1$ and $S2$, because event ordering and prefix relations are always with respect to an entire computation—a sequence of trace-sets—while $S1$ and $S2$ are with respect to individual trace-sets. We employ $\Box$ to convert a specification to being on entire computations. By letting $A = \text{ORDERING} \wedge \text{PREFIX}$, we can prove that if $S1$ is a precise specification for N and $S2$ a valid specification for N , then

$$(\Box S1 \wedge A) \Rightarrow \Box S2. \quad (4.5)$$

In addition, we will argue that ORDERING and PREFIX are necessary for this—if either axiom is removed from A then we can find a network N with precise and valid specifications $S1$ and $S2$ (respectively) such that $\Box S1$ and A do not imply $\Box S2$.

4.4.1 Sufficiency

We begin with a definition and a key lemma.

Definition 4.4.1 Let ρ be any sequence of trace-sets. $Compress(\rho)$ is the sequence obtained from ρ by eliminating each trace-set that duplicates its immediate predecessor in the sequence.

Lemma 4.4.2 (Well-formedness) For any sequence of trace-sets ρ , $Compress(\rho)$ is well-formed if and only if $\rho \models ORDERING \wedge PREFIX$.

Proof: ($\Rightarrow$) If $Compress(\rho)$ is well-formed then $\rho \models ORDERING \wedge PREFIX$:

Since $Compress(\rho)$ is well-formed, by the soundness of $ORDERING$ and $PREFIX$ (Theorems 4.2.3 and 4.3.3) we know

$$Compress(\rho) \models ORDERING \wedge PREFIX.$$

Repetition of trace-sets cannot invalidate $ORDERING$ or $PREFIX$, hence $\rho \models ORDERING \wedge PREFIX$.

($\Leftarrow$) If $\rho \models ORDERING \wedge PREFIX$ then $Compress(\rho)$ is well-formed:

We prove the (equivalent) contrapositive: If $Compress(\rho)$ is not well-formed, then ρ does not satisfy $ORDERING$ and $PREFIX$. In fact, because repetition of trace-sets does not affect the validity of $ORDERING$ or $PREFIX$, it suffices to show that $Compress(\rho)$ —rather than ρ itself—does not satisfy $ORDERING$ and $PREFIX$. Thus, we show that any ρ that is not well-formed and has no repeated trace-sets does not satisfy $ORDERING$ and $PREFIX$. By Definition 3.6.2 of well-formedness, ρ then must exhibit at least one of the following conditions:

1. In the initial trace-set all channel traces are not empty.
2. Some channel trace decreases in length.
3. Some channel trace increases in length by more than 1.

4. Two channel traces increase in length at the same step.
5. Some channel trace element takes on more than one value. (A value changes spontaneously between trace-sets on a path.)

(The negation of the first part of Definition 3.6.2 is case 1, while negating the second part of Definition 3.6.2 results in cases 2–5.) We must show that in each case, one of *ORDERING* and *PREFIX* is violated. The proof proceeds by induction on the length of ρ .

Base Case: $|\rho| = 1$. Since ρ has only one trace-set, ρ must be ill-formed due to case 1—all channel traces are not empty in $\rho.0$. Let $|c| = x$ in $\rho.0$ for some channel c and some $x \geq 1$. Then $\rho \models \Box(|c| \geq 0 \Rightarrow |c| \geq x)$. Trivially, $\rho \models \Box(|c| \geq x \Rightarrow |c| \geq 0)$, so $\rho \models \Box(|c| \geq 0 \equiv |c| \geq x)$. By *ORDERING* we conclude $\rho \models \Box(|c| < 0 \wedge |c| < x)$. This last assertion is not true, so *ORDERING* does not hold for ρ .

Induction: $|\rho| = n + 1$, $n \geq 1$. Suppose, as the induction hypothesis, that any ρ' of length n that is not well-formed violates *ORDERING* and/or *PREFIX*. Now consider ρ . If $\langle \rho.0, \dots, \rho.(n-1) \rangle$ is not well-formed, then, by the induction hypothesis, $\langle \rho.0, \dots, \rho.(n-1) \rangle$ violates *ORDERING* or *PREFIX*, so ρ does also. Therefore, assume that $\langle \rho.0, \dots, \rho.(n-1) \rangle$ is well-formed. Then the ill-formedness of ρ is due to trace-sets $\rho.(n-1)$ and $\rho.n$, and must be of type 2, 3, 4, or 5 above. By cases:

2. Some channel trace decreases in length:

Let $|c| = x$ in $\rho.(n-1)$ and $|c| = y$ in $\rho.n$, for some c and $x > y \geq 0$. Then $c \preceq oc$ does not hold in $\rho.(n-1)$, $\Box(c \preceq oc)$ is not valid for ρ , and *PREFIX* is violated.

3. Some channel trace increases in length by more than 1:

Suppose $|c| = x$ in $\rho.(n-1)$ and $|c| = x + y$ in $\rho.n$, for some c , $x \geq 0$, and

$y \geq 2$. Recall that $\langle \rho.0, \dots, \rho.(n-1) \rangle$ is well-formed (by hypothesis), so we know $\langle \rho.0, \dots, \rho.(n-1) \rangle \models \Box(|c| \leq x)$, since $|c| = x$ in $\rho.(n-1)$. Therefore, $\rho \models \Box(|c| \geq x+1 \Rightarrow |c| \geq x+y)$. Now since $\Box(|c| \geq x+y \Rightarrow |c| \geq x+1)$ holds trivially, we obtain $\rho \models \Box(|c| \geq x+1 \equiv |c| \geq x+y)$. It is not the case, however, that $\rho \models \Box(|c| < x+1 \wedge |c| < x+y)$. Thus *ORDERING* does not hold.

4. Two channel traces increase in length at the same step:

Let $|c1| = x$ and $|c2| = y$ in $\rho.(n-1)$, and let $|c1| = x+1$ and $|c2| = y+1$ in $\rho.n$, for some $c1, c2, x \geq 0$, and $y \geq 0$. Since $\langle \rho.0, \dots, \rho.(n-1) \rangle$ is well-formed, $\rho \models \Box(|c1| \geq x+1 \equiv |c2| \geq y+1)$. Then by *ORDERING* it should be the case that $\rho \models \Box(|c1| < x+1 \wedge |c2| < y+1)$. This assertion is not valid, so *ORDERING* is violated.

5. A channel trace element takes on more than one value:

Suppose there is a channel-trace element $c.x$ such that $c.x = a$ in $\rho.(n-1)$, $c.x = b$ in $\rho.n$, and data items a and b are not identical. Then $c \preceq \circ c$ does not hold in $\rho.(n-1)$, $\Box(c \preceq \circ c)$ is not valid for ρ , and *PREFIX* is violated.

(*End Induction*)

We have shown that if ρ exhibits one of the five cases above, then ρ does not satisfy both of *ORDERING* and *PREFIX*. Suppose that in fact ρ is ill-formed in more than one way. Then consider a condition that involves a single channel—only case 4 involves two channels—and reasoning as above guarantees that one of *ORDERING* and *PREFIX* is still violated. Thus, we have shown that for any ρ satisfying *ORDERING* and *PREFIX*, $\text{Compress}(\rho)$ is well-formed. Together with the first half of the proof: if ρ is a sequence of trace-sets then $\text{Compress}(\rho)$ is well-formed if and only if $\rho \models \text{ORDERING} \wedge \text{PREFIX}$. $\otimes$

With this lemma in hand, we can easily prove that our two axioms are sufficient to guarantee the validity of implication (4.5).

Theorem 4.4.3 (Sufficiency of the Axioms) If $S1$ is a precise specification for network N and $S2$ a valid specification for N , then

$$\Box S1 \wedge ORDERING \wedge PREFIX \Rightarrow \Box S2.$$

Proof: We show that any sequence of trace-sets ρ satisfying $\Box S1$, $ORDERING$, and $PREFIX$, also satisfies $\Box S2$. Since $\rho \models ORDERING \wedge PREFIX$, by Lemma 4.4.2 we know that $Compress(\rho)$ is well-formed. Furthermore, $Compress(\rho) \models \Box S1$ since $\rho \models \Box S1$. (Note that for any trace-set ρ and specification S , $\rho \models \Box S$ if and only if $Compress(\rho) \models \Box S$.) Now recall from the formal definition of preciseness (Definition 3.6.3) that any well-formed sequence satisfying a precise specification is a path in the computation tree for the corresponding process or network. Thus, since $S1$ is precise, $Compress(\rho) \in Comps(N)$. By the validity of $S2$, every sequence in $Comps(N)$ satisfies $\Box S2$. Hence $Compress(\rho) \models \Box S2$ and consequently $\rho \models \Box S2$. $\otimes$

4.4.2 Necessity

Theorem 4.4.3 tells us that, with $ORDERING$ and $PREFIX$, we ensure that any valid network specification follows from a precise specification for the network. (In fact, by Preciseness-Preservation Theorem 3.6.4, only precise specifications for the component processes are needed.) Both axioms are necessary for the implication to always hold, as well as sufficient, as is shown by the following theorem.

Theorem 4.4.4 (Necessity of the Axioms) There exist networks $N1$, $N2$, and $N3$, with precise specifications $S1_P$, $S2_P$, and $S3_P$ (respectively) and valid specifications $S1_V$, $S2_V$, and $S3_V$ (respectively), such that

1. $\Box S1_P \wedge ORDERING \not\Rightarrow \Box S1_V$;
2. $\Box S2_P \wedge PREFIX \not\Rightarrow \Box S2_V$;
3. $\Box S3_P \not\Rightarrow \Box S3_V$.

Proof:

1. Let $N1$ be the example network of Section 3.7.4.
2. Let $N2$ be the example network of Section 3.7.3.
3. Let $N3$ be the example network of Section 3.7.3 or the example network of Section 3.7.4.

⊗

4.5 Example Encodings of the Axioms

We have proved that axiomatizations of the temporal ordering and prefix properties are necessary to achieve relative completeness. Since these two axioms are essential components of a relatively complete proof system, it is interesting to look at existing complete systems and identify how the axioms are represented.

Several proof systems involve explicit reasoning about every possible interleaving of communication events [8,17,38]. It is clear that such a logic will be relatively complete, since an exhaustive list of potential computations is an exact characterization of process and network behavior, including (implicitly) the constraints of the temporal ordering and prefix properties.

The proof system of Zwiers et. al. [56] is designed both for the specification of sequential processes and for the verification of their behavior when connected into a network. Thus, Hoare-style triples and inference rules are given (in the style of [2,30]), as well as a means for reasoning about specifications over channel traces. The logic includes a statement of the prefix property, written essentially as

$\{Tr = c\} Pgm \{Tr \preceq c\}$, where Pgm is any program segment.³ Reasoning about the temporal ordering property, however, is achieved only by enumerating all possible interleavings of the communication events of interest.

In [56], the authors also discuss the incompleteness of the Misra/Chandy system [39] and suggest a rule that would render it relatively complete. (A similar rule is proposed by Nguyen in [41].) Informally, the rule asserts the following: Let S be a valid specification for network N and let t be an interleaved trace of all communication events during any computation of N . Then every prefix of t satisfies S . This rule certainly captures the prefix property, and the temporal ordering property is encoded as well. To see this, suppose specification S constrains two communication events $c1.x$ and $c2.y$ (say) to occur simultaneously. Any trace t including only one of $c1.x$ and $c2.y$ will not satisfy S , so it cannot be a computation of N . Suppose, then, that both events are included in t . Consider any prefix p of t that contains one event but not the other. (Such a prefix must exist.) Then p will not satisfy S , since only one of $c1.x$ and $c2.y$ appears in p . Hence no computation of N can include either event.

This suggested rule is stated as an axiom in the relatively complete proof system of Hehner/Hoare [17]. (The axiom is the second of two given in our initial description of [17] in Section 1.4.4, and is restated in Section 3.8.) In Section 3.8, in fact, we use the rule to verify a valid specification for the Every-Other network of Section 3.7.2. The network illustrates the need for reasoning with the temporal ordering property; the axiom's role in the correctness proof parallels our explanation of how the rule encodes this property.

In [25], Jonsson identifies the fact (and problem) that valid specifications do not always follow from precise specifications, but no solution is proposed. The

³Recall the interpretation of this triple: If execution of Pgm is begun in any state in which channel trace c has value Tr , and if Pgm terminates, then upon termination Tr is a prefix of c .

author does suggest adding a proof rule of the form

$$\frac{N \text{ sat } S1}{N \text{ sat } S2}$$

which can be used whenever $S1$ and $S2$ are such that any network satisfying $S1$ will also satisfy $S2$. With a rule of inference like this, the issue of behavioral properties such as temporal ordering can essentially be ignored, but no formal method is given for deciding when a pair of specifications is a candidate for an application of the above rule.

The proof system of Nguyen et. al. [42] is based on temporal logic, so it is straightforward to formulate ordering constraints between network events in the logic. In addition, the “axioms for behaviors” (recall Section 1.4.5) include assertions that all traces are initially empty, that only one communication event can occur in a single time-step, that the prefix property holds, etc. These axioms are stated in temporal logic. Our *ORDERING* and *PREFIX* axioms could be similarly formulated, since we use temporal operators that are subsumed by the corresponding operators of temporal logic.

4.6 Expressing the Axioms

We have proved that axiomatizations of the temporal ordering and prefix properties are sufficient to achieve relative completeness. Therefore, for a relatively complete proof system to be as simple as possible, the system must be just strong enough to express and allow reasoning about temporal ordering and prefix axioms. We were unable to express *ORDERING* and *PREFIX* in STL, but were able to express them using a subset of temporal logic.

We are interested, then, in the exact expressive power a logic must have in order to form the basis of a relatively complete network proof system. We know that any logic in which one can express and reason about *ORDERING* and

PREFIX is strong enough. But *ORDERING* and *PREFIX*, although necessary and sufficient, may not be the only way to axiomatize the properties of computation that must be encoded.

Recall Lemma 4.4.2: given a sequence ρ of trace-sets, $Compress(\rho)$ is well-formed if and only if $\rho \models ORDERING \wedge PREFIX$. From this we conclude, in Theorem 4.4.3, that $\Box S1 \wedge A \Rightarrow \Box S2$ whenever $S1$ and $S2$ are precise and valid specifications (respectively) for a network N , where $A = ORDERING \wedge PREFIX$. Any A , however, such that $Compress(\rho)$ is well-formed if and only if $\rho \models A$, guarantees $\Box S1 \wedge A \Rightarrow \Box S2$. Thus, the actual function of A is to characterize well-formed computation.

In the next three chapters we discuss—first in general terms and then with respect to subsets of temporal logic—the expressive power required of a proof system in order to characterize well-formed network computation and thereby achieve relative completeness.

Chapter 5

A Model-Based Generalization

We are interested in the expressiveness required of a logic if it is to axiomatize those properties of computation that must be encoded in any relatively complete trace-based network proof system. From the results of Chapter 4, we know that any such logic must be at least strong enough to characterize well-formed network computation. That is, the logic must permit distinction of states¹ that are reachable by a well-formed computation from those states that are not.

As an example, recall the single-process network of Sections 3.7.3 and 4.2. The precise specification given for P is

$$(c = \langle \rangle \wedge d = \langle \rangle) \vee (c = \langle a \rangle \wedge d = \langle a \rangle).$$

A state in which $c = \langle \rangle$ and $d = \langle \rangle$ is reachable by a well-formed computation—the computation in which nothing happens. The only way a state in which $(c = \langle a \rangle \wedge d = \langle a \rangle)$ can be reached, however, is by a computation in which two messages are transmitted simultaneously, an illegal computation. The language of STL is not strong enough to express the fact that only an illegal computation can yield $(c = \langle a \rangle \wedge d = \langle a \rangle)$, so in STL this disjunct cannot be ruled out.

¹Throughout this chapter, the term *state* informally denotes the set of channel traces produced by a computation.

In trace-based proof systems, the specifications for processes and networks are often written in a first-order language over sequences [10,20,22,25,39], as in the first-order trace logic of STL. Therefore, to formalize the question of expressiveness, we begin with the standard notion of *models of first-order formulas* [5,9,13,49].² Typically, a model of a first-order formula is an assignment of the formula's variables to values such that performing the corresponding substitution renders the formula true. In our case, we are interested in assignments that both render a formula true and are reachable by actual network computation; we call these assignments the *computational models* of a formula. We want to determine exactly how expressive a logic must be if it is to distinguish computational models from all other models.

5.1 Models of First-Order Formulas

Let L be a language for first-order formulas and Var the set of variables in L . In mathematical logic [13,49], a model of a formula in L consists of

1. a domain V ,
2. an element of V for every constant symbol in L ,
3. an n -ary function on V for every n -ary function symbol in L (for all $n \geq 1$),
4. an n -ary relation on V for every n -ary relation symbol in L (for all $n \geq 1$),
5. an assignment $\alpha: Var \rightarrow V$ from the variables of L to values in V .

Now suppose that L is the language of *Trace-Formulas* (formally defined in Table 3.1). As in Section 3.1.1, we simplify our model by

²In Chapter 2 we defined a model of network computation and used it to prove properties of STL. The models of first-order formulas introduced for our purposes here are closely related; several corresponding definitions are given.

1. fixing a finite or infinite domain $V = \{v_1, v_2, v_3, \dots\}$, where v_j is a constant denoting the corresponding element of V ,
2. fixing a set of functions $Fns = \{F_1, F_2, F_3, \dots\}$ on V , and
3. fixing a set of predicates $Prds = \{G_1, G_2, G_3, \dots\}$ on V .

For our purposes, there is no need to distinguish between constant symbols and actual constants, function symbols and actual functions, or predicate symbols and actual predicates. Furthermore, although variables (and constants, predicates, and functions) in first-order trace logic can be of type integer or of type V , without loss of generality we assume that V subsumes—or even consists exactly of—the integers. Hence all that remains in our models of first-order formulas is item 5, an assignment from variables to values.

Recall that the free variables of a first-order trace formula are channel-trace variables; let C be the set of all such variables. In an assignment for a trace formula, then, we want to map variables not to elements of V , but to sequences of elements of V . Let Val be the set of all channel-trace values: all finite sequences of elements of V .

Definition 5.1.1 A *model* of a first-order trace formula f is an *assignment* $\alpha: C \rightarrow Val$, where C and Val are the sets of all channel-trace variables and values, respectively.

If α is an assignment taking c_i to v_i (say), $1 \leq i \leq k$, then we may also denote α by

$$\alpha = [c_1 \mapsto v_1, c_2 \mapsto v_2, \dots, c_k \mapsto v_k].$$

For any assignment α and variable c , $\alpha(c)$ denotes the value c is mapped to under α .

Definition 5.1.2 Let t be a term of first-order trace logic containing variables from C (only), and let α be an assignment. The *value of term t under assignment α* , denoted by $t|_\alpha$, is defined inductively as follows:

1. $k|_\alpha = k$, for a constant $k \in V$;
2. $c|_\alpha = \alpha(c)$, for a variable $c \in C$;
3. $F(t_1, \dots, t_n)|_\alpha = F(t_1|_\alpha, \dots, t_n|_\alpha)$, for a function $F \in Fns$ and terms $t_1, \dots, t_n$.

Definition 5.1.3 Let f be a formula of first-order trace logic and α an assignment. $\alpha \models f$, read as α *models* f , is defined inductively as follows:

1. $\alpha \models G(t_1, \dots, t_n)$ iff $G(t_1|_\alpha, \dots, t_n|_\alpha)$, for a predicate $G \in Prds$ and terms $t_1, \dots, t_n$;
2. $\alpha \models f1 \vee f2$ iff $\alpha \models f1$ or $\alpha \models f2$, for formulas $f1$ and $f2$;
3. $\alpha \models \neg f$ iff not $\alpha \models f$, for a formula f ;
4. $\alpha \models (\exists x: f)$ iff there exists a constant $k \in V$ such that $\alpha \models f[k/x]$, for a trace formula f , where $f[k/x]$ denotes formula f with all occurrences of variable x replaced by constant k .

5.2 Computational Models of First-Order Formulas

A model of a first-order trace formula is an assignment of the channel-trace variables to sequences of transmittable values. There is a clear correspondence between models and actual network computation: an assignment $\alpha = [c_1 \mapsto v_1, \dots, c_k \mapsto v_k]$ represents the state reached by a network N with communication channels $c_1, \dots, c_k$, in which sequences $v_1, \dots, v_k$ are the respective traces of channels $c_1, \dots, c_k$.

Once again consider process P with precise specification

$$S: (c = \langle \rangle \wedge d = \langle \rangle) \vee (c = \langle a \rangle \wedge d = \langle a \rangle).$$

Let α_1 be the assignment that maps both c and d to $\langle \rangle$, i.e. $\alpha_1 = [c \mapsto \langle \rangle, d \mapsto \langle \rangle]$. Clearly $\alpha_1 \models S$. Now let $\alpha_2 = [c \mapsto \langle a \rangle, d \mapsto \langle a \rangle]$, so $\alpha_2 \models S$ as well. Both α_1 and α_2 are models of S , but only α_1 represents a state that can be reached by actual network computation. Therefore, α_1 is a computational model of S and α_2 is not.

We now develop a formal definition of computational models.

Definition 5.2.1 Define $\mathcal{A}$ to be the set of all possible assignments $\alpha: C \rightarrow Val$.

Let the relation $R: \mathcal{A} \rightarrow \mathcal{A}$ be defined as follows:

For any assignments $\alpha_1, \alpha_2 \in \mathcal{A}$, $(\alpha_1, \alpha_2) \in R$ if and only if

1. there exists some $c \in C$ such that $\alpha_1(c) \preceq \alpha_2(c)$ and $|\alpha_1(c)| + 1 = |\alpha_2(c)|$,
and
2. for all $c' \in C$ such that $c' \neq c$, $\alpha_1(c') = \alpha_2(c')$.

Thus, informally, assignments α_1 and α_2 are related by relation R if and only if the state represented by α_2 can be reached from the state represented by α_1 in a single legal computation step. (R is analogous to the parent-child relationship in the computation trees of Chapter 2.)

Definition 5.2.2 Let f be a first-order trace formula. Define the *restriction of R to f* to be the relation R/f such that for any assignments $\alpha_1, \alpha_2 \in \mathcal{A}$, $(\alpha_1, \alpha_2) \in R/f$ if and only if $(\alpha_1, \alpha_2) \in R$, $\alpha_1 \models f$, and $\alpha_2 \models f$.

Hence R/f is the maximal subset of R such that all assignments in the relation are models of f .

Let α_0 be the distinguished assignment in which all channel-trace variables map to $\langle \rangle$, i.e. for all $c \in C$, $\alpha_0(c) = \langle \rangle$. α_0 represents the initial state of

any network computation. Let $(R/f)^*$ be the reflexive transitive closure [24] of relation R/f .

Definition 5.2.3 (Computation Model) If f is a first-order trace formula and α is an assignment, α is a *computational model of f* if and only if $(\alpha_0, \alpha) \in (R/f)^*$.³

Definition 5.2.3 formalizes the notion that a computational model for a formula f represents a state that is reachable from the initial state in which all channel traces are empty by a sequence of legal single steps (R) that always maintain f . For any first-order trace formula f , the computational models of f are a subset of the models of f .

5.3 Computational Models and Proof Systems

In most trace-based proof systems, the specifications for processes and networks are first-order trace formulas [10,20,22,25,39,53]. Thus far, we have formally defined models and computational models of first-order trace formulas and, hence, of process and network specifications. We showed (in Chapter 4) that any relatively complete proof system must somehow encode axioms that characterize well-formed computation, and we showed (in Section 5.2) that computational models represent exactly those states that are reachable by a well-formed computation. Therefore, a “computational model test” is implicitly present in any relatively complete proof system. We explore this relationship between computational models and proof systems.

³Note that if $\alpha_0 \not\models f$ then there can be no $(\alpha_0, \alpha) \in (R/f)^*$ and consequently f has no computational models. This is exactly what we expect for a formula not satisfied by an initial state.

5.3.1 Valid and Precise Specifications

Consider a network N with precise (STL) specification $S1$. Recall, from Definition 3.6.3, that since $S1$ is precise, every well-formed computation always satisfying $S1$ is a possible computation of N . A computational model of any formula f represents a state reachable by a well-formed computation always satisfying f . Therefore, by transitivity, if α is a computational model of $S1$, then α represents a state reachable by a possible computation of N . Furthermore, by the validity of $S1$, any state reachable by a possible computation of N is represented by a computational model of $S1$.

Now suppose we are given a specification $S2$ and would like to determine, based on $S1$, whether $S2$ is valid for N . Specification $S2$ is valid if every computation of N , up to any point in time, yields channel traces that satisfy $S2$. We know that the computational models of $S1$ represent all and only those states reachable by N . Hence $S2$ is valid for N if and only if every computational model of $S1$ is a model of $S2$. The following definitions and theorems formalize this result.

(We must first redefine the notion of valid and precise specifications in terms of models of first-order formulas. The correspondence with Definitions 3.1.4 and 3.6.3 is clear.)

Definition 5.3.1 A specification S is *valid* for a network N if and only if, for all assignments α such that α represents a state reachable by N , $\alpha \models S$.

Definition 5.3.2 A specification S is *precise* for a network N if and only if

1. S is valid for N , and
2. every computational model of S represents a state reachable by N .

Definition 5.3.3 For a specification S , let $Models(S)$ denote the set of models of S and $CompModels(S)$ denote the set of computational models of S .

Theorem 5.3.4 Given a network N with precise specification $S1$, a specification $S2$ is valid for N if and only if $CompModels(S1) \subseteq Models(S2)$.

Proof: Consider an arbitrary network N with precise specification $S1$.

($\Rightarrow$) If specification $S2$ is valid for N , then $CompModels(S1) \subseteq Models(S2)$:

Since $S2$ is valid for N , every assignment α representing a state reachable by N is a model of $S2$. By the preciseness of $S1$, every computational model of $S1$ represents a state reachable by N . Therefore, every computational model of $S1$ is a model of $S2$, and $CompModels(S1) \subseteq Models(S2)$.

($\Leftarrow$) If $CompModels(S1) \subseteq Models(S2)$, then $S2$ is valid for N .

Suppose every computational model of $S1$ is a model of $S2$. By the preciseness of $S1$, every computational model of $S1$ represents a state reachable by N . Then, by supposition, every assignment representing a state reachable by N is a model of $S2$. Hence $S2$ is a valid specification for N . $\otimes$

Corollary 5.3.5 Given a network N with precise specification $S1$, a specification $S2$ is valid for N if and only if $CompModels(S1) \subseteq CompModels(S2)$.

Proof: By Theorem 5.3.4, we know that $S2$ is valid for N if and only if $CompModels(S1) \subseteq Models(S2)$. It suffices to show, then, that for any specifications $S1$ and $S2$,

$$\begin{aligned} &CompModels(S1) \subseteq Models(S2) \text{ iff} \\ &CompModels(S1) \subseteq CompModels(S2). \end{aligned}$$

($\Rightarrow$) If $CompModels(S1) \subseteq Models(S2)$
then $CompModels(S1) \subseteq CompModels(S2)$:

Suppose every computational model of $S1$ is a model of $S2$, and consider any such model α . We must show that α is a computational model of $S2$. By

Definition 5.2.3 of a computational model, $(\alpha_0, \alpha) \in (R/S1)^*$. Therefore, by the definition of $(R/S1)^*$, $\alpha_0 \models S1$ and there is a sequence of assignments $\alpha_1, \alpha_2, \dots, \alpha_m$, such that

1. $\alpha_1 \models S1, \alpha_2 \models S1, \dots, \alpha_m \models S1$,
2. for all $i, 0 \leq i < m, (\alpha_i, \alpha_{i+1}) \in R$,
3. $(\alpha_m, \alpha) \in R$.

Thus, the α_i 's, $0 \leq i \leq m$, are all computational models of $S1$. Then, by assumption, α_i is a model of $S2$, $0 \leq i \leq m$. We already know that $\alpha \models S2$. So, by the same sequence $\alpha_0, \alpha_1, \dots, \alpha_m, \alpha$ of assignments, $(\alpha_0, \alpha) \in (R/S2)^*$. Hence α is a computational model of $S2$.

($\Leftarrow$) If $CompModels(S1) \subseteq CompModels(S2)$
 then $CompModels(S1) \subseteq Models(S2)$:

Follows transitively from the fact that $CompModels(S2) \subseteq Models(S2)$. $\otimes$

Figure 5.1 illustrates Theorem 5.3.4 (and Corollary 5.3.5) by showing the relationship among the sets of models and computational models of specifications $S1$ and $S2$, when $S1$ and $S2$ are valid and precise, respectively, for some network N . Note that the pictured inclusions need not be strict.

5.3.2 Computational Models in STL

A compositional proof system for networks of processes is complete if and only if, for any network N , every valid specification for N can be deduced from a precise specification for N . A precise specification S can be obtained, using the network composition rule, from precise specifications for N 's component processes. If all valid specifications for N can be deduced from S , then all valid specifications are provable. Conversely, if all valid specifications for N are provable, then the

Figure 5.1: Models and computational models for specifications $S1$ and $S2$

proof system must include a means for deducing valid specifications from precise specifications.

Computational models characterize the relationship between valid and precise specifications. Let N be a network with precise specification $S1$. By Theorem 5.3.4, specification $S2$ is valid for N if and only if $CompModels(S1) \subseteq Models(S2)$. Using this fact, we might consider extending STL to permit a consequence rule as follows.

Definition 5.3.6 (Model-Based Consequence Rule)

$$\frac{N \text{ sat } S1; \alpha \in CompModels(S1) \Rightarrow \alpha \in Models(S2), \text{ for all } \alpha}{N \text{ sat } S2}$$

Incorporating the Model-Based Consequence Rule appears to result in a relatively complete proof system, since every valid specification for a network is provable. We cannot simply insert this rule into STL, however. Recall that, for relative completeness, we assumed the provability of all valid first-order formulas (Section 3.6.2). Statements of the form $S1 \Rightarrow S2$, as in the original STL consequence rule, are formulas of first-order logic; statements of the form $\alpha \in CompModels(S1) \Rightarrow \alpha \in Models(S2)$ are not. Thus, in STL, we have no method for proving $\alpha \in CompModels(S1) \Rightarrow \alpha \in Models(S2)$, even if it is valid.

The system is still incomplete. Relative completeness is achieved only by adding a facility for formal reasoning over formulas of the form $\alpha \in \text{CompModels}(S1)$ and $\alpha \in \text{Models}(S2)$.

Recall Theorem 4.4.3: If $S1$ is a precise specification for network N and $S2$ a valid specification for N , then

$$\Box S1 \wedge \text{ORDERING} \wedge \text{PREFIX} \Rightarrow \Box S2.$$

This result suggests extending STL to permit the following consequence rule.

Definition 5.3.7 (Axiom-Based Consequence Rule)

$$\frac{N \text{ sat } S1, (\Box S1 \wedge \text{ORDERING} \wedge \text{PREFIX}) \Rightarrow \Box S2}{N \text{ sat } S2}$$

Here, again, we are dealing with formulas not written in first-order logic—formulas of the form $(\Box S1 \wedge \text{ORDERING} \wedge \text{PREFIX}) \Rightarrow \Box S2$. Therefore, as with Definition 5.3.6, including such a consequence rule does not automatically yield a relatively complete proof system.

In both cases, we constructed a consequence rule that guarantees the inference of valid specifications from precise specifications, but the rule is not written in first-order logic. (In fact, in Chapter 7 we formally prove that it is impossible to write such a rule in first-order logic.) It is interesting to look at the relationship between the two revised consequence rules.

In rule 5.3.6, the relevant formula is

$$\alpha \in \text{CompModels}(S1) \Rightarrow \alpha \in \text{Models}(S2). \quad (5.1)$$

Informally, (5.1) asserts that any state reachable by a well-formed computation always satisfying $S1$ will also satisfy $S2$. In rule 5.3.7, the relevant formula is

$$(\Box S1 \wedge \text{ORDERING} \wedge \text{PREFIX}) \Rightarrow \Box S2. \quad (5.2)$$

This formula states that any computation always satisfying $S1$, *ORDERING*, and *PREFIX*, will also always satisfy $S2$. Recall, from Lemma 4.4.2, that a computation is well-formed if and only if it satisfies *ORDERING* and *PREFIX*. Therefore, formula (5.2) asserts that any well-formed computation that always satisfies $S1$ will also always satisfy $S2$. Formula (5.2) is thus equivalent to formula (5.1).

It is expected that the temporal ordering and prefix axioms are an encoding of the *CompModels* membership test of formula (5.1). As discussed in Section 4.6, the purpose of *ORDERING* and *PREFIX* is to eliminate the non-computational models. Rule 5.3.6 is a model-based description of the deductive reasoning power that must be available in any relatively complete proof system. In the next section we discuss how this computational model test is encoded in other trace-based proof systems.

5.3.3 Computational Models in Other Trace Logics

The trace logics of [10,20,22,39] are not strong enough to distinguish computational models from non-computational models and, consequently, are incomplete. STL is similarly incomplete, but we have shown how the STL consequence rule can be extended to include computational model recognition (Definition 5.3.6). With the appropriate reasoning tools, this extended consequence rule allows any valid specification for a network to be deduced from a precise specification for that network. Although an encoding of such a computational model test must appear in every relatively complete trace-based proof system (Theorem 5.3.4), the test need not necessarily appear in a consequence rule.

In both [17] and [42], computational model recognition is encoded as a set of axioms, which are informally described in Sections 1.4.4 and 1.4.5, respectively. (See also Sections 3.8 and 4.5.) In essence, both sets of axioms state:

For any network N and specification S :

If S is valid for N then $CM(S)$ is valid for N ,

where $CM(S)$ is a specification derived from S such that a model α satisfies $CM(S)$ if and only if α is a computational model of S . Providing such a set of axioms is equivalent to providing an extended consequence rule like Definition 5.3.6. In a proof system with computational model axioms, to derive a valid specification $S2$ for some network N from a precise specification $S1$ for N , one first uses the axioms to prove that $CM(S1)$ is valid for N . By Theorem 5.3.4, $CM(S1) \Rightarrow S2$, since $S1$ is precise and $S2$ is valid. Therefore, a standard consequence rule (e.g. Definition 3.2.2) can then be used to prove that $S2$ is valid for N . This procedure is equivalent to directly proving the validity of $S2$ through an extended consequence rule like 5.3.6.

Using the notation of STL, these axioms for computational model recognition can be encoded as the following inference rule:

$$\frac{N \text{ sat } S}{N \text{ sat } CM(S)} \quad (5.3)$$

Such a rule is informally suggested by Jonsson in [25] (recall Section 4.5), but no actual encoding of $CM(S)$ is provided.

Inference rule (5.3) can also be directly incorporated into the network composition rule:

$$\frac{P_1 \text{ sat } S_1, P_2 \text{ sat } S_2, \dots, P_n \text{ sat } S_n}{P_1 \parallel P_2 \parallel \dots \parallel P_n \text{ sat } CM(\bigwedge_i S_i)} \quad (5.4)$$

although we have not seen such an encoding in existing proof systems. With inference rule (5.4), precise specifications for a set of processes are used to directly infer a network specification that is not only precise, but, by Theorem 5.3.4, implies every valid specification for that network.

We have described several methods by which trace-based proof systems can incorporate the computational model test to achieve relative completeness. Each method consists of some tool (or tools) to strengthen a proven network specification into a specification that is satisfied only by computational models. The

stronger specification can then be used to deduce any valid specification for the network. We now consider the expressive power required for such strengthening of specifications.

5.4 Recognizing Computational Models

In any relatively complete trace-based proof system, it is necessary to eliminate states that are reachable only by illegal computations, that is, it is necessary to be able to distinguish computational models from all other models. Formulas (5.1) and (5.2) are examples of such distinction of computational models. Neither formula is written in first-order trace logic, so incorporating such a formula into a proof system based on first-order trace logic requires extending the reasoning capabilities of the system. Since we want trace logics to remain as simple as possible, we would like to determine the minimum reasoning power necessary for the recognition of computational models.

Let f be any formula of first-order trace logic; f may have both computational and non-computational models. To distinguish the computational models of f from other models, a formula $CM(f)$ must be constructed such that a model α satisfies $CM(f)$ if and only if α is a computational model of f . Clearly $\alpha \in CompModels(f)$ is exactly such a formula—by the definition of $CM(f)$ —but it reveals very little about what actually is being expressed. For this we must return to the formal definition of a computational model, Definition 5.2.3: α is a computational model of f if and only if $(\alpha_0, \alpha) \in (R/f)^*$.

In the proof of Corollary 5.3.5, we noted that $(\alpha_0, \alpha) \in (R/f)^*$ if and only if there exists a sequence $\alpha_1, \alpha_2, \dots, \alpha_m$ of assignments such that

1. $\alpha_0 \models f, \alpha_1 \models f, \dots, \alpha_m \models f,$
2. for all $i, 0 \leq i < m, (\alpha_i, \alpha_{i+1}) \in R,$

3. $(\alpha_m, \alpha) \in R$.

Therefore, $CM(f)$ must assert the existence of a sequence of assignments satisfying criteria 1–3. Since this sequence can be arbitrarily long, it is natural to write $CM(f)$ inductively.

Suppose, without loss of generality, that the communication channels of all networks under consideration are $c_1, \dots, c_k$, $k \geq 1$.⁴ We use $\bar{t} = [t_1, \dots, t_k]$ to denote a k -tuple of channel traces.

Definition 5.4.1 (Recursive Definition: $CM_R(f)$)

$$CM_R(f) \equiv LFP(CM-Rec(f, [\langle \rangle, \dots, \langle \rangle], [c_1, \dots, c_k])),$$

where

1. $LFP(F)$ is the least fixed point [51] of function F , and

2. $CM-Rec(f, \bar{t}, \bar{u}) \equiv$

$$\begin{aligned} & f[\bar{t}/\bar{c}]^5 \wedge \\ & (\bar{t} = \bar{u}) \vee \\ & (\exists j, v: 1 \leq j \leq k, v \in V: \\ & \quad CM-Rec(f, \bar{t}[t_j \cdot \langle v \rangle / t_j]^6, \bar{u})). \end{aligned}$$

Definition 5.4.1, although somewhat unwieldy, amounts only to an explicit encoding of the reflexive-transitive closure membership test: $(\alpha_0, \alpha) \in (R/f)^*$. For any formula f and assignment α , $\alpha \models CM_R(f)$ if and only if α is a computational model of f .⁷

⁴Parameterizing a definition of $CM(f)$ to permit arbitrary sets of communication channels is straightforward.

⁵ $f[\bar{t}/\bar{c}]$ denotes formula f with free variables $c_1, \dots, c_k$ replaced by channel-trace values $t_1, \dots, t_k$.

⁶ $\bar{t}[t_j \cdot \langle v \rangle / t_j]$ denotes tuple $\bar{t}$ with channel trace t_j extended by the value v .

⁷If $f[\langle \rangle, \dots, \langle \rangle / c_1, \dots, c_k] \equiv false$, then there is no α such that $\alpha \models CM_R(f)$. This corresponds to $\alpha_0 \not\models f$ (Section 5.2.3), the case in which f has no computational models because it is not satisfied by an initial state.

We can also write $CM(f)$ non-recursively by allowing quantification over a variable representing an arbitrarily long sequence of tuples, thereby “unrolling” the recursion.

Definition 5.4.2 (Non-recursive Definition: $CM(f)$)

$$\begin{aligned}
 CM(f) \equiv & \\
 & (\exists \langle \bar{t}^0, \bar{t}^1, \dots, \bar{t}^n \rangle : \\
 & \quad \bar{t}^0 = [\langle \rangle, \dots, \langle \rangle] \wedge \\
 & \quad \bar{t}^n = [c_1, \dots, c_k] \wedge \\
 & \quad (\forall i: 0 \leq i \leq n: f[\bar{t}^i / \bar{c}]) \wedge \\
 & \quad (\forall i: 0 \leq i < n: \\
 & \quad \quad (\exists j, v: 1 \leq j \leq k, v \in V: \\
 & \quad \quad \quad \bar{t}^{i+1} = \bar{t}^i[t_j^i \cdot \langle v \rangle / t_j^i]))))
 \end{aligned}$$

Again, for any formula f and assignment α , $\alpha \models CM(f)$ if and only if α is a computational model of f .

It would be unwise to actually use Definitions 5.4.1 or 5.4.2 in a proof system. Theoretically, with the appropriate reasoning tools, the formulas yield relatively complete systems. In practice, verifying such formulas requires either explicit or implicit construction of arbitrarily long sequences drawn from an exponentially large pool of possibilities—i.e. the enumeration of all possible computations. (This is the approach taken in [17,38], where the proof systems are of purely theoretical interest.)

We want to find a more appropriate language for the expression of formulas such as $CM(f)$. By defining a mapping between candidate languages and the extended first-order language of Definition 5.4.2, $CM(f)$ can itself be used to establish expressiveness bounds: a language must contain operators necessary and sufficient for writing a formula equivalent to $CM(f)$.

The semantics of temporal logic [37,47] are based on sequences of states, so

the language of temporal logic is appropriate for our goals. In the next two chapters, we isolate a subset of temporal logic that is both an upper and a lower bound on the expressiveness required for the recognition of computational models; this subset is thus a minimally sufficient language on which to base a relatively complete trace-based proof system.

Chapter 6

Temporal Logic and Computational Models

A relatively complete trace logic must include enough reasoning power to discriminate between computational and non-computational models. Therefore, given a formula f of first-order trace logic, we are interested in a formula $CM(f)$ such that an assignment α is a model of $CM(f)$ if and only if α is a computational model of f . Using a first-order language extended to allow quantification over an arbitrarily long sequence of states (the states are encoded as tuples), Definition 5.4.2 is one such formula. Definition 5.4.2 is not very useful, however, since verifying $CM(f)$ can involve enumerating a large number of sequences. We want to find a suitable yet minimal language in which to write a formula equivalent to $CM(f)$.

Temporal logic (TL), as defined in [37,47], is a logic for formal reasoning over sequences.¹ TL is based on first-order predicate logic, but also includes *temporal operators* for assertions that range over time—assertions that refer to a sequence of states rather than to a single state. TL is not as expressive as one might need for certain applications [18,55], and extensions and modifications have been

¹We consider only the *linear-time* version of temporal logic [47].

Figure 6.1: Semantics-preservation

suggested [18,31,55]. It is known, however, that TL is at least strong enough for building a relatively complete trace-based proof system [42]. Thus, for our purposes, we are interested only in subsets of TL. In particular, we want to find a subset of TL that is necessary and sufficient to encode Definition 5.4.2 of $CM(f)$.

Let *EPL* (for *Extended Predicate Logic*) be the language used in Definition 5.4.2: first-order trace logic with sequences of tuples. To compare a formula in TL with $CM(f)$ —a formula in EPL—we need a syntactic transformation function $\mathcal{T}$ from formulas in TL to formulas in EPL. Let M be a set of models for formulas in TL, and let t be some function mapping models in M to models of EPL formulas (i.e. to assignments). Transformation $\mathcal{T}$ must be *semantics-preserving*, in that for any formula f_{TL} in TL and model $\mu \in M$,

$$\mu \models f_{TL} \text{ iff } t[\mu] \models \mathcal{T}[f_{TL}].$$

Semantics-preservation is illustrated by the commutative diagram of Figure 6.1. Once a semantics-preserving transformation function $\mathcal{T}$ has been defined, a TL formula f_{TL} is proven equivalent to an EPL formula f_{EPL} by showing $\mathcal{T}[f_{TL}] \equiv f_{EPL}$.

In this chapter, a formalism is developed for proving equivalence of TL formulas and $CM(f)$. The syntax and semantics of TL are given, and a semantics-

preserving transformation function $\mathcal{T}: \text{TL} \rightarrow \text{EPL}$ is defined. To accommodate the definition of $\mathcal{T}$, $CM(f)$ is modified into an equivalent formula containing a different set of free variables; we call the revised formula $CM_{EPL}(f)$.

In Chapter 7, a hierarchy of temporal logic subsets is introduced, and transformation $\mathcal{T}$ is used to prove that a certain subset is an upper and a lower bound on the power needed to express a formula equivalent to $CM_{EPL}(f)$. This subset is therefore a minimal language on which to build a relatively complete trace-based proof system.

6.1 Temporal Logic

A model of a first-order formula is an assignment from variables to values, as defined in Chapter 5. During the course of a computation, variables may take on different values at different times. For this reason, in a programming logic it is convenient to use a language modeled by sequences of states—mappings from variables to values at successive points in time. Temporal logic is one such language.

6.1.1 Syntax

The syntax of TL consists of the full syntax of first-order predicate logic, with the addition of four temporal (or *modal*) operators:

1. The *Henceforth* (or *Always*) operator, $\Box$. Informally, $\Box f$ is valid if and only if TL formula f is valid at the current point in time and at every point in the future. (A restricted version of $\Box$ was used in Chapters 3–5.)
2. The *Eventually* operator, $\Diamond$. Informally, $\Diamond f$ is valid if and only if TL formula f is valid either at the current point in time or at some point in the future. Operator $\Diamond$ is the dual of $\Box$, in that for any TL formula f , $\Box f \equiv \neg \Diamond \neg f$ and $\Diamond f \equiv \neg \Box \neg f$.

3. The *Next* operator, $\circ$. Informally, $\circ f$ is valid if and only if TL formula f is valid at the next point in time. The *Next* operator is also defined over terms: for a TL term t , $\circ t$ is the value of t at the next point in time. (A simplified version of $\circ$ over terms was used in Chapters 3–5.)
4. The *Until* operator, $\mathcal{U}$. Informally, $f1 \mathcal{U} f2$ is valid for TL formulas $f1$ and $f2$ if and only if $f2$ is valid either at the current point in time or at some point in the future, and $f1$ is valid at all points from the current point to the point at which $f2$ becomes valid.²

The only free variables in Definition 5.4.2 of $CM(f)$ are $c_1, c_2, \dots, c_k$, so we are still interested in formulas in which all free variables are channel-trace variables. We use a version of TL based on first-order trace logic: the set of free variables is $\{c_1, \dots, c_k\}$, and we continue to assume a fixed domain V , a fixed set of functions Fns , and a fixed set of predicates $Prds$, as in Chapter 5. The full syntax of TL is given in Table 6.1; Table 6.2 lists some syntactic abbreviations.

6.1.2 Semantics

A semantics for TL is given by providing a definition for validity of TL formulas with respect to a set of formal models. Models of TL formulas are sequences of states. Let $C = \{c_1, \dots, c_k\}$ be the set of free variables, and let Val be the set of all channel-trace values: all finite sequences of elements of V .

Definition 6.1.1 A model of a TL formula is an infinite *state-sequence* $\sigma = \langle \sigma.0, \sigma.1, \sigma.2, \dots \rangle$, where each $\sigma.i$, $i \geq 0$, is an assignment from the variables in C to values in Val .

²Some definitions of temporal logic instead use a *Weak Until* operator, in which $f2$ need not ever become valid as long as $f1$ is always valid. In the context of TL, the two versions of $\mathcal{U}$ are expressively equivalent [55].

Table 6.1: Syntax of Temporal Logic

<i>TL-Formula</i>	$::=$	$G(t_1, \dots, t_n)$	$G \in Prds; t_1, \dots, t_n \text{ TL-Terms}$
		$f1 \vee f2$	$f1 \text{ and } f2 \text{ TL-Formulas}$
		$\neg f$	$f \text{ a TL-Formula}$
		$(\exists x: f)$	$f \text{ a TL-Formula};$ $x \text{ a variable ranging over } V$
		$\Box f$	$f \text{ a TL-Formula}$
		$\circ f$	$f \text{ a TL-Formula}$
		$f1 \mathcal{U} f2$	$f1 \text{ and } f2 \text{ TL-Formulas}$
<i>TL-Term</i>	$::=$	k	$k \text{ a constant in } V$
		x	$x \text{ a variable}$
		$F(t_1, \dots, t_n)$	$F \in Fns; t_1, \dots, t_n \text{ TL-Terms}$
		$\circ t$	$t \text{ a TL-Term}$

Table 6.2: Syntactic Abbreviations

$f1 \wedge f2$	for	$\neg(\neg f1 \vee \neg f2)$	$f1 \text{ and } f2 \text{ TL-Formulas}$
$f1 \Rightarrow f2$	for	$\neg f1 \vee f2$	$f1 \text{ and } f2 \text{ TL-Formulas}$
$(\forall x: f)$	for	$\neg(\exists x: \neg f)$	$f \text{ a TL-Formula};$ $x \text{ a variable ranging over } V$
$\Diamond f$	for	$\neg \Box \neg f$	$f \text{ a TL-Formula}$

We do not lose generality by allowing only infinite state-sequences. Any finite sequence σ can be converted to an infinite sequence σ' by indefinitely repeating σ 's last state: $\sigma' = \langle \sigma.0, \sigma.1, \dots, \sigma.(|\sigma|-1), \sigma.(|\sigma|-1), \sigma.(|\sigma|-1), \dots \rangle$. (This is effectively what was done in Section 4.3, when we defined $\circ c = c$ in the last state of a sequence.) A similar conversion from finite to infinite sequences is used in [33,34,37,45].

For a state-sequence σ and index $i \geq 0$, let $\sigma[i..]$ denote sequence σ beginning with element $\sigma.i$, i.e. $\sigma[i..] = \langle \sigma.i, \sigma.(i+1), \sigma.(i+2), \dots \rangle$.

Definition 6.1.2 Let t be a term of TL containing variables from C (only), and let σ be a state-sequence. The *value of term t in state-sequence σ* , denoted by $t|_\sigma$, is defined inductively as follows:

1. $k|_\sigma = k$, for a constant $k \in V$;
2. $c|_\sigma = \sigma.0(c)$, for a variable $c \in \{c_1, \dots, c_k\}$;
3. $F(t_1, \dots, t_n)|_\sigma = F(t_1|_\sigma, \dots, t_n|_\sigma)$, for a function $F \in Fns$ and TL terms $t_1, \dots, t_n$.
4. $\circ t|_\sigma = t|_{\sigma[1..]}$, for a TL term t .

Definition 6.1.3 Let σ be a state-sequence and f a formula of TL. $\sigma \models f$, read as σ *models* f , is defined inductively as follows:

1. $\sigma \models G(t_1, \dots, t_n)$ iff $G(t_1|_\sigma, \dots, t_n|_\sigma)$, for a predicate $G \in Prds$ and TL terms $t_1, \dots, t_n$;
2. $\sigma \models f1 \vee f2$ iff $\sigma \models f1$ or $\sigma \models f2$, for TL formulas $f1$ and $f2$;
3. $\sigma \models \neg f$ iff not $\sigma \models f$, for a TL formula f ;
4. $\sigma \models (\exists x: f)$ iff there exists a constant $k \in V$ such that $\sigma \models f[k/x]$, for a TL formula f .
5. $\sigma \models \Box f$ iff for all $i \geq 0$, $\sigma[i..] \models f$, for a TL formula f ;

by transforming all free variables c_j of f_{TL} to terms of the form $s.i(c_j)$ —remember that $s.i(c_j)$ is actually “implemented” as $(s.i)_j$ —where $i \geq 0$. If $\mathcal{T}$ is defined such that for any TL formula f_{TL} and state-sequence σ ,

$$\sigma \models f_{TL} \text{ iff } [s \mapsto \sigma] \models \mathcal{T}[\![f_{TL}]\!],$$

then $\mathcal{T}$ is an appropriate semantics-preserving transformation function. A formal definition of $\mathcal{T}$ is given in the next section, followed by an example of the mapping and a proof of its correctness.

6.2.1 Definition of the Mapping

An inductive definition of transformation function $\mathcal{T}$, from TL formulas to EPL formulas, is given in Table 6.3. The definition directly parallels Definition 6.1.3 of validity of TL formulas. In defining $\mathcal{T}$, a second transformation function, $\mathcal{R}$, from TL terms to EPL terms, is needed. $\mathcal{R}$ is defined in Table 6.3 as well.

6.2.2 Example of the Mapping

We illustrate the mapping on a small example. Let c be a channel-trace variable and k a constant. Consider the TL formula $f_{EX} \equiv \Box(\circ c.0 = k)$, which has no significance beyond its syntactic properties. $\mathcal{T}$ is used to transform f_{EX} into an EPL formula, $\mathcal{T}[\![f_{EX}]\!]$, such that for any state-sequence σ , $\sigma \models f_{EX}$ iff $[s \mapsto \sigma] \models \mathcal{T}[\![f_{EX}]\!]$.

$$\begin{aligned} & \mathcal{T}[\![\Box(\circ c.0 = k)]\!] \\ &= (\forall i: i \geq 0: \mathcal{T}[\![\circ c.0 = k]\!][s[i..]/s]) \\ &= (\forall i: i \geq 0: (\mathcal{R}[\![\circ c.0]\!] = \mathcal{R}[\![k]\!])[s[i..]/s]) \\ &= (\forall i: i \geq 0: (\mathcal{R}[\![c.0]\!][s[1..]/s] = \mathcal{R}[\![k]\!][s[i..]/s])) \\ &= (\forall i: i \geq 0: ((\mathcal{R}[\![c]\!].\mathcal{R}[\![0]\!])[s[1..]/s] = \mathcal{R}[\![k]\!])[s[i..]/s])) \end{aligned}$$

Table 6.3: Transformation Function $\mathcal{T}$

$\mathcal{T}[G(t_1, \dots, t_n)]$	$= G(\mathcal{R}[t_1], \dots, \mathcal{R}[t_n])$	$G \in Prds;$ $t_1, \dots, t_n$ TL terms
$\mathcal{T}[f1 \vee f2]$	$= \mathcal{T}[f1] \vee \mathcal{T}[f2]$	$f1$ and $f2$ TL formulas
$\mathcal{T}[\neg f]$	$= \neg \mathcal{T}[f]$	f a TL formula
$\mathcal{T}[(\exists x: f)]$	$= (\exists k: \mathcal{T}[f[k/x]])$	f a TL formula; k a constant in V
$\mathcal{T}[\Box f]$	$= (\forall i: i \geq 0: \mathcal{T}[f][s[i..]/s])$	f a TL formula
$\mathcal{T}[\circ f]$	$= \mathcal{T}[f][s[1..]/s]$	f a TL formula
$\mathcal{T}[f1 \mathcal{U} f2]$	$= (\exists i: i \geq 0: (\mathcal{T}[f2][s[i..]/s]) \wedge$ $(\forall j: 0 \leq j < i:$ $\mathcal{T}[f1][s[j..]/s]))$	$f1$ and $f2$ TL formulas
$\mathcal{R}[k]$	$= k$	k a constant in V
$\mathcal{R}[c]$	$= s.0(c)$	c a variable in $\{c_1, \dots, c_k\}$
$\mathcal{R}[F(t_1, \dots, t_n)]$	$= F(\mathcal{R}[t_1], \dots, \mathcal{R}[t_n])$	$F \in Fns;$ $t_1, \dots, t_n$ TL terms
$\mathcal{R}[\circ t]$	$= \mathcal{R}[t][s[1..]/s]$	t a TL term

$$\begin{aligned}
&= (\forall i: i \geq 0: ((s.0(c). \mathcal{R}[[0]])[s[1 \dots]]/s) = \mathcal{R}[[k]](s[i..]/s)) \\
&= (\forall i: i \geq 0: ((s.0(c).0)[s[1 \dots]]/s) = \mathcal{R}[[k]](s[i..]/s)) \\
&= (\forall i: i \geq 0: (s[1..].0(c).0 = \mathcal{R}[[k]](s[i..]/s)) \\
&= (\forall i: i \geq 0: (s.1(c).0 = \mathcal{R}[[k]](s[i..]/s)) \\
&= (\forall i: i \geq 0: (s.1(c).0 = k)(s[i..]/s)) \\
&= (\forall i: i \geq 0: s[i..].1(c).0 = k) \\
&= (\forall i: i \geq 0: s.(i+1)(c).0 = k).
\end{aligned}$$

Therefore, for any state-sequence σ ,

$$\sigma \models \Box(\circ c.0 = k) \text{ iff } [s \mapsto \sigma] \models (\forall i: i \geq 0: s.(i+1)(c).0 = k).$$

6.2.3 Correctness of the Mapping

We prove that $\mathcal{T}$ is a correct semantics-preserving transformation: for any state-sequence σ and TL formula f_{TL} , $\sigma \models f_{TL}$ iff $[s \mapsto \sigma] \models \mathcal{T}[[f_{TL}]]$. The proof is based on Definitions 5.1.2 and 5.1.3 of Chapter 5—value and validity of first-order terms and formulas, respectively—as well as on the corresponding TL Definitions 6.1.2 and 6.1.3 of Section 6.1.2.

Lemma 6.2.1 Let t_{TL} be any TL term containing variables from $\{c_1, \dots, c_k\}$ (only), and let σ be any state-sequence. Then $t_{TL}|_\sigma = \mathcal{R}[[t_{TL}]]|_{[s \mapsto \sigma]}$.

Proof: The proof proceeds by structural induction on t_{TL} .

Base Cases:

1. $t_{TL} = k$, k a constant.

$$k|_\sigma = k.$$

$$\mathcal{R}[[k]]|_{[s \mapsto \sigma]} = k|_{[s \mapsto \sigma]} = k.$$

2. $t_{TL} = c$, c a variable.

$$c|_{\sigma} = \sigma.0(c).$$

$$\mathcal{R}[\![c]\!]|_{[s \mapsto \sigma]} = s.0(c)|_{[s \mapsto \sigma]} = \sigma.0(c).$$

Induction:

3. $t_{TL} = F(t_1, \dots, t_n)$, F a function, $t_1, \dots, t_n$ terms.

By the induction hypothesis, $t_i|_{\sigma} = \mathcal{R}[\![t_i]\!]|_{[s \mapsto \sigma]}$, for all $1 \leq i \leq n$.

$$F(t_1, \dots, t_n)|_{\sigma} = F(t_1|_{\sigma}, \dots, t_n|_{\sigma}).$$

$$\begin{aligned} & \mathcal{R}[\![F(t_1, \dots, t_n)]\!]|_{[s \mapsto \sigma]} \\ &= F(\mathcal{R}[\![t_1]\!], \dots, \mathcal{R}[\![t_n]\!])|_{[s \mapsto \sigma]} \\ &= F(\mathcal{R}[\![t_1]\!]|_{[s \mapsto \sigma]}, \dots, \mathcal{R}[\![t_n]\!]|_{[s \mapsto \sigma]}) \\ &= F(t_1|_{\sigma}, \dots, t_n|_{\sigma}). \quad (\text{by the induction hypothesis}) \end{aligned}$$

4. $t_{TL} = \circ t$, t a term.

By the induction hypothesis, $t|_{\sigma} = \mathcal{R}[\![t]\!]|_{[s \mapsto \sigma]}$.

$$\circ t|_{\sigma} = t|_{\sigma[1..]}.$$

$$\begin{aligned} & \mathcal{R}[\![\circ t]\!]|_{[s \mapsto \sigma]} \\ &= \mathcal{R}[\![t]\!][s[1..]/s]|_{[s \mapsto \sigma]} \\ &= (\mathcal{R}[\![t]\!]|_{[s \mapsto \sigma]})[\sigma[1..]/\sigma] \\ &= (t|_{\sigma})[\sigma[1..]/\sigma] \quad (\text{by the induction hypothesis}) \\ &= t|_{\sigma[1..]}. \end{aligned}$$

⊗

Theorem 6.2.2 For any TL formula f_{TL} and state-sequence σ , $\sigma \models f_{TL}$ iff $[s \mapsto \sigma] \models \mathcal{T}[\![f_{TL}]\!]$.

Proof: The proof proceeds by structural induction on f_{TL} .

Base Case:

1. $f_{TL} = G(t_1, \dots, t_n)$, G a predicate, $t_1, \dots, t_n$ terms.

By Lemma 6.2.1, $t_i|_\sigma = \mathcal{R}[[t_i]]|_{[s \mapsto \sigma]}$, for all $1 \leq i \leq n$.

$$\sigma \models G(t_1, \dots, t_n) \text{ iff } G(t_1|_\sigma, \dots, t_n|_\sigma).$$

$$[s \mapsto \sigma] \models \mathcal{T}[[G(t_1, \dots, t_n)]]$$

$$\text{iff } [s \mapsto \sigma] \models G(\mathcal{R}[[t_1]], \dots, \mathcal{R}[[t_n]])$$

$$\text{iff } G(\mathcal{R}[[t_1]]|_{[s \mapsto \sigma]}, \dots, \mathcal{R}[[t_n]]|_{[s \mapsto \sigma]})$$

$$\text{iff } G(t_1|_\sigma, \dots, t_n|_\sigma). \quad (\text{by Lemma 6.2.1})$$

Induction:

2. $f_{TL} = f1 \vee f2$, $f1$ and $f2$ formulas.

By the induction hypothesis, $\sigma \models f1$ iff $[s \mapsto \sigma] \models \mathcal{T}[[f1]]$ and

$$\sigma \models f2 \text{ iff } [s \mapsto \sigma] \models \mathcal{T}[[f2]].$$

$$\sigma \models f1 \vee f2 \text{ iff } \sigma \models f1 \text{ or } \sigma \models f2.$$

$$[s \mapsto \sigma] \models \mathcal{T}[[f1 \vee f2]]$$

$$\text{iff } [s \mapsto \sigma] \models \mathcal{T}[[f1]] \vee \mathcal{T}[[f2]]$$

$$\text{iff } [s \mapsto \sigma] \models \mathcal{T}[[f1]] \text{ or } [s \mapsto \sigma] \models \mathcal{T}[[f2]]$$

$$\text{iff } \sigma \models f1 \text{ or } \sigma \models f2. \quad (\text{by the induction hypothesis})$$

3. $f_{TL} = \neg f$, f a formula.

By the induction hypothesis, $\sigma \models f$ iff $[s \mapsto \sigma] \models \mathcal{T}[[f]]$.

$$\sigma \models \neg f \text{ iff not } \sigma \models f.$$

$$[s \mapsto \sigma] \models \mathcal{T}[[\neg f]]$$

$$\text{iff } [s \mapsto \sigma] \models \neg \mathcal{T}[[f]]$$

$$\text{iff not } [s \mapsto \sigma] \models \mathcal{T}[[f]]$$

$$\text{iff not } \sigma \models f. \quad (\text{by the induction hypothesis})$$

4. $f_{TL} = (\exists x: f)$, f a formula.

By the induction hypothesis, $\sigma \models f[k/x]$ iff $[s \mapsto \sigma] \models \mathcal{T}[\![f[k/x]]\!]$, for any constant k .

$\sigma \models (\exists x: f)$ iff there exists a k such that $\sigma \models f[k/x]$.

$[s \mapsto \sigma] \models \mathcal{T}[\![\exists x: f]\!]$

iff $[s \mapsto \sigma] \models (\exists k: \mathcal{T}[\![f[k/x]]\!])$

iff there exists an l such that $[s \mapsto \sigma] \models (\mathcal{T}[\![f[k/x]]\!])[l/k]$

iff there exists an l such that $[s \mapsto \sigma] \models \mathcal{T}[\![f[l/x]]\!]$

iff there exists an l such that $\sigma \models f[l/x]$ (by the induction hypothesis)

iff there exists a k such that $\sigma \models f[k/x]$.⁴

5. $f_{TL} = \Box f$, f a formula.

By the induction hypothesis, $\sigma \models f$ iff $[s \mapsto \sigma] \models \mathcal{T}[\![f]\!]$, for any state-sequence σ .

$\sigma \models \Box f$ iff for all $i \geq 0$, $\sigma[i..] \models f$.

$[s \mapsto \sigma] \models \mathcal{T}[\![\Box f]\!]$

iff $[s \mapsto \sigma] \models (\forall i: i \geq 0: \mathcal{T}[\![f]\!][s[i..]/s])$

iff for all $i \geq 0$, $[s \mapsto \sigma] \models (\mathcal{T}[\![f]\!][s[i..]/s])$

iff for all $i \geq 0$, $[s \mapsto \sigma[i..]] \models \mathcal{T}[\![f]\!]$

iff for all $i \geq 0$, $\sigma[i..] \models f$. (by the induction hypothesis)

6. $f_{TL} = \circ f$, f a formula.

By the induction hypothesis, $\sigma \models f$ iff $[s \mapsto \sigma] \models \mathcal{T}[\![f]\!]$, for any state-sequence σ .

$\sigma \models \circ f$ iff $\sigma[1..] \models f$.

⁴In the proof for this case, bound variable k is renamed as l but then must be renamed back to k . In subsequent cases, such circular renaming is omitted.

$$\begin{aligned}
[s \mapsto \sigma] &\models \mathcal{T}[\circ f] \\
\text{iff } [s \mapsto \sigma] &\models \mathcal{T}[f][s[1..]/s] \\
\text{iff } [s \mapsto \sigma[1..]] &\models \mathcal{T}[f] \\
\text{iff } \sigma[1..] &\models f. \quad (\text{by the induction hypothesis})
\end{aligned}$$

7. $f_{TL} = f1 \mathcal{U} f2$, $f1$ and $f2$ formulas.

By the induction hypothesis, $\sigma \models f1$ iff $[s \mapsto \sigma] \models \mathcal{T}[f1]$ and $\sigma \models f2$ iff $[s \mapsto \sigma] \models \mathcal{T}[f2]$, for any state-sequence σ .

$\sigma \models f1 \mathcal{U} f2$ iff there exists an $i \geq 0$ such that $\sigma[i..] \models f2$ and
for all j , $0 \leq j < i$, $\sigma[j..] \models f1$.

$$\begin{aligned}
[s \mapsto \sigma] &\models \mathcal{T}[f1 \mathcal{U} f2] \\
\text{iff } [s \mapsto \sigma] &\models (\exists i: i \geq 0: (\mathcal{T}[f2][s[i..]/s]) \wedge \\
&\quad (\forall j: 0 \leq j < i: \mathcal{T}[f1][s[j..]/s])) \\
\text{iff there exists an } i &\geq 0 \text{ such that} \\
[s \mapsto \sigma] &\models (\mathcal{T}[f2][s[i..]/s] \wedge (\forall j: 0 \leq j < i: \mathcal{T}[f1][s[j..]/s])) \\
\text{iff there exists an } i &\geq 0 \text{ such that } [s \mapsto \sigma] \models \mathcal{T}[f2][s[i..]/s] \text{ and} \\
[s \mapsto \sigma] &\models (\forall j: 0 \leq j < i: \mathcal{T}[f1][s[j..]/s]) \\
\text{iff there exists an } i &\geq 0 \text{ such that } [s \mapsto \sigma] \models \mathcal{T}[f2][s[i..]/s] \text{ and} \\
\text{for all } j, 0 \leq j < i, &[s \mapsto \sigma] \models \mathcal{T}[f1][s[j..]/s] \\
\text{iff there exists an } i &\geq 0 \text{ such that } [s \mapsto \sigma[i..]] \models \mathcal{T}[f2] \text{ and} \\
\text{for all } j, 0 \leq j < i, &[s \mapsto \sigma[j..]] \models \mathcal{T}[f1] \\
\text{iff there exists an } i &\geq 0 \text{ such that } \sigma[i..] \models f2 \text{ and} \\
\text{for all } j, 0 \leq j < i, &\sigma[j..] \models f1. \quad (\text{by the I.H.})
\end{aligned}$$

⊗

6.3 Revisions to Formula $CM(f)$

Let f_{TL} be a formula in TL and f_{EPL} a formula in EPL. Using transformation function $\mathcal{T}$, formulas f_{TL} and f_{EPL} are equivalent if and only if $\mathcal{T}[\![f_{TL}]\!] \equiv f_{EPL}$.

Recall Definition 5.4.2 of $CM(f)$:

$$\begin{aligned}
 CM(f) \equiv & \\
 & (\exists \langle \bar{t}^0, \bar{t}^1, \dots, \bar{t}^n \rangle : \\
 & \quad \bar{t}^0 = [\langle \rangle, \dots, \langle \rangle] \wedge \\
 & \quad \bar{t}^n = [c_1, \dots, c_k] \wedge \\
 & \quad (\forall i: 0 \leq i \leq n: f[\bar{t}^i / \bar{c}]) \wedge \\
 & \quad (\forall i: 0 \leq i < n: \\
 & \quad \quad (\exists j, v: 1 \leq j \leq k, v \in V: \\
 & \quad \quad \quad \bar{t}^{i+1} = \bar{t}^i [t_j^i \cdot \langle v \rangle / t_j^i]))))
 \end{aligned}$$

Our goal is find the minimal subset of TL powerful enough to express a formula equivalent to $CM(f)$. Therefore, we will need to construct some TL formula $CM_{TL}(f)$ such that $\mathcal{T}[\![CM_{TL}(f)]\!] \equiv CM(f)$. When transformation $\mathcal{T}$ is applied, however, all TL formulas are mapped to EPL formulas with one free variable, state-sequence s . Thus, to establish equivalence between $CM_{TL}(f)$ and $CM(f)$, $CM(f)$ must be rewritten so that its only free variable is s .

Suppose, for the moment, that state-sequence variables can have finite length. We then rewrite $CM(f)$, letting s represent bound sequence variable $\langle \bar{t}_0, \bar{t}_1, \dots, \bar{t}_n \rangle$, as follows.

Definition 6.3.1 (Revised Non-recursive Definition: $CM_{Rev}(f)$)

$$\begin{aligned}
 CM_{Rev}(f) \equiv & \\
 & s.0 = [\langle \rangle, \dots, \langle \rangle] \wedge \\
 & (\forall i: 0 \leq i < |s|: f[s.i / \bar{c}]) \wedge
 \end{aligned}$$

$$\begin{aligned}
& (\forall i: 0 \leq i < |s| - 1: \\
& \quad (\exists j, v: 1 \leq j \leq k, v \in V: \\
& \quad \quad s.(i+1) = s.i[(s.i)_j \cdot \langle v \rangle / (s.i)_j]))
\end{aligned}$$

By this definition, an assignment α is a computational model of f —equivalently, $\alpha \models CM(f)$ —if and only if there exists some finite state-sequence σ such that $\sigma.(|\sigma|-1) = \alpha$ and $[s \mapsto \sigma] \models CM_{Rev}(f)$.

The following definition and theorem show that, in fact, finite-length state-sequence variables are unnecessary—an equivalent formula can be written using only an infinite state-sequence variable s .

Definition 6.3.2 (Final EPL Definition: $CM_{EPL}(f)$)

$$\begin{aligned}
CM_{EPL}(f) \equiv & \\
& s.0 = [\langle \rangle, \dots, \langle \rangle] \wedge \\
& (\forall i: i \geq 0: f[s.i/\bar{c}]) \wedge \\
& (\forall i: i \geq 0: \\
& \quad s.(i+1) = s.i \vee \\
& \quad (\exists j, v: 1 \leq j \leq k, v \in V: \\
& \quad \quad s.(i+1) = s.i[(s.i)_j \cdot \langle v \rangle / (s.i)_j]))
\end{aligned}$$

Theorem 6.3.3 Let f be a formula of first-order trace logic and α an assignment. There exists a finite state-sequence σ such that $\sigma.(|\sigma|-1) = \alpha$ and $[s \mapsto \sigma] \models CM_{Rev}(f)$ if and only if there exists an infinite state-sequence σ' and index $i \geq 0$ such that $\sigma'.i = \alpha$ and $[s \mapsto \sigma'] \models CM_{EPL}(f)$.

Proof: ($\Rightarrow$) Suppose σ exists. Let σ' be constructed from σ by indefinitely repeating σ 's last state: $\sigma' = \langle \sigma.0, \sigma.1, \dots, \sigma.(|\sigma|-1), \sigma.(|\sigma|-1), \sigma.(|\sigma|-1), \dots \rangle$. (Recall that we similarly converted finite state-sequences to infinite state-sequences for Definition 6.1.1.) Since $\sigma'.(|\sigma|-1) = \sigma.(|\sigma|-1)$ and $\sigma.(|\sigma|-1) = \alpha$, by transitivity $\sigma'.(|\sigma|-1) = \alpha$. Furthermore, by $[s \mapsto \sigma] \models CM_{Rev}(f)$, we know

that $[s \mapsto \sigma'] \models CM_{EPL}(f)$, since for all $i \geq |\sigma| - 1$, $\sigma'.i = \sigma'.(i+1)$.

($\Leftarrow$) Suppose σ' and i exist. By assumption, $\sigma'.i = \alpha$ for some $i \geq 0$. Let $\sigma'[..i]$ denote sequence σ' through element $\sigma'.i$ (hence $\sigma'[..i]$ is finite), and let σ be constructed from $\sigma'[..i]$ by removing each state that duplicates its immediate predecessor in the sequence. We know $\sigma.(|\sigma|-1) = \sigma'.i$, so, by definition, $\sigma.(|\sigma|-1) = \alpha$. Furthermore, by $[s \mapsto \sigma'] \models CM_{EPL}(f)$, we know that $[s \mapsto \sigma] \models CM_{Rev}(f)$, since σ is formed from σ' by truncation and removal of repeated states. $\otimes$

Using the original $CM(f)$, an assignment α is a computational model of a formula f if and only if $\alpha \models CM(f)$; using $CM_{EPL}(f)$, α is a computational model of f if and only if there exists an infinite state-sequence σ and index $i \geq 0$ such that $\sigma.i = \alpha$ and $[s \mapsto \sigma] \models CM_{EPL}(f)$. Since recognition of computational models differs for $CM(f)$ and $CM_{EPL}(f)$, we must formally prove that the two formulas do, in fact, perform the identical function with respect to a proof system.

Recall Corollary 5.3.5: given a network N with precise specification $S1$, a specification $S2$ is valid for N if and only if

$$CompModels(S1) \subseteq CompModels(S2). \quad (6.1)$$

Therefore, we are interested in verifying formulas of the form

$$CM(S1) \Rightarrow CM(S2), \quad (6.2)$$

since (6.2) implies (6.1). We show that formula (6.2) holds when and only when

$$CM_{EPL}(S1) \Rightarrow CM_{EPL}(S2) \quad (6.3)$$

holds. Hence, $CM_{EPL}(f)$ and $CM(f)$ are equivalent with respect to trace-based proof systems.

Theorem 6.3.4 For any formulas $f1$ and $f2$ of first-order trace logic,

$$CM(f1) \Rightarrow CM(f2) \text{ iff } CM_{EPL}(f1) \Rightarrow CM_{EPL}(f2).$$

Proof: By definition, $CM(f1) \Rightarrow CM(f2)$ if and only if, for all assignments α ,

$$\alpha \models CM(f1) \Rightarrow \alpha \models CM(f2).$$

Similarly, $CM_{EPL}(f1) \Rightarrow CM_{EPL}(f2)$ if and only if, for all state-sequences σ ,

$$[s \mapsto \sigma] \models CM_{EPL}(f1) \Rightarrow [s \mapsto \sigma] \models CM_{EPL}(f2).$$

($\Rightarrow$) If $CM(f1) \Rightarrow CM(f2)$ then $CM_{EPL}(f1) \Rightarrow CM_{EPL}(f2)$:

Suppose $\alpha \models CM(f1) \Rightarrow \alpha \models CM(f2)$, for all assignments α . We must show, for any state-sequence σ , that $[s \mapsto \sigma] \models CM_{EPL}(f1) \Rightarrow [s \mapsto \sigma] \models CM_{EPL}(f2)$. Consider an arbitrary state-sequence σ and suppose $[s \mapsto \sigma] \models CM_{EPL}(f1)$. We need to show $[s \mapsto \sigma] \models CM_{EPL}(f2)$. The only conjunct in $CM_{EPL}(f)$ with a reference to f is $(\forall i: i \geq 0: f[s.i/\bar{c}])$, so, by $[s \mapsto \sigma] \models CM_{EPL}(f1)$, it suffices to show $[s \mapsto \sigma] \models (\forall i: i \geq 0: f2[s.i/\bar{c}])$. Equivalently, we must show that for all $i \geq 0$, $\sigma.i \models f2$.

Consider an arbitrary $\sigma.i$. By $[s \mapsto \sigma] \models CM_{EPL}(f1)$ and Theorem 6.3.3, there exists a finite state-sequence σ' such that $\sigma'.(|\sigma'| - 1) = \sigma.i$ and $[s \mapsto \sigma'] \models CM_{Rev}(f1)$. Hence, by Definition 6.3.1 of $CM_{Rev}(f)$, $\sigma.i \models CM(f1)$. By assumption, then, $\sigma.i \models CM(f2)$, and consequently, by Definition 5.4.2 of $CM(f)$, $\sigma.i \models f2$.

($\Leftarrow$) If $CM_{EPL}(f1) \Rightarrow CM_{EPL}(f2)$ then $CM(f1) \Rightarrow CM(f2)$:

Suppose $[s \mapsto \sigma] \models CM_{EPL}(f1) \Rightarrow [s \mapsto \sigma] \models CM_{EPL}(f2)$, for all state-sequences σ . We must show, for any assignment α , that $\alpha \models CM(f1) \Rightarrow \alpha \models CM(f2)$. Consider an arbitrary assignment α and suppose that $\alpha \models CM(f1)$. By Definition 6.3.1 of $CM_{Rev}(f)$, there exists a finite state-sequence σ such that $\sigma.(|\sigma| - 1) = \alpha$ and $[s \mapsto \sigma] \models CM_{Rev}(f1)$. Then, by Theorem 6.3.3, there exists an infinite state-sequence σ' and index $i \geq 0$ such that $\sigma'.i = \alpha$ and $[s \mapsto \sigma'] \models CM_{EPL}(f1)$. Since $[s \mapsto \sigma'] \models CM_{EPL}(f1)$, by assumption, $[s \mapsto \sigma'] \models CM_{EPL}(f2)$. Then, by $\sigma'.i = \alpha$ and Theorem 6.3.3, there exists

a finite state-sequence σ'' such that $\sigma''.(|\sigma''|-1) = \alpha$ and $[s \mapsto \sigma''] \models CM_{Rev}(f2)$.
 Consequently, by Definition 6.3.1, $\alpha \models CM(f2)$. $\otimes$

We have defined an EPL formula, $CM_{EPL}(f)$, such that for any first-order trace formulas $f1$ and $f2$,

$$CompModels(f1) \subseteq CompModels(f2) \text{ iff } CM_{EPL}(f1) \Rightarrow CM_{EPL}(f2).$$

Therefore, if a language is strong enough to express a formula equivalent to $CM_{EPL}(f)$, the language is strong enough to be used as the basis of a relatively complete trace-based proof system. In the next chapter, we define a subset of temporal logic that has the necessary and sufficient expressive power to encode $CM_{EPL}(f)$.

Chapter 7

Expressiveness Bounds for Relative Completeness

In Chapter 6, we showed that any relatively complete trace logic must be expressive enough to encode a formula equivalent to formula $CM_{EPL}(f)$ of Definition 6.3.2. We defined a transformation function $\mathcal{T}$ from formulas in Temporal Logic to formulas in Extended Predicate Logic; since $CM_{EPL}(f)$ is a formula in EPL, using $\mathcal{T}$ we can prove that a given subset of TL is necessary and sufficient to express a formula equivalent to $CM_{EPL}(f)$. Such a subset, therefore, has the exact expressive power required for a relatively complete trace logic.

To prove the necessity and sufficiency of a TL subset, we must determine exactly the possible subsets of TL. First, we introduce three new temporal operators—strictly weaker versions of *Always*, *Next*, and *Until*—that enrich the subset structure of TL and thereby “tighten” the attainable expressiveness bounds. The resulting subsets do not form a linear order since temporal operators $\Box$, $\circ$, and $\mathcal{U}$ are disjoint—no operator can be encoded by the other two (likewise for the three new operators). The subsets do, however, form a partial order, and this hierarchy is used to prove that the TL subset consisting of first-order trace logic with a version of the *Always* operator is an upper and a lower

bound on the power needed to express $CM_{EPL}(f)$. This result is then applied in the description of a trace-based proof system that is minimally complete with respect to our hierarchy of TL subsets.

7.1 Temporal Operators over Non-Temporal Formulas

In Chapter 4, an *Always* operator was used to convert STL specifications to being on entire computations (rather than single states). The implication in Theorem 4.4.3 includes formulas of the form $\Box S$, where S is a formula of first-order trace logic. This *Always* operator—over first-order formulas only—is a restricted version of the $\Box$ operator defined in Chapter 6. In general, by permitting nested temporal operators, quite complex properties can be expressed, e.g. $\Box(f1 \Rightarrow (\Box f2 \vee \circ f3))$, which asserts that whenever $f1$ is valid, either $f2$ is valid thereafter or $f3$ is valid at the next point in time. This property cannot be expressed using temporal operators only over first-order formulas.

Since we are interested in using some TL subset to express a formula equivalent to $CM_{EPL}(f)$, and in particular because all f 's under consideration are non-temporal (i.e. contain no temporal operators), we introduce an additional set of temporal operators that are restricted to operate only over non-temporal terms and formulas. These operators are strictly weaker than their fully temporal counterparts; the new operators thus increase the number of possible subsets of TL.

The syntax of TL is extended to include the new operators as follows; note that *Trace-Formula* and *Trace-Term* refer to the syntactic definition of first-order trace logic given in Table 3.1.

1. *TL-formula* $::= \bar{\Box} f$, f a *Trace-Formula*;
 $\bar{\Box}$ is the *Restricted Always* operator.
2. *TL-formula* $::= \bar{\circ} f$, f a *Trace-Formula*;
TL-term $::= \bar{\circ} t$, t a *Trace-Term*;
 $\bar{\circ}$ is the *Restricted Next* operator.
3. *TL-formula* $::= f_1 \bar{\mathcal{U}} f_2$, f_1 and f_2 *Trace-Formulas*;
 $\bar{\mathcal{U}}$ is the *Restricted Until* operator.

A *Restricted Eventually* operator is not included, since $\Diamond f$ is simply a syntactic abbreviation for $\neg \bar{\Box} \neg f$. We sometimes refer to $\Box$, $\circ$, and $\mathcal{U}$ as *Unrestricted Always*, *Unrestricted Next*, and *Unrestricted Until*, respectively, to distinguish them from the corresponding restricted operators.

Although Definitions 6.1.2 and 6.1.3 for validity of TL formulas with respect to state-sequences can be used directly for the new operators—the operators are subsumed by their unrestricted counterparts—we instead make explicit the simpler semantics of $\bar{\Box}$, $\bar{\circ}$, and $\bar{\mathcal{U}}$. Let σ be any state-sequence. Recall, for all $i \geq 0$, that $\sigma.i$ is an assignment—a model of a first-order trace formula. The following definitions are derived from Definitions 6.1.2 and 6.1.3, and from the fact that $\bar{\Box}$, $\bar{\circ}$, and $\bar{\mathcal{U}}$ operate only over non-temporal terms and formulas.

1. $\bar{\circ} t|_{\sigma} = t|_{\sigma.1}$, for a trace term t , where $t|_{\sigma.1}$ is as defined in Definition 5.1.2 of the value of first-order trace terms with respect to assignments;
2. $\sigma \models \bar{\Box} f$ iff for all $i \geq 0$, $\sigma.i \models f$, for a trace formula f , where $\sigma.i \models f$ is as defined in Definition 5.1.3 of validity of first-order trace formulas with respect to assignments;
3. $\sigma \models \bar{\circ} f$ iff $\sigma.1 \models f$, for a trace formula f , where $\sigma.1 \models f$ is as defined in Definition 5.1.3;

4. $\sigma \models f1 \bar{U} f2$ iff there exists an $i \geq 0$ such that $\sigma.i \models f2$ and for all j , $0 \leq j < i$, $\sigma.j \models f1$, for trace formulas $f1$ and $f2$, where $\sigma.i \models f$ and $\sigma.j \models f$ are as defined in Definition 5.1.3.

The transformation functions $\mathcal{T}$ and $\mathcal{R}$ defined in Table 6.3 need not be extended to accommodate formulas of the form $\bar{\square} f$, $\bar{o} f$, or $f1 \bar{U} f2$, or terms of the form $\bar{o} t$. The definitions of $\mathcal{T}$ and $\mathcal{R}$ for these formulas and terms are identical to $\mathcal{T}[\square f]$, $\mathcal{T}[\circ f]$, $\mathcal{T}[f1 \mathcal{U} f2]$, and $\mathcal{R}[\circ t]$, respectively.

7.2 Subsets of Temporal Logic

TL consists of first-order trace logic with the addition of six temporal operators: $\square$, $\circ$, $\mathcal{U}$, $\bar{\square}$, $\bar{o}$, and $\bar{U}$. A subset of TL is constructed by choosing any subset of the temporal operators; for example, trace logic with $\square$ and $\bar{o}$ is a (strict) subset of TL. We want to find the TL subset that is both necessary and sufficient to express a formula equivalent to $CM_{EPL}(f)$.

The subsets of TL form a rather complicated hierarchy. Although operator $\square$ subsumes operator $\bar{\square}$, $\circ$ subsumes $\bar{o}$, and $\mathcal{U}$ subsumes $\bar{U}$, operators $\square$, $\circ$, and $\mathcal{U}$ are disjoint, as are $\bar{\square}$, $\bar{o}$, and $\bar{U}$. Recall, however, the definition of transformation function $\mathcal{T}$ on formulas of the form $f1 \mathcal{U} f2$:

$$\mathcal{T}[f1 \mathcal{U} f2] = (\exists i: i \geq 0: (\mathcal{T}[f2][s[i..]/s]) \wedge (\forall j: 0 \leq j < i: \mathcal{T}[f1][s[j..]/s]))$$

Definition 6.3.2 of $CM_{EPL}(f)$ contains no eventuality components of the form $(\exists i: i \geq 0: f'(s.i))$. Thus, it is clear that operator $\mathcal{U}$ will not be of use in any TL formula f_{TL} such that $\mathcal{T}[f_{TL}] \equiv CM_{EPL}(f)$. Similarly, there is no need to consider operator $\bar{U}$.

The TL subsets of interest therefore correspond to the subsets of $\{\square, \circ, \bar{\square}, \bar{o}\}$. These subsets—and their interrelationship—are described by the partial order

Figure 7.1: Subsets of temporal logic

pictured in Figure 7.1. Remember that restricted operators are subsumed by their unrestricted counterparts, so, for example, the subset of TL corresponding to $\{\bar{\square}, \square\}$ is equivalent to the subset corresponding to $\{\square\}$.

7.3 Necessity and Sufficiency of the *Unrestricted Always Operator*

Subset 4 of Figure 7.1, first-order trace logic with an *Unrestricted Always* operator, has the necessary and sufficient expressive power to encode $CM_{EPL}(f)$. We prove this subset to be sufficient by using trace logic with *Unrestricted Always* operators (only) to write a formula equivalent to $CM_{EPL}(f)$. We then prove that the subset is an absolute lower bound: each subset lower than or incomparable to subset 4 in the hierarchy of Figure 7.1 is not expressive enough to encode $CM_{EPL}(f)$. This is proven by showing that no formula equivalent to $CM_{EPL}(f)$ can be expressed in subset 8—first-order trace logic with a *Restricted Always*

Figure 7.2: Expressiveness bounds

and an *Unrestricted Next* operator. Consequently, all subsets except 4, 7 and 9 are insufficient. The resulting division of the subset hierarchy is illustrated in Figure 7.2.

7.3.1 Sufficiency

Let $TL_{\square}$ be the language of first-order trace logic with an *Unrestricted Always* operator—subset 4 of Figures 7.1 and 7.2. To prove that $TL_{\square}$ is sufficiently powerful to encode $CM_{EPL}(f)$, we exhibit a formula $CM_{\square}(f)$ in $TL_{\square}$ such that $\mathcal{T}[CM_{\square}(f)] \equiv CM_{EPL}(f)$. $CM_{\square}(f)$ is derived from the temporal ordering and prefix axioms of Chapter 4, as follows.

Definition 7.3.1 (TL Definition: $CM_{\square}(f)$)

$$\begin{aligned}
CM_{\square}(f) \equiv & \\
& \square f \wedge \\
& (\forall m, n, x, y: 1 \leq m \leq k, 1 \leq n \leq k, x \geq 1, y \geq 0, m \neq n \vee x \neq y: \\
& \quad (\square(|c_m| \geq x \equiv |c_n| \geq y)) \Rightarrow \square(|c_m| < x \wedge |c_n| < y)) \wedge \\
& (\forall m, x, v: 1 \leq m \leq k, x \geq 0, v \in V: \\
& \quad \square((c_m.x = v) \Rightarrow \square(c_m.x = v)))^1
\end{aligned}$$

The second conjunct of $CM_{\square}(f)$ corresponds directly to Definition 4.2.2 of axiom *ORDERING*. The third conjunct of $CM_{\square}(f)$ is a restatement of axiom *PREFIX* (Definition 4.3.2), eliminating the use of a *Next* operator.

We must prove $\mathcal{T}[[CM_{\square}(f)]] \equiv CM_{EPL}(f)$. Using the definition of $\mathcal{T}$,

$$\begin{aligned}
\mathcal{T}[[CM_{\square}(f)]] \equiv & \\
& (\forall i: i \geq 0: f[s.i(\bar{c})/\bar{c}])^2 \wedge \\
& (\forall m, n, x, y: 1 \leq m \leq k, 1 \leq n \leq k, x \geq 1, y \geq 0, m \neq n \vee x \neq y: \\
& \quad (\forall i: i \geq 0: |s.i(c_m)| \geq x \equiv |s.i(c_n)| \geq y) \Rightarrow \\
& \quad (\forall i: i \geq 0: |s.i(c_m)| < x \wedge |s.i(c_n)| < y)) \wedge \\
& (\forall m, x, v: 1 \leq m \leq k, x \geq 0, v \in V: \\
& \quad (\forall i: i \geq 0: (s.i(c_m).x = v) \Rightarrow (\forall j: j \geq 0: s.(i+j)(c_m).x = v))).
\end{aligned}$$

¹Quantifying over channel-trace variables—which we effectively do here by quantifying over channel-name indices—is forbidden by the syntax of TL (Table 6.1). In this case, however, quantification is being used only for notational brevity. An equivalent legal formula can be written by expanding the quantified clauses to a finite number of conjuncts, one conjunct for each channel or channel pair.

²Note that for any non-temporal formula f , $\mathcal{T}[[f]] = f[s.0(c_1), \dots, s.0(c_k)/c_1, \dots, c_k]$. We use $f[s.i(\bar{c})/\bar{c}]$ as an abbreviation for $f[s.i(c_1), \dots, s.i(c_k)/c_1, \dots, c_k]$.

Recall Definition 6.3.2 of $CM_{EPL}(f)$:

$$\begin{aligned}
 CM_{EPL}(f) \equiv & \\
 & s.0 = [\langle \rangle, \dots, \langle \rangle] \wedge \\
 & (\forall i: i \geq 0: f[s.i/\bar{c}]) \wedge \\
 & (\forall i: i \geq 0: \\
 & \quad s.(i+1) = s.i \vee \\
 & \quad (\exists j, v: 1 \leq j \leq k, v \in V: \\
 & \quad \quad s.(i+1) = s.i[(s.i)_j \cdot \langle v \rangle / (s.i)_j]))^3
 \end{aligned}$$

The first conjunct of $\mathcal{T}\llbracket CM_{\square}(f) \rrbracket$ is equivalent to the second conjunct of $CM_{EPL}(f)$. Let $ORDERING'$ denote the second conjunct of $\mathcal{T}\llbracket CM_{\square}(f) \rrbracket$:

$$\begin{aligned}
 ORDERING' \equiv & \\
 & (\forall m, n, x, y: 1 \leq m \leq k, 1 \leq n \leq k, x \geq 1, y \geq 0, m \neq n \vee x \neq y: \\
 & \quad (\forall i: i \geq 0: |s.i(c_m)| \geq x \equiv |s.i(c_n)| \geq y) \Rightarrow \\
 & \quad (\forall i: i \geq 0: |s.i(c_m)| < x \wedge |s.i(c_n)| < y))
 \end{aligned}$$

Let $PREFIX'$ denote the third conjunct of $\mathcal{T}\llbracket CM_{\square}(f) \rrbracket$:

$$\begin{aligned}
 PREFIX' \equiv & \\
 & (\forall m, x, v: 1 \leq m \leq k, x \geq 0, v \in V: \\
 & \quad (\forall i: i \geq 0: (s.i(c_m).x = v) \Rightarrow (\forall j: j \geq 0: s.(i+j)(c_m).x = v)))
 \end{aligned}$$

To show $\mathcal{T}\llbracket CM_{\square}(f) \rrbracket \equiv CM_{EPL}(f)$, then, we must prove the following theorem.

³Remember: in $CM_{EPL}(f)$, variable s is an encoding of a state-sequence rather than the state-sequence itself. Hence $(s.i)_j$ in $CM_{EPL}(f)$ is equivalent to $s.i(c_j)$ in $\mathcal{T}\llbracket CM_{\square}(f) \rrbracket$.

Theorem 7.3.2

$$ORDERING' \wedge PREFIX' \equiv$$

$$s.0 = [\langle \rangle, \dots, \langle \rangle] \wedge$$

$$(\forall i: i \geq 0:$$

$$s.(i+1) = s.i \vee$$

$$(\exists j, v: 1 \leq j \leq k, v \in V:$$

$$s.(i+1) = s.i[(s.i)_j \cdot \langle v \rangle / (s.i)_j]))$$

A formal proof of Theorem 7.3.2 is omitted; it exactly parallels the proof of Lemma 4.4.2 in Chapter 4 (the well-formedness lemma). Recall Lemma 4.4.2: for any sequence of trace-sets ρ , $\rho \models ORDERING \wedge PREFIX$ if and only if $Compress(\rho)^4$ is well-formed. By Definition 3.6.2, a sequence of trace-sets is well-formed if and only if all traces are empty in the first trace-set and all other trace-sets in the sequence extend exactly one trace of the preceding set by exactly one element. Therefore, by making the obvious correspondences— $ORDERING'$ with $ORDERING$, $PREFIX'$ with $PREFIX$, and state-sequence s with sequence of trace-sets ρ —the proof of Theorem 7.3.2 is isomorphic to the proof of Lemma 4.4.2.

We conclude that a formula equivalent to $CM_{EPL}(f)$ can be written in $TL_{\square}$, so $TL_{\square}$ has sufficient expressive power for a relatively complete trace-based proof system. A proof system based on $TL_{\square}$ is described in Section 7.5.

7.3.2 Necessity

We now show that $TL_{\square}$ is necessary, as well as sufficient, for a relatively complete proof system: any TL subset weaker than or incomparable to $TL_{\square}$ cannot be used to express a formula equivalent to $CM_{EPL}(f)$. By the hierarchy of Figure 7.1, proving the necessity of $TL_{\square}$ only requires proving that no formula equivalent

⁴Recall that $Compress(\rho)$ is the sequence obtained from ρ by eliminating each trace-set that duplicates its immediate predecessor in the sequence.

to $CM_{EPL}(f)$ can be expressed in the TL subset consisting of trace logic with a *Restricted Always* and an *Unrestricted Next* operator—subset 8 of Figures 7.1 and 7.2.

Let TL_8 be the language of first-order trace logic with operators $\bar{\square}$ and $\circ$. We show that there is no formula $CM_8(f)$ in TL_8 such that $\mathcal{T}[CM_8(f)] \equiv CM_{EPL}(f)$. To prove this, we consider the nesting depth of *Next* operators in TL_8 formulas. Since every formula has only a finite number of $\circ$'s, the maximum depth of $\circ$ nesting in a given formula is a well-defined nonnegative integer.

Definition 7.3.3 For any formula f_{TL} in TL_8 , let $\circ\text{-nesting}(f_{TL})$ denote the maximum nesting of *Next* operators in f_{TL} . $\circ\text{-nesting}(f_{TL})$ is defined inductively, on the structure of f_{TL} , as follows.

1. $f_{TL} = G(t_1, \dots, t_n)$:
 $\circ\text{-nesting}(G(t_1, \dots, t_n)) = \max(\circ\text{-nesting}(t_1), \dots, \circ\text{-nesting}(t_n)).$
2. $f_{TL} = f1 \vee f2$:
 $\circ\text{-nesting}(f1 \vee f2) = \max(\circ\text{-nesting}(f1), \circ\text{-nesting}(f2)).$
3. $f_{TL} = \neg f$:
 $\circ\text{-nesting}(\neg f) = \circ\text{-nesting}(f).$
4. $f_{TL} = (\exists x: f)$:
 $\circ\text{-nesting}((\exists x: f)) = \circ\text{-nesting}(f).$
5. $f_{TL} = \bar{\square} f$:
 $\circ\text{-nesting}(\bar{\square} f) = \circ\text{-nesting}(f).$
6. $f_{TL} = \circ f$:
 $\circ\text{-nesting}(\circ f) = \circ\text{-nesting}(f) + 1.$
7. Term $t_{TL} = k$:
 $\circ\text{-nesting}(k) = 0.$

8. Term $t_{TL} = c$:
 $\circ\text{-nesting}(c) = 0$.
9. Term $t_{TL} = F(t_1, \dots, t_n)$:
 $\circ\text{-nesting}(F(t_1, \dots, t_n)) = \max(\circ\text{-nesting}(t_1), \dots, \circ\text{-nesting}(t_n))$.
10. Term $t_{TL} = \circ t$:
 $\circ\text{-nesting}(\circ t) = \circ\text{-nesting}(t) + 1$.

Using this definition, we prove a key lemma: there is no formula f_{TL} in TL_8 such that $\mathcal{T}[\![f_{TL}]\!]$ and $CM_{EPL}(true)$ are satisfied by the same set of assignments. The final result—that there is no formula $CM_8(f)$ in TL_8 such that $\mathcal{T}[\![CM_8(f)]\!] \equiv CM_{EPL}(f)$ —follows directly from the lemma. Let CM_{EPL} be an abbreviation for $CM_{EPL}(true)$. To prove that there is no formula f_{TL} in TL_8 such that $\mathcal{T}[\![f_{TL}]\!]$ and CM_{EPL} are satisfied by the same set of assignments, we show, for every f_{TL} , that there is either an assignment that models $\mathcal{T}[\![f_{TL}]\!]$ but does not model CM_{EPL} , or there is an assignment that models CM_{EPL} but does not model $\mathcal{T}[\![f_{TL}]\!]$.

For every f_{TL} in TL_8 there is some $n \geq 0$ — n being the maximum nesting depth of *Next* operators in f_{TL} —such that beyond the n th state in any state-sequence, $\mathcal{T}[\![f_{TL}]\!]$ can refer to states only by universal quantification (resulting from *Restricted Always* operators in f_{TL}). Now, if no state-sequences satisfy $\mathcal{T}[\![f_{TL}]\!]$, or the only satisfying sequences have all repeated states beyond their n th state, then it is straightforward to construct a sequence σ such that assignment $[s \mapsto \sigma]$ models CM_{EPL} but does not model $\mathcal{T}[\![f_{TL}]\!]$. Otherwise, by constructing a sequence σ that violates *ORDERING'* or *PREFIX'* beyond the n th state, $[s \mapsto \sigma]$ does not model CM_{EPL} ; however, $[s \mapsto \sigma]$ does model $\mathcal{T}[\![f_{TL}]\!]$, as long as σ is constructed by rearranging states from a sequence known to satisfy $\mathcal{T}[\![f_{TL}]\!]$.

A rigorous proof follows.

Lemma 7.3.4 For any formula f_{TL} in TL_8 , there exists a state-sequence σ such that either

$$[s \mapsto \sigma] \models \mathcal{T}[f_{TL}] \quad \text{and} \quad [s \mapsto \sigma] \not\models CM_{EPL}^5, \text{ or}$$

$$[s \mapsto \sigma] \not\models \mathcal{T}[f_{TL}] \quad \text{and} \quad [s \mapsto \sigma] \models CM_{EPL}.$$

Proof: Consider an arbitrary f_{TL} in TL_8 and let $n = \circ\text{-nesting}(f_{TL})$. Three cases must be considered:

Case 1. There is no σ such that $[s \mapsto \sigma] \models \mathcal{T}[f_{TL}]$:

We must then show that there is some σ such that $[s \mapsto \sigma] \models CM_{EPL}$. One such σ is the state-sequence in which every channel is always mapped to the empty trace:

$$\begin{aligned} \sigma = & \langle [c_1 \mapsto \langle \rangle, c_2 \mapsto \langle \rangle, \dots, c_k \mapsto \langle \rangle], \\ & [c_1 \mapsto \langle \rangle, c_2 \mapsto \langle \rangle, \dots, c_k \mapsto \langle \rangle], \dots \rangle. \end{aligned}$$

$[s \mapsto \sigma] \not\models \mathcal{T}[f_{TL}]$, since there is no σ such that $[s \mapsto \sigma] \models \mathcal{T}[f_{TL}]$, but, by Definition 6.3.2 of $CM_{EPL}(f)$, $[s \mapsto \sigma] \models CM_{EPL}$.

Case 2. There exists a σ such that $[s \mapsto \sigma] \models \mathcal{T}[f_{TL}]$ and every such σ has only repeating states after state $\sigma.n$ (recall $n = \circ\text{-nesting}(f_{TL})$), i.e. whenever $[s \mapsto \sigma] \models \mathcal{T}[f_{TL}]$, $\sigma.m = \sigma.(m-1)$ for all $m > n$:

Consider an arbitrary σ such that $[s \mapsto \sigma] \models \mathcal{T}[f_{TL}]$. If $[s \mapsto \sigma] \not\models CM_{EPL}$, we are done. Suppose, then, that $[s \mapsto \sigma] \models CM_{EPL}$. Construct σ' from σ by extending one trace by one element between states $\sigma.n$ and $\sigma.(n+1)$:

$$\begin{aligned} \sigma' = & \langle \sigma.0, \sigma.1, \dots, \sigma.n, \\ & \sigma.n[\sigma.n(c_1) \cdot \langle v \rangle / \sigma.n(c_1)], \sigma.n[\sigma.n(c_1) \cdot \langle v \rangle / \sigma.n(c_1)], \dots \rangle, \end{aligned}$$

for an arbitrary $v \in V$. Then $[s \mapsto \sigma'] \not\models \mathcal{T}[f_{TL}]$, since $\sigma'.(n+1) \neq \sigma'.n$. But, by

⁵We use $\alpha \not\models f$ as an abbreviation for not $\alpha \models f$.

$[s \mapsto \sigma] \models CM_{EPL}$ and the definition of $CM_{EPL}(f)$, $[s \mapsto \sigma'] \models CM_{EPL}$.

Case 3. There exists a σ such that $[s \mapsto \sigma] \models \mathcal{T}[\llbracket f_{TL} \rrbracket]$ and σ has a non-repeating state after $\sigma.n$ (i.e. $\sigma.m \neq \sigma.(m-1)$ for some $m > n$):

Consider such a σ , and consider the smallest m satisfying $\sigma.m \neq \sigma.(m-1)$. By definition,

$$\sigma = \langle \sigma.0, \sigma.1, \dots, \sigma.n, \sigma.(n+1), \dots, \sigma.(m-1), \sigma.m, \dots \rangle$$

such that for all x , $n \leq x < m$, $\sigma.x = \sigma.n$, but $\sigma.m \neq \sigma.n$. If there does not exist a j and a v , $1 \leq j \leq k$, $v \in V$, such that $\sigma.m = \sigma.n[\sigma.n(c_j) \cdot \langle v \rangle / \sigma.n(c_j)]$, then $[s \mapsto \sigma] \not\models CM_{EPL}$, and we are done. Suppose, then, that $\sigma.m = \sigma.n[\sigma.n(c_j) \cdot \langle v \rangle / \sigma.n(c_j)]$, for some $1 \leq j \leq k$ and $v \in V$. Let σ' be constructed from σ by repeating state $\sigma.n$ and inserting a copy of state $\sigma.m$ between the repetition:

$$\sigma' = \langle \sigma.0, \sigma.1, \dots, \sigma.n, \sigma.m, \sigma.n, \sigma.(n+1), \sigma.(n+2), \dots \rangle.$$

Note that $\sigma'.(n+1) = \sigma.m$ and $\sigma'.(n+2) = \sigma.n$. Let x be the index of the last element in $\sigma'.(n+1)(c_j)$. (We know $\sigma'.(n+1)(c_j)$ is non-empty, since $\sigma'.(n+1) = \sigma.m = \sigma.n[\sigma.n(c_j) \cdot \langle v \rangle / \sigma.n(c_j)]$.) $[s \mapsto \sigma'] \not\models CM_{EPL}$, since $PREFIX'$ is contradicted: $\sigma'.(n+1)(c_j).x = \sigma.m(c_j).x = v$, but $\sigma'.(n+2)(c_j).x = \sigma.n(c_j).x$ is undefined. Therefore, by showing $[s \mapsto \sigma'] \models \mathcal{T}[\llbracket f_{TL} \rrbracket]$, the proof is complete.

Recall that we are assuming $[s \mapsto \sigma] \models \mathcal{T}[\llbracket f_{TL} \rrbracket]$. We prove $[s \mapsto \sigma'] \models \mathcal{T}[\llbracket f_{TL} \rrbracket]$ by structural induction on f_{TL} .

Base Case:

1. $f_{TL} = G(t_1, \dots, t_n)$:

The only temporal operators that can appear in $G(t_1, \dots, t_n)$ are *Next* operators over terms. Therefore, by the definition of mapping $\mathcal{R}$ (Table 6.3) and the fact that $\circ\text{-nesting}(G(t_1, \dots, t_n)) \leq n$, the index on every occurrence

of state-sequence variable s in $\mathcal{T}[G(t_1, \dots, t_n)]$ is at most n . By the definition of σ' , σ' and σ are identical in their first $n + 1$ elements. Therefore, $[s \mapsto \sigma'] \models \mathcal{T}[G(t_1, \dots, t_n)]$ follows from $[s \mapsto \sigma] \models \mathcal{T}[f_{TL}]$.

Induction:

2. $f_{TL} = f1 \vee f2$:

By $[s \mapsto \sigma] \models \mathcal{T}[f_{TL}]$, the definition of $\mathcal{T}$, and the definition of $\models$, $[s \mapsto \sigma] \models \mathcal{T}[f1]$ or $[s \mapsto \sigma] \models \mathcal{T}[f2]$. If $[s \mapsto \sigma] \models \mathcal{T}[f1]$, then, by the induction hypothesis, $[s \mapsto \sigma'] \models \mathcal{T}[f1]$. If $[s \mapsto \sigma] \models \mathcal{T}[f2]$, then, by the induction hypothesis, $[s \mapsto \sigma'] \models \mathcal{T}[f2]$. Therefore $[s \mapsto \sigma'] \models \mathcal{T}[f1]$ or $[s \mapsto \sigma'] \models \mathcal{T}[f2]$, and consequently $[s \mapsto \sigma'] \models \mathcal{T}[f1 \vee f2]$.

3. $f_{TL} = \neg f$:

By $[s \mapsto \sigma] \models \mathcal{T}[f_{TL}]$, the definition of $\mathcal{T}$, and the definition of $\models$, it is not the case that $[s \mapsto \sigma] \models \mathcal{T}[f]$. Then, by the induction hypothesis, it is not the case that $[s \mapsto \sigma'] \models \mathcal{T}[f]$. Therefore $[s \mapsto \sigma'] \models \mathcal{T}[\neg f]$.

4. $f_{TL} = (\exists x: f)$:

By $[s \mapsto \sigma] \models \mathcal{T}[f_{TL}]$, the definition of $\mathcal{T}$, and the definition of $\models$, there exists a k such that $[s \mapsto \sigma] \models \mathcal{T}[f[k/x]]$. Then, by the induction hypothesis, there exists a k such that $[s \mapsto \sigma'] \models \mathcal{T}[f[k/x]]$. Therefore $[s \mapsto \sigma'] \models \mathcal{T}[(\exists x: f)]$.

5. $f_{TL} = \bar{\square} f$:

We know $[s \mapsto \sigma] \models \mathcal{T}[\bar{\square} f]$. Therefore, by the definition of $\mathcal{T}$, $[s \mapsto \sigma] \models (\forall i: i \geq 0: \mathcal{T}[f][s[i..]/s])$. By the definition of $\models$, then, for all $i \geq 0$, $\mathcal{T}[f][\sigma[i..]/s]$. Since $\bar{\square}$ is the restricted version of *Always*, f contains no temporal operators. Therefore, the only references to variable s in $\mathcal{T}[f]$ are references to $s.0$. Consequently, from $\mathcal{T}[f][\sigma[i..]/s]$ we

know $\mathcal{T}[\![f]\!][\sigma.i/s.0]$, for all $i \geq 0$. Now, for all $\sigma'.j$, $j \geq 0$, there exists some $\sigma.i$, $i \geq 0$, such that $\sigma'.j = \sigma.i$ (by the definition of σ'). Therefore, $\mathcal{T}[\![f]\!][\sigma'.j/s.0]$, for all $j \geq 0$, follows from $\mathcal{T}[\![f]\!][\sigma.i/s.0]$, for all $i \geq 0$. Hence $[s \mapsto \sigma'] \models (\forall j: j \geq 0: \mathcal{T}[\![f]\!][s[j..]/s])$, and $[s \mapsto \sigma'] \models \mathcal{T}[\![\bar{\square} f]\!]$.

6. $f_{TL} = \circ f$:

We know $[s \mapsto \sigma] \models \mathcal{T}[\![\circ f]\!]$. By the definition of $\mathcal{T}$, and since $\circ\text{-nesting}(\circ f) \leq n$, every occurrence of variable s in $\mathcal{T}[\![\circ f]\!]$ is either

- A. $s.x$, for some $0 \leq x \leq n$, resulting from at most n nested $\circ$ operators, or
- B. $s.(x+i)$, for some universally quantified i and $0 \leq x \leq n$, resulting from a $\bar{\square}$ operator nested within at most n $\circ$ operators. (No temporal operators can be nested within $\bar{\square}$, since $\bar{\square}$ operates only over non-temporal formulas).

We need to show $[s \mapsto \sigma'] \models \mathcal{T}[\![\circ f]\!]$. Since $[s \mapsto \sigma] \models \mathcal{T}[\![\circ f]\!]$, we can prove $[s \mapsto \sigma'] \models \mathcal{T}[\![\circ f]\!]$ by showing that assignments $[s \mapsto \sigma]$ and $[s \mapsto \sigma']$ yield the same values for all occurrences of variable s in $\mathcal{T}[\![\circ f]\!]$. Consider the two types of occurrences of s , as defined above:

- A. $s.x$, $0 \leq x \leq n$. By the definition of σ' , $\sigma'.x = \sigma.x$ for all $0 \leq x \leq n$.
- B. $s.(x+i)$, i universally quantified and $0 \leq x \leq n$. Under assignment $[s \mapsto \sigma]$, the $s.(x+i)$'s range over the set $S_\sigma = \{\sigma.x, \sigma.(x+1), \sigma.(x+2), \dots\}$. Under assignment $[s \mapsto \sigma']$, the $s.(x+i)$'s range over the set $S_{\sigma'} = \{\sigma'.x, \sigma'.(x+1), \sigma'.(x+2), \dots\}$. By the definition of σ' , $S_\sigma = S_{\sigma'}$ for every possible x , $0 \leq x \leq n$.

Therefore, by $[s \mapsto \sigma] \models \mathcal{T}[\![\circ f]\!]$, we conclude $[s \mapsto \sigma'] \models \mathcal{T}[\![\circ f]\!]$.

Theorem 7.3.5 There is no formula $CM_8(f)$ in TL_8 such that $\mathcal{T}[CM_8(f)] \equiv CM_{EPL}(f)$.

Proof: Consider an arbitrary formula $CM_8(f)$ in TL_8 . $\mathcal{T}[CM_8(f)] \equiv CM_{EPL}(f)$ if and only if, for all trace formulas f and state-sequences σ ,

$$[s \mapsto \sigma] \models \mathcal{T}[CM_8(f)] \text{ iff } [s \mapsto \sigma] \models CM_{EPL}(f).$$

Let $f \equiv \text{true}$. By Lemma 7.3.4, there exists a state-sequence σ such that either

$$[s \mapsto \sigma] \models \mathcal{T}[CM_8(f)] \text{ and } [s \mapsto \sigma] \not\models CM_{EPL}(f), \text{ or}$$

$$[s \mapsto \sigma] \not\models \mathcal{T}[CM_8(f)] \text{ and } [s \mapsto \sigma] \models CM_{EPL}(f).$$

Hence $\mathcal{T}[CM_8(f)] \not\equiv CM_{EPL}(f)$. $\otimes$

We conclude that a formula equivalent to $CM_{EPL}(f)$ cannot be written in TL_8 —first-order trace logic with a *Restricted Always* and an *Unrestricted Next* operator. Therefore, this language, as well as any weaker language, is not expressive enough for a relatively complete proof system. By the sufficiency result of Section 7.3.1 and the hierarchy of Figure 7.1, first-order trace logic with an *Unrestricted Always* operator is both an upper and a lower bound on the required expressive power.

7.4 Refining the Subset Hierarchy

We have been considering a hierarchy of temporal logic subsets in which a distinction is made between temporal operators that operate only over first-order formulas—operators $\bar{\square}$ and $\bar{\circ}$ —and temporal operators that operate over temporal formulas—operators $\square$ and $\circ$. With respect to this hierarchy (pictured in Figure 7.1), we have proven that first-order trace logic with operator $\square$ is necessary and sufficient to express a formula equivalent to $CM_{EPL}(f)$. This TL

subset can therefore be used as the basis of a relatively complete trace-based proof system.

The result is strengthened by further refining the hierarchy of TL subsets. By definition, the restricted and unrestricted versions of a temporal operator can be nested zero and arbitrarily many times, respectively, within TL formulas. Rather than distinguishing only between these two types of operators, we consider an infinite set of temporal operators based on allowable nesting depth. For example, given any $x > y \geq 0$, the *Always* operator restricted to operate over formulas with at most x nested *Always* operators is strictly stronger than the *Always* operator restricted to operate over formulas with at most y nested *Always* operators. Letting $\Box_x$ denote the version of $\Box$ restricted to operate over formulas with no more than x nested $\Box$'s, by definition $\bar{\Box} = \Box_0$ and $\Box = \Box_\infty$. The *Next* operator can similarly be refined, with $\bar{\circ} = \circ_0$ and $\circ = \circ_\infty$. Using these infinite sets of temporal operators, we obtain an infinite hierarchy of TL subsets.

Now consider $TL_{\Box_1}$, the TL subset consisting of first-order trace logic with operator $\Box_1$. Given the results of Section 7.3, it is easy to show that, with respect to the refined subset hierarchy, $TL_{\Box_1}$ is necessary and sufficient to express a formula equivalent to $CM_{EPL}(f)$. $TL_{\Box_1}$ is clearly sufficient, since all $\Box$ operators in Definition 7.3.1 of $CM_{\Box}(f)$ are instances of $\Box_1$; i.e. the *Always* operators in $CM_{\Box}(f)$ are nested only once. Furthermore, for any $x \geq 0$, the TL subset based on operators $\Box_0$ and $\circ_x$ is not sufficient to express a formula equivalent to $CM_{EPL}(f)$. This is a consequence of Theorem 7.3.5, which states that there is no formula f_{TL} of first-order trace logic with $\Box_0$ and $\circ_\infty$ operators such that $\mathcal{T}[\llbracket f_{TL} \rrbracket] \equiv CM_{EPL}(f)$. When considering the hierarchy of TL subsets based on nesting of temporal operators, the subset consisting of first-order logic with operator $\Box_1$ is an upper and a lower bound on the expressiveness required to encode a formula equivalent to $CM_{EPL}(f)$.

Any network proof system based on a subset of temporal logic must include

a facility for proving the correctness of formulas in the given subset. (Recall that, when defining STL in Chapter 3, we assumed some means of verifying valid formulas of first-order trace logic.) The question thus arises as to whether there exists a relatively complete proof system in which the provable theorems are exactly the valid formulas of $TL_{\Box_1}$. Although we know that $TL_{\Box_1}$ is less expressive than $TL_{\Box}$, we would also like to show that proofs of $TL_{\Box_1}$ formulas do not involve reasoning over formulas with $\Box_x$ operators, for any $x > 1$.

A *tableau method* [4,54] is a formal decision procedure for formulas in a given logic. Using a tableau method, formulas are verified or refuted based on systematic examination of their constituent parts. If there exists a tableau method for a logic, then there also exists a complete axiomatization of that logic in which formulas are compositionally verified [4]. It has been shown that tableau methods exist for temporal logic [47,54]. By restricting such a tableau method to formulas of subset $TL_{\Box_1}$, we obtain a complete axiomatization of $TL_{\Box_1}$. Furthermore, since the tableau method and derived axiomatization are based on the structure of formulas in $TL_{\Box_1}$, by the definition of $\Box_1$ there is no need to perform intermediate reasoning over formulas with temporal operators stronger than $\Box_1$. (The complexity of a proof system can, of course, be judged by other criteria, e.g. the number of axioms and inference rules, the lengths of proofs, etc. Investigating and comparing axiomatizations of TL subsets is not considered here.)

In the next section, we return to the finite hierarchy of TL subsets introduced in Section 7.2, and we describe a relatively complete proof system based on first-order trace logic with the *Unrestricted Always* operator.

7.5 A Minimally Complete Proof System

In our hierarchy of temporal logic subsets, the language $TL_{\Box}$ consisting of first-order trace logic with an *Unrestricted Always* operator (or $TL_{\Box_1}$, if we consider

the refined hierarchy) is necessary and sufficient for a relatively complete trace-based proof system. Therefore, a proof system based on $TL_{\square}$ is minimally complete with respect to the TL subsets we have defined—any weaker language results in an incomplete proof system. In this section, we describe such a relatively complete proof system based on $TL_{\square}$; we call the proof system *Extended Trace Logic*, or *ETL*.

ETL is derived directly from STL (Chapter 3), but we revise the consequence rule to incorporate formula $CM_{\square}(f)$ and thereby achieve relative completeness. Incorporating $CM_{\square}(f)$ requires ETL formulas to include formulas of $TL_{\square}$ as well as formulas $PN \text{ sat } S$. (Formulas of first-order trace logic are subsumed by formulas of $TL_{\square}$.) A full ETL proof system must include some facility for formal reasoning over formulas in $TL_{\square}$. We do not present such a facility here; the interested reader is referred to discussions in Section 7.4 and [47].

As in STL, specifications for processes and networks are first-order trace formulas; $PN \text{ sat } S$ is used to denote the fact that specification S is satisfied by all possible computations of process or network PN . The axioms of ETL consist of formulas $P \text{ sat } S$ —one for each (primitive) process P of interest—such that S is a valid specification for P . Specifications for networks are derived from specifications for their component processes using the network composition rule of Definition 3.2.1:

$$\frac{P_1 \text{ sat } S_1, P_2 \text{ sat } S_2, \dots, P_n \text{ sat } S_n}{P_1 \parallel P_2 \parallel \dots \parallel P_n \text{ sat } \bigwedge_i S_i}$$

Definition 3.2.2 of the STL Consequence Rule is replaced by:

Definition 7.5.1 (ETL Consequence Rule)

$$\frac{N \text{ sat } S1, CM_{\square}(S1) \Rightarrow CM_{\square}(S2)}{N \text{ sat } S2}$$

Definition 7.5.1 corresponds to the model- and axiom-based consequence rules of Section 5.3.2 (Definitions 5.3.6 and 5.3.7). The model-based consequence rule uses the clause

$$\alpha \in \text{CompModels}(S1) \Rightarrow \alpha \in \text{Models}(S2);$$

the axiom-based consequence rule uses the clause

$$(\Box S1 \wedge \text{ORDERING} \wedge \text{PREFIX}) \Rightarrow \Box S2.$$

We now know that both of these clauses are equivalent to clause

$$CM_{\Box}(S1) \Rightarrow CM_{\Box}(S2)$$

of the ETL consequence rule. All three implications describe the relationship that must hold between a specification $S1$ that is precise for some network N and a specification $S2$ that is valid for N . We have shown that $CM_{\Box}(S1) \Rightarrow CM_{\Box}(S2)$ is written in a temporal language that is both necessary and sufficient for expressing this relationship.

For relative completeness, we assumed the provability of all valid formulas of first-order trace-logic (Section 3.6.2). Statements of the form $CM_{\Box}(S1) \Rightarrow CM_{\Box}(S2)$ are formulas of first-order trace logic with the addition of an *Always* operator. Therefore, we extend the relative completeness assumption to include provability of all valid formulas of $TL_{\Box}$. Under this assumption, we show that ETL is relatively complete.

Theorem 7.5.2 (Relative Completeness of ETL) Let $N = P_1 || P_2 || \dots || P_n$ be a network and S_i a precise specification for P_i , $1 \leq i \leq n$. If S is a valid specification for N , then $N \text{ sat } S$ is provable using ETL.

Proof: As axioms, we are given $P_i \text{ sat } S_i$, $1 \leq i \leq n$. The network composition rule can then be used to obtain $N \text{ sat } \bigwedge_i S_i$. By Theorem 3.6.4, the preciseness-preservation theorem, $\bigwedge_i S_i$ is a precise specification for N . By Corollary 5.3.5,

then, a specification S' is valid for N if and only if

$$\text{CompModels}(\bigwedge_i S_i) \subseteq \text{CompModels}(S').$$

Since S is valid for N (by assumption),

$$\text{CompModels}(\bigwedge_i S_i) \subseteq \text{CompModels}(S).$$

Therefore, by Theorem 6.3.4,

$$CM_{EPL}(\bigwedge_i S_i) \Rightarrow CM_{EPL}(S).$$

For any trace-logic formula f , $CM_{EPL}(f) \equiv \mathcal{T}[\![CM_{\square}(f)]\!]$ (recall Section 7.3.1).

Consequently,

$$\mathcal{T}[\![CM_{\square}(\bigwedge_i S_i)]\!] \Rightarrow \mathcal{T}[\![CM_{\square}(S)]\!].$$

Finally, since $\mathcal{T}$ is semantics-preserving,

$$CM_{\square}(\bigwedge_i S_i) \Rightarrow CM_{\square}(S).$$

We have shown that $TL_{\square}$ formula $CM_{\square}(\bigwedge_i S_i) \Rightarrow CM_{\square}(S)$ is valid. Therefore, by the relative completeness assumption, this formula is provable. Hence the consequence rule can be used to obtain $N \text{ sat } S$, and $N \text{ sat } S$ is provable using ETL. $\otimes$

A formal ETL proof may include reasoning over formulas in $TL_{\square}$. Consequently, it is unnecessary to restrict the specification language of processes and networks to first-order trace logic—tools for reasoning over formulas of a stronger logic must be present in any relatively complete proof system. Therefore, we extend ETL to allow process and network specifications using *Unrestricted Always* operators; the proof system needs no modification, all results remain valid, and a richer language is available for describing process and network behavior. $TL_{\square}$ is a minimal language usable both for process specifications and for building a relatively complete trace-based network proof system.

Chapter 8

Conclusions

In this final chapter, we summarize our results and discuss implications of the work. Potential extensions are considered, and we describe how changes in the assumptions can affect the methods and results. We conclude by proposing future directions of research.

8.1 Summary

We consider completeness in trace-based network proof systems, isolating computational properties that must be axiomatized within a relatively complete proof system and determining the expressiveness required of a logic if it is to axiomatize these properties.

A simple trace logic, STL, is defined. STL captures the essential components of trace-based network proof systems. Like other simple trace logics, STL is incomplete. The incompleteness is illustrated by several example networks, two consisting of a single process. The single-process examples are used to identify the temporal ordering and prefix properties; we show that axiomatizations of these two properties are necessary and sufficient for achieving relative completeness in a trace-based network proof system.

Since temporal ordering and prefix axioms are not expressible in STL, we consider the expressiveness required of a trace logic if it is to axiomatize such properties. An extended predicate logic, EPL, is used to define a formula that exactly characterizes the required expressiveness. Since EPL is unsuitable for use in a proof system, we define a mapping from temporal logic (TL) to EPL. A hierarchy of TL subsets is defined, and the mapping is used to isolate a subset that is necessary and sufficient to express a formula equivalent to the EPL formula. This subset thus has the exact expressive power required of a relatively complete trace logic. We apply this result in the description of a trace-based network proof system that is minimally complete with respect to our hierarchy of TL subsets; any weaker logic results in an incomplete proof system.

8.2 Implications

In Chapters 2–4, we prove that axioms *ORDERING* and *PREFIX* are necessary and sufficient for deducing any valid trace-based network specification from a precise trace-based specification for that network. In Chapters 5–7, the notion of computational models is used to generalize the behavioral properties expressed in *ORDERING* and *PREFIX*. This characterization allows us to establish bounds on the expressiveness required of a logic if it is to be used for a relatively complete proof system.

From the necessity of *ORDERING* and *PREFIX*, we learn that every relatively complete trace logic must somehow encode these axioms. (In Section 4.5, we look at existing complete systems and identify how the axioms are represented.) More generally, every complete system must include the reasoning power to distinguish computational models from non-computational models. We prove that first-order trace logic is not strong enough for this. Therefore, any network proof system based entirely on first-order logic will be incomplete.

From the sufficiency of *ORDERING* and *PREFIX*, we learn that a relatively complete trace logic, to be as simple as possible, must be just strong enough to express and allow reasoning with these axioms. More generally, every complete system must have just enough reasoning power to distinguish computational models from non-computational models. We consider subsets of temporal logic, proving that first-order trace logic with a version of the temporal logic *Always* operator ($TL_{\Box}$) is necessary and sufficient to make this distinction. Therefore, $TL_{\Box}$ is a minimal temporal language on which to base a relatively complete trace-based proof system.

8.3 Extensions

Our work can be extended in several ways. We have already discussed allowing both synchronous and asynchronous communication channels, and we have considered hierarchically structured networks. In Chapter 2, we show how the computation-tree model of network behavior can be extended to accommodate asynchronous communication and hierarchical networks. In Chapter 3, we show that the specification language and proof system of STL can be similarly extended. The changes required for these extensions are primarily syntactic; they do not affect our fundamental results.

We have described only the specification and verification of *safety properties* of network behavior—properties that hold invariantly throughout computation. Formalisms for reasoning about *liveness properties*—properties that eventually hold during the course of computation—have not been considered. (For further discussion of safety and liveness properties, see [6,29,32].) Most trace logics are concerned only with safety properties [10,20,22,39,56], but trace-based proof systems for specifying and verifying both safety and liveness properties have been presented as well [17,25,42]. Methods similar to those used here may be applied

to explore expressiveness and completeness in trace-based proof systems that include facilities for reasoning about liveness properties.

We have been considering networks defined in terms of some given set of component processes. These processes can, in the case of hierarchical networks, be defined as sub-networks of processes, which, in turn, can be defined as sub-sub-networks, etc. We have not, however, considered recursively-defined networks. A proof system for specifying and verifying recursive networks must include an *induction principle*—an inference rule for verifying network specifications based on specifications for recursively-generated processes. Recursive networks are not discussed in [39], are described as an extension to the proof system in [42], and are integrated into the formalisms of [17,22,56]. The consideration of trace-based proof systems for recursively-defined networks is a reasonable extension to our expressiveness and completeness results.

8.4 Changing the Assumptions

Throughout this work, a number of assumptions have been made. It is interesting to determine how changes in these assumptions can affect our methods and results. In Chapter 2, we make several assumptions about the networks under consideration. We assume that communication is synchronous, but we have also described how our methods can be extended to allow asynchronous communication. We assume a simple two-level structure of processes and networks, but we have shown that our work can also accommodate hierarchical networks. A fixed domain of transmittable values is assumed (along with a fixed set of functions and predicates), but this assumption is made only for simplicity of presentation. Our methods and results are equally valid for arbitrary finite and infinite data domains.

Consider our assumption that message transmissions cannot occur simultaneously, i.e. that there is a total order on the communication events of a given computation. (All trace logics we know of make this assumption [7,10,17,20,22,39,42,56].) Suppose, instead, that a model of network computation is used in which communication events may occur simultaneously, i.e. that there is only a partial order on communication events. If message transmissions can actually be programmed to occur simultaneously (by some synchronization mechanism, for example), then, since specifications must reflect this fact, our model and proof system can be modified to treat such simultaneous events as a single event. If communication events cannot be synchronized, then any network with a possible computation in which two events occur simultaneously must also have two additional possible computations in which the events occur in either order. In this case, generality is not lost by simply ignoring the computation in which the events occur simultaneously—network behavior is fully captured by the existence of the two ordered computations.

Finally, throughout the work we have assumed that valid specifications are given for all primitive processes of interest. The same assumption is made in [17,39,42], although several trace logics do include rules for verifying sequential processes [10,20,22,56]. Since numerous proof systems have been devised for sequential program verification¹, for our purposes it is reasonable to assume that correct process specifications are given and to concentrate on the compositional verification of specifications for process networks.

¹Proof systems for sequential programs are given in [12,16,19], as well as within logics for concurrent programs, e.g. [22,30,43,56].

8.5 Future Work

Several areas of potential research stem from the work described here. In Section 8.3, we discuss trace-based formalisms for reasoning about liveness properties, suggesting that the methods used here also could be used to explore expressiveness and completeness in such systems. Since liveness properties can be of significant importance in concurrent and distributed programming [32,42,45], this extension is worthy of investigation.

In Chapter 4, we prove that the temporal ordering and prefix properties must be axiomatized within any relatively complete trace logic. We then consider the expressiveness required of a logic if it is to axiomatize such properties. Subset $TL_{\square}$ of temporal logic is isolated as necessary and sufficient for this, but formulas in $TL_{\square}$ can become fairly complex and unintuitive. (Recall, for example, Definition 7.3.1 of $CM_{\square}(f)$.) Our initial results could also be used to devise a relatively complete trace logic that, rather than establishing expressiveness bounds, captures the intuition behind the temporal ordering and prefix properties. In particular, both properties are based on the notion of *causality*—that one event must precede another.² We are interested in the development of causality-based reasoning for process networks. (Similar applications of causality are suggested by Brock and Ackerman in [7] and discussed by Pratt in [46], but formal proof systems are not described.)

In addition to proving the correctness of specifications for process networks, we would like to have the ability, given a specification, to systematically use the specification as a guide in building an efficient process network that meets the requirements. Such methodologies exist for sequential programming [12,16], but

²The temporal ordering property is directly based on causality (see Definition 4.2.1). The prefix property (Definition 4.3.1) also can be stated in terms of causality by considering relationships between elements within each channel trace.

the case of process networks appears to be considerably more difficult. Having determined the necessary and sufficient components of relatively complete compositional network proof systems, and having isolated the expressive power required of such systems, we have developed the tools for investigating this open area.

Bibliography

- [1] K.R. Apt. Ten years of Hoare's logic: a survey — part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.
- [2] K.R. Apt, N. Francez, and W.P. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2(3):359–385, July 1980.
- [3] H. Barringer. A survey of verification techniques for parallel programs. In *Lecture Notes in Computer Science 191*, Springer-Verlag, New York, 1985.
- [4] J.L. Bell and M. Machover. *A Course in Mathematical Logic*. North-Holland, Amsterdam, 1977.
- [5] J.L. Bell and A.B. Slomson. *Models and Ultraproducts: an Introduction*. North-Holland, Amsterdam, 1974.
- [6] M. Ben-Ari. *Principles of Concurrent Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1982.
- [7] J.D. Brock and W.B. Ackerman. Scenarios: a model of non-determinate computation. In *Formalization of Programming Concepts, Lecture Notes in Computer Science 107*, pages 252–259, Springer-Verlag, New York, 1981.
- [8] S.D. Brookes. A semantics and proof system for communicating processes. In *Logics of Programs, Lecture Notes in Computer Science 164*, pages 68–85, Springer-Verlag, New York, 1984.

- [9] C.C. Chang and H.J. Keisler. *Model Theory*. North-Holland, Amsterdam, 1973.
- [10] Z.C. Chen and C.A.R. Hoare. Partial correctness of communicating sequential processes. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, pages 1–12, Paris, April 1981.
- [11] S.A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, February 1978.
- [12] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.
- [13] H.D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic*. Springer-Verlag, New York, 1984.
- [14] L. Flon and N. Suzuki. The total correctness of parallel programs. *SIAM Journal on Computing*, 10(2):227–246, May 1981.
- [15] R. Floyd. Assigning meaning to programs. In *Mathematical Aspects of Computer Science*, pages 19–32, XIX American Mathematical Society, 1967.
- [16] D. Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.
- [17] E.C.R. Hehner and C.A.R. Hoare. A more complete model of communicating processes. *Theoretical Computer Science*, 26:105–120, September 1983.
- [18] M. Hennessy and C. Stirling. The power of the future perfect in program logics. *Information and Control*, 67:23–52, 1985.
- [19] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.
- [20] C.A.R. Hoare. A calculus of total correctness for communicating processes. *Science of Computer Programming*, 1(4):49–72, October 1981.

- [21] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.
- [22] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.
- [23] C.A.R. Hoare. *A model for communicating sequential processes*. Technical monograph PRG-22, Programming Research Group, Oxford University Computing Laboratory, 1981.
- [24] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.
- [25] B. Jonsson. A model and proof system for asynchronous networks. In *Proceedings of the Fourth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 49–58, August 1985.
- [26] R.M. Keller. Denotational models for parallel programs with indeterminate operators. In E.J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 337–366, North-Holland, New York, August 1977.
- [27] L. Lamport. The ‘Hoare logic’ of concurrent programs. *Acta Informatica*, 14:21–37, 1980.
- [28] L. Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Transactions on Programming Languages and Systems*, 1(1):84–97, July 1979.
- [29] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.
- [30] G.M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15:281–302, 1981.

- [31] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Logics of Programs, Lecture Notes in Computer Science 193*, pages 196–218, Springer-Verlag, New York, 1985.
- [32] Z. Manna and A. Pnueli. Adequate proof principles for invariance and liveness properties of concurrent programs. *Science of Computer Programming*, 4(3):257–289, December 1984.
- [33] Z. Manna and A. Pnueli. How to cook a temporal proof system for your pet language. In *Proceedings of the Tenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 141–154, January 1983.
- [34] Z. Manna and A. Pnueli. Verification of concurrent programs: a temporal proof system. In *Proceedings of the Fourth School on Advanced Programming*, pages 163–255, Amsterdam, June 1982.
- [35] Z. Manna and A. Pnueli. Verification of concurrent programs: proving eventualities by well-founded ranking. Unpublished.
- [36] Z. Manna and A. Pnueli. Verification of concurrent programs: temporal proof principles. In *Logics of Programs, Lecture Notes in Computer Science 131*, pages 200–252, Springer-Verlag, New York, 1982.
- [37] Z. Manna and A. Pnueli. Verification of concurrent programs: the temporal framework. In R.S. Boyer and J.S. Moore, editors, *The Correctness Problem in Computer Science*, pages 215–273, International Lecture Series in Computer Science, Academic Press, London, 1981.
- [38] R. Milner. A calculus of communicating systems. In *Lecture Notes in Computer Science 92*, Springer-Verlag, New York, 1980.
- [39] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(7):417–426, July 1981.

- [40] P. Naur. Proofs of algorithms by general snapshots. *BIT*, 6:310–316, 1966.
- [41] V. Nguyen. The incompleteness of Misra and Chandy’s proof systems. *Information Processing Letters*, 21:93–96, August 1985.
- [42] V. Nguyen, A. Demers, D. Gries, and S. Owicki. A model and temporal proof system for networks of processes. *Distributed Computing*, 1(1):7–25, January 1986.
- [43] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.
- [44] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.
- [45] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.
- [46] V.R. Pratt. On the composition of processes. In *Proceedings of the Ninth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 213–223, January 1982.
- [47] N. Rescher and A. Urquhart. *Temporal Logic. Library of Exact Philosophy*, Springer-Verlag, Vienna, 1971.
- [48] R.D. Schlichting and F.B. Schneider. Using message passing for distributed programming: proof rules and disciplines. *ACM Transactions on Programming Languages and Systems*, 6(3):402–431, July 1984.
- [49] J.R. Schoenfield. *Mathematical Logic*. Addison-Wesley, Reading, Massachusetts, 1967.

- [50] N. Soundararajan. Axiomatic semantics of communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 6(4):647–662, October 1984.
- [51] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, Cambridge, Massachusetts, 1977.
- [52] M. Wand. *Induction, Recursion, and Programming*. North-Holland, New York, 1980.
- [53] J. Widom, D. Gries, and F.B. Schneider. Completeness and incompleteness of trace-based network proof systems. In *Proceedings of the Fourteenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 27–38, January 1987.
- [54] P. Wolper. The tableau method for temporal logic: an overview. Unpublished.
- [55] P. Wolper. Temporal logic can be more expressive. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, pages 340–348, October 1981.
- [56] J. Zwiers, W.P. de Roever, and P. van Emde Boas. *Compositionality and concurrent networks: soundness and completeness of a proofs system*. Report 57, Informatica/Computer Graphics, Faculty of Science, Nijmegen University, The Netherlands, December 1984.