

Abstract

VENKATRAMAN, MAHADEVAN. Specifying and Verifying Compliance in Commitment Protocols (Under the direction of Dr. Munindar P. Singh).

Interaction protocols are specific, often standard, constraints on the behaviors of autonomous agents in a multiagent system. Protocols are essential to the functioning of open systems, such as those that arise in most interesting web applications. A variety of common protocols in negotiation and electronic commerce are best treated as *commitment protocols*, which may be defined and analyzed in terms of the creation, satisfaction, or manipulation of the commitments among the participating agents.

When protocols are employed in open environments, such as the Internet, they must be executed by agents that behave more or less autonomously and whose internal designs are not known. In such settings, therefore, there is a risk that the participating agents may fail to comply with the given protocol. Without a rigorous means to verify compliance, the very idea of protocols for interoperation is subverted. We develop an approach for verifying whether the behavior of an agent complies with a given commitment protocol. Our approach requires the specification of commitment protocols in temporal logic, and involves a novel way of synthesizing and applying ideas from distributed computing and logics of program.

SPECIFYING AND VERIFYING COMPLIANCE IN COMMITMENT PROTOCOLS

BY

MAHADEVAN VENKATRAMAN

A THESIS SUBMITTED TO THE GRADUATE FACULTY OF
NORTH CAROLINA STATE UNIVERSITY
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

RALEIGH

SEPTEMBER 1999

APPROVED BY:

CHAIR OF ADVISORY COMMITTEE

Biography

Mahadevan Venkatraman was born on February 16th, 1976 in Mumbai, India. He lived in Anushaktinagar, Mumbai until January 1997. His schooling was from Atomic Energy Central School until tenth grade, and from Atomic Energy Junior College until the twelfth grade.

Mahadevan obtained his Bachelor of Technology degree in Computer Science and Engineering from the Indian Institute of Technology-Bombay, India in 1997. He was a Masters student at North Carolina State University in the Department of Computer Science from Fall 1997 to August 1999. He worked as a research assistant under Dr. Munindar Singh during this period. Mahadevan did an internship at Ericsson in Summer 1998. He was elected to the Honor Society of Phi Kappa Phi in 1999.

Acknowledgments

This work is supported by the National Science Foundation under grants IIS-9529179 and IIS-9624425, and IBM corporation. I am very grateful to my advisor Dr. Munindar Singh, without whose constant guidance, support and encouragement, this thesis would not have been possible. I am indebted to him for all the time he has put into this work and for all the brainstorming sessions that we have had together. I am also very thankful to Dr. K.C. Tai and Dr. Injong Rhee for all the useful ideas and pointers they have suggested to me regarding this work. I am also indebted to Feng Wan and Sudhir Rustogi for useful discussions and to Jie Xing and Yu Bin for their related presentations. I am very thankful to Sudhir Rustogi for his help in preparing this document.

Table of Contents

List of Figures	vi
1 Introduction	1
1.1 Commitments in an Open Architecture	6
1.2 Languages for Defining Protocols	8
1.3 Contributions	9
1.4 Organization	10
2 Technical Framework	11
2.1 Potential Causality	11
2.2 Temporal Logic	13
2.3 Social Commitments	15
3 Specification of a Commitment Protocol	20
3.1 Specifying Commitment Protocols	20
3.1.1 Protocol Server	22
3.1.2 BNF	23
3.2 The Layering Approach	25
3.2.1 Commitment	26
3.2.2 Coordination	26
3.2.3 Authentication	28
3.3 Representing Well-Known Protocols	28
4 Verification	36
4.1 The Quasimodel and the General Model	38
4.2 Implications of the Specification Language	41
4.3 Reasoning with the Quasimodel	43
4.4 Model Checking	47
4.5 Handling Message Losses and Delays	50

4.5.1	Message Losses	51
4.5.2	Message Delays	52
5	Conclusion	54
5.1	Literature	56
5.2	Future Directions	57

List of Figures

2.1	Vector clocks in the fish market protocol	13
2.2	Message pattern for delegate(l) and assign(r).	18
3.1	Different layers and their functionality	26
3.2	Skeletons of the haggling protocol	31
3.3	Skeletons of the fish market protocol	34
4.1	An example quasimodel and its corresponding general model	40
4.2	Observations for auctioneer and a bidder in the fish market protocol	41
4.3	Illustration of Theorem 3	44
4.4	State transition diagram of a commitment	48
4.5	State maintenance of commitments	49

Chapter 1

Introduction

Interaction among agents is the distinguishing property of multiagent systems. However, ensuring that only the desirable interactions occur is one of the most challenging aspects of multiagent system analysis and design. This is especially so when the given multiagent system is meant to be used as an open system, for example, in web-based applications.

Because of its ubiquity and ease of use, the web is rapidly becoming the platform of choice for a number of important applications, such as trading, supply-chain management, and in general electronic commerce. However, the web can enforce few constraints on the behavior of agents. Current approaches to security on the web emphasize how the different parties to a transaction may be authenticated or how their data may be encrypted to prevent unauthorized access. Even with authentication and controlled access, the parties would neither have support beyond conventional protocol techniques (such as finite state machine models) to specify the desired interactions nor to detect any violation. Authentication and access control are conceptually orthogonal to ensuring that the parties behave and interact correctly. Even when the parties are authenticated, they may act undesirably through error or malice. It is important in such situations that agents be *accountable* for their actions.

Definition 1 Accountability is the property whereby the association of a unique originator

with an object or action can be proved to a third party [Kailar, 1995]. ■

Without proper accountability assurances in electronic commerce transactions, there would be no means to reliably enforce punitive measures against fraudulent individuals or organizations [Kailar, 1995, Kessler and Neuman, 1998]. Non-repudiation is another essential security service required in electronic commerce. Non-repudiation services protect the parties involved in a transaction against the other party denying that a particular action or event took place. The two basic non-repudiation services required to establish accountability of parties are *non-repudiation of origin (NRO)* and *non-repudiation of receipt (NRR)* [Zhou and Gollmann, 1997]. *NRO* provides the recipient with evidence of origin of a message, which protects against any attempt by the originator to falsely deny having sent the message. In this respect, *NRO* is similar to accountability. On the other hand, *NRR* provides the originator of a message with evidence of receipt, which protects against any attempt by the recipient to falsely deny having received the message.

However, accountability and non-repudiability are not sufficient to ensure correct behavior by an agent in situations where the time corresponding to a particular action is important. In the Internet, where messages are lost or delayed and not necessarily delivered in a FIFO manner, accountability becomes a rather weak property. As an example to demonstrate the importance of time consider the following example of a fish market protocol [Rodríguez-Aguilar et al., 1998].

Example 1 In the fish market protocol, we are given agents of two roles: a single auctioneer and one or more potential bidders. The fish market protocol is designed to sell fish. The seller or auctioneer announces the availability of a bucket of fish at a certain price. The bidders gathered around the auctioneer can scream back *Yes* if they are interested and *No* if they are not; they may also stay quiet, which is interpreted as a lack of interest or *No*. If exactly one bidder says *Yes*, the auctioneer will sell him the fish; if no one says *Yes*,

the auctioneer lowers the price; if more than one bidder says *Yes*, the auctioneer raises the price. In either case, if the price changes, the auctioneer announces the revised price and the process iterates. ■

Suppose, in the above example, a bidder responds with a *Yes* to a certain price quote, but the message gets delayed to a point where the advertised price becomes much higher. In this case, the buyer will suffer a loss if he is unable to further evidence regarding the time he sent the message. This situation illustrates the importance of associating time with actions. Even if we assume that we have an effective timestamping mechanism which allows us to determine that the message was an old one, we are still faced with the problem of determining if the agents are following the behavioral rules of the given protocol. For example, if the auctioneer backs out of the auction, even if there is exactly one bidder who said *Yes*, then there is no way to automatically *prove* the incorrect behavior of the auctioneer. It is up to a human being to examine the trace of each program and determine that the auctioneer committed a violation. Specification and verification of such *behavioral* properties of agents is the main concern of this study. It is also worthwhile to note that we do not concern ourselves with fairness properties of such protocols. As an instance, the fish market protocol does not incorporate fairness criteria based on varying network delays among different bidders. If a bidder has a slow network connection to the auctioneer, then he is clearly at a disadvantage with respect to other bidders and we do not explicitly model this fact in our system.

The web provides an excellent infrastructure through which agents can communicate with one another. But the above problems are exacerbated when agents are employed in the web. In contrast with traditional programs and interfaces, neither the behaviors and interactions nor the construction of web agents is fixed or under the control of a single authority. In general, in an open system, the member agents are contributed by several

sources and serve different interests. Thus, these agents must be treated as

- *autonomous*—with few constraints on behavior, reflecting the independence of their users, and
- *heterogeneous*—with few constraints on construction, reflecting the independence of their designers.

Effectively, a multiagent system is specified as a kind of standard that its member agents must respect. In other words, a multiagent system can be thought of as specifying a protocol that governs how its member agents must act. For our purposes, the standard may be *de jure* as created by a standards body, or *de facto* as may emerge from practice or even because of the arbitrary decisions of a major vendor or user organization. All that matters for us is that a standard imposes some restrictions on the agents.

The fish market protocol is an example of such a standard in an electronic agent marketplace. Because of its relationship to protocols in electronic commerce and because it is more general than the popular English and Dutch auctions, the fish market protocol has become an important one in the recent multiagent systems literature [Rodríguez-Aguilar et al., 1998]. Accordingly, we use it as our main example in this study.

Because of the autonomy and heterogeneity requirements of open systems, compliance verification can be based neither on the internal designs of the agents nor on concepts such as beliefs, desires, and intentions that map to internal representations [Singh, 1998a]. Belief logics are not concerned about proving something to a third party and as a result cannot be employed in electronic commerce where *provability* is a necessity [Kessler and Neuman, 1998]. The only way in which compliance can be verified is based on the external behavior of the participating agents. In this respect, it is similar to the traditional problem of black-box testing or debugging of distributed programs. But there are notable differences between testing and debugging and our approach.

Testing is the process of executing a program with selected tests in order to detect faults [Tai and Carver, 1996]. Testing is not applicable to our domain, because the program code or even the executable code for an agent might not be available for testing. This may be so because the vendors might not want to share their trade secrets. We could, in principle, try to test an agent by verifying its compliance in different situations, but again this is subject to availability of the code.

Debugging is the process of locating and correcting the faults detected during testing [Tai and Carver, 1996]. Our approach is, in principle, similar to debugging in the sense that we attempt to check compliance of a particular run of a system with a predefined specification. The differences between our approach and debugging are two-fold. In our approach, the specification language is based only on the interactions between the agents, while in debugging the emphasis includes individual process semantics. In our approach, we respect the autonomy of agents and hence specify an agent at a very high level that does not constrain the agent to perform unnecessary actions, except to resolve the commitments it creates. Furthermore, an agent can perform certain predefined operations on commitments, which makes the language more expressive in terms of supporting flexible commitments. In debugging, the exact interprocess semantics needs to be defined, or in other words, all the synchronization events need to be modeled explicitly. Hence, debugging approaches tend to be rather rigid in their definition of protocols, and are not suitable for electronic commerce applications. Despite these differences, some underlying concepts are shared between both debugging and compliance verification. Both approaches involve a formal specification language and corresponding methods to test compliance, which typically rely on temporal logics and potential causality. Cheng & Wallentine describe a language, DEBL, that focuses on interprocess semantics and in which the debugging is done by examining traces of execution, rather than by interactively controlling the program. This debugging method

comes close to our approach, except for the differences noted above.

Another important distinction is that verification may be performed by a central authority or by any of the participating agents whereas testing and debugging are performed centrally. This verification is based solely on the behavior of an agent, in other words, only on the messages that it generates. However, the requirements for behavior in multiagent systems can be quite subtle. Thus, along with languages for specifying such requirements, we need corresponding techniques to verify compliance.

1.1 Commitments in an Open Architecture

There are three levels of architectural concern in a multiagent system. One deals with individual agents; another deals with the systemic aspects of how different services and brokers are arranged. Both of these have received much attention in the literature. In the middle is the multiagent *execution* architecture, which has not been as intensively studied within the multiagent research community. An execution architecture must ultimately be based on distributed computing ideas albeit with an open flavor, e.g., [Agha and Jamali, 1999, Carriero and Gelernter, 1992, Francez and Forman, 1996]. A well-defined execution functionality can be given a principled design, and thus facilitate the construction of robust and reusable systems. Some recent work within multiagent systems, e.g., Ciancarini *et al.* [1996, 1998] and Singh [1998c], has begun to address this level.

Much of the work on this broad theme, however, focuses primarily on coordination, which we think of as the lowest level of interaction. Coordination deals with how autonomous agents may align their activities in terms of what they do and when they do it. However, there is more to interaction in general and to compliance in particular. Specifically, interaction must include some consideration of the commitments that the agents

enter into with each other. The commitments of the agents are not only base-level commitments dealing with what actions they must or must not perform, but also metacommitments dealing with how they adjust their base-level commitments [Singh, 1999]. Commitments provide a layer of coherence to the agents' interactions with each other. They are especially important in environments where we need to model any kind of contractual relationships among the agents. The ContractNet approach discussed in [Smith, 1980] tries to model such kind of contractual relationships, but it lacks a formal verification methodology.

Such contractual environments are crucial wherever open multiagent systems must be composed on the fly, e.g., in electronic commerce of various kinds on the Internet. The presence of commitments as an explicit first-class object results in considerable flexibility of how the protocols can be realized in changing situations. We term such augmented protocols *commitment protocols*.

Verifying compliance deals with checking if the different agents resolve their commitments to one another. These commitments are created as agents send messages to each other. Also, some commitments are created when the agents bind themselves to a given protocol, and may be thought of as rules of the protocol. The task of determining if different agents are resolving their commitments to each other are performed by the agents themselves. In case of conflict, they submit their evidence to a trusted third party who then resolves the conflict based on the evidence.

Example 2 We informally describe the protocol of Example 1 in terms of commitments. When a bidder says *Yes*, he commits to buying the bucket of fish at the advertised price. When the auctioneer advertises a price, he commits that he will sell the fish at that price if he gets a unique *Yes*. Neither commitment is irrevocable. For example, if the fish are spoiled, the auctioneer releases the bidder from paying for them. Specifying all possibilities in terms of irrevocable commitments would complicate each commitment, but would still fail

to capture the practical meanings of such a protocol. For instance, the auctioneer may not honor his offering price if a sudden change in weather indicates that fishing will be harder for the next few days. ■

Commitments could also be associated with a continuous penalty level, where commitments vary from unbreakable to breakable as a continuum by assigning a commitment breaking cost to each commitment separately [Sandholm and Lesser, 1995]. Agents form and break commitments to maximize their own gain. The advantages of a leveled commitment protocol are formally analyzed in [Sandholm and Lesser, 1995]. Although we do not discuss leveled commitments in our study, we believe the same could be incorporated in our framework through appropriate metacommitments.

1.2 Languages for Defining Protocols

The need for languages to define protocols is well understood in the literature. In the computer security community, the focus has been on defining languages for cryptographic protocols or authentication protocols. On the other hand, this study is concerned with the high-level behavioral properties of agents rather than low-level authentication properties. As a result, we need a language that helps us specify typical kinds of protocols that are prevalent in electronic commerce applications. Desirable properties for such a language include

- Expressiveness
- Verifiability of compliance with the stated requirements
- Efficiency of the associated verification procedure

With these needs in mind, we develop a language based on temporal logic and potential causality. We show applicability of the language to standard protocols to establish the

expressiveness of the language.

1.3 Contributions

Agents participating in a commitment protocol send messages to each other forming social commitments. In addition, there are metacommitments specific to the given protocol that exist from the start of the system. These could be thought of as laws of the system. These commitments are expressed in a specification language based on operations on commitments and the Computation Tree Logic (CTL).

It is essential for the participating agents to determine if other agents are indeed discharging their commitments to them. The agents can do so by creating a quasimodel based on the causal order of messages and apply a model checking algorithm over it. We do not concern ourselves with the details of the model checking, but concentrate more on the conditions under which an individual agent may check others' commitments to it and also optionally keep track of its own pending commitments.

An agent can perform this local verification on the quasimodel itself by transforming the temporal formulae associated with the commitments. Traditionally, one would blow up the quasimodel into a finitely branching, infinitely deep tree model by considering all possible interleavings of the causally unrelated messages. We show that it is unnecessary to do so. This is one of the salient features of the specification language that we develop. Our main contributions include

- developing a high-level specification language to specify behavioral properties of agents.
- incorporating potential causality and temporal logic for the specification and verification of a commitment protocol.

- showing that it is unnecessary to consider all interleavings between causality unrelated messages, and thereby suggesting an efficient procedure to check compliance.
- showing how to verify compliance based on local information.
- handling message losses and delays
- achieving clear delineation between the roles of the commitment layer and the coordination layer.

1.4 Organization

The rest of this document is organized as follows. Chapter 2 presents our technical framework, which combines commitments, potential causality, and temporal logic. Chapter 3 describes the components of a commitment protocol. It also discusses our specification language and gives a BNF for the same. Chapter 4 presents our approach for verifying (non-)compliance of agents with respect to a commitment protocol. Chapter 5 concludes with a discussion of our major themes, the literature, and some important issues that remain outstanding.

Chapter 2

Technical Framework

Commitment protocols as defined here are a multiagent concept. They are far more flexible and general than commitment protocols in distributed computing and databases, such as *two-phase commit* [Gray and Reuter, 1993, pp. 562–573]. This is because our underlying notion of commitment is flexible, whereas traditional commitments are rigid and irrevocable. However, because multiagent systems are distributed systems and commitment protocols are protocols, it is natural that techniques developed in classical computer science will apply here. Accordingly, our technical framework integrates approaches from distributed computing, logics of program, and distributed artificial intelligence.

2.1 Potential Causality

The key idea behind potential causality is that the ordering of events in a distributed system can be determined only with respect to an observer [Lamport, 1978]. If event e precedes event f with respect to an observer, then e may *potentially* cause f . The observed precedence suggests the possibility of an information flow from e to f , but without additional

knowledge of the internals of the agents, we cannot be sure that true causation was involved. It is customary to define the local time of an agent as the number of steps it has executed. [Schwarz and Mattern, 1994] and [Fidge, 1991] introduced the notion of a vector clock, which defines a partial order between events of a distributed system. A *vector clock* is a vector, each of whose elements corresponds to the local time of each communicating agent. A vector v is considered later than a vector u if v is later on some, and not sooner on any, element.

Definition 2 A clock over n agents is an n -ary vector $v = \langle v_1 \dots v_n \rangle$ of natural numbers. The starting clock is $\vec{0} \triangleq \langle 0 \dots 0 \rangle$. ■

Notice that the vector representation is just a convenience. We could just as well use pairs of the form $\langle \text{agent-id}, \text{local-time} \rangle$, which would allow us to model systems of varying membership more easily.

Definition 3 Given n -ary vectors u and v , $u \prec v$ if and only if $(\forall i : 1 \leq i \leq n : u_i \leq v_i)$ and $(\exists i : 1 \leq i \leq n : u_i < v_i)$. ■

Each agent starts at $\vec{0}$. It increments its entry in that vector whenever it performs a local event. It attaches the entire vector as a timestamp to any message it sends out. When an agent receives a message, it updates its vector clock to be the element-wise maximum of its previous vector and the vector timestamp of the message it received. Intuitively, the message brings news of how far the system has progressed; for some agents, the recipient may have better news already. However, any message it sends after this receive event will have a later timestamp than the message just received.

Example 3 Figure 2.1 illustrates the evolution of vector timestamps for one possible run of the fish market protocol. In the run described here, the auctioneer (A) announces a price

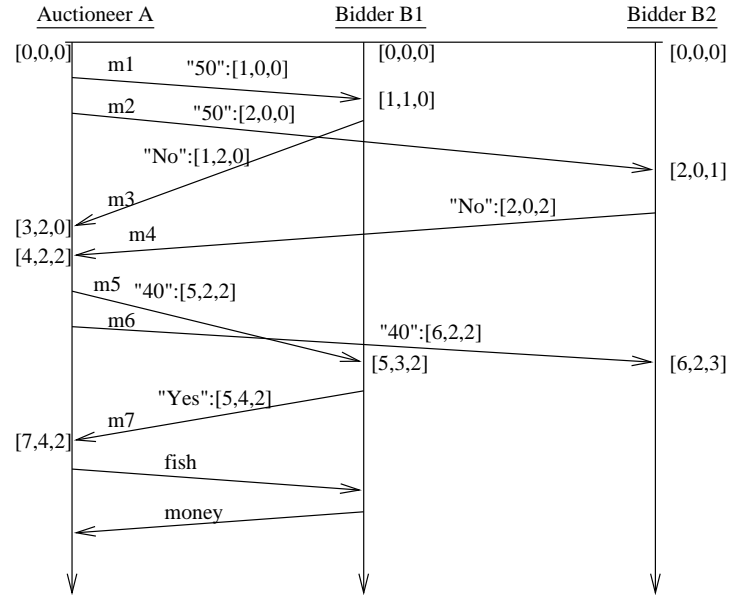


Figure 2.1: Vector clocks in the fish market protocol

of 50 for a certain bucket of fish. Bidders B1 and B2 both decline. A lowers the price to 40 and announces it. This time B1 says *Yes*, leading A to transfer the fish to B1 and B1 to send money to A. For uniformity, the last two steps are also modeled as communications. The messages are labeled m_i to facilitate reference from the text. ■

2.2 Temporal Logic

The progression of events, which is inherent in the execution of any protocol, suggests the need for representing and reasoning about time. Temporal logics provide a well-understood means of doing so, and have been applied in various subareas of computer science. Because of their naturalness in expressing properties of systems that may evolve in more than one possible way and for the efficiency of reasoning that they support, the branching-time logics have been especially popular in this regard [Emerson, 1990]. Of these, the best known

is Computation Tree Logic (CTL), which we adapt here in our formal language \mathcal{L} . Conventionally, a model of CTL is expressed as a tree. Each node in the tree is associated with a state of the system being considered; the branches of the tree or *paths* thus indicate the possible courses of events or ways in which the system's state may evolve. CTL provides a natural means by which to specify acceptable behaviors of the system.

The following Backus-Naur Form (BNF) grammar with a distinguished start symbol L gives the syntax of \mathcal{L} . \mathcal{L} is based on a set Φ of atomic propositions. Below, *slant* typeface indicates nonterminals; \longrightarrow and $|$ are metasymbols of BNF specification; \ll and \gg delimit comments; the remaining symbols are terminals. As is customary in formal semantics, we are only concerned with abstract syntax.

L1. $L \longrightarrow Prop \ll\text{atomic propositions: members of } \Phi \gg$

L2. $L \longrightarrow \neg L \ll\text{negation}\gg$

L3. $L \longrightarrow L \wedge L \ll\text{conjunction}\gg$

L4. $L \longrightarrow A P \ll\text{universal quantification over paths}\gg$

L5. $L \longrightarrow E P \ll\text{existential quantification over paths}\gg$

L6. $P \longrightarrow L U L \ll\text{until: operator over a single path}\gg$

The meanings of formulas generated from L are given relative to a model and a state in the model. The meanings of formulas generated from P are given relative to a path and a state on the path. The boolean operators are standard. Useful abbreviations include $\text{false} \equiv (p \wedge \neg p)$, for any $p \in \Phi$, $\text{true} \equiv \neg \text{false}$, $p \vee q \equiv \neg p \wedge \neg q$ and $p \rightarrow q \equiv \neg p \vee q$. The temporal operators A and E are quantifiers over paths. Informally, pUq means that on a given path from the given state, q will eventually hold and p will hold until q holds. Fq

means “eventually q ” and abbreviates $\text{true} \cup q$. Gq means “always q ” and abbreviates $\neg F \neg q$. Therefore, $E p \cup q$ means that on some future path from the given state, q will eventually hold and p will hold until q holds.

Definition 4 $M = \langle S, <, \mathbf{I} \rangle$ is a formal model for \mathcal{L} . S is a set of states; $< \subseteq S \times S$ is a partial order indicating branching time, and $\mathbf{I} : S \mapsto \mathcal{P}(\Phi)$ is an interpretation, which tells us which atomic propositions are true in a given state. For $t \in S$, \mathbf{P}_t is the set of paths emanating from t . ■

$M \models_t p$ expresses “ M satisfies p at t ” and $M \models_{P,t} p$ expresses “ M satisfies p at t along path P .”

M1. $M \models_t \psi$ iff $\psi \in \mathbf{I}(t)$, where $\psi \in \Phi$

M2. $M \models_t p \wedge q$ iff $M \models_t p$ and $M \models_t q$

M3. $M \models_t \neg p$ iff $M \not\models_t p$

M4. $M \models_t \mathbf{A}p$ iff $(\forall P : P \in \mathbf{P}_t \Rightarrow M \models_{P,t} p)$

M5. $M \models_t \mathbf{E}p$ iff $(\exists P : P \in \mathbf{P}_t \text{ and } M \models_{P,t} p)$

M6. $M \models_{P,t} p \cup q$ iff $(\exists t' : t \leq t' \text{ and } M \models_{P,t'} q \text{ and } (\forall t'' : t \leq t'' \leq t' \Rightarrow M \models_{P,t''} p))$

The above is an abstract semantics. In Section 4.3, we specify the concrete form of Φ , S , $<$, and \mathbf{I} , so the semantics can be exercised in our computations.

2.3 Social Commitments

As discussed earlier, commitments give the core meaning of a protocol, and capture the behavioral relationships that agents have with one another. Our approach builds on a flexible

and powerful variety of social commitments, which are the commitments of one agent to another [Singh, 1999]. These commitments are defined relative to a *context*, which is typically the multiagent system itself. The *debtor* refers to the agent that makes a commitment, and the *creditor* to the agent who receives the commitment. Thus we have the following logical form.

Definition 5 A commitment is an expression $C(x, y, G, p)$, where x is the debtor, y the creditor, G the context, and p the condition committed to. ■

Definition 6 A commitment $c = C(x, y, G, p)$ is *base-level* if p does not refer to any other commitments; c is a *metacommitment* if p refers to a base-level commitment. ■

Intuitively, a protocol definition is a set of metacommitments for the different roles (along with a mapping of the message tokens to operations on commitments). In combination with what the agents communicate, these lead to base-level commitments being created or manipulated, which is primarily how a commitment may be referred to within a protocol. The violation of a base-level commitment can give us proof or the “smoking gun” that an agent is noncompliant.

The following *operations* on commitments define how they may be created or manipulated. When we view commitments as an abstract data type, the operations are methods of that data type.

Each operation is realized through a simple message pattern, which states what messages must be communicated among which of the participants and in what order. For the operations on commitments that we consider, the patterns are simple. As described below, most patterns require only a single message, but some require three messages. Obeying the specified patterns ensures that the local models have the information necessary for verifying compliance. That the given operation can be performed at all depends on whether

the protocol, through its metacommitments, allows that operation. However, when an operation is allowed, it affects the agents' commitments. For simplicity, we assume that the operations on commitments are given a deterministic interpretation. Here z is an agent and $c = C(x, y, G, p)$ is a commitment.

- O1. $Create(x, c)$ instantiates a commitment c . $Create$ is typically performed as a consequence of the commitment's debtor promising something contractually or by the creditor exercising a metacommitment previously made by the debtor. $Create$ usually requires a message from the debtor to the creditor.
- O2. $Discharge(x, c)$ satisfies the commitment c . It is performed by the debtor concurrently with the actions that lead to the given condition being satisfied, e.g., the delivery of promised goods or funds. For simplicity, we treat the $discharge$ actions as performed only when the proposition p is true. Thus the $discharge$ actions are *detached*, meaning that p can be treated as true in the given moment. We model the $discharge$ as a single message from the debtor to the creditor.
- O3. $Cancel(x, c)$ revokes the commitment c . It can be performed by the debtor as a single message. At the end of this action, $\neg c$ holds. However, depending on the existing metacommitments, the $cancel$ of one commitment may lead to the $create$ of other commitments.
- O4. $Release(G, c)$ or $release(y, c)$ essentially eliminates the commitment c . This is distinguished from both $discharge$ and $cancel$, because $release$ does not mean success or failure, although it lets the debtor off the hook. At the end of this action, $\neg c$ holds. The $release$ action may be performed by the context or the creditor of the given commitment, also as a single message. Because $release$ is not performed by the debtor, different metacommitments apply than for $cancel$.

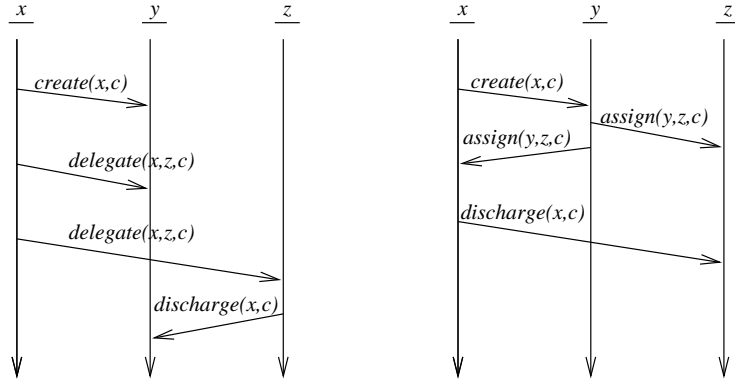


Figure 2.2: Message pattern for $\text{delegate}(l)$ and $\text{assign}(r)$.

- O5. $\text{Delegate}(z, c)$ shifts the role of the debtor to another agent within the same context, and can be performed by the old debtor or the context. Let $c' = C(z, y, G, p)$. At the end of the delegate action, $c' \wedge \neg c$ holds.

To prevent the risk of miscommunication, we require the creditor to also be involved in the message pattern. Figure 2.2(l) shows the associated pattern. The first message sets up the commitment c from x to y and is not part of the pattern. When x delegates the commitment c to z , x tells both y and z that the commitment is delegated. z is now committed to y . Later z may discharge the commitment. The two delegate messages constitute the pattern.

- O6. $\text{Assign}(z, c)$ transfers a commitment to another creditor within the same context, and can be performed by the present creditor or the context. Let $c' = C(x, z, G, p)$. At the end of the assign action, $c' \wedge \neg c$ holds.

Here we require that the new creditor and the debtor are also involved as shown in Figure 2.2(r). The figure shows only the general pattern. Here x is committed to y . When y assigns the commitment to z , y tells both x and z (so z knows it is the new creditor). Eventually, x should discharge the commitment to z . A potentially

tricky situation is if x discharges the commitment c even as y is assigning c to z (i.e., the messages cross). In this case, we require y to discharge the commitment to z —essentially by forwarding the contents of the message from x . Thus the worst case requires three messages.

We write the operations as propositions indicating successful execution. Based on the applicable metacommitments, each operation may entail additional operations that take place implicitly.

These constitute the formal definition of commitments and the operations that can be performed on them. We employ these definitions in our language for specification of a commitment protocol.

Chapter 3

Specification of a Commitment Protocol

This chapter discusses the components of a commitment protocol, and the specification language in particular, that we use to specify and verify commitment protocols.

We present a BNF to define the language and discuss the salient properties of the language with respect to electronic commerce protocols. We also discuss the underlying coordination layer and the role it plays in our verification framework.

3.1 Specifying Commitment Protocols

Protocols in general can be represented by means of communicating finite state machines. We could set up the system in such a way that all agents are constrained to satisfy their skeletal finite state machine thus always resulting in correct and coherent behavior. But, this destroys the autonomy of the agents in being able to perform desired actions. Agents should be able to do anything they want. But specifying and verifying correct behavior in such situations is difficult.

What we need is a trade-off between autonomy and correct behavior. The solution to

this is to have a skeletal program for each role in a protocol. A *skeleton* is a coarse description of how an agent may behave [Singh, 1998c]. A skeleton is associated with each role in the given multiagent system to specify how an agent playing that role may behave in order to coordinate with others. The coordination layer knows about the skeleton and ensures that agents act according to their skeletons. Coordination includes the simpler aspects of interaction, e.g., turn-taking. Coordination is required so that the agents' commitments make sense. For instance, a bidder should not make a bid prior to the advertisement; otherwise, the commitment content of the bid would not even be fully defined.

The skeletons may be constructed by introspection or through the use of a suitable methodology [Singh, 1998d]. No matter how they are created, the skeletons are the first line of compliance verification, because an agent that does not comply with the skeleton for its role is automatically in violation. So as to concentrate on commitments in this study, we postulate that a *proxy* object is interposed between an agent and the rest of the system. The proxy agent ensures that the agent follows the dictates of the skeleton of its role. Proxies are discussed further in Section 3.2.2.

Every agent is forced to comply with the skeleton. Further, every agent in complying with the skeleton creates some commitments. To satisfy these commitments is up to the agent. Thus, we respect the autonomy of the agents in that they are able to decide for themselves whether or not to break commitments. The less restrictive the skeleton, the more autonomous the agents. In the light of the above discussion, we can now state what a protocol specification must include.

Definition 7 A commitment protocol specification has five components :

- Roles and context group
- Domain specific propositions and actions
- Metacommitments

- Protocol tokens whose meaning is described in terms of commitments
- Skeletal programs for all roles. The skeletons describe the basic ordering of actions between agents, i.e., the basic coordination between agents. We use a finite state machine augmented with programming language syntax for this. The representation of the skeleton is shown in the examples.

■

This describes the basic components of a commitment protocol. We now elaborate on how this is realized.

3.1.1 Protocol Server

The protocol server is the main entity that stores the definition of the commitment protocol. It plays the role of a trusted third party in the system. It allows agents to bind themselves to the roles defined in the specification of the protocol and provides them with the coordination stub corresponding to their role in the protocol. The need for this stub is discussed in Section 3.2.2 below. Once the role binding process is complete, the communication between agents is completely distributed and the messages that agents send to each other need not be routed through the protocol server.

The protocol officially begins when the protocol server broadcasts a *start* message to each agent to start the protocol. The protocol ends when each of the individual agents express their desire to end the execution. When that happens, the protocol server broadcasts an *end* message to each agent, and this is when the protocol officially ends. The protocol server can be thought of as the context group of the execution.

Upon completion of the protocol, each agent has to submit a trace of its execution to the protocol server. This could be automatically done by the coordination stub, since it

knows about all the events that happens at the local space of an agent. This trace contains information about the events that happened in an agent's local space. The protocol server may then verify the compliance of all agents in the system with regard to the commitments they formed. Any unresolved or broken commitment constitutes a violation. If a metacommitment exists that describes what needs to be done in such a situation, then the protocol server takes the appropriate action. However, if no metacommitment deals with the situation, then human intervention is required. This describes a retrospective approach to verify compliance but we believe the same could be extended for online checking and control also.

This study is not concerned with how a violation is handled by the authorities once it is detected, but in generating a proof that a violation occurred or, in other words, *provability* of (non-)compliance is the main concern. It is up to the authorities to handle the situation appropriately.

3.1.2 BNF

As discussed above, a commitment protocol consists of metacommitments and tokens whose meaning is expressed in terms of commitments. It turns out that most of the standard protocols in electronic commerce, typically in auction settings, follow the same structure. It would be nice if we had a language for commitment protocols that basically capture the essential structure of protocols, i.e., some kind of a metamodel for protocols. We could then come up with a standard verification algorithm for all protocols belonging to the language. It is also important that the language we define be expressive enough so as to encompass most of the standard commitment protocols in literature. Below we define the BNF syntax corresponding to our language, and apply it to two protocols, namely, the fish market protocol and the haggling protocol. The fish market protocol is well-known in literature

[Rodríguez-Aguilar et al., 1998]. The latter is a hypothetical protocol, but often seen in use in human interactions.

We now define the syntax of the specification language through the following grammar whose start symbol is *Protocol*. The braces { and } indicate that the enclosed item is repeated 0 or more times. The angled brackets \ll and \gg delimit comments. Note that our language is context-free.

The following syntax describes the structure of the commitments (both meta and ordinary) that can be used in a commitment protocol.

- $Protocol \longrightarrow \{Meta\} \{Message\}$
- $Message \longrightarrow Token: \{Act\} \ll\text{each message corresponds to a sequence of actions}\gg$
- $Commitment \longrightarrow C(Debtor, Creditor, Context, Prop)$
- $Prop \longrightarrow AG[BoolAct \Rightarrow AFSendAct] \mid AG[SendAct \Rightarrow BoolAct] \mid AFsend_DomAct$
- $Meta \longrightarrow C(Debtor, Creditor, Context, MetaProp)$
- $MetaProp \longrightarrow AG[BoolAct \Rightarrow AFsend_OpAct] \mid AG[send_OpAct \Rightarrow BoolAct] \ll\text{a metacommitment always has a nested commitment}\gg$
- $BoolAct \longrightarrow \ll\text{boolean combinations of } Act \text{ and } Commitment\gg$
- $Act \longrightarrow SendAct \mid RecvAct$
- $SendAct \longrightarrow send_OpAct \mid send_DomAct \ll\text{all the send events}\gg$
- $RecvAct \longrightarrow recv_OpAct \mid recv_DomAct \ll\text{all the receive events}\gg$
- $OpAct \longrightarrow Operation(Agent, Commitment)$
- $Operation \longrightarrow \ll\text{the six operations}\gg$
- $DomAct \longrightarrow \ll\text{domain-specific actions, e.g., } fish(A, B)\gg$

The above language embeds a subset of CTL, \mathcal{L} , that we defined in Chapter 2. Our

verification approach is to detach the outer actions and commitments, so we can process the inner \mathcal{L} part as a temporal logic. By using commitments and operations on them, instead of simple domain propositions, we can capture a variety of subtle situations, e.g., to distinguish between *release* and *cancel* operations both of which result in the given commitment being removed.

The above syntax allows commitments like $C(A, B, G, AFsend_fish(C, D))$ although these do not make sense. As a rule, we allow only those commitments in which the debtor can verify the truth of the proposition contained in the commitment, or in other words, the debtor sees all the actions present in the definition of the commitment. Specifically, the debtor must be mentioned in the scope of any action present in the commitment definition. A simple algorithmic check can identify commitments of the form described above. Alternatively, we could avoid such commitments from the syntax explicitly by using labels corresponding to agents in the syntax above, but we choose not to do so for simplicity. We just assume commitments of the form described above are not allowed. Protocols where the debtor has less control over the commitments that it creates do not fit our model, but it is worthwhile to note that in competitive scenarios such as electronic commerce, this is often not the case.

3.2 The Layering Approach

Interaction-oriented programming, as described by Singh in [1998c], defines a layered approach to multiagent systems. We use the same concept here, and give it a more concrete form in our approach. For our present purposes, we add a cryptographic layer at the bottom and neglect the collaboration layer. There are three layers that play an important role in our system. These are illustrated in Figure 3.1.

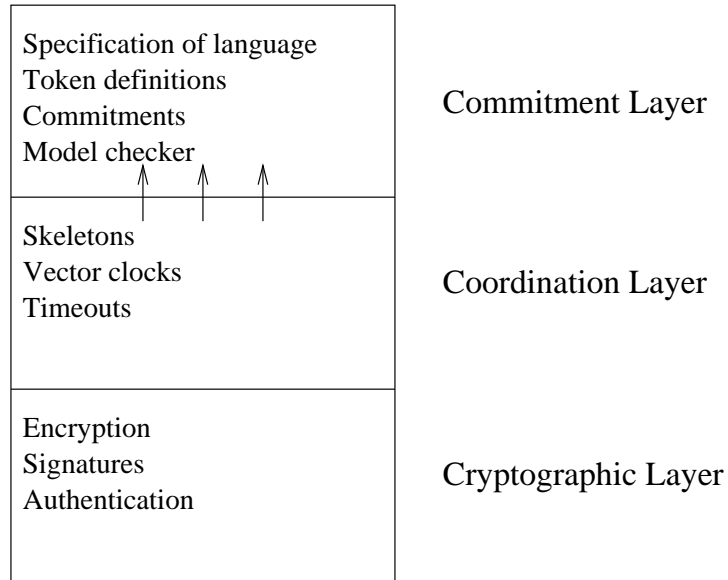


Figure 3.1: Different layers and their functionality

3.2.1 Commitment

The commitment layer bears the high-level definition of the commitment protocol, i.e., the metacommitments and the definitions of the various tokens of the protocol. This study primarily deals with the commitment layer and its verification methodology as applied to our specification language.

3.2.2 Coordination

The coordination layer describes the basic ordering of events in the protocol. Every agent has to comply with its basic coordination constraints so that the messages that agents send to each other make sense. In our approach, we achieve this by defining a coordination skeleton for each role in the protocol. This skeleton is described by using a finite state machine augmented with programming language syntax. There are two ways of imposing the coordination constraints on an agent. One way is to hold an agent responsible if it does not

comply with its skeleton. The other way is to use the coordination layer as the medium or the proxy object through which the agents communicate. The proxy can ensure that agents are indeed complying with their skeleton, since all the messages are routed through it. All it needs to do is to check if the agent's action is consistent with the current state of the finite state machine corresponding of the role. The coordination layer timestamps the messages according to a timestamping mechanism as discussed in Section 2.1. It is possible that agents could forge their timestamps, but there are mechanisms by which these could be prevented [Reiter and Gong, 1993]. For simplicity, we assume that the coordination skeleton corresponding to every role is provided by the protocol server at bind time, and this acts as the trusted stub through which agents communicate with one another. Thus we do not risk the situation in which malicious agents forge their timestamps. Alternatively, techniques for handling reliable message delivery despite byzantine failures could be adopted [Reiter and Gong, 1993]. Also, the coordination layer could sign the messages so that it is not possible to use a different stub from the one provided.

Skeleton specification

We use the following representation for skeletons and tokens. A skeleton is primarily a non-deterministic finite state machine with some additional annotations. Each transition is annotated with a label of the form p/q . Here, p is the action that happens at the agent's local space, and q is an assignment statement. The label is read as *if p then q* . Here, p is the pre-condition for a transition and q is the statement that is executed if the transition is fired. We need to have q because the protocol tokens are not independently representable in terms of commitments but depend on the execution history. We use temporary variables to capture the history of the execution. For instance, in the fish market protocol, if a buyer responds by saying *Yes* to a price quote, the commitment he creates is dependent on the

price quote made by the auctioneer earlier. Note that we are only storing the last statement by the auctioneer rather than the entire history. This illustrates that the commitment layer is dependent upon the coordination layer to provide it with values of variables. This dependence may be thought of as the point where the coordination layer and the commitment layer interface with each other. This is similar to the *service access point (SAP)* concept in the OSI reference model, whereby a higher layer requests services from a lower layer. Figures 3.2 and 3.3 below illustrate the skeleton specification details. In these figures, the first element of every p/q pair is an action or an event that corresponds to the *send* or the *receive* of a token. We distinguish between send and receive events as always. The labels of the some transitions do not have the q part. The symbol e corresponds to the empty string or the epsilon transition, i.e., this transition can be fired without any local event taking place.

Every agent complies with its skeleton in terms of providing the basic coordination. For instance, a seller cannot send 2 price quotes one after the other without receiving a price quote from the buyer. This is easily seen from Figure 3.2. Once an agent reaches its end state, it can receive messages, but cannot send any messages.

3.2.3 Authentication

This layer is the lowest level of compliance checking, and includes cryptographic means by which messages can be authenticated. A key distribution protocol could be used to authenticate messages. We do not concern ourselves with these aspects as they already well-known in literature [Burrows et al., 1990].

3.3 Representing Well-Known Protocols

We now represent some well-known protocols in our language.

Example 4 This is the classical buyer-seller *haggling* protocol. First, a seller proposes a price for an item. The buyer responds with another price quote (smaller if the buyer is rational). The seller then responds with another price quote, and so on.

- If after a certain price quote by the seller, the buyer is satisfied, then he sends the money to the seller upon which the seller sends the item to the buyer.
- If after a certain price quote by the buyer, the seller is satisfied, then he sends the item to the buyer upon which the buyer sends the money to the seller.

We represent the haggling protocol in our framework as follows, thus giving a formal and unambiguous definition to the protocol.

Roles and Context:

- Seller, henceforth called S.
- Buyer, henceforth called B.
- Context group for the haggling protocol, henceforth called HP.

Domain specific propositions and actions:

- $ItemAct(S, B)$ corresponds to the action of giving the item to the buyer.
- $MoneyAct_p(B, S)$ corresponds to the action of giving the money to the seller (p dollars).

Metacommitments:

There are no metacommitments for this protocol.

Protocol Tokens and Skeletons:

In the example, we use temporary variables called *currBuyerPrice* and *currSellerPrice* to store the most recent price quotes by the two parties. Then, the tokens can be associated with actions involving these variables. First, we define some abbreviations for simplicity.

- $CS_p = C(S, B, HP, AG[recv_Money_Act_p(B, S) \Rightarrow AFsend_ItemAct(S, B)])$
- $CB_p = C(B, S, HP, AG[recv_ItemAct(S, B) \Rightarrow AFsend_MoneyAct_p(B, S)])$

Now we define the four tokens that are used in the protocol.

- *SellerPrice_p* : This token is equivalent to releasing the buyer from his previous commitment to the seller, and creates a new commitment for the seller.
release(S, CB_{currBuyerPrice}) followed by
create(S, CS_p)
- *BuyerPrice_p* : This token is equivalent to releasing the seller from his previous commitment to the buyer, and creates a new commitment for the buyer.
release(B, CS_{currSellerPrice}) followed by
create(B, CB_p)
- *ItemAct* : This token corresponds to the actual sending of the item to the buyer.
discharge(S, C(S, B, HP, AFsend_ItemAct(S, B)))
- *MoneyAct_p* : This token corresponds to the actual sending of the money to the seller.
discharge(B, C(B, S, HP, AFsend_MoneyAct_p(B, S)))

Fig 3.2 shows the skeletons of the seller and the buyer in the haggling protocol. ■

The above was a simple example in which the maximum order of a commitment was one. In general, protocols are more complex, and we need higher order commitments. With a view to this, we look at the fish market protocol, which is a more general form of an auction protocol. Example 5 applies the above specification language on the fish market protocol.

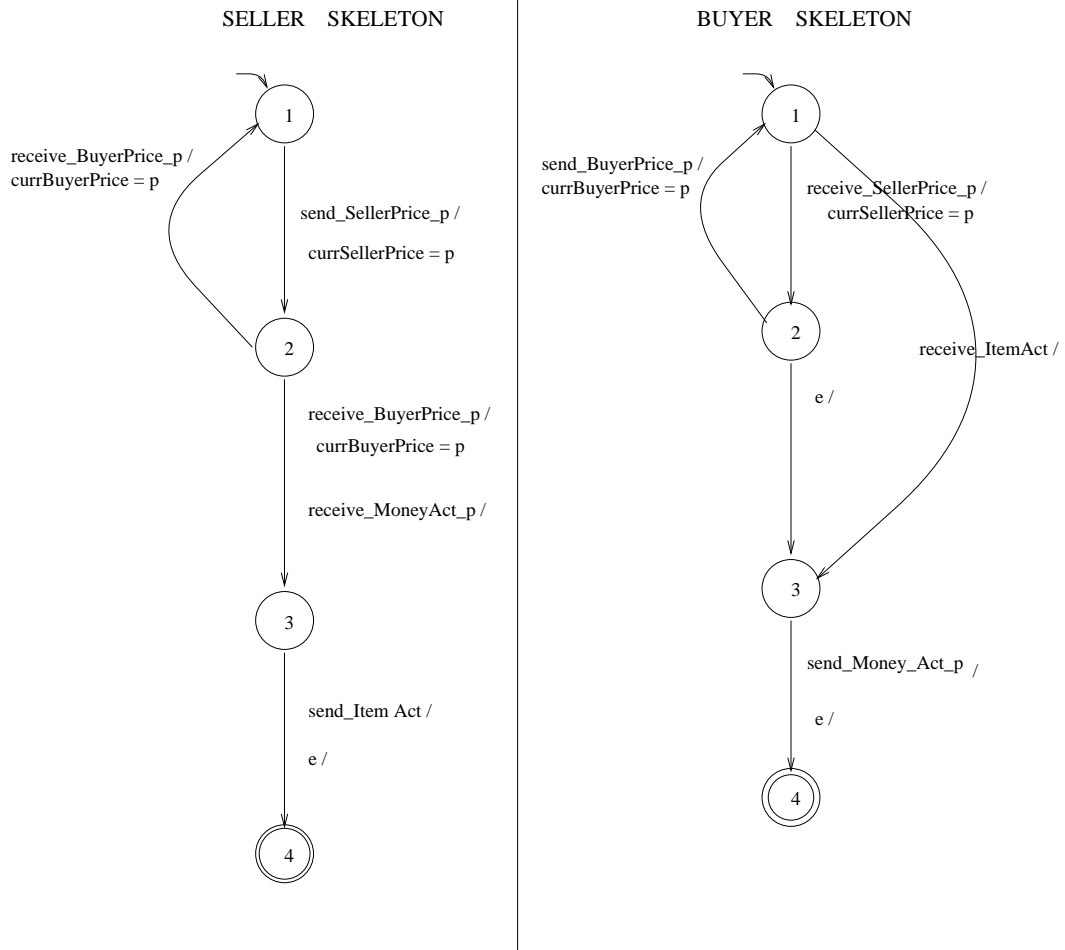


Figure 3.2: Skeletons of the haggling protocol

Example 5 The messages in Figure 2.1 can be given a content based on the following definitions. Here FM is the fish market context.

Roles and Context:

- Auctioneer, henceforth called A
- Bidders, henceforth called B_j
- Context group for the fish market protocol, henceforth called FM .

Domain specific propositions and actions:

- *fish*: a domain proposition meaning the fish is delivered
- *money_i*: a domain proposition meaning that the appropriate money is paid
(subscripted to allow different prices)
- *bad*: a domain proposition meaning the fish is spoiled

Metacommitments:

The protocol includes metacommitments that are not associated with any single message. In the present protocol, there are two metacommitments. The *high demand* metacommitment says that the auctioneer can cancel his commitment to supply fish at a given price, provided there is high demand at that price. The *bad fish* metacommitment is of the context itself to release a committing party under certain circumstances. For practical purposes, we could treat these as metacommitments of the creditor. Alternatively, we could have modeled this commitment in the same way as we modeled the *high demand commitment*, by allowing the bidder to cancel his commitment if the fish were bad.

- High demand: $C(A, B_j, FM, AG[cancel_i(B_j) \Rightarrow demand_i])$
- Bad fish: $C(FM, B_j, FM, AG[bad \Rightarrow AFrelease(FM, C(B_j, A, FM, AFsend_money_i))])$

Protocol Tokens and Skeletons:

First, we define some abbreviations for simplicity.

- $Bid_i(B_j)$: $C(B_j, A, FM, AG[recv_fish \Rightarrow AFsend_money_i])$ —meaning the bidder promises to pay *money_i* if given the fish
- $Ad_i(B_j)$: $C(A, B_j, FM, AG[recv_Bid_i(B_j) \Rightarrow AFsend_fish])$ —meaning the auctioneer offers to deliver the fish if he gets a bid for *money_i*
- $demand_i$: $(\exists j, k : j \neq k \wedge Bid_i(B_j) \wedge Bid_i(B_k))$ —meaning that at least two bidders have bid for the fish at price *i*
- $cancel_i(B_j)$: $cancel(A, Ad_i(B_j))$

Armed with the above, we can now state the commitments associated with the different messages (tokens) in the fish market protocol.

- *money_i*
Payment of *i* from B_j : $discharge(B_j, C(B_j, A, FM, AFsend_money_i))$
- *fish*
Delivering fish to B_j : $discharge(A, C(A, B_j, FM, AFfish))$
- *Yes*
Yes from B_j (for price *currPrice*): $create(B_j, Bid_{currPrice}(B_j))$
- *No*
No from B_j (for price *currPrice*): true
- *price = i*
Advertise to B_j (for price *i*): $create(A, Ad_i(B_j))$
- *cancel*
Cancel the ad to B_j (for price *currPrice*): $cancel_{currPrice}(B_j)$

In addition, in a monotonic framework, we would also need to state the completion requirements to ensure that only the above actions are performed.

The auctioneer is not committed to a price if no bid is received. If more than one bid is received, the auctioneer is allowed to cancel his commitment. Notice that the auctioneer can exit the market or adjust the price in any direction if a unique *Yes* is not received for the current price *money_i*. It would neither be rational for the auctioneer to raise the price if there are no takers at the present price, nor to lower the price if takers are available. However, the protocol *per se* does not legislate against either behavior. ■

The *No* messages have no significance on commitments. They serve only to assist in the coordination so the context can determine if enough bids are received. Fig 3.3 shows the skeletons of the auctioneer and the bidders in the fish market protocol. Now we can see

how the reasoning takes place in a successful run of the protocol.

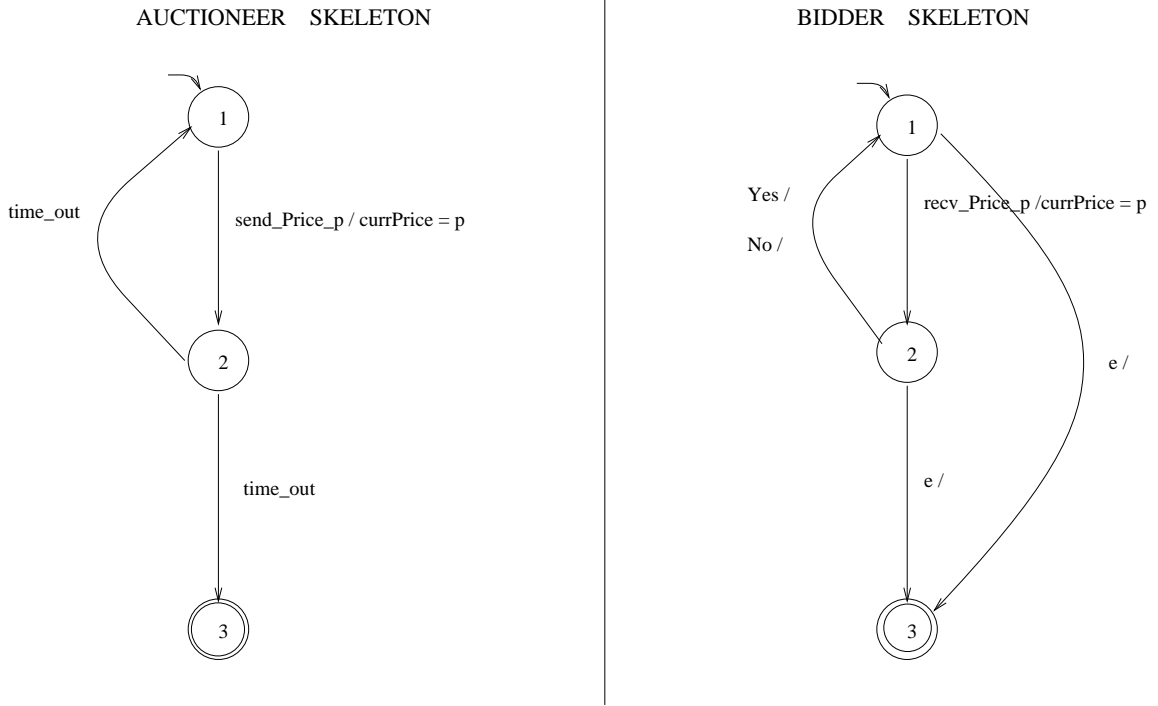


Figure 3.3: Skeletons of the fish market protocol

Example 6 The auctioneer sends out an advertisement, which commits him to supplying the fish if he receives a suitable bid. This commitment will be discharged if $AG[Bid_i(B_j) \Rightarrow AF_{send_fish}]$ holds. When $Bid_i(B_j)$ is sent by B_j , the bidder is committed to the bid, which is discharged if $AG[fish \Rightarrow AF_{money_i}]$ holds. To discharge the advertisement, the auctioneer must eventually supply the fish. If he does not supply the fish, he is in violation. If he supplies the fish, the bidder is then committed to eventually supply the money. If the bidder does so, the protocol is executed successfully, i.e., without any violation. ■

It could be gauged from the above examples that it is not trivial to write a protocol description formally from the English description. It is a difficult problem to map the

different tokens of the protocol to commitments. Moreover, there could be different ways to describe the same protocol. For this purpose, we assume that formal specifications of protocols are written by experts. It is also possible to come up with design methodologies to support specifications of commitment protocols, but we defer this to future work.

Chapter 4

Verification

In their generic forms, both causality and temporal logic are well-known. However, applying them in combination and in the particular manner suggested here is novel to this study. Temporal logic model checking is usually applied for design-time reasoning [Emerson, 1990, pp. 1042–1046]. We are given a specification and an implementation, i.e., program, that is supposed to meet it. A model is generated from the program. A model checking algorithm determines whether the specification is true in the generated model. However, in an open, heterogeneous environment, an agent’s design may not be available at all. For example, the vendors who supply the agents may consider their designs to be trade secrets.

By contrast, ours is a run-time approach, and can meaningfully apply model checking even in open settings. This is because it uses a model generated from the joint executions of the agents involved. Model checking in this setting simply determines whether the present execution satisfies the specification. We are only concerned with resolution of commitments in the system and not about other aspects such as fairness etc. If all the agents resolve all their commitments in a given execution, then we have a successful run. Verification deals with checking if commitments are resolved in a particular run of the protocol. Even an execution respects the given protocol, some other execution may be inappropriate

in other circumstances. However, if an execution is inappropriate, that does entail that the system does not satisfy the protocol. Consequently, although we are verifying specific executions of the multiagent system, we can only falsify (but not verify) the correctness of the construction of the agents in the system.

Model checking of the form introduced above may be applied by any observer in the multiagent system. A useful case is when the observer is one of the participating agents. Another useful case is when the observer is some agent dedicated to the task of managing or auditing the interactions of some of the agents in the multiagent system.

Potential causality is most often applied in distributed systems to ensure that the messages being sent in a system satisfy causal ordering [Birman, 1993]. Causality motivates vector clocks and vector timestamps on messages, which help ensure correct ordering by having the messaging subsystem reorder and retransmit messages as needed. This application of causality can be important, but is controversial [Birman, 1994, Cheriton and Skeen, 1993], because its overhead may not always be justifiable.

In our approach, the delivery of messages may be non-causal. However, causality serves the important purpose of yielding accurate models of the observations of each agent. These are needed, because in a distributed system, the global model is not appropriate. Creating a monolithic model of the execution of the entire system requires imposing a central authority through which all messages are routed. Adding such an authority would take away many of the advantages that make distributed systems attractive in the first place. Consequently, our method of constructing and reasoning with models should

- not require a centralized message router
- work from a single vantage of observation, but be able to handle situations where some agents pool their evidence.

Such a method turns out to naturally employ the notion of potential causality.

The observations made by each agent are essentially a record of the messages it has sent or received. Since each message is given a vector timestamp, the observations can be partially ordered. In general, this order is not total, because messages received from different agents may be mutually unordered.

4.1 The Quasimodel and the General Model

For the purposes of the semantics, we must define a global model with respect to which commitment protocols may be specified. The formal semantics of our language are then defined over this global model. Our specific concrete model identifies states with messages. Recall that the timestamp of a message is the clock vector attached to it. The states are ordered according to the timestamps of the messages. The proposition true in a state is the one corresponding to the operation that is performed by the message.

The quasimodel refers to the model generated based on the causal ordering between messages. As a result, it has two states corresponding to each message, one for the *send* event and one for the *receive* event. There are two additional states, the *start* and the *end* states. We assume that a *start* message and an *end* message is broadcast to every participant of the protocol. This is done by the protocol server. We do not distinguish between the *send* and *receive* events of *start* and *end* messages. Thus, we arrive at the following definition for the set of states Q .

Definition 8 $Q = \{send(m) : m \text{ is a message}\} \cup \{recv(m) : m \text{ is a message}\} \cup \{start, end\}$ ■

We now define a partial order between states in Q based on the causality relation \prec defined in section 2.2.

Definition 9 Let $<$ denote the minimum relation on Q such that $<^+ = \prec$. Here, *start* corresponds to $\vec{0}$. Note that $(\forall q \in Q : start < q)$ and $(\forall q \in Q : q < end)$. ■

Definition 10 For $s \in \mathbf{Q}$, $\mathbf{I}(s) = \{\text{the operations or actions mapped to or executed by message } s\}$ ■

The structure $M_Q = \langle \mathbf{Q}, <, \mathbf{I} \rangle$ is a *quasimodel* (Here and below, we assume that $<$ and \mathbf{I} are appropriately projected to the available states). M_Q is structurally a model, because it matches the requirements of Definition 4. However, M_Q is not a model of the computations that may take place, because the branches in M_Q are concurrent events and do not individually correspond to a single path. A quasimodel can be mapped to a model, $M_S = \langle \mathbf{S}, <, \mathbf{I} \rangle$ with an initial state $\vec{0}$, by including all possible interleavings of the causally unordered states. That is, \mathbf{S} would include a distinct state for every message in each possible ordering of the states in \mathbf{Q} that is consistent with the temporal order $<$ of M_Q . The relation $<$ can be suitably defined for M_S . M_S can be interpreted as a finitely branching, infinitely tall rooted tree. The *end* state is repeated infinitely many times below each distinct ordering, so as to be consistent with the CTL model. In figure 4.1, the state e represents the *end* state. The root corresponds to the *start* state. Figure 4.1 shows an example quasimodel and its corresponding general model. However, there is a potentially exponential blowup in the size of \mathbf{S} as a state in Q could possibly map to many states in S .

Definition 11 Let state $u \in S$ map to state $v \in Q$, i.e., to say both u and v correspond to either the send or the receive of the same message. Then, we say $v = quasi(u)$. ■

Here, $quasi : S \mapsto Q$ is a many-to-one function. In fact, the number of elements in S could be an exponential function of the number of elements in Q . This is in fact the motivation for working with the quasimodel rather than explicitly constructing the model itself.

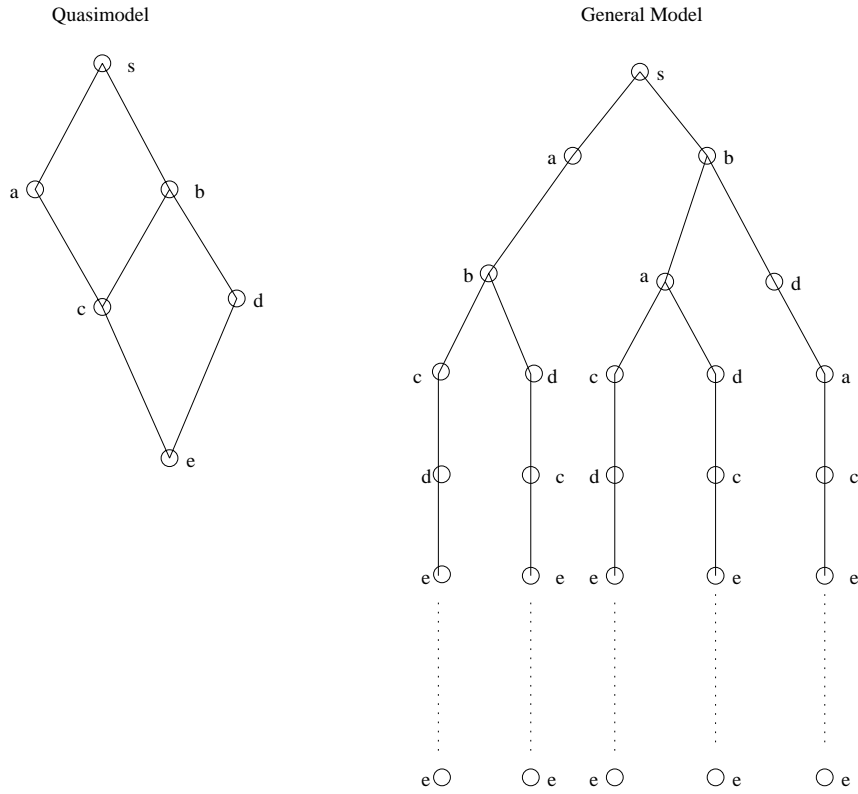


Figure 4.1: An example quasimodel and its corresponding general model

Example 7 Figure 4.2 shows the models constructed locally from the observations of the auctioneer and a bidder in the run of Example 3. ■

Although a straightforward application of causality, the above example shows how local models may be constructed. The quasimodel is in some sense the union of all the local models. Given local models, $M_i = \langle Q_i, <_i, I_i \rangle$, the corresponding global quasimodel is $M = \langle Q, <, I \rangle$ where $Q = \bigcup_i Q_i$, $I(q) = I_j(q)$ where $q \in Q_j$, and $<$ is the minimum relation such that $<^+ = \prec$.

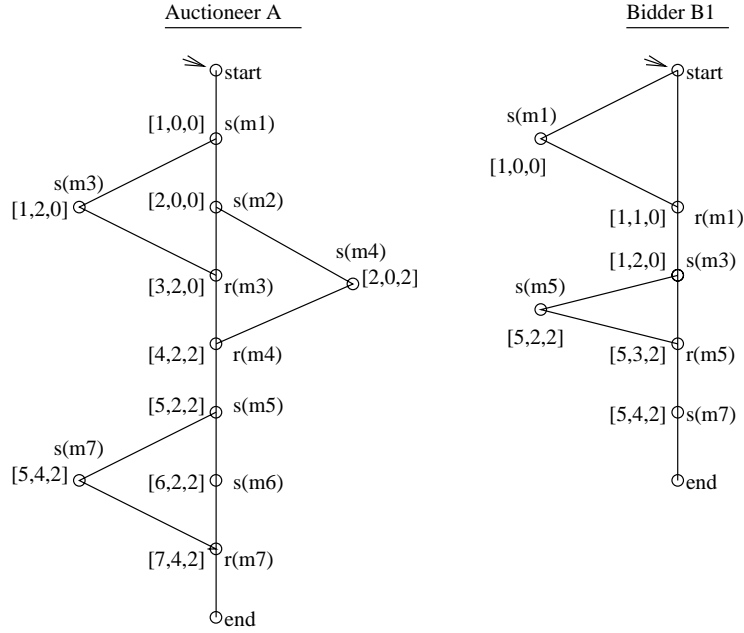


Figure 4.2: Observations for auctioneer and a bidder in the fish market protocol

4.2 Implications of the Specification Language

First, we define some important concepts that apply to commitments.

Definition 12 A commitment c is *resolved* through a *release*, *discharge*, *cancel*, *delegate*, or *assign* performed on c . The commitment c ceases to exist when resolved. However, a new commitment is created for *delegate* or *assign*. ■

New commitments created because of some existing metacommitment are not included in the definition of resolution. Theorem 1 states that the creditor knows the disposition of any commitment due to it, if all the actions in the definition of the commitment contain it as either the sender or the receiver. This result helps establish that the creditor can determine compliance of others relative to what was committed to it, for certain types of commitments.

Theorem 1 If event e_i creates commitment c and event e_j resolves c , then the creditor of c sees both e_i and e_j , provided all actions in the definition of c contain it as either the sender or the receiver.

Proof. Since all the actions in the definition of c contain the creditor of c , it can always verify $discharge(c)$. For other operations, an inspection of the corresponding message patterns defined in Section 2.3 helps establish this fact. This is because all operations require a message to be sent to the creditor. ■

Definition 13 A commitment c is *ultimately resolved* through a *release*, *discharge* or *cancel* performed on c where it is resolved, or where any commitments created by the *delegate* or *assign* of c are ultimately resolved. ■

Theorem 2 essentially states that the creation and ultimate resolution of a commitment occur along the same causal path. This is important, because it legitimizes a significant optimization below. Indeed, we defined the above message patterns so we would obtain Theorem 2.

Theorem 2 If message m_i creates commitment c and message m_j ultimately resolves c , then $m_i \prec m_j$.

Proof. As stated in section 3.1.2, only commitments in which the debtor is able to verify the discharge of the commitments are allowed in our framework. This is the reason that all the actions leading to the *discharge* of c are seen by the debtor and hence occur on the same causal path. For other operations, an inspection of their message patterns helps establish this fact. This is because all the other operations require a message to be sent or received by the debtor, and thus are causally related to the initial commitment. ■

In the following definitions let $M = \langle \mathbf{S}, \prec, \mathbf{I} \rangle$ be a model that matches the requirements of Definition 4. In particular, M could be either M_S or M_Q .

Definition 14 Let c be any commitment, meta or ordinary. We say that $M \triangleright c$ iff c is resolved in M . ■

Definition 15 Let c be a commitment created in state $q \in S$. Then, $M \models_p c$ iff $q < p$ and the commitment is not resolved on any path from q to p . In other words, the commitment exists in the system in state p and has not yet been resolved. ■

It is our goal to apply the model-checking procedure on the quasimodel itself, so that we can eliminate the potential blowup in the number of states. For this purpose, we need to transform the temporal formulae encapsulated inside the commitments, so that they can be applied to the quasimodel. Accordingly, we define the following transformation.

Definition 16 Define $M_Q^T = \langle Q, <, \mathbf{I}^T \rangle$ where each commitment c in I is transformed to c^T by replacing every eventuality formula AF in c by EF. ■

The basic structure of all commitments remains the same. All that changes is the temporal formula inside a commitment.

4.3 Reasoning with the Quasimodel

Now we explain the main reasoning steps in our approach and show that they are sound. A protocol execution is considered compliant with the specifications if all commitments created in it are resolved. This means that inappropriate actions such as *cancel* must be prevented or constrained, unless they are allowed by the metacommitments.

Theorem 3 shows that for the resolution of commitments, it is correct to naively treat a quasimodel as if it were a model. It is not necessary to consider all possible interleavings in the general model as our specification language only includes commitments of the form,

where the debtor is committed to perform an action eventually. In such a case, it does not matter how the different events are ordered among themselves. All that matters is that the discharge action is causally after the creation of the commitment. Also, since all the events relevant to a commitment occur on the same causal path, it is sufficient to look at paths in the quasimodel rather than look at interleavings in the general model.

Our construction ensures that all the events relevant to another event are totally ordered with respect to each other. Notice that, as shown in Figure 2.2, the construction may appear to require one more message than necessary for the *assign* and *delegate* operations. As we show below, however, this linear amount of extra work (for the entire set of messages) pays off in reducing the complexity of our reasoning algorithm.

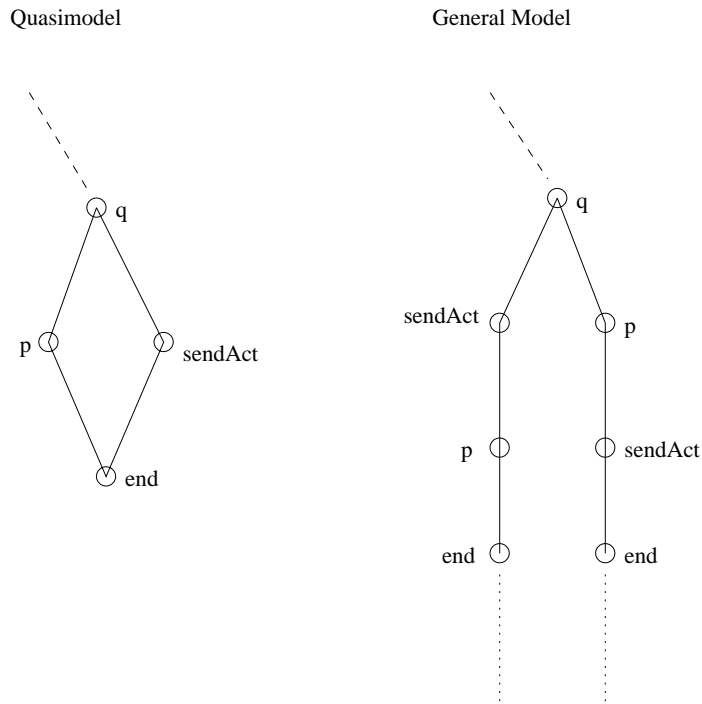


Figure 4.3: Illustration of Theorem 3

Theorem 3 $M_Q^T \triangleright c^T$ iff $M_S \triangleright c$.

Proof. Figure 4.3 shows an example quasimodel where $M_S \models_q \text{bool}$. Let the proposition contained in c be $\text{AG}[\text{bool} \Rightarrow \text{AFsendAct}]$, which means $c^T = \text{AG}[\text{bool} \Rightarrow \text{EFsendAct}]$. First we prove $M_S \triangleright c \Rightarrow M_Q^T \triangleright c^T$.

We assume that $M_S \models_q \text{bool}$, which means $M_S \models_q \text{AFsendAct}$, which means all paths rooted at q in M_S contain a state corresponding to sendAct below. If we assume that sendAct and q were causally unrelated, then there must exist a path along which sendAct happens before q in M_S . But this is contradictory to the fact that $M_S \models_q \text{AFsendAct}$. Hence our assumption is false and sendAct is causally after state q . This means there exists a causal path from q to sendAct , which means $M_Q^T \models_q \text{EFsendAct}$, and this shows $M_Q^T \triangleright c^T$.

Now, we prove the converse, i.e., $M_Q^T \triangleright c^T \Rightarrow M_S \triangleright c$. If $M_Q^T \models_q \text{EFsendAct}$, then there is a causal path from state q to the state corresponding to sendAct , and hence this state will be below q along all interleavings, which means $M_S \models_q \text{AFsendAct}$. ■

The above results show that compliance can be verified and without blowing up the model unnecessarily. However, we would like to test for compliance based on local information—so that any agent can decide for itself whether it has been wronged by another. For this reason, we would like to be able to project the global model onto local models for each agent, while ensuring that the local models carry enough information that they are indeed usable in isolation from other local models. Accordingly, we can define the construction of local models corresponding to an agent’s observations. This is simply by defining a subset of \mathbf{S} for a given agent a .

Definition 17 $\mathbf{S}_a = \{\text{send}(m) : m \text{ is a message from or to } a\} \cup \{\text{recv}(m) : m \text{ is a message from or to } a\}$
 $M_a^T = \langle \mathbf{S}_a, <, \mathbf{I}^T \rangle$. ■

Theorem 4 shows that if we restrict our attention to commitments whose resolution the given agent can verify, then the projected quasimodel yields all and only the correct conclusions relative to the global quasimodel. Thus, if the interested party is vigilant, it can check if anyone else violated the protocol.

Theorem 4 $M_a^T \triangleright c^T$ if and only if $M_Q^T \triangleright c^T$, provided that a sees all the commitments mentioned in c .

Proof. Since a sees all the commitments mentioned in c , this means M_a^T has all the information required to verify compliance. Furthermore, the CTL state formula $EF\textit{sendAct}$ requires the existence of one causal path along which $\textit{sendAct}$ occurs, which is the same for both M_a^T and M_Q^T , except possibly that there might be some intermediate, though irrelevant, states missing in M_a^T . Hence, the result follows. ■

Example 8 If one of the bidders backs down from a successful bid, the auctioneer immediately can establish that he is cheating, because the auctioneer is the creditor for the bidder's commitment. However, a bidder cannot ordinarily decide whether the auctioneer is noncompliant, because the bidder does not see all relevant commitments based on which the auctioneer may be released from a commitment to the bidder. ■

Theorem 5 lifts the above results to sets of agents. Thus, a set of agents may pool their evidence in order to establish whether a third party is noncompliant. Thus, in a setting with two bidders, a model that includes all their evidence can be used to determine whether the auctioneer is noncompliant. Ordinarily, the bidders would have to explicitly pool their information to do so. However, in a broadcast-based or outcry protocol (like a traditional fish market in which everyone is screaming), the larger model can be built by anyone who hears all the messages. Let A be a set of agents.

Definition 18 $S_A = \bigcup_{a \in A} S_a$. $M_A^T = \langle S_A, <, \mathbf{I}^T \rangle$. ■

Theorem 5 Let the commitments observed by agents in A include all the commitments in c . Then $M_A^T \triangleright c^T$ iff $M_Q^T \triangleright c^T$.

Proof. From the construction of M_A and the proof of Theorem 4. The set of agents could be thought of as a single agent who observes all the events for any agent, and the proof for Theorem 4 can be applied here as well. ■

4.4 Model Checking

Given the theorems above, we need to apply model checking to the quasimodel. The states in the model correspond to operations on commitments, and domain actions. Operations like *delegate* and *assign* trivially resolve a commitment but create new commitments. The *release* and *cancel* operations serve in deleting commitments from the system, but we need to check if they are allowed by the metacommitments.

In this section, we present pseudocode for performing various types of commitment analyses on the quasimodel both for online and retrospective verification. The *create*, *delegate* and *assign* operations create new commitments. Any commitment is *trivially resolved* if one of *cancel*, *release*, *delegate* or *assign* is performed on it. We assume that these operations are performed only if allowed by the metacommitments. This assumption is suitable because the agents interact through predefined tokens, and the coordination layer would not allow sending arbitrary messages.

There are three types of commitments allowed by our specification language namely:

- INEVITABLE(I): This refers to commitments of the form
 $C(x, y, G, \text{AFdischarge_action})$
- STIMULUS-RESPONSE(S-R): This refers to commitments of the form
 $C(x, y, G, \text{AG}[\text{antecedent} \Rightarrow \text{AFdischarge_action}])$

- **CONDITIONAL(C)**: This refers to commitments of the form $C(x, y, G, AG[\text{antecedent} \Rightarrow \text{consequent}])$

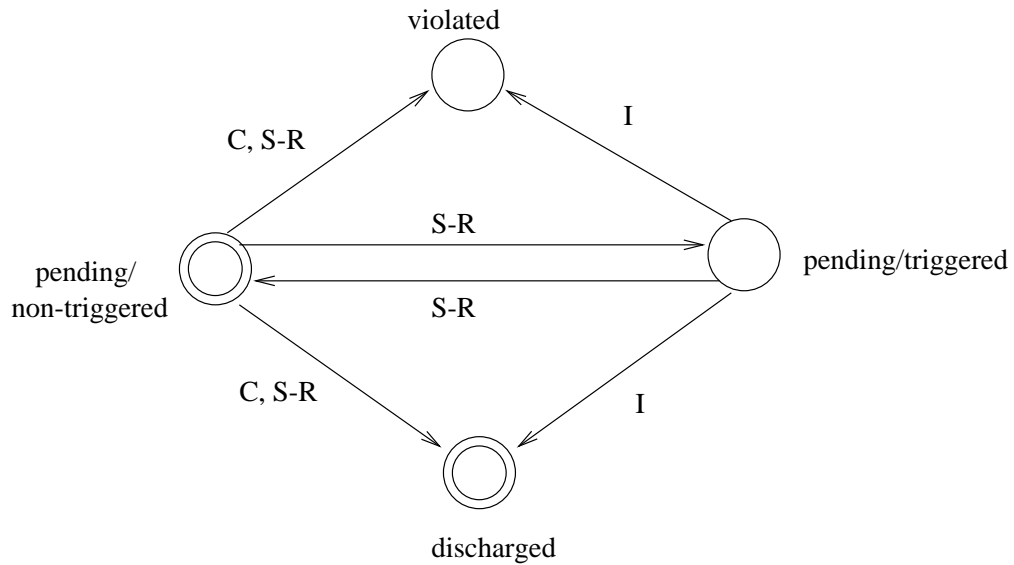


Figure 4.4: State transition diagram of a commitment

Furthermore, during the protocol execution each commitment is associated with one or more of the following states:

- *pending*: this means the commitment has not yet been resolved
- *violated*: this means the commitment has already been violated
- *discharged*: this means the commitment has already been discharged
- *triggered*: this means an action is required on the part of the debtor to discharge the commitment. An inevitable commitment is always in the triggered state until it is discharged or violated.

At any point of time, a commitment is either pending, discharged, or violated. Figure 4.4 shows the state transition diagram of a commitment during the execution of procedure 4.5.

Procedure 4.5 keeps track of the state of the various commitments in the system. Each

```

update_pending_commitments( $M_A, A$ )
   $p = \text{prev\_maximum\_state}(\text{agent})$ 
   $m = \text{maximum\_state}(\text{agent})$ 
   $\text{bfs\_iterator} = \text{begin\_bfs}(M_A, p)$ 
   $s = \text{bfs\_iterator.next\_state}(M_A, p)$ 
  while ( $s \leq m$ )
    forall ( $C_1 \in \text{new\_commitments}(M_A, s)$ )
      pending = pending  $\cup \{C_1\}$ 
      if ( $\text{type\_of}(C_1) == \text{INEVITABLE}$ )
        triggered = triggered  $\cup \{C_1\}$ 

    forall  $C \in \text{pending}$ 
      unmark( $C$ )
    while ( $\exists C \in \text{pending} : C$  is unmarked)
      mark( $C$ )
      if ( $\text{trivial\_resolve}(M_A, s, C)$ )
        pending = pending  $- \{C\}$ 
        discharged = discharged  $\cup \{C\}$ 
      else if ( $\text{type\_of}(C) == \text{INEVITABLE}$ )  $\wedge (M_A \models_s \text{discharge\_action}(C))$ 
        pending = pending  $- \{C\}$ 
        triggered = triggered  $- \{C\}$ 
        discharged = discharged  $\cup \{C\}$ 
      else if ( $\text{type\_of}(C) == \text{STIMULUS-RESPONSE}$ )
        if ( $\text{triggered}(C) \wedge M_A \models_s \text{discharge\_action}(C)$ )
          triggered = triggered  $- \{C\}$ 
        else if ( $\neg \text{triggered}(C) \wedge M_A \models_s \text{antecedent}(C)$ )
          triggered = triggered  $\cup \{C\}$ 
        else if ( $M_A \models_s (\text{antecedent}(C) \wedge \neg \text{consequent}(C))$ )
          pending = pending  $- \{C\}$ 
          violated = violated  $\cup \{C\}$ 
       $s = \text{bfs\_iterator.next\_state}(M_A, p)$ 

    if ( $m == \text{end}$ )
      discharged = discharged  $\cup (\text{pending} - \text{triggered})$ 
      pending = triggered
      violated = violated  $\cup \text{pending}$ 

boolean exists_commitment( $M_A, C, s$ )
  return ( $C \in \text{pending}$ )

```

Figure 4.5: State maintenance of commitments

time the procedure is called it performs a breadth first search from the maximum state it previously visited. It examines each state in the quasimodel and updates the state of each pending commitment according to the state transition diagram of Figure 4.4. A commitment exists at any point of time iff it is in the set of pending commitments. The procedure **new_commitments**(M_A, s) returns the new commitments created at state s in the quasimodel M_A either due to an explicit *create* operation or due to *delegate* and *assign* operations. The procedure **trivial_resolve**(M_A, s, C) returns true if C is *trivially resolved* at state s .

Procedure 4.5 visits each state in the quasimodel exactly once in a breadth first manner. At each state, every pending commitment is checked for resolution. A naive analysis would assume that the number of pending commitments is $O(n)$ at any time, and thus the complexity of the procedure is $O(n^2)$, where n is the number of messages sent. But, in reality, the number of commitments created is much less than the number of messages. In fact, we could improve the above procedure by maintaining a hashtable which associates actions and commitments. In this way, we need not check for all pending commitments at a state, but only check for those commitments that are associated with the actions at that state. Also, since we examine each state in the quasimodel exactly once, we may discard all the states from the quasimodel after each call to the above procedure. This optimization improves the space complexity of our algorithm significantly.

4.5 Handling Message Losses and Delays

Till now, we have only demonstrated situations where the messages are reliably sent and received. However, in the Internet such an assumption is not entirely justifiable. It is upto our system to handle delayed and lost messages. In this section, we show how this can be incorporated in our framework.

4.5.1 Message Losses

Our specification language distinguishes between the *send* and *receive* of messages. This gives us considerable flexibility in specifying commitments. Some kinds of messages do not require an acknowledgment from the receiving party. Such messages, though not very common, can be captured by our specification language as is. Example 9 illustrates this fact.

Example 9 Consider an example of a bank agent interacting with a customer agent. Suppose the bank is required to send a receipt as proof of deposit of a certain amount of money in the customer's account. Further assume that the bank is only committed to send the receipt. It does not care if the receipt is reliably received by the buyer. We could represent this using a simple commitment as follows.

$$C(B, C, G, AG[receiveMoney(C, B) \Rightarrow AFsendReceipt])$$

Now even if the receipt is lost, then the bank can still prove that it satisfied its commitment in sending the receipt, and hence is not in violation. ■

However, there are many situations where a message needs to be resent until it is acknowledged by the other party. Example 10 illustrates this situation.

Example 10 Consider an example of an electronic marketplace where a buyer is committed to send his credit card number to a vendor upon receipt of a certain product. The protocol stipulates that the buyer has to send his credit card number to the vendor and must establish that the credit card number was received successfully. We could represent this as follows :

$$C(B, V, G, AG[receiveProduct \Rightarrow AF[sendCardNum]])$$

$$C(B, V, G, AG[sendCardNum \Rightarrow AF[recvCardNumAck]])$$

The above two commitments capture the fact that the buyer has to send his credit card number and receive an *ack* for the same. The task of determining to what message the *ack* belongs is the job of the coordination layer. In the above example, the vendor is also committed to send an *ack* to the buyer.

$$C(V, B, AG[recvCardNum \Rightarrow AFsendCardNumAck])$$

Note that if the *ack* gets lost then the buyer has to resend his credit card number to satisfy his commitment, and the vendor has to resend the *ack*. This continues until the buyer receives an *ack* for his message. ■

4.5.2 Message Delays

A message could either be delayed that it causes a timeout for the receiving party, or it may arrive out of order. In the former case, it is up to the coordination layer of the receiving party to just ignore the message. Basically, the coordination layer generates a timeout event, and all messages that are received after the timeout but sent causally prior to it are ignored. Example 11 illustrates this scenario.

Example 11 The auctioneer sends out an advertisement, and waits for a predetermined amount of time to receive the bids from the bidders. Then, it generates a timeout event, and all bids that were sent prior to the timeout event but received after it are ignored. Thus, in the next round, bids from the previous round would not cause confusion. ■

In the latter case, if the receiving party receives a message in an order not consistent with the causal order, then the timestamp of the message determines the actual position of the message, and the commitment layer itself deals with it. There are other possibilities in this case. The coordination layer could always present the commitment layer with messages

in a causal order, although this approach is potentially inefficient in an environment where there are unexpected delays.

The underlying coordination layer ensures that the commitment layer does not have to deal with issues regarding timeout and the details of message losses and message delays.

Chapter 5

Conclusion

Given the autonomy and heterogeneity of agents, the most natural way to treat interactions is as communications. A communication protocol involves the exchange of messages with a streamlined set of tokens. Traditionally, these tokens are not given any meaning except through reference to the beliefs or intentions of the communicating agents. By contrast, our approach assigns *public*, i.e., observable, meanings in terms of social commitments. Viewed in this light, *every communication protocol is a commitment protocol*.

Formulating and verifying compliance of autonomous and heterogeneous agents is a key prerequisite for the effective application of multiagent systems in open environments. As asserted by Chiariglione, minimal specifications based on external behavior will maximize interoperability [1998]. The research community has not paid sufficient attention to this important requirement. A glaring shortcoming of most existing semantics for agent communication languages is their fundamental inability to allow verification for the compliance of an agent [Singh, 1998a, Wooldridge, 1998]. The present approach shows how that might be carried out. Moreover, most of the agent communication languages deal with coordination details contrary to our specification language which attempts to capture the commitment aspects.

The specification language we develop is designed to represent common protocols found in electronic commerce applications like auction protocols, payment protocols, or other contract based protocols, where commitments are the most natural form of specifying contracts. Though, a nice specification language for protocols is necessary, it is practically useless without a verification framework that checks compliance of different agents following the protocol.

While the purpose of the protocols is to specify legal behavior, they should not specify rational behavior. Rational behavior may result as an indirect consequence of obeying the protocols. However, not adding rationality requirements leads to more succinct specifications and also allows agents to participate even if their rationality cannot be established by their designers.

The compliance checking procedure can be used by any agent who participates in, or observes, a commitment protocol. There are two obvious uses. One, the agent can track which of the commitments made by others are still pending or have been violated. Two, it might track which of its own commitments are pending or whose satisfaction has not been acknowledged by others. The agent can thus use the compliance checking procedure as an input to its normal processes of deliberation to guide its interactions with other agents.

We have so far discussed how to detect violations. Once an agent detects a violation, as far as the above method is concerned, it may proceed in any way. However, some likely candidates are the following. The wronged agent may

- inform the agents who appeared to have violated their commitments and ask them to respect the applicable metacommitments
- inform the context, who might penalize the guilty parties, if any; the context may require additional information, e.g., certified logs of the messages sent by the different agents, to establish that some agents are in violation.

- inform other agents in an attempt to spoil the reputation of the guilty parties. Reputation is often the only hold an agent has on another agent in an open environment.

5.1 Literature

From the multiagent systems standpoint, some of the important strands of research of relevance to commitment protocols have been carried out before. However, the synthesis, enhancement, and application of these techniques on multiagent commitment protocols is a novel contribution of this study. Interaction (rightly) continues to draw much attention from researchers. Still, most current approaches do not consider an explicit execution architecture (however, there are some notable exceptions, e.g., [Ciancarini et al., 1996, 1998, Singh, 1998c]). Other approaches lack a formal underpinning; still others focus primarily on monolithic finite-state machine representations for protocols. Such representations can capture only the lowest levels of a multiagent interaction, and their monolithicity does not accord well with distributed execution and compliance verification. Model checking has recently drawn much attention in the multiagent community, e.g., [Benerecetti et al., 1998, Singh, 1998b]. However, the above approaches consider knowledge and related concepts and are thus not directly applicable for behavior-based compliance.

From a distributed computing standpoint, there has been a lot of work on testing and debugging distributed programs. As discussed in Chapter 1, both testing and especially debugging have considerable overlap with our work, but these approaches are generally tailored to meet the rigid debugging requirements of distributed programs in contrast to our approach which relies on more flexible definitions of protocols. However, it is notable to mention [Cheng and Wallentine, 1989], which defines a language DEBL based on inter

process semantics and hence is similar to our approach. DEBL provides both synchronous and asynchronous communication mechanisms. In contrast, we only allow asynchronous events, although synchronous events could be modeled through an explicit acknowledgment protocol. In DEBL, each process is specified in terms of its possible interactions with other processes. These possibilities are represented by a tree. DEBL allows for aggregate events and is useful when users elect to examine aspects of the program such as the detection of a unique communication pattern. In our approach, aggregate events include the *delegate* and *assign* operations and the $C(x, y, G, p)$ construct that corresponds to the existence of a commitment at any point of time. Apart from these, DEBL uses Temporal Event Logic System (TEL), an interval based temporal logic to model time. This is because nested time intervals are a natural implication of DEBL's aggregate event structure.

In the computer security community, most of the research work has been on low-level authentication and accountability protocols [Kailar, 1995, Kessler and Neuman, 1998]. Although these topics are important and provide the first line of compliance testing, our concerns are more high-level in nature, as we try to specify and verify properties that are behavior oriented, in other words, focus on the behavioral properties of agents.

5.2 Future Directions

The present approach highlights the synergies between distributed computing and multi-agent systems. Since both fields have advanced in different directions, a number of important technical problems can be addressed by their proper synthesis.

One aspect relates to situations where the agents may suffer a Byzantine failure or act maliciously. Such agents may fake messages or deny receiving them. How can such situations be detected by the other agents? In the current study, we resolve this problem

by assuming a trusted coordination stub. However, more advanced solutions that rely on distributed computing ideas to determine inconsistencies in the values of logical clocks or some such methodology may be essential.

Another aspect is to capture additional structural properties of the interactions so that noncompliant agents can be more readily detected. Alternatively, we might offer an assistance to designers by synthesizing skeletons of agents who participate properly in commitment protocols. Another future direction is to develop a formal commitment calculus with axiomatic rules so that reasoning may be more readily performed. Lastly, it is well-known that there can be far more potential causes than real causes [Schwarz and Mattern, 1994]. Can we analyze conversations or place additional, but reasonable, restrictions on the agents that would help focus their interactions on the true relationships between their respective computations? We defer these topics to future research.

Bibliography

- Gul A. Agha and Nadeem Jamali. Concurrent programming for distributed artificial intelligence. In *Weiß [1999]*, chapter 12, pages 505–534. 1999.
- Massimo Benerecetti, Fausto Giunchiglia, and Luciano Serafini. Model checking multi-agent systems. *Journal of Logic and Computation*, 8(3):401–423, 1998.
- Kenneth P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, 1993.
- Kenneth P. Birman. A response to Cheriton and Skeen’s criticism of causal and totally ordered communication. *Operating Systems Review*, 28(1):11–21, 1994.
- Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.
- Nicholas Carriero and David Gelernter. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, 1992.
- Wan-Hong S. Cheng and Virgil E. Wallentine. DEBL: A knowledge based language for specifying and debugging distributed programs. In *Communications of the ACM*, volume 32, pages 1079–1084. 1989.

- David R. Cheriton and Dale Skeen. Understanding the limitations of causally and totally ordered communication. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP)*, pages 44–57. ACM Press, 1993.
- Leonardo Chiariglione. Foundation for intelligent physical agents (FIPA) scope, 1998. www.fipa.org/library/scope.html.
- Paolo Ciancarini, Andreas Knoche, Robert Tolksdorf, and Fabio Vitali. PageSpace: An architecture to coordinate distributed applications on the web. *Computer Networks and ISDN System*, 28(7–11):941–952, 1996. Proceedings of the 5th International World Wide Web Conference.
- Paolo Ciancarini, Robert Tolksdorf, Fabio Vitali, Davide Rossi, and Andreas Knoche. Coordinating multiagent applications on the WWW: A reference architecture. *IEEE Transactions on Software Engineering*, 24(5):362–375, 1998.
- E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. North-Holland, Amsterdam, 1990.
- Colin Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, 1991.
- Nissim Francez and Ira R. Forman. *Interacting Processes: A Multiparty Approach to Coordinated Distributed Programming*. ACM Press and Addison-Wesley, New York, 1996.
- Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, 1993.

- Rajashekar Kailar. Reasoning about accountability in protocols for electronic commerce. In *IEEE Symposium on Security and Privacy*, pages 236–250, California, 1995. IEEE Computer Society Press.
- Volker Kessler and Heike Neuman. A sound logic for analysing electronic commerce protocols. In *Computer Security - ESORICS*, pages 345–360. Springer, Belgium, 1998.
- Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
- Michael Reiter and Li Gong. Preventing denial and forgery of causal relationships in distributed systems. In *IEEE Symposium on Research in Security and Privacy*, pages 30–40, California, 1993.
- Juan A. Rodríguez-Aguilar, Francisco J. Martín, Pablo Noriega, Pere Garcia, and Carles Sierra. Towards a test-bed for trading agents in electronic auction markets. *AI Communications*, 11(1):5–19, 1998.
- Tuomas Sandholm and Victor Lesser. Issues in automated negotiation and electronic commerce: Extending the contract net framework. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 328–335, California, 1995. AAAI Press/The MIT Press.
- Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.
- Munindar P. Singh. Agent communication languages: Rethinking the principles. *IEEE Computer*, 31(12):40–47, 1998a.

- Munindar P. Singh. Applying the mu-calculus in planning and reasoning about action. *Journal of Logic and Computation*, 8(3):425–445, 1998b.
- Munindar P. Singh. A customizable coordination service for autonomous agents. In *Intelligent Agents IV: Proceedings of the 4th International Workshop on Agent Theories, Architectures, and Languages (ATAL-97)*, pages 93–106. Springer-Verlag, 1998c.
- Munindar P. Singh. Developing formal specifications to coordinate heterogeneous autonomous agents. In *Proceedings of the 3rd International Conference on Multiagent Systems (ICMAS)*, pages 261–268. IEEE Computer Society Press, 1998d.
- Munindar P. Singh. An ontology for commitments in multiagent systems: Toward a unification of normative concepts. *Artificial Intelligence and Law*, 7:97–113, 1999.
- Reid G. Smith. The contract-net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, C-29(12):1104–1113, 1980.
- K. C. Tai and Richard H. Carver. *Parallel and Distributed Computing Handbook*, chapter 33, pages 955–978. McGraw-Hill, 1996.
- Gerhard Weiß, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, Cambridge, MA, 1999.
- Michael J. Wooldridge. Verifiable semantics for agent communication languages. In *Proceedings of the 3rd International Conference on Multiagent Systems (ICMAS)*, pages 349–356. IEEE Computer Society Press, 1998.
- Jiaying Zhou and Dieter Gollmann. An efficient non-repudiation protocol. In *Proceedings of the 10th Computer Security Foundations Workshop*, pages 126–132, 1997.