Abstract

The need for communication and coordination between agents in a distributed system distinguishes distributed problems from uni-processor problems. The level of coordination achievable in a system determines which problems are solvable in that system. Often, it is not clear from the informal and operational descriptions of systems in the literature just what level of coordination is possible. This leaves us with the practical question: Which properties of a given system are relevant to another system?

We introduce a method for building a problem-based taxonomy of distributed systems, where a system is classified according to the distributed problems which it can solve. We expect that such a taxonomy would ease the process of applying results from one system to a new system.

The method is non-operational, based on specifications of knowledge-transfer among agents over time. It has the following components:

1. A propositional modal language and associated decidable logic with knowledge and temporal operators for specifying the knowledge transfer behavior and requirements of distributed systems, protocols and problems;

2. The concept of generic distributed problems which, along with full-information protocols, allow systems to be studied and compared in isolation from the influences of protocol choice and specific problem requirements;

3. A small model theorem which allows Kripke structures to be mechanically produced from system and protocol specifications, and a bisimulation result which allows those Kripke structures to be formally compared.

To classify a system by our method, one would first formally specify the system, full-information protocol and problem. A Kripke structure modeling the system and protocol behavior can then be mechanically generated and checked to determine whether the problem specification is met. Kripke structures modeling various systems can be compared by finding bisimulations between knowledge states of the structures. This classification method is formal, mechanizable and comprehensive with the flexibility to compare diverse systems.

A Formal Method for Classifying Distributed Systems

Deborah Shands

May 27, 1994

To my parents.

Acknowledgments

Many people have contributed to this dissertation by giving their time, expertise and encouragement. I would first like to thank my reading committee, Profs. Ten-Hwang Lai, Spiro Michaylov, and Neelamegam Soundararajan, for suggestions which have improved the final draft. I am most grateful to my advisor and friend, Spiro, for his encouragement and not-so-gentle prodding. I would not have completed a dissertation without his help. Spiro was amazingly patient in reading many poor drafts and the reader may be very glad for his frequent reminders that *other people have to read this.*

The correctness of the technical aspects of this thesis is primarily due to Prof. Michael Rathjen. Michael helped me to choose the specification language and logic and to write the proofs of the theorems. He gave me hints when I was confused and towed me through proofs when I was stuck. His help was essential to this research. I am also very grateful to Dr. Chung-Kuo Chang for working with and encouraging me. Our discussions defined the problem and approach at the outset and led to our joint paper.

Several other people gave me technical help with the project. Prof. Anish Arora made suggestions that clarified the use of terminating full-information protocols. Prof. Timothy Long helped me to determine the complexity of the Kripke structure building process. Prof. Yoram Moses pointed out, and helped to fix technical problems in an extended abstract of this work. Dr. Mark Tuttle discussed much of the project with me, suggested that I use full-information protocols, and helped me to get reference materials. Dr. Ronald Fagin pointed me toward related small model theorems, and Dr. Murray Mazer suggested using bisimulation equivalence to formalize the relationship between Kripke structures.

I am also grateful to Prof. Jeff Hirst and Dr. Yao-Nan Lien. Jeff was an excellent logic teacher, able to overcome many of my inadequacies as a logic student. I especially appreciate his willingness to teach a course in modal logic. Yao-Nan taught me how to look critically at a problem and how to choose a research topic.

The office staff of the CIS department, particularly Ms. Marty Marlatt, Ms. Deanna Tavenner, Mr. Thomas Fletcher, Ms. Sandy Hill, Ms. Elizabeth O'Neill and Ms. Jane Grissom, deftly managed the piles of paperwork necessary for me to complete my degree. I am very grateful for their efforts as crisis managers and counselors. I would also like to thank Ms. Eleanor Quinlan and Mr. Carl Phillips. As the department coordinator of teaching associates, Eleanor made my teaching years very pleasant. She has also been a very supportive friend. As the computer facilities manager, Carl has developed and maintained our enormous and reliable network of computers. Without his effort, the production of this thesis would have been much more difficult. I am particularly grateful for his willingness to answer my questions, help me with system problems, and, especially, for his friendship along the way.

Many friends have made my stay in Columbus a great pleasure. I would particularly like to thank Barbara Becker, Melpi Magliveras, Kevin Rodgers, Paolo Bucci, Manas Mandal, Mahendra Ramachandran, J. Ramanujam, Pierce and Liz Krouse, Al and Heidi Fencl, Bill Bradley, Pete Ware and Keith and Katherine Dufrane. I am especially grateful to officemates Sanjay Gadkari and Wayne Heym for their patience and willingness to listen to half-baked ideas, panicked squawking and wrong proofs. They offered advice and suggestions and encouraged me when I was sure I couldn't do it.

Profs. Donna and Timothy Long gave me a place in their home and under their wings. I am completely grateful for their confidence in me. I would also like to thank my sister, Jeannie Shands, who has encouraged me from the start. Finally, I am deeply grateful to my phenomenally wonderful husband, Doug Roble. He has supported me throughout this process and, most impressively, he has NEVER asked (not even once, quietly), "What's taking you so long?"

Contents

Chapter 1

Introduction

A distributed system is a collection of computers, connected by a network of communication channels. Assumptions about the characteristics of the computers and communication links distinguish one such system from another. Because computing tasks can be divided among the computers in the system, a distributed system can often solve a problem faster than can a single computer, working alone. However, the division of tasks introduces the problem of combining intermediate and final results of the individual computations. This necessitates communication and coordination between the computers, which distinguishes the study of distributed systems from that of uni-processors.

Distributed problems require coordination among computers in the system. For example, a large banking database may be split over many sites, with duplicate records for quick access and back-ups in case of failure. Computers maintaining copies must coordinate to keep the copies consistent. The level of coordination achievable in a system determines the problems which are solvable in that system. Often, it is not clear from the descriptions of systems in the literature just what level of coordination is possible. In fact, literature on fault tolerant computing (see, for example, [DDS87, DLS88, BT85, YK88, AGMT92]) has shown that problem-solving ability varies drastically between systems, even between apparently similar systems. This leaves us with the practical question: Which properties of a given system are relevant to another system?

A problem-based taxonomy, where a system is classified according to the distributed problems which it can solve, would ease the process of applying results to new systems. In this thesis, we describe a method for building a taxonomy of distributed systems. The method is:

- knowledge-transfer oriented, as it considers what knowledge is exchanged among the computers, rather than specifics of message-passing along the communication channels;
- non-operational, as specifications of the computer's states of knowledge are used in place of operational details of system behavior;
- mechanizable, as there is an algorithm for determining whether a given problem can be solved in a given system.

In the remainder of this chapter, we define our assumptions about distributed systems, explain our method for comparing them and identify our contributions to the field.

1.1 Distributed Systems, Protocols and Problems

Informally speaking, a *distributed system* is composed of agents and communication channels. An *agent* is our smallest state-bearing entity—it may be a discrete piece of processing machinery (a processor, process, node, neuron, etc.) or, perhaps, several pieces of machinery working together to perform a specific function. An agent's state (assignments to variables, database entries, etc.) determines what it *knows*. Sets of assumptions and requirements about the knowledge states of the agents describe a *problem* to be solved in a particular distributed system. Given a problem and a system, a *protocol* prescribes the behavior of agents in the system with respect to the problem.

Characteristics of a system are usually stated informally and operationally in the literature. For example, a paper might describe a system with a set of fault-free processes, connected by unidirectional channels which do not corrupt or lose messages, but may reorder them. Such operational descriptions can exaggerate differences among systems: two systems may be able to solve similar problems (each running an appropriate protocol) but may operate very differently. We will avoid this problem by considering just specifications of the knowledge transfer behavior of agents within a system. Knowledge transfer occurs when an agent sends, via communication channels, information to other agents. Halpern and Moses, in [HM84] (a revised version appears in [HM90]), provide an excellent introduction to using knowledge to reason about the behavior of distributed systems, and Chandy and Misra ([CM86]) discuss fundamentals of knowledge transfer. We will specify the knowledge transfer behavior of systems (with protocols) using formulas of a formal specification language. A distributed problem is specified by a *problem assumption* and a *problem requirement*, both of which are formulas from our specification language. We must check that the system specification, protocol specification and problem assumption together imply the problem requirement.

1.2 The Method for Classifying Systems

In this section, we outline our approach to specifying diverse systems, handling protocols and problems, so that they don't interfere with our classification of systems, and building models of systems which show knowledge-transfer behavior.

1.2.1 The Specification Language

Our specification language must be formal, so that the process of classifying systems is mechanical, and show knowledge transfer behavior in systems, protocols and problems. To this end, we choose a propositional modal language with knowledge and temporal operators.

This type of language allows us to specify how the knowledge of individual agents must change over time. Halpern and Moses, [HM85], introduce the logic of knowledge to distributed systems, while Pnueli, [Pnu77], introduce temporal logic for reasoning about concurrent programs.

1.2.2 Isolating Systems from Protocols and Problems

The functions of system, protocol and problem specifications are necessarily interrelated, since a system must run a protocol to solve a problem. If we claim that a system, running a given protocol, is unable to solve a problem, one could argue that the system might be capable of solving the problem if it were running a "better" protocol. If we are to study and classify *systems*, we must eliminate the factors of problems and protocol choice on the analysis. To eliminate protocol choice, we apply the idea of a *full-information protocol*, described in the literature (see [PSL80, DM86, MT88, Mic89].) A full-information protocol is one in which each agent sends a copy of its state to every other agent as often as possible, forcing maximal knowledge transfer at the fastest rate allowed by the system. Therefore, each system has a native full-information protocol which can be used whenever a protocol is required: the protocol is, essentially, made a function of the system. Protocols are thus hidden and protocol choice is no longer an issue.

To neutralize the impact of problem choice on the comparison of systems, we take the opposite approach, highlighting problems by using them to define our taxonomic classes. For this purpose, we introduce the notion of a *generic distributed problem* with its components: the *generic problem assumption* and *generic problem requirement.* A generic problem is one which specifies that knowledge must be transferred, but does not specify *what* knowledge must be transferred. Thus, a leader election problem may be specified in exactly the same way that a consensus problem is specified–we would not be able to determine from the specification whether votes or other values were being transferred. In our approach, it is most convenient for us to choose a standard generic problem assumption and allow one generic problem to differ from another only by its requirement. Hence, taxonomic classes can be defined by varying generic problem requirements and a system (running its full-information protocol) with the standard generic problem assumption may or may not fall into a given class, depending on whether it can meet the generic problem requirement of the class.

1.2.3 Classifying Systems

Once we have problem classes to define the taxonomy and a language for describing the systems we intend to classify, it remains to establish methods for placing systems into classes. Typically, there are two ways to determine that an individual should fall into a given class: directly, by showing that it meets the criteria which define the class, or indirectly, by showing that it is sufficiently similar to an already proven member of the class (a representative of the class.)

To make a direct determination that a given system belongs to a particular problem class, we can take either of two approaches: The first is to deduce the problem requirement from the axioms of our logic, using the system specifications (with full-information protocol) and standard generic problem assumption. This deduction may be accomplished either by hand or by using a theorem prover. The second approach is to build a *Kripke structure* (introduced by Kripke in [Kri63]), which appears as a graph, showing the potential knowledge transfer behavior of the system. See [Gol92] for an introduction to Kripke structures. We can then check the structure to determine the validity of the problem requirement. The process of building and checking the Kripke structure is mechanizable. Both the deductive and the structure-building approaches can be used with our method, but the Kripke structure, showing paths of potential knowledge transfer for a system, may be quite useful for studying the behavior of the system.

To make an indirect determination that a given system belongs to a particular problem class, we first produce a Kripke structure for the system and then find a bisimulation relation (introduced by Park in [Par81]) from states in our structure to states in a structure for a representative member of the class. The existence of the bisimulation between the structures allows us to conclude that the two systems are equivalent under the given problem requirement and, hence, belong in the same class. The bisimulation also shows how the knowledge states of the two systems are related.

1.3 Contributions

This thesis discusses the engineering of a method for making non-operational knowledge transfer comparisons of distributed systems. Theoretical components of the method are mostly modifications of tools and concepts from mathematical logic and theoretical distributed systems literature. In our method, each component is required to serve a function which it has not served in previous literature.

In particular, considerations of agents' knowledge have been applied in the literature to the analysis of distributed systems. Most of this analysis is very abstract, producing results which address the knowledge-transfer behavior of all distributed systems. In the instances where knowledge has been applied to the behavior of an individual system, it has been used in assertions or guards of operational programs. Our approach uses knowledge in non-operational specifications for individual systems, problems and protocols. The situation is similar with regard to the use of temporal logic. In the literature, temporal logic has been used to analyze and sometimes even construct protocols for parallel or distributed systems.

Logics of knowledge and time have been thoroughly discussed in the literature, but there have been few logics which have included both knowledge and temporal operators. The few logics that have included both types of operators have combined the operators within the same axioms, determining the memory characteristics of agents in the system and essentially fixing the type of system which may be discussed using the logic. This must be avoided if we are to study a variety of systems. Our logic uses both knowledge and temporal operators and specifies as little as possible about their relationship, to avoid fixing the systems in advance.

This gives us the flexibility to compare diverse systems.

Full-information protocols have been used in the literature for essentially the same purpose for which we use them–eliminating the issue of protocol choice, so that either systems or problems may be studied. We take this approach one step further by introducing the concept of generic distributed problems. This allows us to study systems more carefully by controlling both the protocol and problem parameters. In the literature addressing taxonomies of distributed systems, the common approach has been to use a full-information protocol and a fixed problem (say, consensus) to allow the study of systems. The introduction of the generic problem concept allows us to avoid fixing a problem and also formalizes the common intuition regarding relationships between knowledge transfer problems.

The small model theorem which we show in Chapter 5 has many analogues in mathematical logic. Logicians commonly use such small model results to show that a given logic is decidable. In our application, the small models are used to show the knowledge transfer behavior of individual distributed systems. The ability to generate them mechanically is a key feature of our approach.

Finally, the bisimulation result shown in Chapter 5 allows us to compare one model with another. Bisimulations are used in the literature to show observational equivalence between two processes. In our work, matching bisimulation points show similar states of knowledge which arise in two different systems.

With this background, we can state our contributions. The primary contribution of this thesis is a method for making non-operational, knowledge-transfer comparisons of distributed systems. The method has the following components:

1. A propositional modal language and associated decidable logic with knowledge and temporal operators for specifying non-operationally the knowledge transfer behavior and requirements of distributed systems, protocols and problems.

2. The concept of generic distributed problems which, along with full-information protocols, allow systems to be studied and compared in isolation from the influences of protocol choice and specific problem requirements.

3. A small model theorem which allows Kripke structures to be mechanically produced from system and protocol specifications, and a bisimulation result which allows those Kripke structures to be formally compared.

1.4 Roadmap

The remainder of the dissertation is organized as follows: Chapter 2 contains a survey of related general literature. References to literature addressing technical details are deferred to appropriate points in other chapters. In Chapter 3, we define the syntax and semantics of our language and associated logic. In Chapter 4, we discuss how to use full-information protocols to neutralize the issue of protocol choice and introduce generic problems for defining taxonomic classes. Chapter 5 details the approach to constructing a Kripke structure from a specification, and requirements for bisimulation relations for comparing structures. Finally, the appendices provide tables summarizing axioms and rules of the logic, quick references to symbols, and proofs of the major results.

Chapter 2

Related Work

In this chapter, we discuss work that has influenced our motivation, overall approach and even certain technical choices. Discussion of literature that addresses particular technical details of our approach is deferred until relevant points in other chapters.

Here, we consider previous methods for comparing the capabilities of distributed systems and disadvantages of these methods which lead us to this work. A major motivation for this work arose from taxonomies developed in the fault tolerant protocols literature. We also discuss influences from the literature on temporal logics for analyzing parallel programs and the literature on using knowledge to analyze distributed systems. We then briefly address two other significant bodies of work–Chandy and Misra's Unity system [CM88] and Fischer and Ladner's PDL [FL79]–and discuss how they relate to our own. Our method for comparing distributed systems uses full-information protocols and introduces generic problem specifications. We survey the development of full-information protocols in the literature and work which addresses the variety of distributed problems. Finally, since a major feature of our approach is its mechanizability, we discuss relevant software tools from the literature.

2.1 Comparisons of Distributed Systems

In the literature, approaches to the comparison of distributed systems fall into two categories: explicit comparisons of systems achieved by operational reasoning, and protocol compilers which allow a protocol to be implemented on a variety of systems by inserting system-appropriate communication primitives. This section surveys the literature in both of these categories and characterizes the strengths and weaknesses of the two approaches.

2.1.1 Operational Taxonomies

The need for taxonomic information about distributed systems has long been recognized in the process of developing fault tolerant algorithms for solving distributed problems, since the ability to tolerate faults varies widely from system to system. Several papers have examined collections of distributed systems with respect to their ability to tolerate certain types of faults, while running a deterministic protocol for the distributed consensus problem. In particular, Dolev, Dwork and Stockmeyer,[DDS87], consider the tolerance of 32 different systems (five system parameters, processor (a) synchrony, communication (a) synchrony, message order (a) synchrony, broadcast or point-to-point transmission, and atomic or separate receive/send, yielding 2^5 systems) to crash failures. The authors show operational proofs of bounds on the number of failures which each system can tolerate–steps in the computation, or information exchange, are examined to show situations in which an algorithm must fail.

In [DLS88], Dwork, Lynch and Stockmeyer extend the results of [DDS87] by considering systems with partial communication and processor synchrony and the potential for omission, authenticated Byzantine or Byzantine failures. Bracha and Toueg, [BT85], consider crash and Byzantine fault tolerance of probabilistic protocols. Though most papers have considered message-passing systems, Afek, et. al, [AGMT92], address systems with shared memory. The systems are distinguished by the types of memory faults which may occur. Yamashita and Kameda, in [YK88], discuss systems that use message-passing, but individual agents cannot be identified (i.e., have no i.d. numbers.) A variety of systems are distinguished by the assumptions made about the availability of topological information: no information about network attributes, a known upper bound on the number of agents, an accurate count of the number of agents, or the actual topology of the network. In addition, the various systems are measured against four different distributed problems: leader election, edge election, spanning tree construction and topology recognition.

This operational method of comparison is helpful in pinpointing key steps in distributed decision-making, but finding a possibility or impossibility proof for one system does not necessarily simplify the process of finding a proof for another system (i.e., the approach is not systematic). In addition, the operational method is not mechanizable, since the language of specification and proof is not formal.

2.1.2 The Protocol Compiler Approach

A second, more scalable, method for comparisons involves identifying fundamental components of a protocol to solve a particular problem and providing communication primitives in various systems to implement those components. In [ST87], Srikanth and Toueg consider algorithms for solving Byzantine agreement and election problems in synchronous systems, assuming that messages are authenticated. They then produce broadcast primitives which do not use digital signatures, but have the essential properties of authenticated messages. These primitives can be used in place of signed messages for the purposes of their algorithms. Bracha, in [Bra87], uses a similar reliable broadcast primitive to reduce the impact of Byzantine faults in an asynchronous system.

In [NT88], Neiger and Toueg show primitives which allow protocols for synchronous systems with omission or even Byzantine failures to be compiled from protocols for a synchronous system allowing only crash failures. A similar protocol compiler for asynchronous systems is discussed by Coan in [Coa88]. Bar-Noy and Dolev, [BND89], provide shared-memory and message-passing primitives for an asynchronous system that permit algorithms to be translated between the two information exchange models. In [Wel87], Welch uses a simulation to demonstrate that a system with synchronous agents and unbounded message delays can be simulated by a system with asynchronous agents when faults may occur.

By considering the implementations of communications primitives and the abilities of various systems to simulate one another, it is possible to compare various systems. Depending on the level of interchangeability of the primitives or generality of the simulations, one may be able to avoid the pairwise system comparisons necessitated by the first approach. From a comparison or classification perspective, the disadvantage of the protocol compiler approach is that rather than recognizing a high-level similarity in the knowledge transfer behavior of two systems, we are forced to show the operational details of a simulation or communication primitive. These details may obscure the high level similarities between the systems. For purposes of simplifying protocol design and allowing modular proofs of protocol correctness, however, the compiler approach is excellent.

2.2 Foundations of the Approach

The work in this thesis is based on well-known features of propositional logics with modal operators to express time and knowledge. In [Gol92], Goldblatt provides an introduction to propositional modal logic, including logics of knowledge and a discussion of linear and branching time (CTL) temporal logics. In this section, we discuss foundational literature addressing the use of knowledge for analyzing distributed systems, temporal logic for analyzing parallel programs, and other related foundations.

2.2.1 Logics of Knowledge

Intuitive descriptions of protocols have often used loose concepts of knowledge and common knowledge: one agent will send a message when it "knows" some fact; it is "common knowledge" among agents in the system that the database has the current balance. Distributed problems can often be stated as knowledge transfer problems. Some initial discussions of the relationships among knowledge, communication and action in distributed computer systems borrow logical puzzles from philosophy to help identify how knowledge advances or limits communication and action. In [MDH86], for example, Moses, Dolev and Halpern discuss variants of the cheating husbands puzzle to show how an agent can gain information by watching other agent's actions when the first agent knows how the second agent's actions are related to facts that it knows.

The formal foundations of knowledge-based reasoning about distributed systems established what could be known and learned by agents in a system. In [HM84] (a revised version appears in [HM90]), Halpern and Moses describe a hierarchy of knowledge states for a system, ranging from implicit knowledge to common knowledge, and show that common knowledge cannot be generated in a realistic distributed system. Chandy and Misra, in [CM86], address the question of how agents acquire knowledge during message exchanges. In particular, Chandy and Misra show that either a gain or a loss of knowledge requires that information be transferred sequentially. In [HF85], Halpern and Fagin describe a formal model relating knowledge, action and communication in a distributed computer system. To that end, they introduce the idea of a knowledge-based protocol–an agent's actions are dependent on its knowledge and message history. Fagin, Halpern and Vardi, in [FHV88] (a revised version appears in [FHV92]), show how the choice of system affects the properties of knowledge in that system. Though all of the literature discussed above assume message-passing systems, Merritt and Taubenfeld, [MT91] (see also [MT92]), apply knowledge-based reasoning to shared memory systems, studying the amount of shared memory necessary for learning certain types of facts and confirm results from message-passing systems.

The result of [HM84], showing the impossibility of attaining common knowledge in a realistic distributed system, prompted a search for useful and attainable variants of common knowledge and weaker versions of individual knowledge. In [HM84], Halpern and Moses present ϵ-common knowledge, likely common knowledge, eventual common knowledge and timestamped common knowledge. In [Mos87], Moses is concerned with the limited computational resources of agents in a realistic distributed system. Agents are not assumed to be logically omniscient (i.e., they may not know all the consequences of their knowledge), due to real-world bounds on the amount of time that an agent can spend on processing. To account for knowledge under these circumstances, Moses defines the notion of resource-bounded knowledge. In [NT87] (a revised version appears in [NT93]), Neiger and Toueg present logical clocks which substitute for real-time clocks within a class of problems that they formally characterize. They use these clocks to produce a broadcast primitive which can achieve timestamped common knowledge: for the problem class they characterize, timestamped common knowledge adequately substitutes for common knowledge. In [PT92], Panangaden and Taylor also address the lack of real-time information in a distributed system by defining and showing the applicability of concurrent common knowledge. This variant of common knowledge uses causality, rather than real-time. Halpern, Moses and Waarts, [HMW90], define the notion of continual common knowledge in order to describe the type of knowledge required by the eventual Byzantine agreement problem. Continual common knowledge is contrasted with ordinary common knowledge, which is required for solving the simultaneous Byzantine agreement problem.

Reasoning about knowledge has been applied to system analysis, proving protocols correct, and protocol development. In [Nei87], Neiger presents an approach to knowledge-based analysis of agent behavior that is helpful in system simulation. The interpretation of knowledge is permitted to vary from system to system as long as no agent can detect the difference. If an interpretation is knowledge-consistent with a given system, then protocols can be written for the interpretation which will run successfully in the system. Dwork and Moses, [DM90], study the behavior of fault-tolerant protocols for performing simultaneous actions by determining which facts can become common knowledge as a protocol is executed. In [MT88], Moses and Tuttle describe a simultaneous action protocol which uses tests for common knowledge. Implementations for the tests are shown for several fault models. Halpern and Zuck, [HZ87] (a revised version appears in [HZ92]), show derivations of knowledge-based protocols for solving the sequence transmission problem and operational proofs of their correctness. Katz and Taubenfeld, in [KT87], present a language and method for proving knowledge assertions about programs. The method allows proofs of assertions about both message-passing and shared-memory protocols. Sanders shows, in [San91], a programming language which incorporates knowledge. K_iA is defined as a function from predicate A to predicates dependent only on variables accessible to $Agent_i$. Such knowledge predicate transformers are then allowed in guards of program statements.

The literature on using logics of knowledge and knowledge transfer concepts to reason about distributed systems makes a strong case for analysis based on an assessment of agents' knowledge. This gives us precedent for basing our method of classifying systems on knowledge transfer capabilities.

2.2.2 Temporal Logics

A thorough introduction to temporal logic can be found in [RU71]. In [Pnu77], Pnueli introduced the idea of using (linear time) temporal logic for reasoning about concurrent programs. A variety of work since then has studied temporal logics with regard to reasoning about programs. Gabbay, et. al., [GPSS80], in particular, consider several desirable properties of concurrent programs–e.g., safety, partial correctness, liveness, deadlock and livelock freedom and fairness–and proposes an expressively complete language that will allow their specification.

One aspect of the work in temporal logics for program analysis is a lively debate over the "right" logic for proving programs correct. Lamport begins the argument in [Lam80] with a demonstration of the incomparability of a linear time and a branching time logic and concludes that the linear time logic is better than the branching time logic for reasoning about concurrent programs. In [EH83], Emerson and Halpern produce a more general language (CTL*) for comparing the logics and conclude that the appropriate choice of linear vs. branching time logic is largely application dependent. Emerson and Lei, in [EL85] cite the lower complexity of model checking as an advantage of branching time over linear time logics. Clarke and Draghicescu, [CD88], provide necessary and sufficient conditions for a formula of CTL* to be expressible in a linear time logic. In [ES88], Emerson and Srinivasan survey branching time systems from the literature, including discussions of their axiomatizations and decision procedures for satisfiability and model-checking problems. They also compare the logics with respect to their expressive power and ability to specify properties of programs, and provide additional evidence for allowing the application to guide the choice of logic.

In [HM85] (a revised version appears in [HM]), Halpern and Moses discuss Kripke semantics for logics of knowledge and the choice of an axiom system. They address issues involving decision procedures, including techniques for finding decision procedures and complete axiomatizations, the relationship between axiom system choice and complexity of its decision procedure, and the complexity of a decision procedure for the well-known knowledge logic S5 with distributed knowledge and common knowledge operators. This work generalizes the results of Lehmann, [Leh84], and Ladner and Reif, [LR86]. Halpern and Vardi, in

[HV87] (a revised version appears in [HV89]) consider the complexity of decision procedures for logics with both temporal and knowledge operators. Lehmann ([Leh84]) provides an axiomatization of a linear time/knowledge theory with a common knowledge operator which also specifies a system's memory characteristics: temporal and knowledge operators appear together in the axioms.

2.2.3 Other Approaches to Analyzing Systems, Protocols and Problems

In [CM88], Chandy and Misra discuss an approach to concurrent programming with a focus on producing non-operational program specifications and proving the correctness of those specifications. Their approach emphasizes problems and programs and explicitly avoids formalizing descriptions of specific systems and mappings from program to systems. Their purpose is to describe a foundation, or small collection of basic laws, for (concurrent) programming. Our purpose is taxonomic: we want to classify systems without regard for particular programs which may run on them.

Fischer and Ladner, [FL77] (a revised version appears in [FL79]), introduced propositional dynamic logic (PDL) for reasoning about sequential programs. PDL is the propositional restriction of Dynamic Logic, introduced by Pratt in [Pra76]. In PDL, programs themselves are modal operators: $\langle \text{Prog} \rangle A$ says that program Prog can terminate with formula A holding, whereas $[\text{Prog}]A$ says that whenever Prog terminates, A holds. Fischer and Ladner, [FL79], introduce the small model technique which we apply in our small model theorem of Chapter 5. In [Pra78], Pratt provides a complete axiomatization for PDL as well as a tableaux-based decision procedure for determining the satisfiability of a PDL formula.

2.3 Isolating System Characteristics from Protocols and Problems

In order to carefully study distributed systems, we must be able to isolate the systems from closely related concerns about protocols and problems. The common approach in the literature is to use a full-information protocol to neutralize the effect of protocol choice on the analysis. A single popular problem is then chosen as representative and analysis proceeds, based on this problem. We discuss first the literature involving full-information protocols and then survey work highlighting analyses of distributed problems.

2.3.1 Full-Information Protocols

Full-information protocols have been used since [PSL80] to solve a variety of coordinated action problems. These protocols are appealing due to their ability to solve a variety of problems and the simplicity of their specifications. A full-information protocol is at least as powerful as any other protocol, and so may be taken as an all-purpose protocol for a given system. Dwork and Moses showed in [DM86] that agents running a full-information protocol in a crash failure model will achieve agreement as early as they could using any other protocol, when corresponding runs were compared. This result was extended to omissions models by Moses and Tuttle in [MT88] and to Byzantine models by Michel in [Mic89].

2.3.2 The Variety of Distributed Problems

We expect that each class of our taxonomy of distributed systems will be identified with the formal specification of a distributed problem, so that the variety of problems considered will determine the granularity of distinctions among systems. Just as we fix a protocol and use problems to study systems, Dwork and Skeen, in [DS84], fix a system and control protocols to study the variety of distributed consensus problems. They produce a taxonomy of problems which are characterized by the communication behavior of protocols which solve them. Fischer, Lynch and Merritt, [FLM86], show proofs of the impossibility of solving several consensus problems (Byzantine agreement, weak agreement, Byzantine firing squad, approximate agreement and clock synchronization) in a variety of systems. In [Cha90], Chaudhuri defines the k-set consensus and agreement problems to study the boundary between possibility and impossibility of reaching a solution in an asynchronous system with the potential for crash failures. In these problems, the number k specifies the size of the allowable solution set and, therefore, the simplicity of the problem. 2-set consensus, for example, requires that the agents decide on at most two different values. The difficulty of the problem can then be adjusted to compensate for the possibility of multiple faults.

2.4 Software Tools

The literature discusses many algorithms and software tools for producing proofs and analyzing Kripke structures. A *theorem prover* automates the proving process and a *model checker* determines whether a formula is satisfiable in a given structure. There are a variety of decision procedures and working theorem provers for propositional modal logics. For discussions of decision procedures, see Rescher and Urquhart, [RU71] and Wolper, [Wol81], and for a survey, Abadi and Manna, [AM86]. Catach, in [Cat91], discusses the general theorem proving system, TABLEAUX, for propositional modal logics.

Clarke, Emerson and Sistla, [CES83], introduced the idea of an automatic model-checker for a branching time temporal logic. In [LP84], Lichtenstein and Pnueli discuss model-checking for a linear time logic, and in [HM], Halpern and Moses discuss model-checking for logics of knowledge.

Chapter 3

The Specification Language $\mathcal{L}$

Our specification language, $\mathcal{L}$, will be used to define the knowledge transfer behavior and requirements of distributed systems, protocols and problems. For our purposes, $\mathcal{L}$ must satisfy two requirements: First, it must be formal, so that the process of classifying systems can be mechanized. Second, it must show the knowledge transfer behavior of systems and protocols and the knowledge transfer requirements of problems. This will provide a basis for comparing even the most disparate systems. A propositional modal language with operators for expressing knowledge and time meets both these criteria.

The language, $\mathcal{L}$, has temporal operators $\oplus$ for "next time", $\ominus$ for "last time", $\mathcal{U}$ for "until", and $\mathcal{S}$ for "since." As discussed in [GPSS80], these allow us to express invariances, such as safety, partial correctness, and mutual exclusion and eventualities, such as liveness, total correctness and livelock freedom. $\mathcal{L}$'s knowledge operators are K_i for "agent i knows" and C for "it is common knowledge that." These allow us to express actual and potential knowledge states of a distributed system. With the combination of the temporal and knowledge operators, we can express individual knowledge retention over time (agent memory) and knowledge transfer within the system, over time.

In Section 3.1, we define the symbols of $\mathcal{L}$ and show how properly to generate its formulas. In Sections 3.2 and 3.3, we give the Kripke semantics for $\mathcal{L}$ and detail our deductive system L . Sections 3.4 and 3.5 show examples of distributed systems, specified in $\mathcal{L}$.

3.1 Syntax

Our language $\mathcal{L}$ contains propositional variables, the logical connectives "not" and "and", and modal temporal and knowledge operators. The pair of linear temporal operators: "next time (last time)", and "until (since)" is known to be expressively complete with respect to the first order logic of linear order [GPSS80] (i.e., first order logic with arbitrary unary predicates, equality, and the binary $<$ relation.) The expressiveness of a branching time logic, however, is related to less intuitive formalisms (see [ES88] for a discussion and further references). We choose linear time variants of the "next time (last time)" and "until (since)" operators (for reasons we discuss in Section 3.2) to express the temporal notions over all possible branches or paths of time. Though not strictly necessary for the sake of expressibility, the past operators are provided as a convenience. With them, one can avoid using history variables, as these two formalisms have been shown in [Zuc86] to be equivalent.

There is currently no standard for the expressive completeness of a collection of knowledge operators. I.e., the expressiveness of a collection of linear temporal operators can be compared against the expressiveness of the first order logic of linear order, but, for knowledge, we have no accepted basis for comparison. In the literature, various combinations of operators have been chosen, including operators for individual, distributed and common knowledge. We have chosen two operators: "agent i knows that" and "it is common knowledge that" to express individual and strong group knowledge. Definition 1 below lists the symbols of $\mathcal{L}$. We sometimes prefer to simplify formulas by using abbreviations listed in Table 3.1.

Definition 1 *Our language $\mathcal{L}$ uses the following symbols:*

1. *Propositional variables: a countable set $\mathcal{P} = \{P_0, P_1, \ldots\}$*
2. *Logical connectives: $\neg, \wedge$*
3. *Other symbols: parentheses*
4. *modal operators: $\oplus$, $\ominus$, $\mathcal{U}$, $\mathcal{S}$, C, K_i for $1 \leq i \leq n$, where n is the number of agents in the system.*

Definition 2 defines the (standard) rules for generating formulas from the symbols of $\mathcal{L}$. We tend to eliminate parentheses from formulas when no ambiguity would result.

Definition 2 *A is a formula of $\mathcal{L}$ ($A \in \mathrm{Form}$) iff*

1. *A is a string of symbols of $\mathcal{L}$; and*
2. *A is generated by a finite number of applications of the BNF equation:*
   $A ::= P_j \mid (\neg A) \mid (A \wedge A) \mid (\oplus A) \mid (\ominus A) \mid (A\ \mathcal{U}\ A) \mid (A\ \mathcal{S}\ A) \mid (K_i A) \mid (CA)$ *for $P_j \in \mathcal{P}$ and $1 \leq i \leq n$, where n is the number of agents in the system.*

Definition 3 defines the subformulas of a formula–any substring of the original formula which is itself a formula.

Definition 3 *A* subformula *of formula A is a formula which is a string of consecutively occurring symbols from A.*

3.2 Semantics

The meaning of a formula of $\mathcal{L}$ is given by a Kripke structure, introduced as a formal model for possible worlds semantics by Kripke in [Kri63]. A formula may be either true or false in various states, or paths of states, of a given Kripke structure. A Kripke structure consists of a set of system states S, where each propositional variable is given a value (true or false) at each state by a valuation function V. A pair of states may be connected by an edge of a

Table 3.1: Abbreviations

Abbreviation	Formula	Explanation
true	$P_j \vee \neg P_j$, for some $P_j \in P$	true
false	$\neg$ true	false
$A_1 \vee A_2$	$\neg(\neg A_1 \wedge \neg A_2)$	or
$A_1 \rightarrow A_2$	$(\neg A_1 \vee A_2)$	implication
$A_1 \leftrightarrow A_2$	$(A_1 \rightarrow A_2) \wedge (A_2 \rightarrow A_1)$	bi-implication
$\oplus^n$A	$\underbrace{\oplus \oplus \cdots \oplus}_{n}$A	n states later A
$\ominus^n$A	$\underbrace{\ominus \ominus \cdots \ominus}_{n}$A	n states earlier A
$\langle F \rangle$A	true $\mathcal{U}$ A	sometime in the future A
$\langle P \rangle$A	true $\mathcal{S}$ A	sometime in the past A
[F]A	$\neg\langle\mathrm{F}\rangle\neg A$	always in the future A
[P]A	$\neg\langle\mathrm{P}\rangle\neg A$	always in the past A
EA	$\bigwedge_i K_i$A	everyone knows A
$E^n A$	$\underbrace{EE \cdots E}_{n}$A	everyone knows n times A
SA	$\bigvee_i K_i$A	someone knows A
$\Box$ A	[P]A $\wedge$ A $\wedge$ [F]A	always A
$\diamond A$	$\langle P \rangle A \vee$ A $\vee \langle F \rangle A$	sometime A

relation. In our Kripke structures, we have a temporal relation *Next* (meaning "next time") and knowledge relations $\sim_i$ (meaning "indistinguishable by agent i"). An agent is said to know a fact if that fact holds in all scenarios the agent deems possible.

We take *Next* to be an irreflexive, serial (i.e., $\forall s \exists t$ such that $(s, t) \in Next$) branching temporal relation and the indistinguishability relations $\sim_i$ to be equivalence relations. It seems natural for the indistinguishability relations to be equivalence relations–if state s is indistinguishable from state t, then surely t is similarly indistinguishable from s.

Choosing a branching *Next* relation over a linear *Next* relation is not nearly so easy to justify, however. It remains unclear whether linear or branching time logics are more appropriate for reasoning about concurrent programs. Lamport in [Lam80] found differences in the expressive power of branching and linear time logics, in particular, he found that some common fairness constraints (e.g. strong eventual fairness: if a process demands infinitely often to be executed, then it will eventually be executed) cannot be expressed in the branching time logic CTL (computational tree logic, introduced by Clarke and Emerson in [CE81]), though they are easily expressed in a linear time logic. Conversely, branching time logics can naturally express non-determinism (different computations lie on various paths through the structure), but linear time logics cannot.

Emerson and Halpern, in [EH83], take issue with Lamport's basis for comparison and provide a more uniform framework for comparing branching with linear time. Their approach is to define the syntax and semantics of a larger branching language (CTL^*) and then discuss properties of its sublanguages, some of which are linear. They are able to find a branching time analogue for any linear time logic, but their results show branching and linear time logics which are incomparable with respect to expressive power.

Our semantics is based on branching time structures because we consider the ability to show non-determinism to be potentially useful. We should also note that extensions to CTL have been proposed which allow fairness constraints to be expressed (see [ES88]). Since our application requires only that we verify a protocol's solution to a given problem, while running in a particular distributed system and that the protocol must solve the problem in every run, we are interested only in those properties that hold over all computation paths. The semantics of our temporal operators, therefore, involves all computation paths (universal-style linear operators). Full CTL^* also provides similar operators whose semantics involves only one computation path (existential-style branching operators). We can, therefore, handle specifications with statements of the form "it always happens that...." If our problem specifications were to involve statements beginning with "it might happen that...," then we should add existential operators to our language. The branching time semantics were chosen to allow this extension.

It is also important to mention that the agents within the system may consider a variety of scenarios to be *possible* at each *actual* state of the computation. The resulting Kripke structure will show a branching computational thread with possible states branching off, whenever an agent considers an alternative.

Alternatives to Kripke semantics are available, for example, in [FHV91], Fagin, Halpern and Vardi discuss *knowledge structures* as an alternative to Kripke structures for a language with knowledge operators. But, since a Kripke structure is a state-based semantics, it provides an intuitive picture of the flow (or potential flow) of information through a distributed system. A single structure can show temporal relationships among system states, as well as relationships perceived by individual agents inside the system. The Kripke semantics are also relatively practical, as they generate a graph which can be exploited by mechanical decision procedures.

In [Bar87], Barwise considers alternative semantics for common knowledge. For purposes of dealing with agents in a distributed computing system, the iterate approach which we choose seems most appropriate. In [ES88], Emerson and Srinivasan compare the expressiveness of branching time temporal logics to that of finite state automata on infinite trees and the monadic second order theory of n successors. Again, the use of the temporal logic with Kripke semantics is convenient.

Definition 4 *A* Kripke structure *is a tuple* $\mathcal{M} = (\mathrm{S}, \mathrm{Next}, \sim_1, \sim_2, \ldots, \sim_{\mathrm{n}}, \mathrm{V})$ *with:*

1. *A set of states* S*;*
2. *An irreflexive, serial, branching temporal relation* $\mathrm{Next} \subseteq \mathrm{S} \times \mathrm{S}$ *which indicates that one state precedes another in time;*
3. *Equivalence relations* $\sim_i \subseteq \mathrm{S} \times \mathrm{S}$*, for* $1 \leq i \leq n$ *which indicate that* $Agent_i$ *is unable to distinguish two states;*
4. *A valuation function* $\mathrm{V} : \mathrm{S} \times \mathcal{P} \rightarrow \{true, false\}$ *which assigns a truth value to each propositional variable at every state.*

The semantics of most of the operators is state-based, but the semantics of the $\mathcal{U}$ and $\mathcal{S}$ operators are based on temporal branches. Definition 5 defines what we mean by a temporal branch (an infinite notion) and a temporal path (a finite notion.) For a finite structure, a branch reduces to a path.

Definition 5 *A* Next*-branch starting at state* s_0 *of* $\mathcal{M}$ *is an infinite sequence* $\langle \mathrm{s}_0, \mathrm{s}_1, \mathrm{s}_2, \ldots \rangle$*, where* $(\mathrm{s}_\mathrm{k}, \mathrm{s}_{\mathrm{k}+1}) \in \mathrm{Next}$ *for all* $k \in N$*. A* Next*-path starting at state* s_0 *of* $\mathcal{M}$ *is a finite sequence* $\langle \mathrm{s}_0, \mathrm{s}_1, \ldots, \mathrm{s}_\mathrm{m} \rangle$*, where* $(\mathrm{s}_\mathrm{k}, \mathrm{s}_{\mathrm{k}+1}) \in \mathrm{Next}$ *for all* $0 \leq k < m$ *and* $m \in N$*.*

We can now define the Kripke semantics of a formula of $\mathcal{L}$ by induction on its structure.

Definition 6 *Formula* A *is* true *at state* s *in structure* $\mathcal{M}$*, written* $(\mathcal{M}, \mathrm{s}) \models A$*, when the following hold:*

The semantics of the $\neg$ *and* $\wedge$ *connectives are standard for propositional logic:*

$$(\mathcal{M}, \mathrm{s}) \models P_j \textit{ iff } \mathrm{V}(\mathrm{s}, P_j) = true$$

$$(\mathcal{M}, \mathrm{s}) \models \neg A \textit{ iff } (\mathcal{M}, \mathrm{s}) \not\models A$$

$$(\mathcal{M}, \mathrm{s}) \models A_1 \wedge A_2 \textit{ iff } \ (\mathcal{M}, \mathrm{s}) \models A_1 \textit{ and } (\mathcal{M}, \mathrm{s}) \models A_2$$

The semantics of the simple temporal and knowledge operators are:

$$(\mathcal{M}, \mathrm{s}) \models \oplus A \textit{ iff for all } \mathrm{t} \in \mathrm{S} \textit{ such that } (\mathrm{s}, \mathrm{t}) \in \mathrm{Next}, (\mathcal{M}, \mathrm{t}) \models A$$

$$(\mathcal{M}, \mathrm{s}) \models \ominus A \textit{ iff for all } \mathrm{t} \in \mathrm{S} \textit{ such that } (\mathrm{t}, \mathrm{s}) \in \mathrm{Next}, (\mathcal{M}, \mathrm{t}) \models A$$

$$(\mathcal{M}, \mathrm{s}) \models K_i A \textit{ iff for all } \mathrm{t} \in \mathrm{S} \textit{ such that } (\mathrm{s}, \mathrm{t}) \in \sim_{\mathrm{i}}, (\mathcal{M}, \mathrm{t}) \models A$$

The semantics of the $\mathcal{U}$ and $\mathcal{S}$ operators are:

$(\mathcal{M}, \mathrm{s}) \models A_1 \mathcal{U} A_2$ *iff for all* Next*-branches* $\langle \mathrm{s} = \mathrm{s}_0, \mathrm{s}_1, \mathrm{s}_2, \ldots \rangle$*, there exists* s_k *on the branch such that* $(\mathcal{M}, \mathrm{s}_\mathrm{k}) \models A_2$*, and for all* j *such that* $0 \leq j < k$*,* $(\mathcal{M}, \mathrm{s}_\mathrm{j}) \models A_1$.

$(\mathcal{M}, \mathrm{s}) \models A_1 \mathcal{S} A_2$ *iff for all* Next*-branches* $\langle \ldots, \mathrm{s}_{\mathrm{k-m}}, \ldots, \mathrm{s}_{\mathrm{k-1}}, \mathrm{s}_\mathrm{k} = s \rangle$*, there exists* $\mathrm{s}_{\mathrm{k-m}}$ *on the branch such that* $(\mathcal{M}, \mathrm{s}_{\mathrm{k-m}}) \models A_2$*, and for all* j *such that* $0 \leq j < m$*,* $(\mathcal{M}, \mathrm{s}_{\mathrm{k-j}}) \models A_1$.

Let $\sim$ *be* $\bigcup_i \sim_i$ *(which is not closed under transitivity) and let* $\sim^*$ *be the transitive closure of* $\sim$*, then the semantics of the* C *operator is:*

$$(\mathcal{M}, \mathrm{s}) \models CA \textit{ iff for all } \mathrm{t} \in \mathrm{S} \textit{ such that } (\mathrm{s}, \mathrm{t}) \in \sim^*, (\mathcal{M}, \mathrm{t}) \models A$$

Finally, we extend the notion of a tautology to formulas with modal operators by the following definition. We intend to include all tautologies in our logic, defined in Section 3.3.

Definition 7 *A formula* A *is a* tautology *iff* A *is the result of uniformly substituting formulas for propositional variables in a tautology of propositional calculus.*

3.3 The Logic L

The logic (i.e., deductive system) L , which we define in this section, forms the basis for our reasoning. The axioms described here list only the most fundamental assumptions about knowledge and time. In particular, these axioms do not identify any specific relationships between knowledge and time–such relationships imply either an agent's understanding of the temporal behavior of knowledge in its system or an agent's memory capacity over time. Such system-dependent capabilities are left to individual system specifications.

The inference rules of L are listed in Appendix B in Table B.1, while the axiom schemata appear in Table B.2. The inference rules permit the deduction of any tautology, as well as formulas resulting from Modus Ponens, knowledge or temporal generalization, or any of three induction rules for common knowledge and henceforth (hitherto). The rules for tautologies and Modus Ponens:

$$\frac{\text{A is a tautology}}{A},$$

and

$$\frac{A_1 \ , \ A_1 \rightarrow A_2}{A_2},$$

are familiar from propositional calculus, but we should comment on the remaining rules. The generalization rules:

$$\frac{A}{CA},$$

and

$$\frac{A}{\Box A},$$

say that any formula derivable from L can always be derived by any (and all) agents. This implies that the axioms and rules of the logic are common knowledge among the agents. The remaining three inference rules:

$$\frac{A_1 \rightarrow E(A_1 \wedge A_2)}{A_1 \rightarrow CA_2},$$

$$\frac{(A_2 \vee (A_1 \wedge \oplus A_3)) \rightarrow A_3}{(A_1 \mathcal{U} A_2) \rightarrow A_3},$$

and

$$\frac{(A_2 \vee (A_1 \wedge \ominus A_3)) \rightarrow A_3}{(A_1 \mathcal{S} A_2) \rightarrow A_3},$$

are essentially induction rules, specifying when a stronger expression of knowledge or time may be claimed, based on a weaker expression. The rule for knowledge says that if a second fact is a particularly obvious consequence of a first fact, then we can claim that the first implies common knowledge of the second. This tells us when we may strengthen claims about individual knowledge to common knowledge. The induction rules for the $\mathcal{U}$ and $\mathcal{S}$ operators provide a similar correspondence between $\oplus$ and $\mathcal{U}$ and between $\ominus$ and $\mathcal{S}$.

Let's now consider the axiom schemata for the knowledge operators. The K_i operators are described by the logic S5. The first axiom of S5:

$$K_i(A_1 \rightarrow A_2) \rightarrow (K_iA_1 \rightarrow K_iA_2),$$

defines knowledge distribution. It says that an agent will make a deduction, given a fact and an implication. Hence, agents are logical thinkers. The second axiom:

$$K_iA \rightarrow A$$

requires that only truths be known, this prevents the conflation of knowledge and belief. The third axiom:

$$K_iA \rightarrow K_iK_iA$$

says that an agent knows what it knows (i.e., there is no hidden knowledge.) This is called positive introspection. The fourth axiom:

$$\neg K_iA \rightarrow K_i\neg K_iA$$

says that an agent is aware of its ignorance: it knows what it doesn't know. This is called negative introspection and is probably the most controversial of the knowledge axioms. Philosophers do not generally accept that humans are always aware of their ignorance, but from the computer science perspective, this is considered to be an acceptable assumption (see [Bar87].) A computer agent has a "closed world", perhaps a database with a field for every fact the agent could know (i.e., store in memory). The agent need only check whether the field is null to know that it does not know the fact.

The common knowledge operator is similarly addressed by the axioms of S5:

$$C(A_1 \rightarrow A_2) \rightarrow (CA_1 \rightarrow CA_2),$$

$$CA \rightarrow A,$$

$$CA \rightarrow CCA,$$

and

$$\neg CA \rightarrow C\neg CA$$

with the addition of the "fixed point" axiom:

$$CA \rightarrow E(A \wedge CA)$$

This axiom recursively defines common knowledge in terms of individual knowledge (remember that EA abbreviates $K_1A \wedge K_2A \wedge \cdots \wedge K_nA$). In fact, common knowledge can be defined as the greatest fixed point of the recursive equation $CA \leftrightarrow E(A \wedge CA)$.

Let's now consider the axiom schemata for the temporal operators. The first pair of axioms:

$$\oplus(A_1 \rightarrow A_2) \rightarrow (\oplus A_1 \rightarrow \oplus A_2),$$

and

$$\ominus(A_1 \to A_2) \to (\ominus A_1 \to \ominus A_2)$$

say that deductions can be made, as expected, at each state in time (i.e., temporal distribution.) The second pair of axioms:

$$\neg\oplus \text{ false},$$

and

$$\neg\ominus \text{ false}$$

say that the $\oplus$ and $\ominus$ operators do not allow us to write falsehoods into either the past or the future. The third axiom:

$$\oplus P_j \to \neg \oplus \neg P_j, \text{ for } P_j \in \mathcal{P},$$

enforces the seriality of *Next*. Finally, the pair of axioms:

$$A_1 \mathcal{U} A_2 \leftrightarrow A_2 \vee (A_1 \wedge \oplus(A_1 \mathcal{U} A_2)),$$

and

$$A_1 \mathcal{S} A_2 \leftrightarrow A_2 \vee (A_1 \wedge \ominus(A_1 \mathcal{S} A_2))$$

give a recursive definition of the $\mathcal{U}$ and $\mathcal{S}$ operators, much like the recursive definition of the C operator. This says that on every branch or path, either A_2 holds, or A_1 holds at all states between now and the time that A_2 holds.

L is sound and complete. A soundness proof is provided in Section C.1. In [HM], Halpern and Moses show both soundness and completeness of the knowledge component (the logic S5). In particular, S5 is determined by (i.e., is sound and complete for) the class of equivalence relations. In [EH85], Emerson and Halpern show soundness and completeness for the temporal component of L , CTL^*. From [HV89], we know that the validity problem for a logic with a common knowledge operator and linear time operators is EXPTIME-complete, if we do not assume that no forgetting or no learning takes place in the system. L , itself, does not make these assumptions, but adding such assumptions at the system formula level could make the logic undecidable. A thorough discussion of these assumptions can be found in [FHV92]. Chapter 5 shows decidability by introducing a small model theorem. At that point, we can also address the complexity of model-checking for L .

3.4 Example: The CSF System

In this section, and in Section 3.5, we show how to specify a system, protocol and problem, using $\mathcal{L}$. We begin by specifying a system, allowing operational primitives for message sending, arrival and receipt to appear in our specification formulas. Such operational descriptions of knowledge transfer are just informal aids for producing a formal, non-operational specification. They are not part of $\mathcal{L}$ and must be eliminated before a Kripke structure can be produced. Our protocol specifications will also contain references to operational details of knowledge transfer. Using the two collections of formulas, we can eliminate these operational references. The combined non-operational specification of knowledge transfer in the system running the protocol can then be added to the (non-operational) problem specification. We now describe a system and produce formulas to specify it, then describe and specify a problem to be solved, and, finally, describe and specify a protocol solving the problem in the system.

3.4.1 An Operational Description of the CSF System

In the CSF system, two agents communicate synchronously and may fail by crashing. Agents which have not failed are said to be *working*. We first address the behavior of CSF agents. Each agent maintains a local clock. Intuitively, we want it always to be the case that:

$$\text{“}Clock_i = \text{a”} \rightarrow (K_i \text{ “}Clock_i = \text{a”} \vee \text{“}failed_i\text{”})$$

where a string between double quotation marks, should, formally, be replaced with a propositional variable. We must be careful to note that our language does not allow statements like “$Clock_i$ = a” which appears to have variables and a function. Instead, this should be interpreted as a finite set of formulas of that form. For example,

$$\text{“The local clock of } Agent_7 \text{ is 3”} \rightarrow$$

$$(K_7\text{“The local clock of } Agent_7 \text{ is 3”} \vee \text{“}failed_7\text{”})$$

Note that this restriction prevents us from expressing infinite sequences of local clock values, round numbers, etc. Without this restriction, however, we would be able to embed statements of set theory and would suffer from the accompanying incompleteness result.

We now return to describing the CSF system. The local clock counts changes in an agent's knowledge state, so when the knowledge of the agent changes, its local clock value is incremented by one. Again, we want it always to be the case that:

$$(\neg\, K_i\text{A} \wedge \oplus K_i\text{A}) \rightarrow ((\text{“}Clock_i = \text{a”}) \rightarrow \oplus(\text{“}Clock_i = \text{a} + 1\text{”}))$$

Since crash failures are permanent (i.e., agents do not recover from crashes), a failed agent remains in a failure state forever, with a stopped local clock. We want it always to be the case that:

$$\text{“}failed_i\text{”} \rightarrow (\text{“}Clock_i = \text{a”} \rightarrow [\text{F}]\text{“}Clock_i = \text{a”}) \wedge [\text{F}]\text{“}failed_i\text{”}$$

A message received by an agent must have arrived on the channel, since agents cannot corrupt a message during receipt. We avoid formulas (facts) whose truth value might change in a knowledge transfer. Case in point: sending a message to say “it is raining” is not useful if the message might arrive after the rain has stopped. We prefer the message “it was raining at 2 p.m. EDT on September 2, 1993 in Columbus Ohio”. This fact is *stable* (or *future permanent*), in that once it holds, it will hold forever after. In our language, a stable fact takes the form [F]A, where A is any formula of $\mathcal{L}$. This says that, henceforth, A holds. We want it always to be the case that:

$$\text{received(i, j, [F]A)} \rightarrow \langle \text{P} \rangle \text{arrived(i, j, [F]A)}$$

where received(i, j, [F]A) means that message [F]A, which was sent from $Agent_i$ has been received by $Agent_j$. The primitive received(i, j, [F]A) must be removed before a Kripke structure can be constructed for CSF, because our language does not allow expressions of second-order notions. Received(i, j, [F]A) addresses an event concerning the formula [F]A, but this is not expressible in a propositional language.

Because agent failure is restricted to crashing, an agent cannot send information that it does not know to be correct (i.e., agents cannot lie.) The following should always hold:

$$\text{sent(i, j, [F]A)} \rightarrow K_i[\text{F}]\text{A}$$

which says that if a message [F]A was sent by $Agent_i$, then $Agent_i$ knows that [F]A holds. The string sent(i, j, [F]A) must be removed from the specification for the same reason that received(i, j, [F]A) must be removed from the previous formula. Since agents cannot lie, an agent knows that any information it receives must be correct:

$$\text{received(i, j, [F]A)} \rightarrow K_j[\text{F}]\text{A}$$

We next address the behavior of CSF channels. Communication between agents is point-to-point via channels which are failure free. A message which is sent eventually arrives at the receiver, and a message that has arrived was actually sent. We distinguish message arrival (i.e., message becomes available for receipt) from message receipt. The following two statements must always hold (though the primitives sent and arrived must be eliminated):

$$\text{sent(i, j, [F]A)} \rightarrow \langle \text{F} \rangle \text{ arrived(i, j, [F]A)}$$

and

$$\text{arrived(i, j, [F]A)} \rightarrow \langle \text{P} \rangle \text{sent(i, j, [F]A)}$$

Finally, we address the synchronous communication of CSF. Each agent can read a shared round counter and messages are received during the same round in which they are sent. The following statement says that the round counter is shared:

$$\text{“Round = x”} \rightarrow (K_i \text{ “Round = x”} \vee \text{“}\mathit{failed}_i)$$

We also have a memory requirement for agents of CSF. For agents to coordinate rounds of message transmission, each agent must be aware of the changing of rounds. This requires that an agent be able to remember the round number at the last time on its internal clock. The following formula makes that requirement:

$$\ominus\ K_i(\text{“}\mathit{Clock}_i = \text{a”} \wedge \text{“Round = x”}) \rightarrow K_i(\text{“}\mathit{Clock}_i = \text{a”} \rightarrow \text{“Round = x”})$$

The last requirement guarantees that a transmission occupies only one round, so that if the round number were x when a message was sent, then the round number would remain x until that message was received:

$$\text{sent(i, j, [F]A)} \wedge \text{“Round = x”} \rightarrow (\text{“Round = x”}\ \mathcal{U} \text{ received (i, j, [F]A))}$$

The operational description of CSF that we have discussed appears in Table 3.2. Each formula should be preceded by the C and $\Box$ operators to ensure that it is common knowledge that the formula always holds.

3.4.2 A Message Exchange Problem

We can now describe an example problem, involving a simple message exchange. We assume that, eventually, $Agent_1$ will know some fact [F]A. To solve the problem, our protocol must ensure that, eventually, $Agent_1$ knows that $Agent_2$ knows fact [F]A. I.e., $Agent_1$ must somehow transmit [F]A to $Agent_2$ and be certain that $Agent_2$ got the message. If either agent fails, then all bets are off. Our problem specification involves two formulas: one stating our assumptions and a second stating the obligation which the protocol must meet:

Problem 1: **assumption:** $\langle \text{F} \rangle\ K_1[\text{F}]\text{A}$
requirement: $\langle \text{F} \rangle\ (K_1 K_2[\text{F}]\text{A} \vee \text{“}\mathit{failed}_1\text{”} \vee \text{“}\mathit{failed}_2\text{”})$

We can now discuss the protocol used to solve Problem 1 in CSF–essentially, a send-acknowledge protocol with failures permitted. Initially, $Agent_1$ is required to send to $Agent_2$ any stable formula that it knows. Of course, $Agent_1$ may fail, and, in that case, the message might not be sent. We must always have that:

Table 3.2: Operational Description of the CSF System

Agents: Crash Failures Permitted	
CSF 1	"$Clock_i$ = a" $\rightarrow$ (K_i "$Clock_i$ = a" $\vee$ "$failed_i$")
CSF 2	($\neg$ K_iA $\wedge$ $\oplus K_i$A) $\rightarrow$ ("$Clock_i$ = a" $\rightarrow$ $\oplus$"$Clock_i$ = a + 1")
CSF 3	"$failed_i$" $\rightarrow$ ("$Clock_i$ = a" $\rightarrow$ [F]"$Clock_i$ = a") $\wedge$ [F]"$failed_i$"
CSF 4	received(i, j, [F]A) $\rightarrow$ $\langle$P$\rangle$arrived(i, j, [F]A)
CSF 5	sent(i, j, [F]A) $\rightarrow$ K_i[F]A
CSF 6	received(i, j, [F]A) $\rightarrow$ K_j[F]A
Channels: non-FIFO, point-to-point, no failures	
CSF 7	sent(i, j, [F]A) $\rightarrow$ $\langle$F$\rangle$ arrived(i, j, [F]A)
CSF 8	arrived(i, j, [F]A) $\rightarrow$ $\langle$P$\rangle$sent(i, j, [F]A)
Communication: Synchronous	
CSF 9	"Round = x" $\rightarrow$ (K_i "Round = x" $\vee$ "$failed_i$")
CSF 10	$\ominus$ K_i("$Clock_i$ = a" $\wedge$ "Round = x") $\rightarrow$ K_i("$Clock_i$ = a" $\rightarrow$ "Round = x")
CSF 11	sent(i, j, [F]A) $\wedge$ "Round = x" $\rightarrow$ ("Round = x" $\mathcal{U}$ received (i, j, [F]A))

$$K_1\text{[F]A} \rightarrow \langle\text{F}\rangle\ (\text{sent(1, 2, [F]A)} \vee \text{“}failed_1\text{”})$$

$Agent_2$, in turn, must acknowledge a message received from $Agent_1$. $Agent_2$'s acknowledgment of the receipt of stable fact [F]A from $Agent_1$ looks like: K_2[F]A. This gives us the requirement that:

$$\text{received(1, 2, [F]A)} \rightarrow \langle\text{F}\rangle\ (\text{sent}(2, 1, K_2\text{[F]A}) \vee \text{“}failed_2\text{”})$$

To this point, we have placed requirements on message transmission and receipt. We must also be sure that an agent picks up (receives) messages available for delivery (arrived). Each agent must eventually receive any message which has arrived in its buffer. We require that the following always hold:

$$\text{arrived(i, j, [F]A)} \rightarrow \langle\text{F}\rangle\ (\text{received(i, j, [F]A)} \vee \text{“}failed_j\text{”})$$

Table 3.3 shows the protocol (Protocol with Failures) used to solve Problem 1 in CSF. Again, each formula should be preceded by the C and $\Box$ operators.

We can now try to check that the PF protocol really does solve Problem 1, while running in the CSF system. Remember that the message-passing primitives "sent", "arrived", and

Table 3.3: Operational Description of the PF Protocol

PF 1	K_1[F]A $\rightarrow \langle$F$\rangle$ (sent(1, 2, [F]A) $\vee$ "$failed_1$")
PF 2	received(1, 2, [F]A) $\rightarrow \langle$F$\rangle$ (sent(2, 1, K_2[F]A) $\vee$ "$failed_2$")
PF 3	arrived(i, j, [F]A) $\rightarrow \langle$F$\rangle$ (received(i, j, [F]A) $\vee$ "$failed_j$")

"received" are not part of $\mathcal{L}$, so cannot formally be included in our specifications. Removing these primitives involves finding an implication from one primitive to the next (for example, sent $\rightarrow \langle$F$\rangle$ arrived) and deducing the second via Modus Ponens. Essentially the same process is used in a (not quite formal) deduction of **requirement** from CSF + PF + **assumption**, shown in Table 3.4. In this proof, the operational primitives "sent", "arrived" and "received" are treated as syntactic black boxes which behave as propositional variables with respect to the deduction. Each line is deduced from formulas of CSF + PF + **assumption** and the preceding lines. We make use of a few simple lemmas which are proved in Section C.2.

This proof shows that, indeed, PF solves Problem 1, while running in CSF. In particular, it shows that

$$\langle \mathrm{F}\rangle K_1[\mathrm{F}]\mathrm{A} \rightarrow \langle \mathrm{F}\rangle(K_1 K_2[\mathrm{F}]\mathrm{A} \vee \text{“}failed_1\text{”} \vee \text{“}failed_2\text{”}).$$

This formula is at the heart of knowledge transfer in CSF, running PF. In fact, there are no other truly non-operational formulas describing that knowledge transfer: all of the formulas of PF include message-passing primitives and many of those in CSF do, as well. Formulas 1, 2, 3, 9 and 10 of CSF do not explicitly describe message-passing and, of these, formulas 1, 2, 3 and 10 specify the workings of agents' internal clocks. Formula 8 addresses agents' awareness of round synchronization, operational overhead for the system. This example shows how, in practice, we might arrive at a non-operational specification and gives us reason to believe that the protocol does solve the problem in the system.

Note that many formulas of CSF were not used in the deduction. This is because PF did not make use of several of the CSF specifications. In particular, PF did not use round numbers or local clocks, since Problem 1 did not require any synchronization of the agents. A protocol that required some synchronization would probably make use of these specifications.

3.5 Example: The CAN System

We now describe a second system which communicates asynchronously with no failures (CAN). CAN differs from CSF in the behavior of its agents (failures were permitted in CSF) and the fact that there are no rounds to coordinate message sending and receiving phases for all the agents. Table 3.5 lists the system formulas of CAN. As usual, each formula should be preceded by the C and $\Box$ operators. Note the CAN formulas are the CSF formulas with references to failures and synchronous communication removed. Channel behavior is

Table 3.4: Proof that PF Solves Problem 1 in CSF

1.	$\langle \mathrm{F} \rangle K_1[\mathrm{F}]\mathrm{A}$	Problem 1 assumption
2.	$K_1[\mathrm{F}]\mathrm{A} \rightarrow \langle \mathrm{F} \rangle(\mathrm{sent}(1, 2, [\mathrm{F}]\mathrm{A}) \vee \text{“}\mathit{failed}_1\text{”})$	PF1
3.	$\langle \mathrm{F} \rangle \langle \mathrm{F} \rangle(\mathrm{sent}(1, 2, [\mathrm{F}]\mathrm{A}) \vee \text{“}\mathit{failed}_1\text{”})$	1, 2, Lemma 1
4.	$\langle \mathrm{F} \rangle(\mathrm{sent}(1, 2, [\mathrm{F}]\mathrm{A}) \vee \text{“}\mathit{failed}_1\text{”})$	3, Lemma 2
5.	$\mathrm{sent}(\mathrm{i}, \mathrm{j}, [\mathrm{F}]\mathrm{A}) \rightarrow \langle \mathrm{F} \rangle \mathrm{arrived}(\mathrm{i}, \mathrm{j}, [\mathrm{F}]\mathrm{A})$	CSF7
6.	$\langle \mathrm{F} \rangle(\langle \mathrm{F} \rangle \mathrm{arrived}(1, 2, [\mathrm{F}]\mathrm{A}) \vee \text{“}\mathit{failed}_1\text{”})$	4,5, Lemma 3
7.	$\langle \mathrm{F} \rangle(\mathrm{arrived}(1,2, [\mathrm{F}]\mathrm{A}) \vee \text{“}\mathit{failed}_1\text{”})$	6, Lemma 4
8.	$\mathrm{arrived}(\mathrm{i}, \mathrm{j}, [\mathrm{F}]\mathrm{A}) \rightarrow \langle \mathrm{F} \rangle(\mathrm{received}(\mathrm{i}, \mathrm{j}, [\mathrm{F}]\mathrm{A}) \vee \text{“}\mathit{failed}_j\text{”})$	PF3
9.	$\langle \mathrm{F} \rangle(\langle \mathrm{F} \rangle(\mathrm{received}(1, 2, [\mathrm{F}]\mathrm{A}) \vee \text{“}\mathit{failed}_2\text{”}) \vee \text{“}\mathit{failed}_1\text{”})$	7, 8, Lemma 3
10.	$\langle \mathrm{F} \rangle(\mathrm{received}(1, 2, [\mathrm{F}]\mathrm{A}) \vee \text{“}\mathit{failed}_2\text{”} \vee \text{“}\mathit{failed}_1\text{”})$	9, Lemma 4
11.	$\mathrm{received}(1, 2, [\mathrm{F}]\mathrm{A}) \rightarrow \langle \mathrm{F} \rangle(\mathrm{sent}(2, 1, K_2[\mathrm{F}]\mathrm{A}) \vee \text{“}\mathit{failed}_2\text{”})$	PF2
12.	$\langle \mathrm{F} \rangle(\langle \mathrm{F} \rangle(\mathrm{sent}(2, 1, K_2[\mathrm{F}]\mathrm{A}) \vee \text{“}\mathit{failed}_2\text{”}) \vee \text{“}\mathit{failed}_2\text{”} \vee \text{“}\mathit{failed}_1\text{”})$	10, 11, Lemma 3
13.	$\langle \mathrm{F} \rangle(\mathrm{sent}(2, 1, K_2[\mathrm{F}]\mathrm{A}) \vee \text{“}\mathit{failed}_2\text{”} \vee \text{“}\mathit{failed}_1\text{”})$	12, Lemma 4
14.	$\langle \mathrm{F} \rangle(\langle \mathrm{F} \rangle \mathrm{arrived}(2, 1, K_2[\mathrm{F}]\mathrm{A}) \vee \text{“}\mathit{failed}_2\text{”} \vee \text{“}\mathit{failed}_1\text{”})$	5, 13, Lemma 3
15.	$\langle \mathrm{F} \rangle(\mathrm{arrived}(2, 1, K_2[\mathrm{F}]\mathrm{A}) \vee \text{“}\mathit{failed}_2\text{”} \vee \text{“}\mathit{failed}_1\text{”})$	14, Lemma 4
16.	$\langle \mathrm{F} \rangle(\langle \mathrm{F} \rangle(\mathrm{received}(2, 1, K_2[\mathrm{F}]\mathrm{A}) \vee \text{“}\mathit{failed}_2\text{”}) \vee \text{“}\mathit{failed}_2\text{”} \vee \text{“}\mathit{failed}_1\text{”})$	8, 15, Lemma 3
17.	$\langle \mathrm{F} \rangle(\mathrm{received}(2, 1, K_2[\mathrm{F}]\mathrm{A}) \vee \text{“}\mathit{failed}_2\text{”} \vee \text{“}\mathit{failed}_1\text{”})$	16, Lemma 4
18.	$\mathrm{received}(2, 1, K_2[\mathrm{F}]\mathrm{A}) \rightarrow K_1 K_2[\mathrm{F}]\mathrm{A}$	PF3
19.	$\langle \mathrm{F} \rangle(K_1 K_2[\mathrm{F}]\mathrm{A} \vee \text{“}\mathit{failed}_2\text{”} \vee \text{“}\mathit{failed}_1\text{”})$	17, 18, Lemma 3

identical in CSF and CAN.

Table 3.5: Operational Description of the CAN System

Agents: No Failures Permitted	
CAN 1	"$Clock_i$ = a" $\rightarrow$ K_i "$Clock_i$ = a"
CAN 2	$(\neg K_i\mathrm{A} \wedge \oplus K_i\mathrm{A}) \rightarrow$ ("$Clock_i$ = a" $\rightarrow \oplus$"$Clock_i$ = a + 1")
CAN 3	received(i, j, [F]A) $\rightarrow \langle$P$\rangle$arrived(i, j, [F]A)
CAN 4	sent(i, j, [F]A) $\rightarrow K_i$[F]A
CAN 5	received(i, j, [F]A) $\rightarrow K_j$[F]A
Channels: non-FIFO, point-to-point, no failures	
CAN 6	sent(i, j, [F]A) $\rightarrow \langle$F$\rangle$ arrived(i, j, [F]A)
CAN 7	arrived(i, j, [F]A) $\rightarrow \langle$P$\rangle$sent(i, j, [F]A)
Asynchronous Communication	
no formulas	

We ask CAN to solve a problem similar to that of Problem 1, but without allowing for failures. The specification of Problem 2 looks like this:

Problem 2: **assumption:** $\langle \mathrm{F}\rangle K_1[\mathrm{F}]\mathrm{A}$
requirement: $\langle \mathrm{F}\rangle K_1 K_2[\mathrm{F}]\mathrm{A}$

Not surprisingly, CAN can solve Problem 2 by running a protocol very similar to PF, again without allowing for failures. The send-acknowledge Protocol with No failures is shown in Table 3.6 (add the C and $\Box$ operators to each.)

Table 3.6: Operational Description of the PN Protocol

PN 1	$K_1[F]A \rightarrow \langle$F$\rangle$ sent(1, 2, [F]A)
PN 2	received(1, 2, [F]A) $\rightarrow \langle$F$\rangle$ sent(2, 1, K_2[F]A)
PN 3	arrived(i, j, [F]A) $\rightarrow \langle$F$\rangle$ received(i, j, [F]A)

We won't show a proof that CAN + PN + **assumption** solves the **requirement** of Problem 2, but the deduction would be very similar to that given in Table 3.4 for CSF, PF and Problem 1. By eliminating the operational message-passing primitives, we get that CAN + PN can be non-operationally specified by

$$\langle \mathrm{F}\rangle K_1[\mathrm{F}]\mathrm{A} \rightarrow \langle \mathrm{F}\rangle(K_1 K_2[\mathrm{F}]\mathrm{A}).$$

We will re-consider CSF and CAN in Chapter 4, where we show more complex protocols for them and again in Chapter 5, where we build their Kripke structures and make a formal comparison between the two.

3.6 Summary

In this chapter, we have introduced the syntax and semantics of our specification language, $\mathcal{L}$, as well as the deductive system, L . This provides the basis for non-operational specifications of distributed systems, protocols and problems. Using the logic, L , we can both check that a given problem is solved by protocol running in a particular system and find a non-operational specification of the system-protocol combination. This is, of course, the most direct approach to checking that a particular system-protocol combination fits a taxonomic class characterized by a given problem specification. In the next chapter, we will show how to neutralize the effect of protocol choice on the analysis of system behavior, effectively decoupling systems and protocols, and discuss the use of problem specifications as taxonomic classes.

Chapter 4

Isolating Systems from Protocols and Problems

In Chapter 3, we showed how to specify a distributed system, protocol and problem. From such a specification, we intend to build a Kripke structure (See Definition 4 of Chapter 3), showing potential paths of knowledge transfer in the system. As we discussed in Section 1.2.2, the functions of system, protocol and problem specifications are necessarily interrelated: a problem is not solved directly by a system, since a protocol must direct the system's activities toward solving the problem. Also, most problems include assumptions. For example, "The broadcasting agent begins with a binary value B. The problem is solved when all of the agents know that B is the broadcast value." A system cannot be expected to achieve the problem requirement without using the problem assumption. If we were to build a Kripke structure based only on a system specification, we would see little, if any, knowledge transfer behavior, since agents in most systems exchange information at the direction of a protocol which was designed to solve a particular problem.

For these reasons, the Kripke structures which we will use to model knowledge transfer behavior of distributed systems (discussed in Chapter 5) must be based on the specifications of system, protocol and problem. The contributions of protocol and problem specifications must not, however, impact the classification of the system. In this chapter, we focus on methods of neutralizing the effects of protocol choice and problem assumption on the analysis of a distributed system. In Section 4.1 we discuss how to eliminate the factor of protocol choice by using the idea of a full-information protocol. A full-information protocol serves as an all-purpose protocol for a given system, so that by fixing the system, the protocol is also determined. In Section 4.2 we introduce the concept of a generic distributed problem. We will show that by choosing a single (generic) problem assumption to be used in all of our problem specifications, we can avoid building a Kripke structure of a system which is dependent on a particular problem. Various problem requirements can then be used to delineate taxonomic classes.

4.1 Using Full-Information Protocols

The all-purpose protocol, native to a given system, is known as its full-information protocol. A full-information protocol requires that each agent send all of its information to all other agents as often as possible. Since all available information is exchanged whenever possible, this protocol effects knowledge transfer at least as fast as any other protocol. As discussed in [PSL80, DM86, MT88, Mic89], if a system cannot solve a problem when running a full-information protocol, it will not be able to solve that problem by running any other protocol. We can, therefore, make a fair capabilities-based comparison of two systems by allowing each to run its own full-information protocol. A high-level representation of a full-information protocol looks like this:

repeat at each opportunity
    send copy of state to every agent
    receive copy of state from every agent
until false

For a synchronous system, "each opportunity" comes once every round, but to avoid excessive message-passing in an asynchronous system, the protocol must restrict the opportunities to send so that an agent sends a message only after it has received messages from other agents.

In the literature, full-information protocols are usually assumed to be non-terminating. This approach is convenient for theoretical discussions, since problem-specific stopping conditions need not be determined. For technical reasons, we must require our protocols to terminate: When running a full-information protocol, an agent begins by broadcasting the facts (A) that it knows. Later, after receiving information from other agents, it broadcasts the facts that everyone knows (EA); the next time, it broadcasts facts that everyone knows that everyone knows (EEA), and so on. This requires that an agent not forget anything while the protocol is running–knowledge can only accumulate. In [HV89], Halpern and Vardi show that a logic for a system with no forgetting is undecidable. Since we want our approach to be mechanizable, we must avoid the assumption of unbounded memory, and, therefore, non-terminating protocols.

Since we want a full-information protocol to be dependent only on the system chosen, we must impose a problem-neutral stopping condition (i.e., termination of a run of the full-information protocol for a given system is not based on any problem requirement.) There are two possible choices for this condition:

- a *state count condition* which is defined by the number of states which the system has entered since the protocol started executing (i.e., the "halt the loop after it has run for a month" approach), or

- a *knowledge level condition* which is based on agents reaching a certain state of knowledge.

The advantage to the state count condition is its problem-neutrality: the protocol depends only on the system, and a (large!) bound l on the number of system states. The disadvantage is that a system which could solve a given problem using more than l states will not be distinguished from a weaker system which could never solve the problem.

The knowledge level condition seems more natural, since we are primarily interested in the knowledge state of the system agents. With this type of stopping condition, we might, for example, halt the protocol when EEA holds for some fact A. Unfortunately, a problem requiring that EEEA hold cannot be solved by a protocol which stops after EEA holds. It is also possible that, for some system, EEA will never hold. The knowledge-state bounded full-information protocol for that system would never terminate.

Since we are concerned with the problem-neutrality of a protocol and must have a protocol that halts, we choose a state count stopping condition. With a choice of a large l, we are unlikely to mis-identify a system as unable to solve the given problem.

After l system states have passed, the protocol will stop (e.g., the primitive proposition "protocol-end" will be true) and agents will cease to send messages to one another. We have the requirements that:

$$\text{“}Clock_i = 0\text{”} \rightarrow \oplus^l \text{“}protocol - end\text{”}$$

and

$$\text{“}protocol - end\text{”} \rightarrow E\text{“}protocol - end\text{”}$$

Agents need not remember protocol information after the protocol stops, so we can moderate our definition of stability and consequent memory requirements. In Chapter 3, we said that a fact is stable if, once it holds, it holds forever after. We now assume that a fact is stable if it holds until the currently running protocol terminates, i.e.,

$$stable(A) \stackrel{def}{\equiv} A\mathcal{U}\text{“}protocol - end\text{”}$$

A stable fact generally serves a special purpose in a protocol–few facts are stable, and those are often intentionally stabilized by methods like timestamping. Therefore, once an agent knows a stable fact, it is usually bound to remember it for the duration of the protocol run. We will make the requirement that:

$$K_i stable(A) \rightarrow stable(K_i A)$$

These modifications allow us to write bounded full-information protocols for specific systems. A run of one of these protocols will terminate after the system enters its l^{th} state. In

Sections 4.1.1 and 4.1.2, we discuss full-information protocols for the two systems, CSF and CAN, introduced in Chapter 3.

4.1.1 A Full-Information Protocol for CSF

We will first consider a full-information protocol for a system with synchronous communication (CSF, described in Section 3.4.) In CSF, an external clock pulse, felt by every agent, ticks through rounds. Messages do not cross rounds, i.e., a message is received in the round in which it was sent. Crash failures are possible.

CSF 9	$\text{"}Round = x\text{"} \rightarrow K_i\text{"}Round = x\text{"}$
CSF 10	$\ominus(\text{"}Clock_i = a\text{"} \wedge \text{"}Round = x\text{"}) \rightarrow$ $K_i(\text{"}Clock_i = a\text{"} \rightarrow \text{"}Round = x\text{"})$
CSF 11	$sent(i, j, [F]A) \wedge \text{"}Round = x\text{"} \rightarrow$ $(\text{"}Round = x\text{"} \mathcal{U} received(i, j, [F]A))$

To implement a full-information protocol in CSF, we must ensure that agents exchange information about their states during each round. This can be accomplished by dividing each round into sending and receiving phases, allowing agents to enter the phases independently. An agent determines that a new round has begun by checking whether the current round number is the same as that at the last time on its internal clock. When a new round begins, the agent enters the sending phase of the protocol:

$$((\text{"}Clock_i = a\text{"} \wedge \text{"}Round = x\text{"}) \wedge$$

$$(\Box\text{"}Clock_i = a - 1\text{"} \rightarrow \text{"}Round = x - 1\text{"})) \rightarrow$$

$$(\neg\text{"}receive - phase_i\text{"} \wedge \text{"}send - phase_i\text{"} \vee \text{"}failed_i\text{"})$$

During the sending phase of the protocol, an agent's state (i.e., all stable facts that it knows) is sent to all other agents. The receiving phase follows immediately:

$$(\text{"}send - phase_i\text{"} \wedge \neg\text{"}protocol - end\text{"}) \rightarrow$$

$$(K_i stable(A_i) \rightarrow (sent(i, j, A_i) \wedge$$

$$\oplus((\neg\text{"}send - phase_i\text{"} \wedge (\text{"}receive - phase_i\text{"} \mathcal{U} \text{"}send - phase_i\text{"}))$$

$$\vee \text{"}failed_i\text{"})))$$

During the receiving phase, any messages which have arrived will be received, barring the failure of the recipient:

$$\text{“}receive - phase_j\text{”} \rightarrow$$

$$(arrived(i,j,A_i) \rightarrow \langle F \rangle (received(i,j,A_i) \vee \text{“}failed_j\text{”}))$$

The receiving phase ends when the next round begins, and the agent enters the sending phase. Table 4.1 shows the operational formulas for the full-information protocol for CSF. Each formula should be preceded by the C and $\Box$ operators, because it should be common knowledge that the formula always holds.

Table 4.1: Operational Description of the F-I-CSF Protocol

F-I-CSF 1	$\text{“}Clock_i = 0\text{”} \rightarrow \oplus^l \text{“}protocol - end\text{”}$
F-I-CSF 2	$\text{“}protocol - end\text{”} \rightarrow E\text{“}protocol - end\text{”}$
F-I-CSF 3	$K_i stable(A) \rightarrow stable(K_i A)$
F-I-CSF 4	$((\text{“}Clock_i = a\text{”} \wedge \text{“}Round = x\text{”}) \wedge$ $\Box(\text{“}Clock_i = a - 1\text{”} \rightarrow \text{“}Round = x - 1\text{”})) \rightarrow$ $(\neg\text{“}receive - phase_i\text{”} \wedge \text{“}send - phase_i\text{”} \vee \text{“}failed_i\text{”})$
F-I-CSF 5	$(\text{“}send - phase_i\text{”} \wedge \neg\text{“}protocol - end\text{”}) \rightarrow$ $(K_i stable(A_i) \rightarrow (sent(i,j,A_i) \wedge$ $\oplus((\neg\text{“}send - phase_i\text{”} \wedge (\text{“}receive - phase_i\text{”} \mathcal{U} \text{“}send - phase_i\text{”}))$ $\vee \text{“}failed_i\text{”})))$
F-I-CSF 6	$\text{“}receive - phase_j\text{”} \rightarrow$ $(arrived(i,j,A_i) \rightarrow \langle F \rangle (received(i,j,A_i) \vee \text{“}failed_j\text{”}))$

By following the approach that we used in Table 3.4, we could show that for any choice of k (where k is the number of rounds of knowledge exchange) CSF and F-I-CSF can be described non-operationally by the formula:

$$\mathrm{C}\Box(K_i stable(A_i) \rightarrow \langle F \rangle (stable(E^k A_i) \vee \text{“}failed_1\text{”} \vee \text{“}failed_2\text{”}))$$

Of course, once l (the limit on the number of system states) has been chosen, k is also fixed. Note that in any given round, there are $O(n^2)$ system states: one state in the sending phase (knowledge is not increased by a message send) and at most n^2 in the receiving phase (each of the n agents may either receive a message from each of the others or crash; agents that crashed in earlier rounds will do neither.)

4.1.2 A Full-Information Protocol for CAN

We can now consider a full-information protocol for a system with asynchronous communication (CAN, described in Section 3.5.) In CAN, there are no link or agent failures to cause problems, but also no external clock pulses to synchronize communication rounds. With no external pulse, we must use some internal event to begin the first sending phase. When an agent's internal clock reads zero, it is in a sending phase:

$$\text{“}Clock_i = 0\text{”} \rightarrow (\neg\text{“}receive-phase_i\text{”} \wedge \text{“}send-phase_i\text{”})$$

As in the protocol for CSF, an agent in its sending phase sends everything it knows to all other agents and then enters its receiving phase:

$$(\text{“}send-phase_i\text{”} \wedge \neg\text{“}protocol-end\text{”}) \rightarrow$$

$$(K_i stable(A_i) \rightarrow (sent(i,j,A_i) \wedge$$

$$\oplus(\neg\text{“}send-phase_i\text{”} \wedge (\text{“}receive-phase_i\text{”} \mathcal{U} \text{“}send-phase_i\text{”})))$$

An agent may begin receiving messages once a new message has arrived from each of the other agents. Note that since received(i, j, [F]A) is treated as a primitive proposition (rather than an imperative), once received(i, j, [F]A) holds, it will continue to hold. This challenges us to distinguish in our specification a newly arrived message from an older message which has been received. In the following formula, we identify a new message as one which has arrived, but has not previously been received:

$$(arrived(i,j,A_i) \wedge (\neg\langle P\rangle received(i,j,A_i))) \rightarrow \text{“}new-msg_{i,j}\text{”}$$

When an agent is in its receiving phase and a new message has arrived from all other agents, it receives all new messages and shifts into its sending phase:

$$(\text{“}receive-phase_j\text{”} \wedge$$

$$(\text{“}new-msg_{1,j}\text{”} \wedge \text{“}new-msg_{2,j}\text{”} \wedge \cdots \wedge \text{“}new-msg_{n,j}\text{”})) \rightarrow$$

$$(((arrived(i,j,A_i) \wedge \neg\langle P\rangle received(i,j,A_i)) \rightarrow received(i,j,A_i)) \wedge$$

$$\oplus(\neg\text{“}receive-phase_j\text{”} \wedge \text{“}send-phase_j\text{”}))$$

Table 4.2 shows the operational formulas for the full-information protocol for CAN. As before, the C and $\Box$ operators should be added in front of each of the formulas.

We should note that the full-information protocol for CAN is not as strict as that for CSF. While running the protocol for CAN, one agent could be in its receiving phase while another was in its sending phase. This is possible due to the potential variance in link speeds and the rate at which a receiving agent processes its queue of arriving messages. It is not possible for agents to differ by more than one phase, however, since each must receive a message from every other before passing into the next sending phase.

Table 4.2: Operational Description of the F-I-CAN Protocol

F-I-CAN 1	"$Clock_i = 0$"$\rightarrow \oplus^l$"$protocol - end$"
F-I-CAN 2	"$protocol - end$"$\rightarrow E$"$protocol - end$"
F-I-CAN 3	$K_i stable(A) \rightarrow stable(K_i A)$
F-I-CAN 4	"$Clock_i = 0$"$\rightarrow (\neg$"$receive - phase_i$"$\wedge$"$send - phase_i$"$)$
F-I-CAN 5	$($"$send - phase_i$"$\wedge\neg$"$protocol - end$"$) \rightarrow$ $(K_i stable(A_i) \rightarrow (sent(i,j,A_i)\wedge$ $\oplus(\neg$"$send - phase_i$"$\wedge($"$receive - phase_i$"$\mathcal{U}$"$send - phase_i$"$)))$
F-I-CAN 6	$(arrived(i,j,A_i) \wedge (\neg\langle P\rangle received(i,j,A_i))) \rightarrow$ "$new - msg_{i,j}$"
F-I-CAN 7	$($"$receive - phase_j$"$\wedge$ $($"$new - msg_{1,j}$"$\wedge$"$new - msg_{2,j} \wedge \cdots \wedge$ "$new - msg_{n,j}$"$)) \rightarrow$ $(((arrived(i,j,A_i) \wedge \neg\langle P\rangle received(i,j,A_i)) \rightarrow received(i,j,A_i))\wedge$ $\oplus(\neg$"$receive - phase_j$"$\wedge$"$send - phase_j$"$))$

As for CSF, we could show that for any choice of k (where k is the number of rounds of knowledge exchange) CAN and F-I-CAN can be described non-operationally by the formula:

$$\mathrm{C}\Box(K_i stable(A_i) \rightarrow \langle F\rangle stable(E^k A_i))$$

To write full-information protocols for other systems, we would proceed as we did for CSF and CAN: establish sending and receiving phases for exchanging state information, delineate the points at which an agent moves from one phase into the other, and synchronize the phase changes with either hardware or software induced rounds. The rounds force agents' level of knowledge to increase at a uniform rate, so that all the agents have approximately the same level of knowledge when the protocol terminates.

4.2 Generic Distributed Problems

A Kripke structure for modeling a distributed system will be constructed from the formulas of the system specification, full-information protocol specification and a problem assumption. If we analyze a single system with regard to its ability to solve a variety of problems, and each problem has a different assumption, then we must generate a new system model (Kripke structure) for each problem. If, instead, we choose a standard problem assumption, for which all system knowledge transfer will begin, then we need only generate a single system model for a given distributed system. In this section, we introduce the concept of the generic distributed problem and show how it allows us to use a standard problem assumption in our system models. This approach will allow us to specify taxonomic classes by problem requirements.

Many problems in distributed systems are considered to be *data independent*, which means that the values of the input data do not determine the system or protocol behavior (see [Wol86] for a discussion of data independence). For example, a message passing system often does not understand or act on the particular messages that it passes. A generic distributed problem is one whose specification is data independent. The specification will show how knowledge is to be transferred, but it is not specific about the facts which are known.

Let's take an example with a common assumption: each agent holds a piece of stable information and if any agent were able to collect all of those pieces of information, then it would be able to deduce a particular result. In our specification language, this statement appears as:

generic assumption: $stable(K_1 A_1) \wedge stable(K_2 A_2) \wedge \ldots \wedge stable(K_n A_n) \wedge$
$C\Box(A_1 \wedge A_2 \wedge \ldots \wedge A_n \rightarrow R)$

where each agent begins with a stable fact A_i, and it is common knowledge that the facts $A_1, \ldots, A_n$ together imply result R. One generic problem differs from another in its requirements on knowledge transfer. We might have the following requirement:

generic requirement: $\langle$F$\rangle$ER

In this problem, the object is for all the agents to eventually know R. An instance of this problem might arise in a replicated database system. Each agent may handle some transactions individually, but eventually, all copies of the database must be updated to provide a consistent view of the database.

A generic problem specification can be viewed as a template from which related, specific problem specifications can be produced by replacing generic facts with problem-specific facts. The generic problem specification above, for example, hints at a replicated database problem. The generic specification identifies the class of specific problems which can be produced by substituting facts in the template. This will allow us to focus on the knowledge transfer requirements of data independent problems. For the remainder of this chapter, we limit our attention to generic problems.

4.2.1 Choosing a Standard Generic Problem Assumption

By establishing a standard generic problem assumption for all distributed problems, a single Kripke structure can model a distributed system (with its full-information protocol.) Once the model is generated, the system can be studied in light of a variety of generic problem requirements with no additional modeling overhead.

Our choice of standard generic assumption will, of course, affect the type of generic problems which can be specified. We are most interested in problems classified as *fixed point computations*, where knowledge is accumulated via a full-information protocol up to a point where some knowledge transfer problem is solved. Problems commonly solved by fixed point computation include leader election, termination detection and various flavors of consensus (see Flon and Suzuki, [FS78].) Our standard generic assumption is convenient for specifying these types of problems, but not well-suited for specifying others. In particular, *reactive systems* ([Pnu86]), in which the goal is to maintain some invariant such as mutual exclusion, consistency of duplicated data, etc., are not specified using our generic assumption. This does not mean that reactive problems cannot be specified in $\mathcal{L}$, only that the assumption is likely to differ for each reactive problem (i.e., no standard assumption can be chosen). This would require additional system modeling overhead for each problem considered, since a Kripke structure will be formed based on system, protocol and problem assumption specifications.

For our standard, we choose the generic assumption discussed in our replicated database example, since it specifies only that certain stable facts are known by individual agents and that all agents are capable of making a deduction, given those facts.

standard generic assumption:

$$stable(K_1A_1) \wedge stable(K_2A_2) \wedge \ldots \wedge stable(K_nA_n) \wedge$$
$$C\Box(A_1 \wedge A_2 \wedge \ldots \wedge A_n \rightarrow R)$$

4.2.2 Example Fixed Point Problem Specifications

We can now discuss some fixed point problems and their specifications based on our standard generic assumption.

Leader Election

In the leader election problem, each agent begins with a vote (perhaps its own i.d. number.) The votes circulate through the system and, eventually, a winner is determined. All of the agents should know who the winner is. In our standard generic problem assumption, each of the A_i's should be interpreted as an individual vote and R should be interpreted as the election result, probably of the form "Winner is $Agent_i$." The generic problem requirement can be stated in $\mathcal{L}$ as:

$$\langle \mathrm{F} \rangle \mathrm{ER}$$

which says that at some time in the future, everyone knows the election result. This requirement can be made stronger by changing the time and/or the knowledge component, for example:

$$\oplus^5 \mathrm{CR}$$

A weaker requirement probably would not be accepted as a variant of the election problem, as a leader of unknown identity would be unable to perform important coordination tasks for centrally controlled distributed computations.

Termination Detection

The specification for the termination detection problem looks similar. In this case, we might require only that someone know that the underlying computation has terminated, allowing that agent to report to the external user. This gives us a generic problem requirement of:

$\langle \mathrm{F} \rangle \mathrm{SR}$

Consensus

The consensus problem requires that common knowledge of a decision be reached. The weaker version specifies that this common knowledge be reached eventually, while stronger versions might give a time bound. The weak generic specification looks like this:

$\langle \mathrm{F} \rangle \mathrm{CR}$

Based on knowledge transfer requirements, election, termination detection and consensus are closely related. It is important to note that this does not give us an indication of the amount of work (messages to be passed, write locks on shared locations, etc.) necessary to solve the problems. In fact, the amount of work required to solve a given problem may vary drastically from one system to another: consensus is trivially solved in a no-fault shared memory system, but impossible in a Byzantine message-passing system. Our specification and modeling approach to classifying distributed systems does not help us to estimate the work required to achieve a particular knowledge transfer, it only allows us to determine whether or not a given system with its full-information protocol can accomplish that transfer.

4.2.3 An Example Reactive System Problem Specification

Now that we have seen how fixed-point problems can be specified using our standard generic problem assumption, we will discuss a reactive system problem and show why this type of problem is not so conveniently handled by our approach. Remember that a reactive system problem such is one in which the system maintains an invariant. In these problems, generally no assumptions are made about the initial knowledge state of the system and the problem is solved if the invariant (i.e., problem requirement) is maintained. This differs from our fixed point knowledge transfer problems in that we are not concerned with the specific knowledge transferred, but with the set (or sets) of knowledge to be maintained.

Mutual exclusion is a common reactive system problem. We expect two properties to hold in a system with mutual exclusion: only one agent may be in its critical section (i.e., have access) at any given time, and an agent that wants access to its critical section cannot be excluded forever. These requirements can be expressed in $\mathcal{L}$ as:

Exclusivity: $\Box\neg(K_i$"$Agent_i$ has access" $\wedge$ "$Agent_j$ has access"), for $i \neq j$

Fairness: $\Box(K_i$"$Agent_i$ wants access" $\rightarrow \langle \mathrm{F} \rangle K_i$"$Agent_i$ has access")

The exclusivity and fairness requirements together make up the problem requirement for mutual exclusion, and the problem assumption is empty. Any mechanisms necessary for achieving the mutual exclusion should be specified in the list of system formulas, since only the full-information protocol will be used for knowledge transfer. The system, with built-in mutual exclusion constraints, will look like a system with mutual exclusion. A system with constraints on fair message delivery will look like a fair system. In general, systems which are expected to solve reactive problems will appear as specifically reactive systems.

So, $\mathcal{L}$ can be used to specify reactive systems and problems and to model these systems by the methods discussed in Chapter 5, but the results are not particularly interesting. In reactive systems problems, knowledge-transfer is not a primary issue, and because the systems are specifically designed for solving these problems, it is not helpful to classify the systems based on the problems that they can solve. In studying systems with respect to fixed point problems, we can use specifications and modeling to help us consider common, general-purpose systems which are not constructed for solving specific problems. It may then be useful to classify systems based on the types of problems that they can solve.

4.3 Summary

In this chapter, we have discussed the use of full-information protocols and generic fixed point problems to help us isolate distributed systems for classification. A full-information protocol is tied to the system in which it runs, effectively eliminating protocol choice as a factor in system classification. The use of a standard generic problem assumption allows us to consider a system's ability to solve a variety of problems with only one model of the system and its full-information protocol. In the next chapter, we discuss how to mechanically create a Kripke structure for modeling a distributed system, given its system and full-information protocol formulas and the standard generic problem assumption. The model can then be used to either directly or indirectly classify the system based on the distributed problems that it can solve.

Chapter 5

Building and Comparing Kripke Models

In this chapter, we show how to produce a formal model of a distributed system (a Kripke structure) and how to compare two such models. In Section 5.1, we discuss how to build a Kripke structure, given a system, protocol and problem specification. These specifications are written in $\mathcal{L}$ (provided in Chapter 3) and the effects of the protocol and problem choices are neutralized by the use of full-information protocols and generic problem specifications (discussed in Chapter 4.) We also look at Kripke structures modeling our example systems, CSF and CAN. In Section 5.2, we discuss bisimulations (functions from one structure to another) which allow us to formally compare distributed systems.

5.1 Building Kripke Models of Distributed Systems

Let Λ be the logic formed from L , together with a set of specification formulas for a system, its full-information protocol, and the standard generic problem assumption. Then any formulas deducible from Λ are problem requirements (interesting or not) which are met by the system. Our Kripke structure for representing the system must "match" Λ, in that the formulas which are deducible from Λ must be exactly those which are valid in the Kripke structure. I.e., the Kripke structure $\mathcal{M}$ must *model* the formulas of Λ, written $\mathcal{M} \models \Lambda$. Because of this requirement, we refer to this structure as the *Kripke model* of Λ. To construct the Kripke model of Λ, we first determine its states, then add $\sim_i$ and *Next* relations between the states, and, finally, provide a valuation function to give truth values to propositional letters at each state in the model.

A state in a Kripke structure represents a logically possible world. Each state must be logically consistent, so, for example, it is not possible for both of the propositions "it is raining" and "it is not raining" to hold true at a state. Note that our language $\mathcal{L}$ is countably infinite, so the number of formulas which might hold true at a given state is also countably infinite. If we list all possible consistent sets of formulas, we get the set containing all of the states. This defines the universe of logical possibility. Note that this set is uncountably infinite. We cannot hope to be able to produce this Kripke model and check it (i.e., potentially visit all of its states) mechanically.

To reduce the number of states in our Kripke model, we make the assumption that many formulas are uninteresting and can be excluded from the model without compromising its utility. To do this, we start with a set γ of "interesting" formulas. For purposes of classifying distributed systems by their problem-solving capabilities, γ would normally consist of problem requirement specifications for a variety of distributed problems. Consistent sets of formulas derived from γ will serve as states in our Kripke model. Thus, we will only be modeling those parts of Λ which are relevant to γ. Once the Kripke model is constructed, we should be able to determine whether a formula holds at a given state by simply looking at the formulas in the state. We call the resulting model the *γ-relevant Kripke model of* Λ. Much of our notation is borrowed from [HM].

We begin by defining the pool of formulas from which the states of our restricted Kripke model are constructed and considering the size of that pool. Definition 8 collects all the formulas which are relevant to γ—all subformulas and those formulas which, by the semantics of the knowledge and temporal operators, are related to subformulas of γ.

Definition 8 *Let* Sub*(γ) be the set of subformulas of γ, together with the formulas*

1. *$E(A \wedge CA)$, $A \wedge CA$, $K_1(A \wedge CA), \ldots, K_n(A \wedge CA)$, for each subformula CA of γ,*
2. *$K_1A, \ldots, K_nA$, for each subformula EA of γ,*
3. *$\langle F\rangle A_2$, $\oplus A_1$, $\oplus A_2$, and $\oplus(A_1 \mathcal{U} A_2)$, for each subformula $A_1 \mathcal{U} A_2$ of γ,*
4. *$\langle P\rangle A_2$, $\ominus A_1$, $\ominus A_2$, and $\ominus(A_1 \mathcal{S} A_2)$, for each subformula $A_1 \mathcal{S} A_2$ of γ,*

Let *length*(γ) be the length of γ over the alphabet $P \bigcup \{\neg, \wedge,), (, K_i, \mathrm{C}, \oplus, \ominus, \mathcal{U}, \mathcal{S}\}$. Then *length*$(\gamma)$ is an upper bound on the number of subformulas of γ (see Section C.3 for a proof sketch.) Note that Definition 8 forms the set *Sub*(γ) by extending the set of subformulas of γ. The maximum number of additional formulas are included when CA is a subformula of γ (see Clause 1 of Definition 8): the n formulas beginning with K, and two others. This gives us that $|Sub(\gamma)| \leq (n+3) length(\gamma)$.

Definition 9 enlarges $Sub(\gamma)$ to $Sub^+(\gamma)$ by including the negations of formulas in $Sub(\gamma)$. This gives us that $|Sub^+(\gamma)| \leq 2(n+3) length(\gamma)$.

Definition 9 *Let* $\mathrm{Sub}^+(\gamma)$ *consist of all formulas of* $\mathrm{Sub}(\gamma)$ *and their negations.*

Definitions 10 and 11 provide the rules by which a state in the γ-relevant Kripke model will be formed. The formulas at each state should be consistent with respect to Λ, so that no "impossible worlds" are formed.

Definition 10 *A set of formulas* $\Delta \subseteq \mathrm{Sub}^+(\gamma)$ *is* Λ-consistent *iff* $\Delta \not\vdash_\Lambda$ *false.*

The state should also be maximal, in that for each formula A from $Sub^+(\gamma)$, either A or $\neg A$ must appear in the state. This ensures that the truth or falsity of every "interesting formula" is clear at each state.

Definition 11 *A set of formulas* $\Delta \subseteq \mathrm{Sub}^+(\gamma)$ *is* Λ-maximal *iff*

1. Δ *is* Λ*-consistent, and*
2. *For any* $A \in \mathrm{Sub}^+(\gamma)$*, either* $A \in \Delta$ *or* $\neg A \in \Delta$.

The Λ-maximal subsets of $Sub^+(\gamma)$ will serve as states for our Kripke model. Definition 12 gives the name $Max(\gamma)$ to the collection of these subsets.

Definition 12 *Let* $\mathrm{Max}(\gamma)$ *be the set of all* Λ*-maximal subsets of* $\mathrm{Sub}^+(\gamma)$.

Since $Max(\gamma)$ is smaller than the power set of $Sub^+(\gamma)$, we have that $|Max(\gamma)| \leq 2^{2(n+3)length(\gamma)}$.

Now that we know how the states of our model will be formed, we can discuss how to produce the valuation function V_γ and the *Next* and $\sim_i$ relations. We want to be able to determine which formulas are true at a given state in the model by simply looking at the set of formulas at that state. If the formula is in the set, it is true at that state; if the formula is not in the set, it is false at that state. The valuation function defines the truth value for propositional letters in exactly this way. The definitions of the *Next* and $\sim_i$ relations must also support this state-lookup approach in order to correctly implement the semantics of the temporal and knowledge operators. Definition 13 describes the γ-relevant Kripke model $\mathcal{M}_\gamma$ of a logic Λ.

Definition 13 *The* γ*-relevant Kripke model of* Λ *is*

$$\mathcal{M}_\gamma = (\mathrm{S}_\gamma, \mathrm{Next}_\gamma, \sim_{1,\gamma}, \sim_{2,\gamma}, \ldots, \sim_{\mathrm{n},\gamma}, \mathrm{V}_\gamma),$$

where

1. $\mathrm{S}_\gamma = \mathrm{Max}(\gamma)$,
2. *For all* $\mathrm{s}, \mathrm{t} \in \mathrm{S}_\gamma$ *and* $A \in \mathrm{Sub}^+(\gamma)$, $(\mathrm{s}, \mathrm{t}) \in \mathrm{Next}_\gamma$ *iff* $\oplus A \in \mathrm{s}$ *implies* $A \in \mathrm{t}$,
3. *For all* $\mathrm{s}, \mathrm{t} \in \mathrm{S}_\gamma$ *and* $A \in \mathrm{Sub}^+(\gamma)$, $(\mathrm{s}, \mathrm{t}) \in \sim_{i,\gamma}$ *iff* $K_i A \in \mathrm{s}$ *implies* $A \in \mathrm{t}$,
4. *For all* $\mathrm{s} \in \mathrm{S}_\gamma$*, and* $P_j \in \mathrm{P}$, $\mathrm{V}_\gamma(\mathrm{s}, P_j) =$ *true if* $P_j \in \mathrm{s}$*, and* $\mathrm{V}_\gamma(\mathrm{s}, P_j) =$ *false otherwise.*

Theorem 1 tells us that $\mathcal{M}_\gamma$ is constructed correctly–for any pre-defined "interesting" formulas, the true formulas are exactly those in the state. The proof for Theorem 1 can be found in Section C.5.

Theorem 1 *For* $A \in \mathrm{Sub}^+(\gamma)$ *and* $\mathrm{s} \in \mathrm{S}_\gamma$,

$$(\mathcal{M}_\gamma, \mathrm{s}) \models A \text{ iff } A \in \mathrm{s};$$

Note that formula A is true in model $\mathcal{M}_\gamma$, written $\mathcal{M}_\gamma \models A$, when it is true at every state in $\mathcal{M}_\gamma$. Corollary 1 is the result that we want: the "interesting" formulas deducible from Λ are exactly those true in $\mathcal{M}_\gamma$, so Λ and $\mathcal{M}_\gamma$ are equivalent formalisms. We then have the choice of looking for a deduction from Λ or building and checking the γ-relevant Kripke model.

Corollary 1 *$\mathcal{M}_\gamma$ determines Λ, i.e. for all $A \in Sub^+(\gamma)$,*

$$\mathcal{M}_\gamma \models A \quad \textit{iff} \quad \vdash_\Lambda A;$$

For a γ-relevant Kripke model, the complexity of model checking is linear in the number of states in the structure–if every state in the model contains the formula in question, then, indeed, the structure is a model of the formula. Of course, we pay a higher price at the time that the model is constructed. The cost of producing the model is dominated by the cost to check the consistency of each of the states in $Max(\gamma)$. The consistency of a state $\{f_1, f_2, \ldots, f_{|Sub^+(\gamma)|}\}$ can be checked by using a theorem-prover for L , adding the formulas of Λ and the formula $f_1 \wedge f_2 \wedge \cdots \wedge f_{|Sub^+(\gamma)|}$ as theorems, and then trying to derive a contradiction. Since L is sound and complete, theorem-proving is equivalent to validity-checking. In [HV89], Halpern and Vardi show that the complexity of the validity problem is EXPTIME-complete in formula length. Our formula, $f_1 \wedge f_2 \wedge \cdots \wedge f_{|Sub^+(\gamma)|}$, is bounded in size by $|Sub^+(\gamma)|$, which, in turn, is less than $2(n+3)length(\gamma)$. Thus, the complexity of checking the consistency of states in the model is in EXPTIME in the length of γ. Since this is the highest cost in producing the model, we also have that model production is in EXPTIME in the length of γ.

5.1.1 A Kripke Model for CSF

Definitions 8 through 13 give us a process for building the γ-relevant Kripke model of a logic Λ. In this section, we assume that Λ is based on CSF and walk through the process of building a model. The first step toward finding the γ-relevant Kripke model of CSF is to determine the set γ of "interesting" formulas. From the requirement of our example Problem 2 of Chapter 3, we choose γ to be the single formula:

$$\langle \mathrm{F} \rangle K_1 K_2 A_1$$

By choosing this γ, we will produce a Kripke model which can reliably address the truth or falsity of $\langle \mathrm{F} \rangle K_1 K_2 A_1$ and that of a few related formulas. Definition 8 says that to form the set $Sub(\gamma)$, we take the subformulas of γ:

$\langle \mathrm{F} \rangle K_1 K_2 A_1$ (which abbreviates $\mathtt{true}\ \mathcal{U} K_1 K_2 A_1$)
$K_1 K_2 A_1$
$K_2 A_1$
A_1

and add to these (by Clause 3) the formulas:

$\oplus K_1 K_2 A_1$
$\oplus \langle \mathrm{F} \rangle K_1 K_2 A_1$

$Sub^+(\gamma)$ is then formed (according to Definition 9 by adding the negation of these formulas:

$$\neg\langle \mathrm{F}\rangle K_1K_2A_1$$
$$\neg K_1K_2A_1$$
$$\neg K_2A_1$$
$$\neg A_1$$
$$\neg \oplus K_1K_2A_1$$
$$\neg \oplus \langle \mathrm{F}\rangle K_1K_2A_1$$

The next step is to find all of the Λ-maximal subsets of $Sub^+(\gamma)$ (i.e., $Max(\gamma)$). Remember that our CSF-based Λ consists of the logic, L , together with the non-operational specification of CSF (from Chapter 4):

$$\mathrm{C}\Box(K_i stable(A_i) \rightarrow \langle F\rangle(stable(E^k A_i) \vee \text{“}failed_1\text{”} \vee \text{“}failed_2\text{”}))$$

Each Λ-maximal subset will have six elements–either the positive or negative version of each formula of $Sub^+(\gamma)$ must appear in the set. At worst, we could have $2^6 = 64$ Λ-maximal subsets of $Sub^+(\gamma)$. In practice, we get only seven Λ-maximal sets. For example, note that the set:

$$A_1$$
$$\neg K_2A_1$$
$$K_1K_2A_1$$
$$\langle \mathrm{F}\rangle K_1K_2A_1$$
$$\oplus K_1K_2A_1$$
$$\oplus\langle \mathrm{F}\rangle K_1K_2A_1$$

is inconsistent, and, therefore, not Λ-maximal. By Axiom 2, it is not possible that both $\neg K_2A_1$ and $K_1K_2A_1$. The consistency requirement is essential because each Λ-maximal set will serve as a state or "possible world" in the Kripke model. An inconsistent set of formulas would be an "impossible world." $Max(\gamma)$ is shown in Figure 5.1 with numbers for identifying the states. Note that states 1 through 5 contain the formula $\langle \mathrm{F}\rangle K_1K_2A_1$, but states 6 and 7 contain its negation.

Now that we have the states of the γ-relevant model, we can add the *Next* and $\sim_i$ relations, according to the rules of Definition 13. Clause 2, for example, says that if for all the formulas $\oplus A_i \in s$ we also have that $A_i \in t$, then we should place a *Next* edge from s to t in our γ-relevant model. Figure 5.2 shows the CSF model with *Next* edges. Consider, for example, state 1, which includes the formula $\oplus K_1K_2A_1$. If there is to be a *Next*-edge from state 1 to another state, then that state must include $K_1K_2A_1$. State 5 is the only state of $Max(\gamma)$ which does include $K_1K_2A_1$. Since it also includes $\langle \mathrm{F}\rangle K_1K_2A_1$ (required by state 1's $\oplus\langle \mathrm{F}\rangle K_1K_2A_1$), we have a *Next*-edge from state 1 to state 5. By contrast, state 6

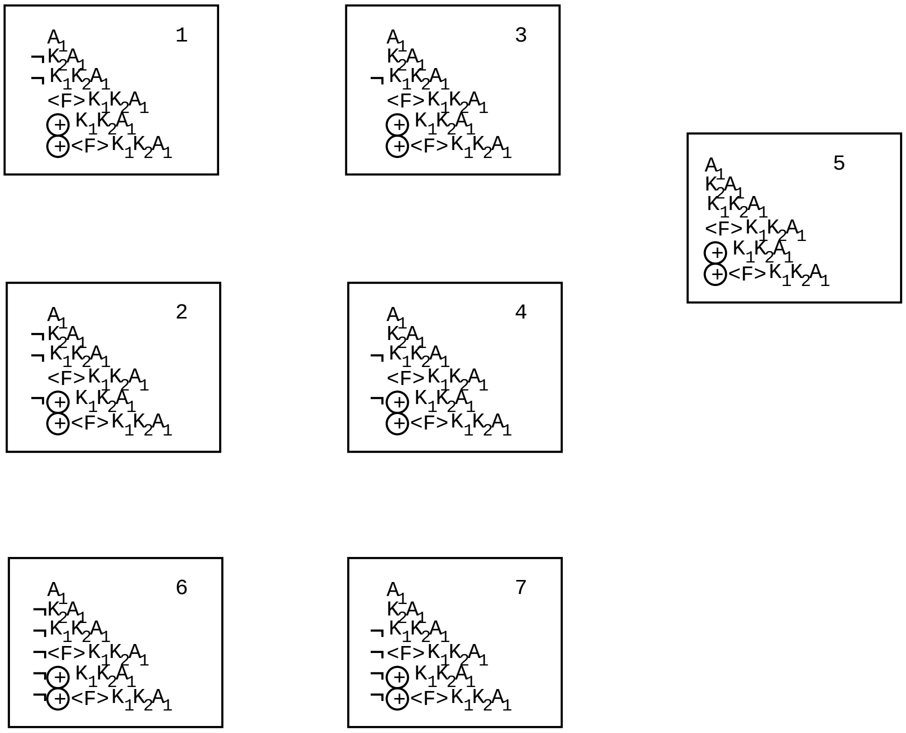

Figure 5.1: $Max(\gamma)$ for Λ based on CSF.

contains no formulas beginning with $\oplus$, so Clause 2 of Definition 13 is trivially satisfied by all other states of $Max(\gamma)$ and there are *Next*-edges from state 6 to every other state.

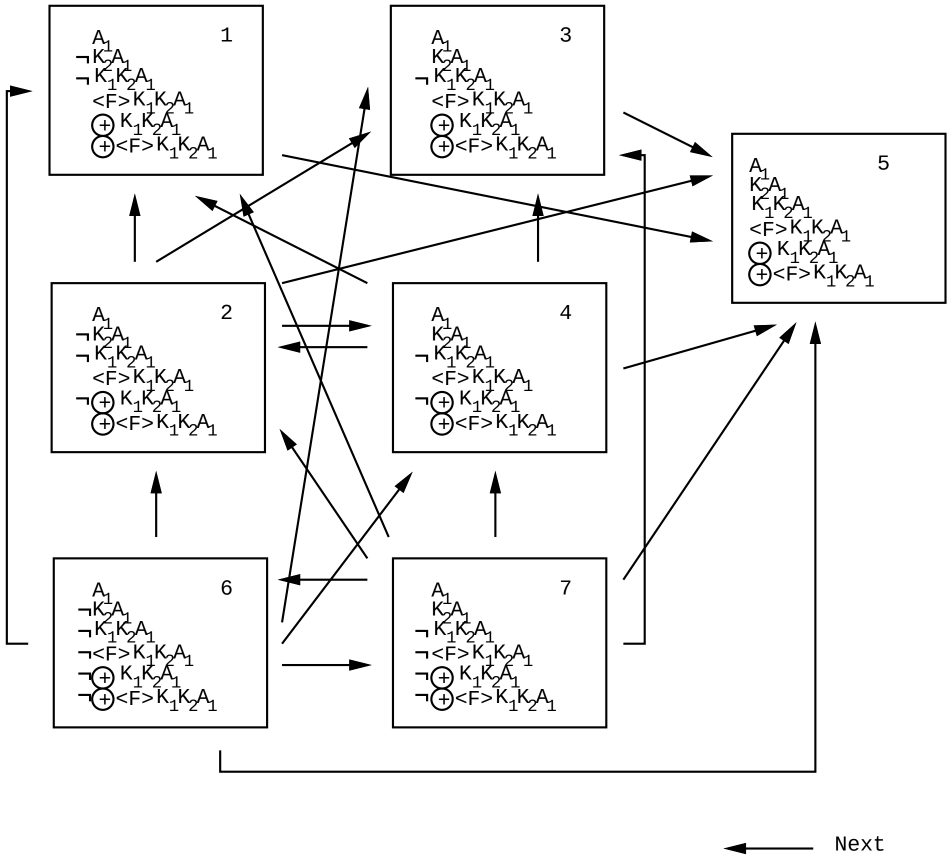

Figure 5.2: A γ-relevant Kripke model for CSF, showing only the *Next* relation.

We take a similar approach to adding $\sim_i$ edges. Clause 3 says that if for all the formulas $K_1A_i \in s$ we also have that $A_i \in t$, then we should add a $\sim_1$, edge from s to t to our model. Figure 5.3 shows the CSF model with the $\sim_1$ and $\sim_2$ edges. $Agent_1$ is unable to distinguish among states 1, 2, 3, 4, 6, and 7 (it has no knowledge in any of these), while $Agent_2$ is unable to distinguish among states 3, 4, 5 and 7 (it knows that A_1 holds in each of these). The γ-relevant Kripke model for CSF is represented by Figures 5.2 and 5.3. For readability, we have not combined the figures.

Let's look at the CSF model in some detail. As we would expect from the non-operational specification of CSF, there are some states in which the formula $\langle \mathrm{F} \rangle K_1K_2A_1$ holds, and some in which its negation holds. Thus, $\mathcal{M}_\gamma \not\models \langle \mathrm{F} \rangle K_1K_2A_1$. We would expect that in those states in which $\langle \mathrm{F} \rangle K_1K_2A_1$ does not hold, either "$failed_1$" or "$failed_2$" would hold. We do not see either of these formulas, however, because they were not included in the set γ of "interesting"

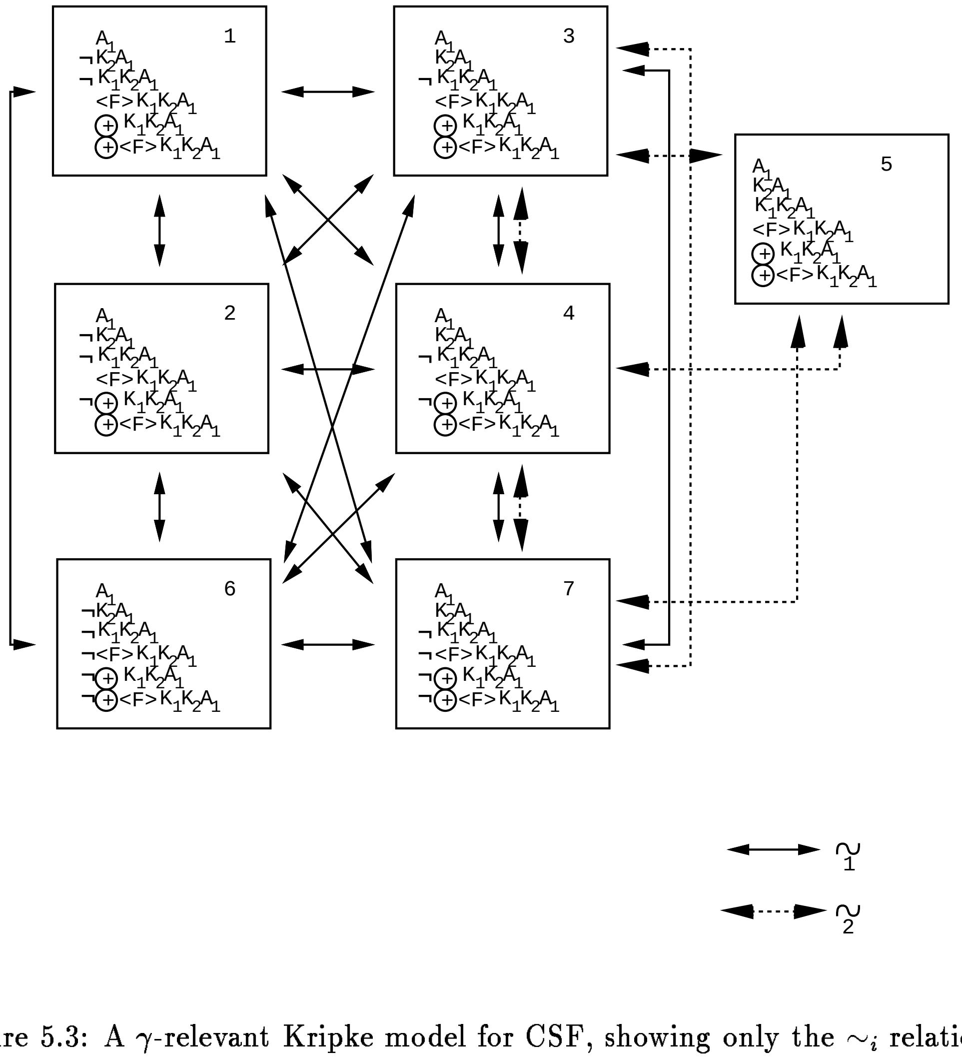

Figure 5.3: A γ-relevant Kripke model for CSF, showing only the $\sim_i$ relations.

formulas. Including the failure propositions would have produced several more states (up to $2^2 \cdot 7 = 28$, total), with a variety of potential failure combinations.

It might also seem odd that the *Next* relation seems to go "backward" at times; e.g., how can there be a *Next* edge from state 3 to state 1 when, in state 3, $Agent_2$ knows A_1, but in state 1, it doesn't know A_1? Why doesn't our CSF restriction to a stable A_1 prevent this? Edges are added to the model following the rules of Definition 13. By Clause 3, we only check formulas in the two states; we cannot somehow refer back to the formulas defining CSF. Since both states 1 and 3 are consistent with CSF, we want them in the model. To prevent such an edge, we would have to include the formula $stable(A_1)$ in γ. By providing γ, we specify not only which formulas are interesting, but also which edges are interesting. Note that the inclusion of this edge is not an error: Corollary 1 does not make promises about the Λ-deducibility of a formula which holds at an individual state, only about those formulas which hold at all states. Note also that the semantics of the operators $\oplus$ and $\langle \mathrm{F} \rangle$ address *all Next* paths, so unless all *Next* paths lead to states containing a particular formula, we cannot draw a conclusion about events in the future.

Given only γ and $\mathcal{M}_\gamma$ (based on CSF), by Corollary 1, the strongest statement that we can make is that $\mathcal{M}_\gamma \models A_1$, and, hence $\vdash_\Lambda A_1$. Of course, we can also see potential knowledge states, and, if we are careful to remember that the $\oplus$ and $\langle \mathrm{F} \rangle$ operators must be interpreted over *all* paths, then we can look at paths of knowledge transfer over time.

5.1.2 A Kripke Model for CAN

The process for building a γ-relevant model of CAN is the same as that for building the model of CSF. By using the same γ for both, we ensure that the models will be completely comparable. Max_γ for CAN is shown in Figure 5.4 . Since γ was chosen to be the same for both models, the potential members of $Max(\gamma)$ are also the same. Because CAN is more restrictive than CSF (failures are not permitted in CAN), fewer subsets of $Sub^+(\gamma)$ are consistent with CAN. Thus, Max_γ for CAN is a subset of Max_γ for CSF. In particular, states 6 and 7 of the CSF model do not appear in the CAN model. These states deny that $K_1K_2A_1$ will become true in the future. For CAN, the following hold: $\mathcal{M}_\gamma \models A_1$, $\mathcal{M}_\gamma \models \langle F \rangle K_1K_2A_1$, and $\mathcal{M}_\gamma \models \oplus\langle F \rangle K_1K_2A_1$. In a pair of systems less closely related than CSF and CAN, we would not expect the set of states of one model to be a subset of the set of states of the other, though the two models might have some states in common.

Figures 5.5 and 5.6 show the *Next* and $\sim_i$ edges for the CAN model. Compare these to Figures 5.2 and 5.3 for the CSF model. Since edges are determined only by formulas in the states of a model and, because the set of states of CAN is a subset of the set of states of CSF, the sets of *Next* and $\sim_i$ edges of CAN are subsets of the sets of *Next* and $\sim_i$ edges of CSF.

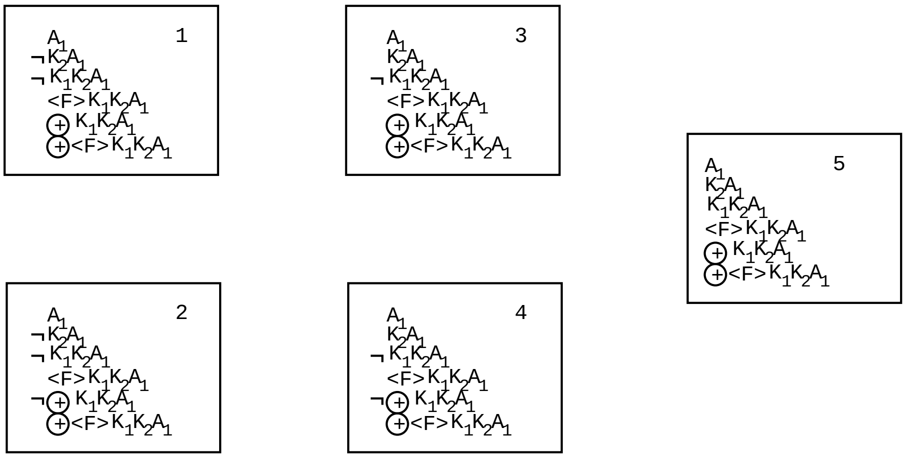

Figure 5.4: $Max(\gamma)$ for Λ based on CAN.

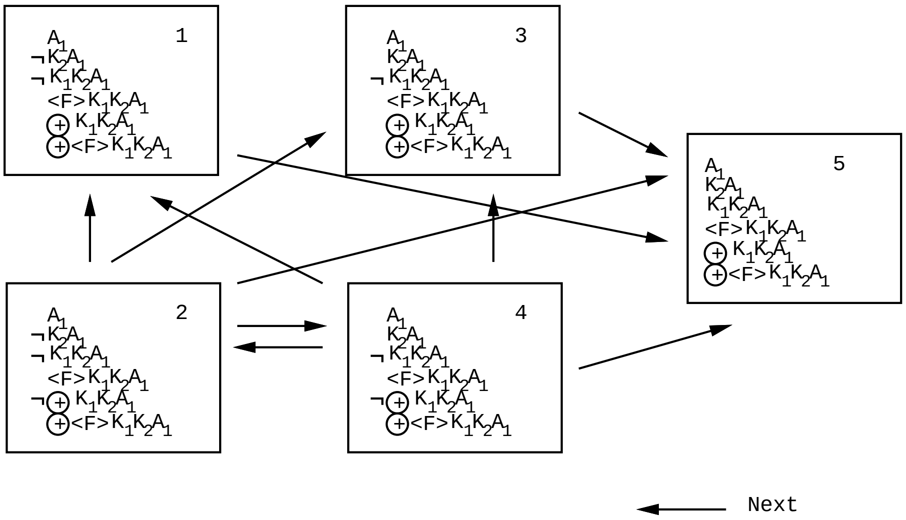

Figure 5.5: A γ-relevant Kripke model for CAN, showing only the *Next* relation.

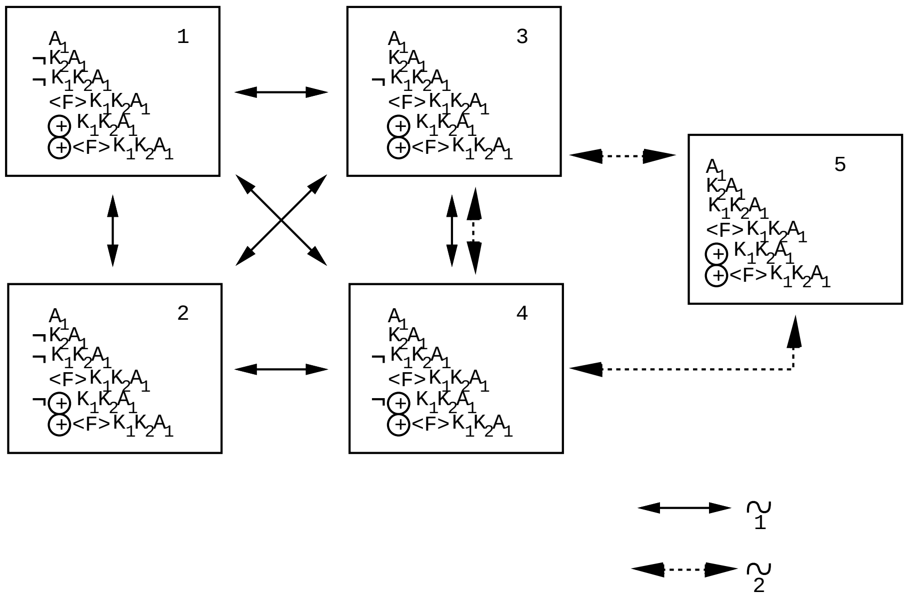

Figure 5.6: A γ-relevant Kripke model for CAN, showing only the $\sim_i$ relations.

5.2 Comparing Kripke Models

At one level, we do not need more apparatus for comparing one γ-relevant model to another. By checking all the states of a model, we can easily list those formulas of γ which hold at every state, and, by Corollary 1, the formulas which are deducible from Λ. This approach does not, however, help us to compare systems which may have some similarities, but which do not model exactly the same formulas of γ. In Sections 5.1.1 and 5.1.2, we found that, despite the many similarities between the CSF and CAN systems, of the formulas of γ, CSF is a model only of A_1, while CAN is a model of A_1, $\langle F\rangle K_1K_2A_1$ and $\oplus\langle F\rangle K_1K_2A_1$. We can easily see, however, that states 1 through 5, as well as the *Next* and $\sim_i$ edges are identical in the two models. In this section, we provide a mechanism for comparing models at the level of individual states.

Definition 14 defines a *bisimulation relation* from states in one model to states in another. If we can find a bisimulation from one model to the other, then Theorem 2 tells us that exactly the same formulas hold at the corresponding states.

Definition 14 *Let* $\mathcal{M}_1 = (\mathrm{S}_1, \mathrm{Next}_1, \sim_{1,1}, \sim_{2,1}, \ldots, \sim_{n,1}, \mathrm{V}_1)$ *and* $\mathcal{M}_2 = (\mathrm{S}_2, \mathrm{Next}_2, \sim_{1,2}, \sim_{2,2}, \ldots, \sim_{n,2}, \mathrm{V}_2)$ *be models,* $\mathrm{s}_1, \mathrm{t}_1 \in \mathrm{S}_1$ *and* $\mathrm{s}_2, \mathrm{t}_2 \in \mathrm{S}_2$*. A relation* $\mathrm{g} \subseteq \mathrm{S}_1 \times \mathrm{S}_2$ *is a* bisimulation *from model* $\mathcal{M}_1$ *to model* $\mathcal{M}_2$ *when the following conditions hold:*

1. *(a) If* $(\mathrm{s}_1, \mathrm{s}_2) \in \mathrm{g}$ *and* $(\mathrm{s}_1, \mathrm{t}_1) \in \mathrm{Next}_1$*, then there is a* $\mathrm{t}_2 \in \mathrm{S}_2$ *such that* $(\mathrm{s}_2, \mathrm{t}_2) \in \mathrm{Next}_2$ *and* $(\mathrm{t}_1, \mathrm{t}_2) \in \mathrm{g}$*,*

(b) *If* $(s_1, s_2) \in g$ *and* $(t_1, s_1) \in \mathrm{Next}_1$, *then there is a* $t_2 \in S_2$ *such that* $(t_2, s_2) \in \mathrm{Next}_2$ *and* $(t_1, t_2) \in g$,

(c) *If* $(s_1, s_2) \in g$ *and* $(s_1, t_1) \in \sim_{i,1}$, *then there is a* $t_2 \in S_2$ *such that* $(s_2, t_2) \in \sim_{i,2}$ *and* $(t_1, t_2) \in g$,

2. (a) *If* $(s_2, s_1) \in g$ *and* $(s_2, t_2) \in \mathrm{Next}_2$, *then there is a* $t_1 \in S_1$ *such that* $(s_1, t_1) \in \mathrm{Next}_1$ *and* $(t_2, t_1) \in g$,

   (b) *If* $(s_2, s_1) \in g$ *and* $(t_2, s_2) \in \mathrm{Next}_2$, *then there is a* $t_1 \in S_1$ *such that* $(t_1, s_1) \in \mathrm{Next}_1$ *and* $(t_2, t_1) \in g$,

   (c) *If* $(s_2, s_1) \in g$ *and* $(s_2, t_2) \in \sim_{i,2}$, *then there is a* $t_1 \in S_1$ *such that* $(s_1, t_1) \in \sim_{i,1}$ *and* $(t_2, t_1) \in g$,

3. *For all* $P_j \in \mathrm{P}$, $(s_1, s_2) \in g$ *implies* $V_1(s_1, P_j) = V_2(s_2, P_j)$.

Figures 5.7 and 5.8 illustrate Clauses 1 and 2 of Definition 14. Note that each of the three components (a, b and c) of the clauses begin with an assumption about two relations (shown with solid arrows) and ends with a statement about two relations (shown with dotted arrows).

Theorem 2 says that if there is a bisimulation edge pairing states from two models, then the same formulas hold at both states in the pair. A proof for Theorem 2 is given in Section C.6.

Theorem 2 *Let* g *be a bisimulation from model* $\mathcal{M}_1$ *to model* $\mathcal{M}_2$. *For any formula* A, *and states* $s_1 \in S_1$, *and* $s_2 \in S_2$, *if* $(s_1, s_2) \in g$, *then*

$$(\mathcal{M}_1, s_1) \models A \text{ iff } (\mathcal{M}_2, s_2) \models A.$$

We can now compare CAN to CSF, using a bisimulation relation g. The numbering of the states suggests the pairing from CAN to CSF shown in Figure 5.9. There are no g-edges from states of CAN to states 6 and 7 of CSF. This comparison is natural, since states 1 through 5 are the "no failure" states of CSF, and, of course, CAN does not permit failures.

Why do we bother with bisimulations? Since truth at a state is equivalent to set inclusion at that state (by Theorem 1), isn't it sufficient to simply match up identical states in the two models? When the sets γ of "interesting" formulas used to form the two models is identical, then there will be a (simple!) one-to-one mapping from one state to its twin in the other model and the bisimulation will be trivial. The situation is more interesting, however, when the γ sets for the two models are not identical. Of course, if the intersection of the two is empty, then no comparison of the models can be made, but if there are formulas in the intersection, there may be a g-edge from a state in the first model to multiple states in the second. In these cases, we can see collections of states in one model simulating collections of states in the other.

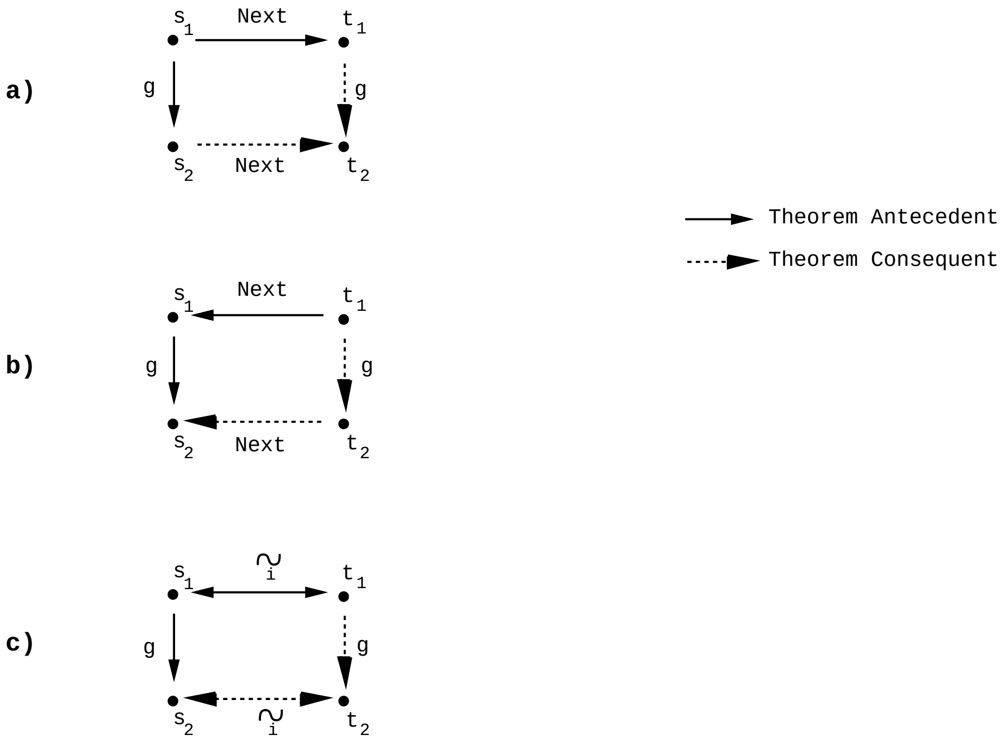

Figure 5.7: Clause 1 of Definition 14 (Bisimulation).

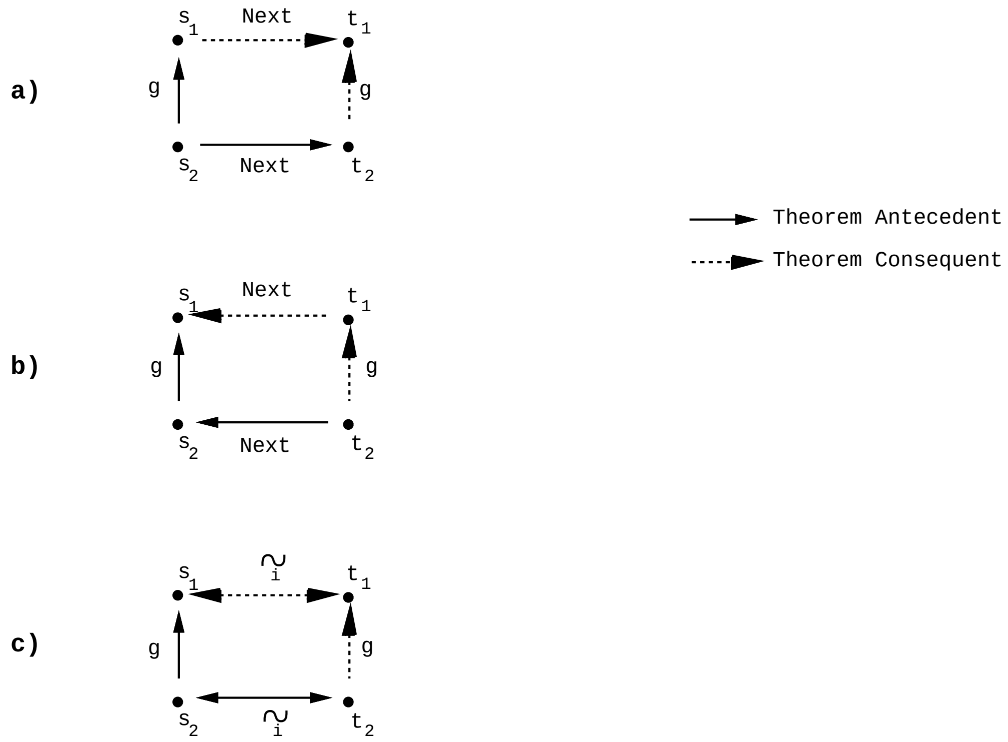

Figure 5.8: Clause 2 of Definition 14 (Bisimulation).

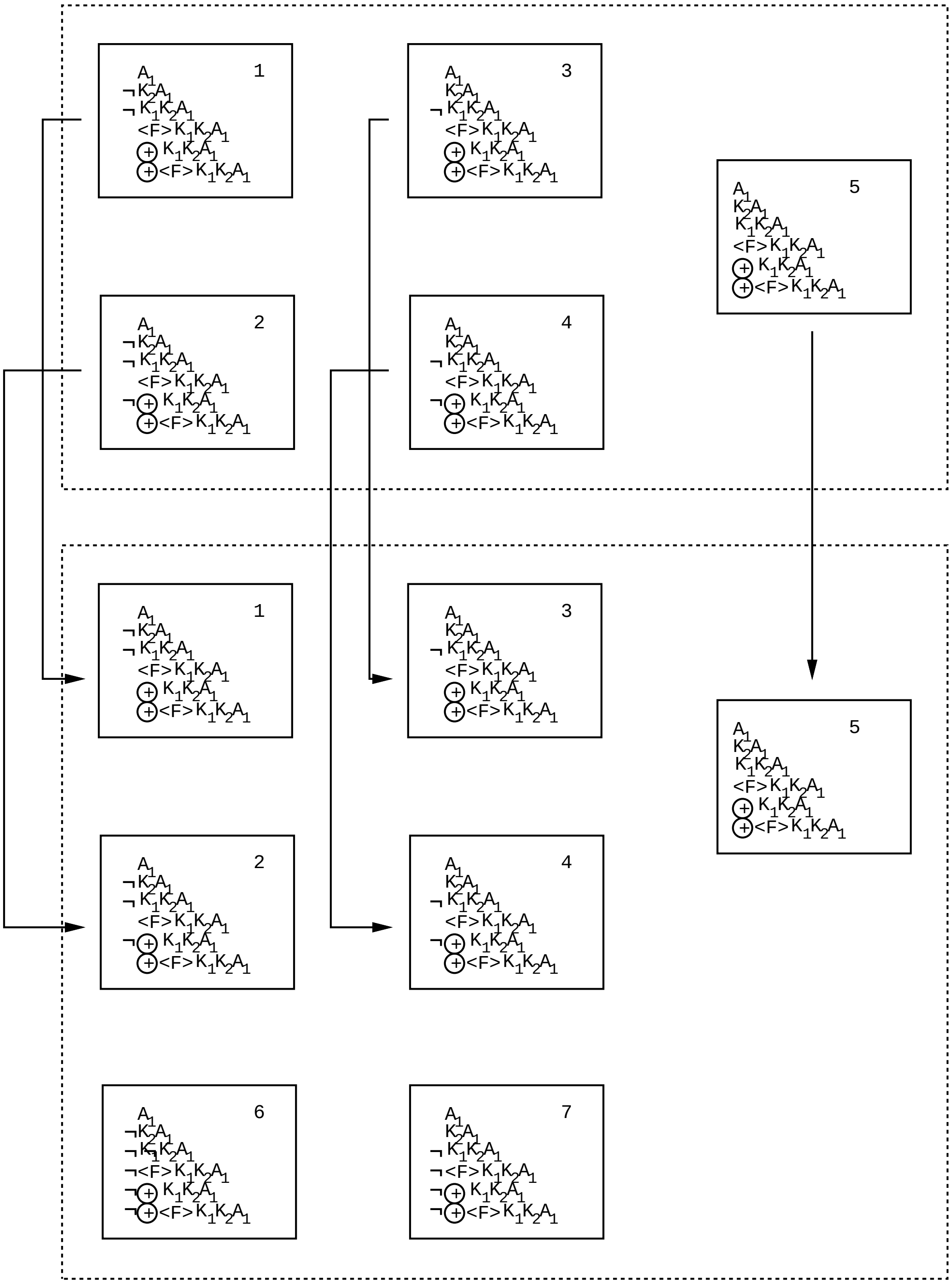

Figure 5.9: Bisimulation from CAN to CSF.

5.3 Summary

In this chapter, we have shown a mechanizable process for building a Kripke model of a system and discussed the use of bisimulation relations for comparing these models. For formulas in a pre-defined set of "interesting" formulas, γ, validity in the γ-relevant model is equivalent to deducibility from the system specification formulas. Two systems are equivalent with respect to the formulas of γ if both model exactly the same formulas. A bisimulation relation allows comparisons at the state level: if there is a bisimulation edge pairing states from two models, then the same formulas hold at both states in the pair.

The tools discussed in Chapters 3 through 5 can be used to build a non-operational problem-based taxonomy of distributed systems. To build a taxonomy, one would begin by choosing and specifying a few important and/or representative problems in distributed systems. The next step would be to develop non-operational specifications of several distributed systems (running their full-information protocols.) One could then choose either the direct or indirect approach to classifying the systems with respect to the problems. The direct approach requires finding a deduction to show that the problem requirement follows from the specification formulas of the system. The indirect approach involves processing the system specifications to produce Kripke models, restricted by a chosen set of "interesting" formulas (based, most likely, on the problem specifications.) The Kripke models can then be checked mechanically to determine which of the problem requirements they satisfy. Producing the Kripke models has the added advantage that the models show knowledge transfer behavior of their systems. Bisimulations can be used to show similarities between individual knowledge states of two Kripke models.

In the next chapter, we summarize the contributions of this thesis and discuss some future research directions.

Chapter 6

Conclusions and Future Work

We have provided tools for specifying distributed systems, building formal models, and then comparing these models. These tools form the basis of a method for comparing distributed systems, and, ultimately, for constructing a problem-based taxonomy of the systems. Because the specifications and comparisons are non-operational it is convenient to compare dissimilar systems (e.g., shared-memory vs. message-passing, systems with various failure models, anonymous agents vs. agents with i.d.'s.)

In summary, the major contribution of this thesis is a method for making non-operational, knowledge-transfer comparisons of distributed systems. The method has the following components:

1. A propositional modal language $\mathcal{L}$ and associated decidable logic, L, with knowledge and temporal operators for specifying non-operationally the knowledge transfer behavior and requirements of distributed systems, protocols and problems.

2. The concept of generic distributed problems which, along with full-information protocols, allow systems to be studied and compared in isolation from the influences of protocol choice and specific problem requirements.

3. A small model theorem which allows Kripke structures to be mechanically produced from system and protocol specifications in $\mathcal{L}$, and a bisimulation result which allows those Kripke structures to be formally compared.

In the future, we hope that this method will be used for comparing a variety of distributed systems. In particular, this method may allow us to find a more intuitive (non-operational) explanation of the results of Dolev, Dwork and Stockmeyer, [DDS87], which showed the crash fault tolerances of a collection of distributed systems, and similar results in the fault tolerance literature.

On a related topic, it would be interesting to build a hierarchy of distributed problems, based on knowledge-transfer specifications: if a given problem is solvable, which others can also be solved? To do this, we would specify a variety of distributed problems and look for instances where one problem requirement implies another.

Appendix A

Symbols

Table A.1: Symbols

Symbol	Explanation
$\mathcal{L}$	specification language
L	logic (i.e., deductive system or axioms and rules)
Λ	the logic formed from L plus the set of specifications for a system, its full-information protocol and the standard generic problem assumption
$\mathcal{P}$	set of propositional variables of $\mathcal{L}$
$\oplus$	"next time" operator
$\ominus$	"last time" operator
$\mathcal{U}$	"until" operator
$\mathcal{S}$	"since" operator
C	"common knowledge" operator
K_i	"$Agent_i$ knows" operator
Form	set of formulas of $\mathcal{L}$
$\mathcal{M}$	a Kripke structure $(S, Next, \sim_1, \sim_2, \ldots, \sim_n, V)$
S	a set of system states
$Next$	an irreflexive, serial, branching temporal relation on system states
$\sim_i$	an equivalence relation on system states indicating which system states $Agent_i$ is unable to distinguish
V	a valuation function assigning truth values to propositional variables at every system state
γ	set of "interesting" formulas for restricting the Kripke model of Λ. Usually a collection of problem requirement specifications.

Appendix B

Rules and Axioms of L

Table B.1: Rules of L

Taut: $\dfrac{\text{A is a tautology}}{A}$

MP: $\dfrac{A_1\ ,\ A_1 \rightarrow A_2}{A_2}$,

Gen: $\dfrac{A}{CA}$, $\dfrac{A}{\Box A}$,

Ind: $\dfrac{A_1 \rightarrow E(A_1 \wedge A_2)}{A_1 \rightarrow CA_2}$, $\dfrac{(A_2 \vee (A_1 \wedge \oplus A_3)) \rightarrow A_3}{(A_1 \mathcal{U} A_2) \rightarrow A_3}$, $\dfrac{(A_2 \vee (A_1 \wedge \ominus A_3)) \rightarrow A_3}{(A_1 \mathcal{S} A_2) \rightarrow A_3}$

Table B.2: Axiom Schemata of L

Axiom 1	$K_i(A_1 \to A_2) \to (K_i A_1 \to K_i A_2)$
	$C(A_1 \to A_2) \to (CA_1 \to CA_2)$
	$\oplus(A_1 \to A_2) \to (\oplus A_1 \to \oplus A_2)$
	$\ominus(A_1 \to A_2) \to (\ominus A_1 \to \ominus A_2)$
Axiom 2	$K_i A \to A$
	$CA \to A$
Axiom 3	$K_i A \to K_i K_i A$
	$CA \to CCA$
Axiom 4	$\neg K_i A \to K_i \neg K_i A$
	$\neg CA \to C \neg CA$
Axiom 5	$CA \to E(A \wedge CA)$
Axiom 6	$\neg\oplus$ false
	$\neg\ominus$ false
Axiom 7	$\oplus P_j \to \neg \oplus \neg P_j$, for $P_j \in \mathcal{P}$
Axiom 8	$A_1 \mathcal{U} A_2 \leftrightarrow A_2 \vee (A_1 \wedge \oplus(A_1 \mathcal{U} A_2))$
	$A_1 \mathcal{S} A_2 \leftrightarrow A_2 \vee (A_1 \wedge \ominus(A_1 \mathcal{S} A_2))$

Appendix C

Proofs

C.1 Proof of soundness of L

Lemmas 1 and 2 are used in proving the soundness of L. Lemma 1 says that the axioms of L are valid. We say that $\models$ A ("A is valid") if for all structures $\mathcal{M}$ and states s of $\mathcal{M}$, $(\mathcal{M}, s) \models A$. Lemma 2 says that the inference rules preserve validity. Theorem 3 (soundness of L) follows directly from the lemmas, since every line of an L proof is either an axiom or is derived by using the inference rules.

Lemma 1 *If A is an axiom of* L *, then* $\models A$.

Proof: Let $\mathcal{M} = (S, \mathit{Next}, \sim_1, \sim_2, \ldots, \sim_n, V)$ be an arbitrary structure and $s \in S$ be some state in that structure. We show that each axiom schema is valid. Let Next^+ be the transitive closure of the *Next* relation. A proof is shown for only one instance of each axiom schema. Other instances would be proved similarly.

> Axiom 1: Assume that $(\mathcal{M}, s) \models \oplus(A_1 \rightarrow A_2)$ and $(\mathcal{M}, s) \models \oplus A_1$. Since $(\mathcal{M}, s) \models \oplus A_1$ and $(\mathcal{M}, s) \models \oplus(A_1 \rightarrow A_2)$, for all $t \in S$ such that $(s,t) \in \mathit{Next}$, $(\mathcal{M}, t) \models A_1$ and $(\mathcal{M}, t) \models (A_1 \rightarrow A_2)$. So, for all $t \in S$ such that $(s,t) \in \mathit{Next}$, $(\mathcal{M}, t) \models A_2$. Thus, $(\mathcal{M}, s) \models \oplus A_2$.
>
> Axiom 2: Assume that $(\mathcal{M}, s) \models K_i A$. Then for all $t \in S$ such that $(s,t) \in \sim_i$, $(\mathcal{M}, t) \models A$. Since $\sim_i$ is reflexive, $(s,s) \in \sim_i$, so $(\mathcal{M}, s) \models A$.
>
> Axiom 3: Assume that $(\mathcal{M}, s) \models K_i A$.
>
> We must show that $(\mathcal{M}, s) \models K_i K_i A$. By the semantics of K_i, it suffices to show that (for all $t \in S$ such that $(s,t) \in \sim_i$, $(\mathcal{M}, t) \models K_i A$). Again, by the semantics of K_i, it suffices to show that (for all $t \in S$ such that $(s,t) \in \sim_i$, (for all $u \in S$ such that $(t,u) \in \sim_i$, $(\mathcal{M}, u) \models A$)). Since $\sim_i$ is transitive, it suffices to show that (for all $u \in S$

such that $(s,u) \in \sim_i$, $(\mathcal{M},u) \models A)$.

By the assumption, $(\mathcal{M},s) \models K_i A$, so for all $u \in S$ such that $(s,u) \in \sim_i$, $(\mathcal{M},u) \models A$.

<u>Axiom 4</u>: Assume that $(\mathcal{M},s) \models \neg K_i A$.

We must show that $(\mathcal{M},s) \models K_i \neg K_i A$. By the semantics of K_i, it suffices to show that (for all $t \in S$ such that $(s,t) \in \sim_i$, $(\mathcal{M},t) \models \neg K_i A)$. By the semantics of $\neg$, it suffices to show that (for all $t \in S$ such that $(s,t) \in \sim_i$, $(\mathcal{M},t) \not\models K_i A)$. By the semantics of K_i, it suffices to show that (for all $t \in S$ such that $(s,t) \in \sim_i$, (there is some $u \in S$ such that $(t,u) \in \sim_i$, but $(\mathcal{M},u) \not\models A)$). Since $\sim_i$ is transitive, it suffices to show that there is some $u \in S$ such that $(s,u) \in \sim_i$, but $(\mathcal{M},u) \not\models A$.

By the assumption, $(\mathcal{M},s) \models \neg K_i A$, so there must be a $u \in S$ such that $(s,u) \in \sim_i$, but $(\mathcal{M},u) \not\models A$.

<u>Axiom 5</u>: Assume that $(\mathcal{M},s) \models CA$.

We must show that $(\mathcal{M},s) \models E(A \wedge CA)$. By the semantics of E, it suffices to show that (for all $t \in S$ such that $(s,t) \in \sim$, $(\mathcal{M},t) \models A \wedge CA)$. By the semantics of $\wedge$, it suffices to show that (for all $t \in S$ such that $(s,t) \in \sim$, $(\mathcal{M},t) \models A$ and $(\mathcal{M},t) \models CA)$.

By the assumption, $(\mathcal{M},s) \models CA$, so for all $u \in S$ such that $(s,u) \in \sim^*$, $(\mathcal{M},u) \models A$. Since $\sim \subseteq \sim^*$, for all $t \in S$ such that $(s,t) \in \sim$, $(\mathcal{M},t) \models A$ and by the transitivity of $\sim^*$, if $(t,u) \in \sim^*$, then $(s,u) \in \sim^*$. So, $(\mathcal{M},t) \models CA$.

<u>Axiom 6</u>: We must show that $(\mathcal{M},s) \models \neg\oplus$ false.

By the semantics of $\neg$, it suffices to show that $(\mathcal{M},s) \not\models \oplus\neg$ true. By the semantics of $\oplus$, it suffices to show that there is some $t \in S$ such that $(s,t) \in \mathit{Next}$, but $(\mathcal{M},t) \not\models \neg$ true. By the semantics of $\neg$, it suffices to show that there is some $t \in S$ such that $(s,t) \in \mathit{Next}$ and $(\mathcal{M},t) \models$ true. Since true abbreviates the formula $(P_j \vee\neg\, P_j)$ for some $P_j \in \mathcal{P}$ and it is always the case that either $(\mathcal{M},t) \models P_j$ or $(\mathcal{M},t) \models \neg P_j$, we have that $(\mathcal{M},t) \models$ true.

<u>Axiom 7</u>: Assume that $(\mathcal{M},s) \models \oplus P_j$.

We must show that $(\mathcal{M}, s) \models \neg \oplus \neg P_j$. By the semantics of $\neg$, it suffices to show that $(\mathcal{M}, s) \not\models \oplus\neg P_j$. By the semantics of $\oplus$, it suffices to show that there is some $t \in S$ such that $(s,t) \in \mathit{Next}$ and $(\mathcal{M}, t) \not\models \neg P_j$.

By the assumption, $(\mathcal{M}, s) \models \oplus P_j$. So, by the semantics of $\oplus$, for all $t \in S$ such that $(s,t) \in \mathit{Next}$, $(\mathcal{M}, t) \models P_j$. Thus, for all $t \in S$ such that $(s,t) \in \mathit{Next}$, $(\mathcal{M}, t) \not\models \neg P_j$.

<u>Axiom 8</u>:

<u>$\rightarrow$</u>

Suppose that $(\mathcal{M}, s) \models A_1 \mathcal{U} A_2$. If $(\mathcal{M}, s) \models A_2$, then $(\mathcal{M}, s) \models A_2 \vee (A_1 \wedge \oplus(A_1 \mathcal{U} A_2))$. So, instead, assume that $(\mathcal{M}, s) \not\models A_2$.

We must first show that $(\mathcal{M}, s) \models A_1$. Since *Next* is a serial relation (i.e., $\forall s \exists t$ such that $(s,t) \in \mathit{Next}$), we can choose a *Next*-branch beginning with s: $\langle s = s_0, s_1, s_2, \ldots \rangle$. Since $(\mathcal{M}, s) \models A_1 \mathcal{U} A_2$, there exists an s_k on the branch such that $(\mathcal{M}, s_k) \models A_2$ and $(\mathcal{M}, s_j) \models A_1$ for $0 \leq j < k$. Because $(\mathcal{M}, s) \not\models A_2$, $k > 0$ (i.e., $s_k \neq s$). Hence, $(\mathcal{M}, s) \models A_1$.

We must also show that $(\mathcal{M}, s) \models \oplus(A_1 \mathcal{U} A_2)$. Let $s, t \in S$ be such that $(s,t) \in \mathit{Next}$. We must show that $(\mathcal{M}, t) \models A_1 \mathcal{U} A_2$. Let $\langle t = t_0, t_1, t_2, \ldots \rangle$ be a *Next*-branch beginning with t and $\langle s = s_0, s_1 = t_0, s_2 = t_1, \ldots \rangle$ be the corresponding *Next*-branch beginning at s. Since $(\mathcal{M}, s) \models A_1 \mathcal{U} A_2$, there is an s_k on the branch such that $(\mathcal{M}, s_k) \models A_2$ and $(\mathcal{M}, s_j) \models A_1$ for $0 \leq j < k$. Because $(\mathcal{M}, s) \not\models A_2$, $k > 0$ (i.e., $s_k \neq s$). Thus, there exists $l = k - 1$ so that $(\mathcal{M}, t_l) \models A_2$ and $(\mathcal{M}, t_j) \models A_1$ whenever $0 \leq j < l$. By the semantics of $\mathcal{U}$, $(\mathcal{M}, t) \models A_1 \mathcal{U} A_2$.

<u>$\leftarrow$</u>

Suppose that $(\mathcal{M}, s) \models A_2 \vee (A_1 \wedge \oplus(A_1 \mathcal{U} A_2))$. If $(\mathcal{M}, s) \models A_2$, then $(\mathcal{M}, s) \models A_1 \mathcal{U} A_2$. So, instead, suppose that $(\mathcal{M}, s) \models A_1 \wedge \oplus(A_1 \mathcal{U} A_2)$. Let $\langle s = s_0, s_1, s_2, \ldots \rangle$ be a *Next*-branch beginning with s. By the semantics of $\oplus$, $(\mathcal{M}, s_1) \models A_1 \mathcal{U} A_2$. Thus, there is an s_k on the branch with $k > 0$ such that $(\mathcal{M}, s_k) \models A_2$ and $(\mathcal{M}, s_j) \models A_1$ for $0 < j < k$. But we also have that $(\mathcal{M}, s = s_0) \models A_1$. Thus, $(\mathcal{M}, s) \models A_1 \mathcal{U} A_2$.

$\square$

Lemma 2 *Inference rules MP, Gen, and Ind preserve validity. I.e.,*

<u>Taut</u>: If A is a tautology, then $\models A$.

<u>MP</u>: If $\models A_1$ and $\models A_1 \rightarrow A_2$, then $\models A_2$.

<u>Gen</u>: If $\models A$, then both $\models CA$ and $\models \Box A$.

<u>Ind$_C$</u>: If $\models A_1 \rightarrow E(A_1 \wedge A_2)$, then $\models A_1 \rightarrow CA_2$.

<u>*Ind*$_{\mathcal{U}}$</u>*: If* $\models A_2 \vee (A_1 \wedge \oplus A_3) \rightarrow A_3$*, then* $\models (A_1 \mathcal{U} A_2) \rightarrow A_3$*.*

Proof: Let $\mathcal{M} = (S, \mathit{Next}, \sim_1, \sim_2, \ldots, \sim_n, V)$ be an arbitrary model. Since each axiom schema has multiple forms, we show validity for only one of the forms. Validity proofs for the remaining forms are similar.

<u>Taut</u>: Suppose that formula A is a tautology. Then A is the result of uniformly substituting formulas $B_1, B_2, \ldots, B_k$ for propositional variables $P_1, P_2, \ldots, P_k$ in a tautology A' of propositional logic. By definition, $V(A') =$ true, regardless of the truth values assigned to the propositional variables of A'. Thus, $\models A'$. Note that $V(B_j) \in$ {true, false}, so the substitution of B_j for P_j in A' does not allow V(A) = false at any valuations of the B_j's. Thus, $\models$ A.

<u>MP</u>: Let $s \in S$ be an arbitrary state and assume that $(\mathcal{M}, s) \models A_1$ and $(\mathcal{M}, s) \models A_1 \rightarrow A_2$. Suppose by way of contradiction that $(\mathcal{M}, s) \not\models A_2$, then since $(\mathcal{M}, s) \models A_1$, $(\mathcal{M}, s) \not\models A_1 \rightarrow A_2$. This contradicts the assumption that $(\mathcal{M}, s) \models A_1 \rightarrow A_2$. So, we must have $(\mathcal{M}, s) \models A_2$.

<u>*Gen*$_{\Box}$</u>: For every state $u \in S$, assume that $(\mathcal{M}, u) \models A$. Suppose by way of contradiction that there is an $s \in S$ such that $(\mathcal{M}, s) \not\models \Box A$, then $(\mathcal{M}, s) \not\models [P]A \wedge A \wedge [F]A$. Then by the semantics of [F] and [P], there is a $t \in S$ such that either $(t, s) \in \mathit{Next}^+$ and $(\mathcal{M}, t) \not\models A$ or $(s, t) \in \mathit{Next}^+$ and $(\mathcal{M}, t) \not\models A$. This contradicts the assumption that for all $u \in S$, $(\mathcal{M}, u) \models A$. So, we must have $(\mathcal{M}, u) \models \Box A$.

<u>*Gen*$_C$</u>: For every state $u \in S$, assume that $(\mathcal{M}, u) \models A$. Suppose by way of contradiction that there is an $s \in S$ such that $(\mathcal{M}, s) \not\models CA$, then there is a $t \in S$ such that $(s, t) \in \sim^*$, but $(\mathcal{M}, t) \not\models A$. This contradicts the assumption that for all $u \in S$, $(\mathcal{M}, u) \models A$. So, we must have $(\mathcal{M}, u) \models CA$.

<u>*Ind*$_C$</u>: For every state $u \in S$, assume that $(\mathcal{M}, u) \models A_1 \rightarrow E(A_1 \wedge A_2)$. Fix an arbitrary state $s \in S$ and assume that $(\mathcal{M}, s) \models A_1$.

We must show that $(\mathcal{M}, s) \models CA_2$. By the semantics of C, it suffices to show that for all $u \in S$ such that $(s, t) \in \sim^*$, $(\mathcal{M}, u) \models A_2$.

If $(s,u) \in\ \sim^*$, then define

$$\mid (s,u) \mid = \begin{cases} 0 & \text{if } s \equiv u \\ 1 & \text{if } (s,u) \in\ \sim \\ \mathit{minimum}(\mid (s,t) \mid) + 1 & \text{otherwise} \\ t \in S \text{ such that } (s,t) \in\ \sim^* & \\ \text{and } (t,u) \in\ \sim & \end{cases}$$

We show by induction on $\mid (s,u) \mid$ that $(\mathcal{M}, u) \models A_1 \wedge A_2$.

Suppose $\mid (s,u) \mid = 1$. Then by the assumptions that $(\mathcal{M}, s) \models A_1 \rightarrow E(A_1 \wedge A_2)$ and $(\mathcal{M}, s) \models A_1$, we have $(\mathcal{M}, s) \models E(A_1 \wedge A_2)$. By the semantics of E, for all $t \in S$ such that $(s,t) \in\ \sim$, $(\mathcal{M}, t) \models A_1 \wedge A_2$.

Assume that for all $t \in S$ such that $\mid (s,t) \mid < n$, $(\mathcal{M}, t) \models A_1 \wedge A_2$.

Suppose $\mid (s,u) \mid = n$. Then there is some $t \in S$ such that $(s,t) \in\ \sim^*$ and $(t,u) \in\ \sim$. By the induction hypothesis, $(\mathcal{M}, t) \models A_1 \wedge A_2$. By assumption, $(\mathcal{M}, t) \models A_1 \rightarrow E(A_1 \wedge A_2)$, so $(\mathcal{M}, t) \models E(A_1 \wedge A_2)$. By the semantics of E, for all $v \in S$ such that $(t,v) \in\ \sim$, $(\mathcal{M}, v) \models A_1 \wedge A_2$. Specifically, since $(t,u) \in\ \sim$, $(\mathcal{M}, u) \models A_1 \wedge A_2$.

$Ind_{\mathcal{U}}$:

For every state $s \in S$, assume that $(\mathcal{M}, s) \models (A_2 \vee (A_1 \wedge \oplus A_3)) \rightarrow A_3$. We must show that for an arbitrary $t \in S$, $(\mathcal{M}, t) \models (A_1 \mathcal{U} A_2) \rightarrow A_3$. So assume that $(\mathcal{M}, t) \models A_1 \mathcal{U} A_2$.

If $(\mathcal{M}, t) \models A_2$, then $(\mathcal{M}, t) \models A_3$ follows from $(\mathcal{M}, s) \models (A_2 \vee (A_1 \wedge \oplus A_3)) \rightarrow A_3$. So, assume, instead, that $(\mathcal{M}, t) \not\models A_2$. Let $v \in S$ and define the set

$\mathit{waiting} = \{t\} \cup \{s \in S : (t,s) \in \mathit{Next}^+$ and $(\mathcal{M}, v) \not\models A_2$ whenever $(t,v) \in \mathit{Next}^+$ and $(v,s) \in \mathit{Next}^+\}$

Let $s \prec s'$ iff $s, s' \in \mathit{waiting}$ and $(s,s') \in \mathit{Next}$.

Observe that there is no infinite descending sequence $s_0 \succ s_1 \succ s_2 \succ \cdots \succ s_k \succ \cdots$, because that would contradict $(\mathcal{M}, t) \models A_1 \mathcal{U} A_2$. Therefore, $\prec$ is well-founded and we may use transfinite induction along $\prec$ to show that $(\mathcal{M}, s) \models A_3$ for all $s \in \mathit{waiting}$. For a discussion of transfinite induction, see [End77].

Let $s \in$ *waiting*. If $(\mathcal{M}, s) \models A_2$, then $(\mathcal{M}, s) \models A_3$ follows from $(\mathcal{M}, s) \models (A_2 \vee (A_1 \wedge \oplus A_3)) \rightarrow A_3$. If $(\mathcal{M}, s) \not\models A_2$, then $(s, v) \in$ *Next* implies $v \in$ *waiting* and $v \prec s$. By induction along $\prec$, we get $(\mathcal{M}, v) \models A_3$ for all v where $(s, v) \in$ *Next*. By the semantics of $\oplus$, $(\mathcal{M}, s) \models \oplus A_3$. Because $(\mathcal{M}, s) \models A_1$ and $(\mathcal{M}, s) \models (A_1 \wedge \oplus A_3) \rightarrow A_3$, we get $(\mathcal{M}, s) \models A_3$.

□

Theorem 3 L *is sound, i.e.,*
If $\vdash_{\mathrm{L}}$ *A, then* $\models A$.

Proof: Suppose that $\vdash_L A$. Every line in a proof of L is either an axiom of L or is derived from an axiom by a finite number of applications of MP, Gen, or Ind, so by Lemmas 1 and 2 , $\models A$. □

C.2 Sketches of Proofs of Deduction Lemmas

These lemmas were used in the formal proof shown in Table 3.4.

C.2.1 Proof of Lemma 1

$\frac{\langle F \rangle A \;,\; \Box(A \rightarrow B)}{\langle F \rangle B}$

1.	$\langle$ F $\rangle$ A	assume
2.	$\Box$(A $\rightarrow$ B)	assume
3.	[P](A $\rightarrow$ B) $\wedge$ (A $\rightarrow$ B) $\wedge$ [F](A $\rightarrow$ B)	remove abbrev. 2
4.	[F](A $\rightarrow$ B)	remove $\wedge$ 3
5.	$\neg\langle$F$\rangle\neg$(A $\rightarrow$ B)	remove abbrev. 4
6.	$\neg\langle$F$\rangle$(A $\wedge\neg$ B)	remove abbrev. 5
7.	$\langle$F$\rangle$B	1, 6, MP

C.2.2 Proof of Lemma 2

$\frac{\langle F \rangle \langle F \rangle A}{\langle F \rangle A}$

1.	$\langle$F$\rangle\langle$F$\rangle$A	assume
2.	true $\mathcal{U}$(true $\mathcal{U}$A)	remove abbrev. 1
3.	(true $\mathcal{U}$A) $\vee\oplus$(true $\mathcal{U}$A)	Axiom 8, MP
4.	true $\mathcal{U}$A	Axiom 8, MP
5.	$\langle$F$\rangle$ A	introduce abbrev. 4

C.2.3 Proof of Lemma 3

$\frac{\langle F\rangle(A\vee B)\ ,\ \Box(A\rightarrow C)}{\langle F\rangle(B\vee C)}$

1.	$\langle F\rangle(A \vee B)$	assume
2.	$\Box(A \rightarrow C)$	assume
3.	$[P](A \rightarrow C) \wedge (A \rightarrow C) \wedge [F](A \rightarrow C)$	remove abbrev. 2
4.	$[F](A \rightarrow C)$	remove $\wedge$ 3
5.	$\neg\langle F\rangle\neg(A \rightarrow C)$	remove abbrev. 4
6.	$\neg\langle F\rangle(A \wedge\neg C)$	remove abbrev. 5
7.	$\langle F\rangle((A \wedge C) \vee B)$	Axiom 8, MP
8.	$\langle F\rangle(B \vee C)$	remove $\wedge$ 7

C.2.4 Proof of Lemma 4

$\frac{\langle F\rangle(\langle F\rangle A\vee B)}{\langle F\rangle(A\vee B)}$

1.	$\langle F\rangle(\langle F\rangle A \vee B)$	assume
2.	$\langle F\rangle\langle F\rangle A \vee\langle F\rangle B$	Axioms 1 and 8, MP 1
3.	$\langle F\rangle A \vee\langle F\rangle B$	Lemma 2, MP 2
4.	$\langle F\rangle(A \vee B)$	Axiom 8, MP 3

C.3 Proof of bound on number of subformulas

We prove by induction on the structure of A that $|subformulas(A)| \leq length(\mathrm{A})$:

<u>$A \in \mathcal{P}$</u>

A has no subformulas, so $|subformulas(A)| = 0$. Since $length(\mathrm{A}) = 1$, $|subformulas(A)| \leq length(\mathrm{A})$.

<u>A has the form $(\bigcirc A_1)$, where $\bigcirc$ represents one of $\neg$, $\oplus$, $\ominus$, C, K_i</u>

$|subformulas(A)| = |subformulas(A_1)| + 1$ and $length(\mathrm{A}) = length(A_1) + 3$. By the induction hypothesis, $|subformulas(A_1)| \leq length(A_1)$, so $|subformulas(A)| \leq length(\mathrm{A})$.

<u>A has either of the forms $(A_1 \wedge A_2)$, $(A_1 \mathcal{U} A_2)$, or $(A_1 \mathcal{S} A_2)$</u>

$|subformulas(A)| = |subformulas(A_1)| + |subformulas(A_2)| + 1$ and $length(\mathrm{A}) = length(A_1) + length(A_2) + 3$. By the induction hypothesis, $|subformulas(A_1)| \leq length(A_1)$ and $|subformulas(A_2)| \leq length(A_2)$, so $|subformulas(A)| \leq length(\mathrm{A})$.

C.4 Proof of Lindenbaum's Lemma

Lemma 3 *(Lindenbaum's Lemma) If Θ is a Λ-consistent set, then there is a Λ-maximal set $\widehat{\Theta} \supseteq \Theta$.*

Proof: (Lemma 3 Lindenbaum's Lemma) By the following construction:

1. Enumerate the formulas of $\mathcal{L}$: $A_0, A_1, \ldots, A_n, \ldots$
2. Let

$$\begin{array}{rcl} \Delta_0 & = & \Theta \\ \Delta_{n+1} & = & \begin{cases} \Delta_n \cup \{A_n\} & \text{if } \Delta_n \not\vdash_\Lambda \neg A_n \\ \Delta_n \cup \{\neg A_n\} & \text{else if } \Delta_n \not\vdash_\Lambda A_n \text{ and } \neg A_n \in \mathcal{L} \\ \Delta_n & \text{else} \end{cases} \\ \Delta & = & \cup_{n\in\omega}\Delta_n \end{array}$$

Let $\widehat{\Theta}$ be the Δ in the construction above. Then,

- $\widehat{\Theta}$ is consistent, since Δ does not contain both A_n and $\neg A_n$ for any formula A_n.
- $\widehat{\Theta}$ is maximal, since:
  1. If $\{A_n, \neg A_n\} \subseteq \mathcal{L}$, then $A_n \in \Delta$, or $\neg A_n \in \Delta$.
  2. If $\neg A_n \notin \mathcal{L}$, then if $\Delta \cup \{A_n\}$ is Λ-consistent, $A_n \in \Delta$.
- $\widehat{\Theta} \supseteq \Theta$, since $\Delta \supseteq \Theta$.

□

C.5 Proof of Theorem 1

We prove the theorem by induction on the structure of A.

Case 1. If $A \in \mathcal{P}$ or A is of the form $A_1 \wedge A_2$, $\neg A_1$, $K_i A_1$, or CA_1, we appeal to the proof of Halpern and Moses in [HM] (Theorem 4.3).
Case 2. We consider the case where A is of the form $\oplus A_1$. The proof for A of the form $\ominus A_1$ is very similar.

$\Longrightarrow$

Suppose that $(\mathcal{M}_\gamma, s) \models \oplus A_1$ and let $w = \{B : \oplus B \in s\}$. We show that $w \cup \{\neg A_1\}$ is inconsistent. Suppose by way of contradiction that $w \cup \{\neg A_1\}$ is consistent. By Lemma 3 (Lindenbaum's Lemma, shown in Section C.4), $w \cup \{\neg A_1\}$ has a Λ-maximal extension, v (i.e., $w \cup \{\neg A_1\} \subseteq v \in S_\gamma$). By definition of v, $(s, v) \in Next_\gamma$, and $(\mathcal{M}_\gamma, v) \models \neg A_1$. Thus, $(\mathcal{M}_\gamma, s) \not\models \oplus A_1$. This contradicts our assumption that $(\mathcal{M}_\gamma, s) \models \oplus A_1$ and gives us that $w \cup \{\neg A_1\}$ is inconsistent.

Since $w \cup \{\neg A_1\}$ is inconsistent, $\vdash_\Lambda B_1 \wedge \cdots \wedge B_n \rightarrow A_1$, for $B_i \in w$. By Axiom 1 and Rules Gen and MP, we have that

$$\vdash_\Lambda \oplus B_1 \wedge \cdots \wedge \oplus B_n \rightarrow \oplus A_1, \text{ for } B_i \in w. \tag{C.1}$$

By definition of w, $\{\oplus B_1, \cdots, \oplus B_n\} \subseteq s$. Since $s \in S_\gamma$, either $\oplus A_1 \in s$ or $\neg \oplus A_1 \in s$. By Equation C.1, $\oplus A_1 \in s$.

$\Longleftarrow$

Suppose that $\oplus A_1 \in s$. We must show that $(s,t) \in Next_\gamma$ implies $(\mathcal{M}_\gamma, t) \models A_1$. Assume that $(s,t) \in Next_\gamma$. By definition of $Next_\gamma$, $(s,t) \in Next_\gamma$ implies that ($\oplus A_1 \in s$ implies $A_1 \in t$). Because $\oplus A_1 \in s$ and $(s,t) \in Next_\gamma$, $A_1 \in t$. By the induction hypothesis, $(\mathcal{M}_\gamma, t) \models A_1$.

Case 3. The case where A is of the form $(A_1 \mathcal{U} A_2)$ (similarly, the form $(A_1 \mathcal{S} A_2)$) is addressed in [Gol92].

□

C.6 Proof of Theorem 2

Let g be a bisimulation from model $\mathcal{M}_1$ to model $\mathcal{M}_2$, A be a formula of $\mathcal{L}$, and $s_1 \in S_1$, $s_2 \in S_2$. Suppose that $(s_1, s_2) \in g$. We show by induction on the structure of A that $(\mathcal{M}_1, s_1) \models A$ iff $(\mathcal{M}_2, s_2) \models A$.

$A \in \mathcal{P}$: $(\mathcal{M}_1, s_1) \models A$ iff $(\mathcal{M}_2, s_2) \models A$ follows immediately from condition 3 of Definition 14.
A is $(\neg A_1)$ or $(A_1 \wedge A_2)$: $(\mathcal{M}_1, s_1) \models A$ iff $(\mathcal{M}_2, s_2) \models A$ follows from the induction hypothesis and the semantics of the $\neg$ and $\wedge$ operators.

A is $(\oplus A_1)$: Suppose that $(\mathcal{M}_1, s_1) \models A$. Then for all $t_1 \in S_1$ such that $(s_1, t_1) \in Next_1$, $(\mathcal{M}_1, t_1) \models A_1$. Choose an arbitrary $t_2 \in S_2$ such that $(s_2, t_2) \in Next_2$, then by condition 1 a) of Definition 14, there exists $t_1 \in S_1$ such that $(t_1, t_2) \in g$ and $(s_1, t_1) \in Next_1$. Then $(\mathcal{M}_1, t_1) \models A_1$. By the induction hypothesis, $(\mathcal{M}_2, t_2) \models A_1$. By the semantics of $\oplus$, $(\mathcal{M}_2, s_2) \models \oplus A_1$. The converse can be shown similarly by appealing to condition 2 a) of Definition 14.

A is (K_iA): This case should be argued as was the case above with references to the $\sim_{i,1}$ and $\sim_{i,2}$ relations and condition 1 c) of Definition 14.

A is $(A_1 \mathcal{U} A_2)$: Suppose that $(\mathcal{M}_1, s_1) \models A$. We must show that $(\mathcal{M}_2, s_2) \models A$. Let $\langle s_2 = t^2_0, t^2_1, t^2_2, \ldots \rangle$ be any $Next_2$-branch that starts at s_2. By condition 1 a) of Definition 14, we can choose a $Next_1$-branch $\langle s_1 = t^1_0, t^1_1, t^1_2, \ldots \rangle$ starting at s_1 such that $({t_j}^1, {t_j}^2) \in g$. Because $(\mathcal{M}_1, s_1) \models A$, there is a k such that $(\mathcal{M}_1, t^1_k) \models A_2$ and $(\mathcal{M}_1, t^1_j) \models A_1$ whenever $0 \leq j < k$. By the induction hypothesis, we have that $(\mathcal{M}_2, t^2_k) \models A_2$ and $(\mathcal{M}_2, t^2_j) \models A_1$ whenever $0 \leq j < k$. This verifies $(\mathcal{M}_2, s_2) \models A$. The converse can be shown similarly by appealing to condition 2 a) of Definition 14.

A is (CA): Suppose that $(\mathcal{M}_1, s_1) \models A$. We must show that $(\mathcal{M}_2, s_2) \models A$. Let $t_2 \in S_2$ be any state such that $(s_2, t_2) \in \sim^*_2$. By condition 1 c) of Definition 14, we can choose a state $t_1 \in S_1$ such that $(s_1, t_1) \in \sim_1^*$ and $(t_1, t_2) \in g$. By the semantics of C, $(\mathcal{M}_1, t_1) \models A_1$. By the induction hypothesis, $(\mathcal{M}_2, t_2) \models A_1$. Since t_2 was an arbitrary state of S_2 satisfying $(s_2, t_2) \in \sim_2^*$, it follows that for all such states, t_2, $(\mathcal{M}_2, t_2) \models A_1$. By the semantics of C, $(\mathcal{M}_2, s_2) \models CA_1$ and, hence, $(\mathcal{M}_2, s_2) \models A$. The converse can be shown similarly by appealing to condition 2 c) of Definition 14.

$\square$

Bibliography

[AGMT92] Y. Afek, D. S. Greenberg, M. Merritt, and G. Taubenfeld. Computing with faulty shared memory (Extended abstract). In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, pages 47–58, 1992.

[AM86] M. Abadi and Z. Manna. Modal theorem proving. In J. H. Siekmann, editor, *Proceedings of the 8th International Conference on Automated Deduction*, pages 172–189, Oxford, England, July 1986. LNCS 230, Springer-Verlag.

[Bar87] J. Barwise. Three views of common knowledge. In M. Y. Vardi, editor, *Proceedings of the 2nd Conference on Theoretical Aspects of Reasoning about Knowledge (TARK II)*, pages 365–379. Morgan Kaufmann, 1987.

[BND89] A. Bar-Noy and D. Dolev. Shared-memory vs. message-passing in an asynchronous distributed environment. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing*, pages 307–318, 1989.

[Bra87] G. Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75:130–143, 1987.

[BT85] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, 1985.

[Cat91] L. Catach. TABLEAUX: a general theorem prover for modal logics. *Journal of Automated Reasoning*, 7(4):489–510, December 1991.

[CD88] E. M. Clarke and I. A. Draghicescu. Expressibility results for linear-time and branching-time logics. In J. W. de Bakker and W.-P. de Roever, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 428–437. (LNCS 354), Springer-Verlag, May/June 1988.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronisation skeletons using branching time temporal logic. In D. Kozen, editor, *Logics of Programs*, pages 52–71. LNCS 131, Springer-Verlag, 1981.

[CES83] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications: a practical approach. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pages 117–126, 1983.

[Cha90] S. Chaudhuri. Agreement is harder than consensus: Set consensus problems in totally asynchronous systems. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, pages 311–324, 1990.

[CM86] K. M. Chandy and J. Misra. How processes learn. *Distributed Computing*, 1(1):40–52, 1986.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley Publishing Company, 1988.

[Coa88] B. Coan. A compiler that increases the fault tolerance of asynchronous protocols. *IEEE Transactions on Computers*, 37(12):1541–1553, 1988.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, 1987.

[DLS88] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[DM86] C. Dwork and Y. Moses. Knowledge and common knowledge in a byzantine environment: Crash failures. In J. Y. Halpern, editor, *Proceedings of the 1986 Conference on Theoretical Aspects of Reasoning about Knowledge (TARK I)*, pages 149–169, 1986.

[DM90] C. Dwork and Y. Moses. Knowledge and common knowledge in a byzantine environment: Crash failures. *Information and Computation*, 88:159–186, 1990.

[DS84] C. Dwork and D. Skeen. Patterns of communication in consensus protocols. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 143–153, 1984.

[EH83] E. A. Emerson and J. Y. Halpern. 'Sometimes' and 'not never' revisited: on branching versus linear time. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 127–140, 1983.

[EH85] E. A. Emerson and J. Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, 30(1):1–24, 1985.

[EL85] E. A. Emerson and C. L. Lei. Modalities for model checking: Branching time strikes back. In *Proceedings 12th ACM Symposium on Principles of Programming Languages*, pages 84–95, 1985.

[End77] H. Enderton. *Elements of Set Theory.* Academic Press, Inc., 1977.

[ES88] E. A. Emerson and J. Srinivasan. Branching time temporal logic. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 123–172. LNCS 354, Springer-Verlag, May/June 1988.

[FHV88] R. Fagin, J. Y. Halpern, and M. Y. Vardi. What can machines know? On the properties of knowledge in distributed systems. Technical Report RJ6250, IBM Almaden Research Center, 1988.

[FHV91] R. Fagin, J. Halpern, and M. Vardi. A model-theoretic analysis of knowledge. *Journal of the ACM*, 38(2):382–428, April 1991.

[FHV92] R. Fagin, J. Y. Halpern, and M. Y. Vardi. What can machines know? On the properties of knowledge in distributed systems. *Journal of the ACM*, 39(2):328–376, April 1992.

[FL77] M. J. Fischer and R. E. Ladner. Propositional modal logic of programs. In *Proceedings of the Nineth Annual ACM Symposium on the Theory of Computing*, pages 286–294, Boulder, CO, May 1977.

[FL79] M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18:194–211, 1979.

[FLM86] M. Fischer, N. Lynch, and M. Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1:26–39, 1986.

[FS78] L. Flon and N. Suzuki. Consistent and complete proof rules for the total correctness of parallel programs. In *Proceedings of the Nineteenth Annual Symposium on Foundations of Computer Science*, 1978.

[Gol92] R. Goldblatt. *Logics of Time and Computation.* CSLI/Stanford, 1992.

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Conference Record of the 7th Annual ACM Symposium on Principles of Programming Languages*, pages 163–173, 1980.

[HF85] J. Y. Halpern and R. Fagin. A formal model of knowledge, action and communication in distributed systems:Preliminary report. In *Proceedings of the 4th ACM Symposium on Principles of Distributed Computing*, pages 224–236, 1985.

[HM] J. Halpern and Y. Moses. A guide to completeness and complexity for modal logics of knowledge and belief. (manuscript 1991).

[HM84] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. In *Proceedings of the 3rd ACM Conference on Principles of Distributed Computing*, pages 50–61, 1984.

[HM85] J. Halpern and Y. Moses. A guide to the modal logics of knowledge and belief. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI-85)*, pages 480–490, 1985.

[HM90] J. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 1990.

[HMW90] J. Y. Halpern, Y. Moses, and O. Waarts. A characterization of eventual Byzantine agreement. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, pages 333–346, 1990.

[HV87] J. Halpern and M. Vardi. The complexity of reasoning about knowledge and time, I: Lower bounds. Technical Report RJ5764, IBM Almaden Research Center, 1987.

[HV89] J. Halpern and M. Vardi. The complexity of reasoning about knowledge and time, I: Lower bounds. *Journal of Computer and System Sciences*, 38(1):195–237, 1989.

[HZ87] J. Halpern and L. Zuck. A little knowledge goes a long way: Simple knowledge-based derivations and correctness proofs for a family of protocols. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 269–280, 1987.

[HZ92] J. Halpern and L. Zuck. A little knowledge goes a long way: Knowledge-based derivations and correctness proofs for a family of protocols. *Journal of the ACM*, 39(3):449–478, July 1992.

[Kri63] S. Kripke. A semantical analysis of modal logic I: normal modal propositional calculi. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[KT87] S. Katz and G. Taubenfeld. What processes know: Definitions and proof methods (prelimimary version). In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 249–262, 1987.

[Lam80] L. Lamport. 'Sometimes' is sometimes 'Not never'. In *Proceeedings of the 7th Annual ACM Symposium on Principles of Programming Languages*, pages 174–185, 1980.

[Leh84] D. Lehmann. Knowledge, common knowledge and related puzzles (Extended summary). In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 62–67, 1984.

[LP84] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages*, pages 97–107, 1984.

[LR86] R. E. Ladner and J. H. Reif. The logic of distributed protocols. In J. Halpern, editor, *Conference on Theoretical Aspects of Reasoning about Knowledge*, pages 207–222. Morgan Kaufmann, 1986.

[MDH86] Y. Moses, D. Dolev, and J. Halpern. Cheating husbands and other stories: A case study of knowledge, action, and communication. *Distributed Computing*, 1:167–176, 1986.

[Mic89] R. Michel. A categorical approach to distributed systems, expressibility and knowledge. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing*, pages 129–143, 1989.

[Mos87] Y. Moses. Resource-bounded knowledge. In *Proceedings of the 2nd Conference on Theoretical Aspects of Reasoning about Knowledge (TARK II)*, pages 261–275, 1987.

[MT88] Y. Moses and M. Tuttle. Programming simultaneous actions using common knowledge. *Algorithmica*, 3:121–169, 1988.

[MT91] M. Merritt and G. Taubenfeld. Knowledge in shared memory systems (Preliminary version). In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, pages 189–200, 1991.

[MT92] M. Merritt and G. Taubenfeld. Errata: Knowledge in shared memory systems (PODC 1991). In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, pages 285–286, 1992.

[Nei87] G. Neiger. Knowledge consistency: A useful suspension of disbelief. In *Proceedings of the 2nd Conference on Theoretical Aspects of Reasoning about Knowledge (TARK II)*, pages 295–308, 1987.

[NT87] G. Neiger and S. Toueg. Substituting for real time and common knowledge in asynchronous distributed systems. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 281–293, 1987.

[NT88] G. Neiger and S. Toueg. Automatically increasing the fault-tolerance of distributed systems (Preliminary Version). In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, pages 248–262, 1988.

[NT93] G. Neiger and S. Toueg. Simulating synchronized clock and common knowledge in distributed systems. *Journal of the ACM*, 40(2):334–367, April 1993.

[Par81] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proceedings of the 5th GI Conference on Theoretical Computer Science*, pages 167–183. LNCS 104, Springer Verlag, 1981.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 1977 IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[Pnu86] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In de Bakker, editor, *Current Trends in Concurrency, LNCS 224*, pages 510–584. Springer-Verlag, 1986.

[Pra76] V. R. Pratt. Semantical considerations on floyd-hoare logic. In *Proceedings of the 17th IEEE Symposium on Foundations of Computer Science*, pages 109–121, 1976.

[Pra78] V. R. Pratt. A practical decision method for propositional dynamic logic. In *10th ACM Symposium on Theory of Computing*, pages 326–337, 1978.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[PT92] P. Panangaden and K. Taylor. Concurrent common knowledge:Defining agreement for asynchronous systems. *Distributed Computing*, 6(2):73–94, 1992.

[RU71] N. Rescher and A. Urquhart. *Temporal Logic*. Springer Verlag, 1971.

[San91] B. Sanders. A predicate transformer approach to knowledge and knowledge-based protocols. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, pages 217–230, 1991.

[ST87] T. Srikanth and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2:80–94, 1987.

[Wel87] J. L. Welch. Simulating synchronous processors. *Information and Computation*, 74:159–171, 1987.

[Wol81] P. Wolper. Temporal logic can be more expressive. In *22nd Annual Symposium on Foundations of Computer Science*, pages 340–348, 1981.

[Wol86] Wolper. Expresing interesting properties of programs in propositional temporal logic. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, pages 184–193, 1986.

[YK88] M. Yamashita and T. Kameda. Computing on anonymous networks. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, pages 117–130, 1988.

[Zuc86] L. Zuck. *Past temporal logic.* PhD thesis, Weizmann Institute of Science, Rehovot, Israel, August 1986.