Abstract

Mallya, Ashok Ullal. Modeling and Enacting Business Processes via Commitment Protocols Among Agents (Under the direction of Professor Munindar P. Singh).

Multiagent systems involve a rich variety of interactions among agents–situated computations that are autonomous in their behavior and heterogenous in structure. These interactions can be realized unambiguously if they are governed by published protocols, since agents diverse in their structure and behavior can interact as long as they respect the protocols. However, traditional protocol specifications are unduly rigid for application in open settings involving autonomous entities. They represent protocols simply as an ordering of steps and stifle the participants' autonomy due to a lack of flexibility during enactment.

Commitments among agents, which are akin to contractual obligations among businesses, are a powerful abstraction for modeling flexible protocols. Commitment-based design enables a more faithful model of the openness of the business world. However, modeling business interactions requires a rich variety of interaction protocols that can capture the needs of different applications. Whereas general (business) protocols might most flexibly characterize the interactions of their participants, protocols often must be refined based on the environment in which they are to be deployed, so as to yield improvements along various properties such as performance and risk outlay, when applied to real-world tasks such as in e-business.

We introduce a formal semantics and an operational characterization for commitment-based protocols wherein traditional software engineering notions such as refinement and aggregation are extended to apply to protocols. We also develop a principled approach for the design of such protocols in addition to methodologies for modeling and handling exceptions in them. We demonstrate, with appropriate examples, the benefits of this approach over traditional ones when applied to business process modeling and enactment.

Our chief contributions are

- A theoretical basis for describing protocol refinement using subsumption hierarchies and an algebra for composing protocols using existing ones.
- A methodology for modeling and handling exceptions in commitment protocols that incorporates the preferences of the protocol designer and policies of the participants and enables specification of exceptions independent of the protocol specification.

- Two methodologies for designing commitment protocols, one by enhancing an existing agent-oriented software engineering methodology, and another by deriving protocols from agent conversations.

Our work draws from and contributes to agent communication, business process modeling and enactment, service-oriented computing, and software engineering.

MODELING AND ENACTING BUSINESS PROCESSES VIA COMMITMENT PROTOCOLS AMONG AGENTS

by

ASHOK ULLAL MALLYA

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial satisfaction of the
requirements for the Degree of
Doctor of Philosophy

COMPUTER SCIENCE

Raleigh

2005

APPROVED BY:

DR. DENNIS R. BAHLER	DR. RADA Y. CHIRKOVA
DR. JAMES C. LESTER	DR. MUNINDAR P. SINGH Chair of Advisory Committee

To Mrudula, for selfless efforts through testing times.

Biography

Ashok Mallya was born in Mangalore, India on September 10, 1978. He holds a Bachelor's degree in Computer Science, awarded in June 2000, from the Karnataka Regional Engineering College, Surathkal, India, which is now known as the National Institute of Technology Karnataka. Ashok joined the North Carolina State University in August 2000, and earned a Masters degree in Computer Science in May 2003. He is currently a doctoral candidate in the Department of Computer Science, North Carolina State University.

Acknowledgment

First and foremost, I thank Dr. Singh for his help, guidance, and support—financial, intellectual, and moral. I could not possibly list here the ways in which he has inspired me. I thank my committee members, Drs. Bahler, Chirkova, and Lester, for insightful comments and Dr. Silverstein for attending my preliminary and final exams as the graduate school representative. This dissertation was funded partially by the NSF under its Digital Society and Technology Program in the Intelligent Information Systems Division (Grant DST-0139037), partially by the DARPA Information Exploitation Office (Contract F30603-00-C-0178), and partially by a North Carolina State University graduate assistantship.

I thank my current and former colleagues at the MultiAgent Systems lab, Pınar Yolum, Raghuram Sreenath, Subhayu Chatterjee, Mike Maximilien, Amit Chopra, Nirmit Desai, Yathi Udupi, Leena Wagle, Payal Chakravarthy, and Arnav Jhala for their comments. They have helped improve the quality of this dissertation and my knowledge of the field. Raghu, Pınar, Amit, Yathi, and Nirmit found time in their busy schedules to patiently proof-read entire chapters of this dissertation, for which I am grateful to them. I thank Michael Huhns from the University of South Carolina, and Rogier Van Eijk and Frank Dignum from the Utrecht University, for providing me a forum for publishing my research early in my graduate studies. I am grateful to Mike Wooldridge from the University of Liverpool, Nanjangud Narendra from IBM Global Services India, and Paolo Giorgini from the University of Trento, Italy for their constructive comments about my research.

I thank Ruben Lobo, Vishwas Puttasubbappa, Bhargava Mallapragada, Naveen Kolli, Mihir Pise, and Raoul Jetley for being great roomates and patient listeners. I thank all my friends for making me the person I am today. Life would not have been half as interesting or fun without Partha Sundaram, Shakti Shenoy, Basty Sridhar, Sandhya Jagannath, Sunil Mathew, Sachin Rasquinha, Bharath Beedu, Nithin Bhat, and Shweta Pai.

Shalini, Veda, and Dinesh Pai provided me a home away from home and valuable advice whenever I needed it. I thank them for their hospitality. I am indebted to Subhash and Rohini Neginhal for their faith in me. I thank Narahari and Arundathi Mallya for a lifetime of sacrifices and encouragement. I thank Rashmi and Vikram Mallya for their love, affection, and their superlative faith in my abilities.

I am eternally grateful to Mrudula for her love, encouragement, and patience through my

long, dull semesters and unearthly work hours, especially in the days leading up to paper submissions. Her pragmatic wisdom has proved invaluable time and again, when common sense has eluded me. I could not have done it without her succor and the intensity of her belief in a bright future for us.

There are many more who deserve to be acknowledged here but are not, due to constraints of space. They will, however, be always gratefully remembered.

Contents

List of Figures

List of Tables

Chapter 1

Introduction: Agents, Business Processes, and the Web

Agents, which are persistent, autonomous software entities situated within a software environment, have been successfully applied to solve problems in which collaboration, cooperation, or communication is required between independent partners, for example, in online auctions, distributed information processing, and human-computer interaction. In this dissertation, we study interactions between agents and develop a formal framework for modeling agent interaction protocols based on the contractual obligations that exist between the participating agents as they communicate with each other and proceed with their interactions. We show how this abstraction of obligations, called *commitments* supports judgments about protocols based on knowledge-engineering notions such as subsumption and composition. Our framework and results contribute to the field of agent communication. Using these concepts, we demonstrate how business processes can be modeled and enacted with flexibility. Our work also introduces a sophisticated exception handling scheme for business processes. Both these contributions advance the state of the art in business process automation. To emphasize the broader area of application for our results, we show how commitment-based agent interaction protocols can be applied to software engineering. We do this by using the TROPOS agent-oriented software engineering methodology as an example.

1.1 Agents and Multiagent Systems

We define an agent as *a persistent, situated computation, that is capable of autonomous behavior*. Agents are persistent because they maintain a history of the environment they perceive.

They are said to be situated because they reside in some software environment that contains other agents with whom communication is possible. An agent's environment can also be viewed from the social angle, in which the agent's (social) relationships with other agents are described in terms of permissions, obligations, and norms. An agent's actions seem autonomous to other agents. Multiagent systems are systems consisting of more than one interacting agents. Whereas developing single-agent systems presents challenges in modeling the reasoning, perception, and behavior of an agent, we concentrate on multiagent systems, in which coordination and communication among the agents is of prime importance.

1.1.1 Commitments

Commitments among agents represent the contractual obligations that exist among them. An agent can commit to bringing about a certain state in the environment in which it exists or to carry out a certain action. Commitments always exist within a well-defined social structure, which forms their *context*. Commitments provide a powerful formalism for modeling and understanding interactive behavior in multiagent systems, as we shall demonstrate in this dissertation. We use the word *commitment* to refer to what is more precisely called a *social commitment*. Social commitments are distinguished from private commitments, which are commitments an agent makes to itself to perform some task or achieve some goal. Private commitments may be used in modeling the reasoning of a single agent. For modeling the behavior of multiagent systems, however, we need publicly observable aspects of agents such as social commitments.

1.1.2 Agent Interaction

One of the key benefits of multiagent systems over single-agent systems is that agents can interact with one another to achieve results that might not have been possible for one agent alone. Interaction among agents is thus an important area of study that aims to model different interaction styles among agents, such as coordination, cooperation, and competition. Software agents can interact by sending point-to-point or broadcast messages, or by using common data stores called blackboards. Agent communication can be modeled and characterized by a variety of schemes such as protocols that specify message ordering or free-form conversations that progress based on the meanings assigned to various messages. Agent communication is an apt model for interactions among autonomous entities.

1.2 Business Processes

One of the most important application areas of intelligent agents has been in automating businesses and developing newer, more productive ways of conducting business. Following this trend, we study how agents and multiagent systems can be applied in business process modeling and enactment. A business process is the way an organization conducts it business. This encompasses the businesses' dealings with its customers, suppliers, and all other entities external to it that it interacts with. A business can have several processes defined, or no documented process at all. Whereas a precise definition and clear documentation makes a business process reproducible, what we aim to achieve is a well defined methodology to *change* a business process to handle external factors such as a change in government regulations or change in market preferences and internal factors such as a change in the businesses goals and policies. To this end, this dissertation develops a theoretical framework for modeling business processes and describes applications of this framework. The Internet, which we present next, is one of the most influential among technologies that have affected business processes.

1.3 The Internet as a Computing Platform

The Internet has become a large, distributed-computing platform. Applications whose reach was limited to a single machine now have access to and incorporate the services of other programs that are distributed over a wide area, span multiple heterogeneous organizations and systems, and are governed by different policies. Two key factors driving this transformation are *The Semantic Web* and *web services*.

1.3.1 The Semantic Web

The vision of the Semantic Web (Berners-Lee et al., 2001) is a transformation of the World Wide Web from a network of human-readable pages of information marked with a description of its presentation (such as by the HyperText Markup Language, HTML) to a network of information marked with a description of its *meaning* or semantics (such as by the Resource Description Framework, RDF) that can be reasoned about by programs. Progress along the lines of this vision has been slow, and only a small fraction of the information on the WWW is currently marked up in RDF (Finin et al., 2005). The Semantic Web vision is however, gaining traction due to active research and industry interest.

1.3.2 Web Services

The vision of web services is to enable programs to dynamically discover and interact with other programs, over the WWW using XML-based standards. Examples of such standards are the Web Service Description Language (WSDL) (Christensen et al., 2001) for describing the data and message formats of a service and the Universal Description, Discovery and Integration (UDDI) (Clement et al., 2004) protocol for describing a directory system for advertising and finding web services, and the Business Process Execution Language for Web Services (BPEL4WS) (Andrews et al., 2003) for describing workflows that involve invocation of web services. Web services are relatively more mature when compared to the Semantic Web, and have been implemented by large corporations such as Google and Amazon. However, web services are low-level specifications, and can form the foundation on which richer frameworks such as the Semantic Web are built to fully realize their potential.

These developments hold great promise for automating business processes just as computerization revolutionized business processes in the last century. The two most important benefits that the Semantic Web and web services offer to businesses are *dynamism* and *global reach* when choosing and interacting with business partners and customers. Modern businesses have moved the processes that they use to interact with their partners to the web, so that they can leverage the wide reach of the Web to increase productivity and profits (Petrie and Sahai, 2004).

1.3.3 The Impact and Scope of the Web

Whereas an Internet-based business interaction scheme provides businesses access to a larger number of partners, suppliers, and customers due to its global reach, the Internet marketplace changes constantly. New vendors and consumers go online every day, as old ones go offline. Businesses therefore need to make partnerships dynamically, changing partners to maintain the profitability of their operations as a partner's value decreases or as laws and market situations change. In such a changing environment, it becomes tedious, if not impossible, to manually keep track of potential partners to interact with. Further, the presence of a large number of potential partners means that processes need to be able to handle interaction with systems that are designed and behave in different ways. The key to process interoperation in such a heterogeneous environment is standardization. When process interfaces are standardized and implementors adhere to these standards, successful interoperation is achieved easily. Web services enable businesses to leverage the dynamism of Internet marketplaces by providing widely adopted, XML-based industry standards

for advertising and discovering services (via UDDI). For its part, the Semantic Web can be used to markup services using a rich vocabulary for specifying elements of quality of service, thus enabling intelligent (automated) selection of services, as done in the OWL-S (Web Ontology Language for Services) (Martin et al., 2004) project.

Since Internet-based operations are more cost effective than traditional brick-and-mortar businesses, even small businesses set up operations online using online marketplaces such as eBay. In this model of online business, the marketplace—here, eBay—provides a common platform, avoiding incompatibilities between the partners' software systems and governing the interaction between them, for a fee. In an open setting, however, businesses will want to eliminate the cost of operating in such marketplaces. Web services, with standards that are free to use, provide a viable solution for such a need. The standardization of message formats (SOAP) (Gudgin et al., 2003) and interface definitions (WSDL) and the wide adoption of these standards enables businesses access partners anywhere in the world. Semantic markup of data and services can help reconcile differences in representation used by different businesses.

1.4 Scope and Relevance

While standardization facilitates interoperability between diverse systems over the Internet, business process specifications rarely specify each step of the process and its possible outcomes (if there is any documented process at all). A business process in most organizations comes about due to a complex interplay between legality of the organization's operating procedures, market trends, and company policies. Further, processes evolve over time due to changes in local policies, governmental policies, and market situations. In an open setting, it is in the best interests of a business to change its partners to adapt to new situations. Web services standards have not incorporated business-level abstractions that are needed when new processes need to be developed or existing ones need to be modified. Recent efforts in business process management and business process composition indicate an increasing interest in the area. If processes are modeled without a clear understanding of the changes that may occur and without a capability to handle such changes easily, standardization efforts are rendered ineffective. Further, the standards themselves might be too restrictive in that they might prevent a participating business from exercising its autonomy. Automating business processes is therefore an exercise in finding the right tradeoff between flexible process models and well-defined standards.

This dissertation deals with solutions to some of the problems faced by process modeling

and enactment. We propose that business processes (cross-enterprise or intra-enterprise) be composed of well-defined *protocols* in which different *roles* interact. We treat protocols as first-class entities and develop a protocol algebra to aid designers of processes. Further, we develop methodologies for dealing with exceptions in protocols. Our theoretical results formalize high-level, semantic aspects of business protocols leading to better and easier process design and enactment.

We propose an agent-based framework and a new way of thinking about interaction protocols. This yields a new approach for synthesizing processes from existing protocols (which are templates for interactions) that aids process designers in developing such hybrid processes. The use of commitments also helps us develop a theory of exception handling and a methodology for the same. Our work has applications to traditional technologies such as workflow management systems and emerging areas such as multi-agent systems. In the course of our efforts we also have contributed significantly to the agent communication research community.

1.5 Approach

We propose a novel approach for designing flexible processes. Simply put, *a process instantiates one or more protocols* among designated parties. We define a protocol as a specification of a logically related set of interactions. For example, we can have protocols for negotiation, for payment, for selecting a shipping company, and so on. Protocols are interfaces. By this we mean that a protocol specifies only the key desired aspects of the interactive behavior; it leaves the details of a local implementation entirely up to those who implement the protocol.

We want general protocols to support flexibility, and specialized processes to support efficiency, security, or risk management. Ultimately, each environment may have a set of desirable protocols. For example, we can imagine a generic payment protocol as well as specializations of it such as payment by cash, credit card, checks, travelers checks, bank draft, wire transfer, in foreign currencies, via vostro and nostro accounts, by Paypal (Paypal), using hawala, and so on. Each of these would have any number of subtleties, e.g., to pay with a check you must show a government ID and a major credit card (which isn't used, but its number is noted), and these protocols could be combined in unusual ways, e.g., some restaurants accept credit cards or travelers checks only; some only accept cash or check. Moreover, the protocols only specify the interactions, not the local policies of the participating entities, such as that they don't take cash after sunset. Protocols enable such policies to be inserted but are not directly concerned with the policies. As long as we recognize that these are payment protocols, our top-level design goal, namely, to enable some form of payment

would be satisfied.

While at first it seems that protocols in which the participants are free to do as they wish would be the best means of preserving their autonomy, standardized protocols are required to enable interaction between disparate systems. Standardization helps check if all parties in an interaction are compliant with the protocol specification. Protocols should therefore be designed with the right balance of participants' autonomy and standardization of the interaction.

We take inspiration from such protocols that exist in the literature already; specifically networking and distributed systems protocols such as IP, TCP, HTTP, and SIP (Rosenberg et al., 2002). These protocols provide published specifications of the behavior of the roles interacting through them. Thus they enable implementors to independently create components that, if compliant, are guaranteed to work together. If we could specify *business protocols* precisely, we could enable the desired business processes in which the autonomy of the participants is maximized (limited only by the requirements of the protocols in effect) and their implementations are heterogeneous. Components could be plugged in or out with ease and processes could be readily adjusted so that the participants can handle exceptions and exploit emerging opportunities.

Whereas networking and distributed systems need only a handful of universal protocols, realistic business settings would need an endless variety of business processes. While some of these processes will be widely deployed, several will be customized to special application domains, industries, and circumstances. For example, the automobile industry would have a variety of processes to manage various aspects of its supply networks. Our basis for this belief arises from the variety and subtlety of existing business processes, such as for supply chain management. Whereas today these processes are coded up in a closed fashion requiring a serious integration and configuration effort, we imagine that by employing well-specified, published protocols to compose processes, the various stake-holders can considerably simplify their integration and configuration efforts (Huhns et al., 2002). Given a set of protocols, they would only need to acquire implementations for the roles of those protocols. Further, if the desired processes are developed by composing existing protocols, integration and configuration is further simplified, because you can put together the necessary software modules in a manner that parallels how the protocols are put together. RosettaNet is already a step in this direction (RosettaNet). It defines over 100 protocols (called Partner Interface Protocols, or PIPs, in their terminology). RosettaNet's protocols are limited to two-party implementations and are mostly two-step protocols. Nevertheless, RosettaNet is in use in the industry and accounts for a considerable volume of cross-enterprise interaction for businesses such as the Intel Corporation (Krazit, 2002).

1.6 Contributions

Protocols will be needed for every aspect of business processes that involves standardized interaction. Payment is just one aspect, but there are as many aspects as business interactions. Clearly, developing sophisticated protocols would not be a trivial undertaking. It needs abstraction support akin to information or object modeling in traditional systems.

The most fundamental computer science approach for dealing with complexity is to enable reuse. Two of the most basic ideas for doing so are to build a specialization-generalization hierarchy and to aggregate components. The contributions of this dissertation are twofold. One, we develop notions akin to traditional subsumption and aggregation that are applicable to protocols. We develop two main classes of abstractions: *refinement* (like the subclass-superclass hierarchy) and *aggregation* (like the part-whole hierarchy). We develop a formal semantics to support the hierarchy and propose an algebra to facilitate reasoning about protocols. This is a fundamental change from viewing workflows as executing applications in a certain order to co-ordinating behaviors of interacting agents. Two, we treat exceptions in process enactment as an integral part of the theory of protocols, not as components that are added on. The introduction of the social structure of agents models the organizational structure that exists in business processes. The existance of organizational structures creates rich theories of exceptions and their handling.

Chapter 2

Background: Flexible Protocols and Commitments among Agents

This chapter describes concepts that underlie the design of interaction protocols and explains the intuitions and formalism we have developed to capture these concepts. Section 2.1 describes the intuition behind our approach to develop modular and flexible protocols. Section 2.2 informally introduces the concept of commitments among agents, which are an important abstraction in our protocol model. This section also describes how commitments are applicable in business interactions(in Section 2.2.2). Section 2.3 outlines an example of a business interaction that we use in many parts of this dissertation to demonstrate our approach.

2.1 Protocol Design Concepts

In essence, each protocol allows a set of computations. More general protocols allow more computations than the protocols that refine them. In simple terms, if we begin with a protocol and eliminate even one of the computations in it, we would have produced a refinement of it. However, in practice, the refinement of a protocol involves more than just selecting some of the computations it allows. Typically, additional constraints and wrinkles must be added to the computations that remain acceptable in the refined protocol. For example, a simple payment protocol might require that the payer transfer funds to the payee. A particular refinement of this might be that the transfer of funds is done through payment with a check. To pay with a check, the payer would send a check to the payee who would deposit the check to his bank, which would present it to the payer's bank, which would (presumably after debiting the appropriate amount from the payer's account) send the

funds to the payee's bank, which would make those funds available to the payee. Thus payment by check is a specialization of payment, but it involves a lot more steps. We know it is a refinement of the payment protocol because, as the payment protocol requires, it ends up transferring funds to the payee.

In general, the refinement of protocols can involve a mix of changes along two dimensions as explained below.

1. A protocol can be refined by a shrinkage along the dimension of the number of computations allowed by it. Protocols that allow many computations are the most flexible because there are numerous ways to satisfy such protocols. Protocols that allow fewer computations have limited ways in which they can be satisfied.

2. A protocol can be refined by an expansion along the dimension of the number of steps involved in an allowed computation. If a protocol dictates a large number of steps that have to be taken achieve the interaction that it models, that protocol unduly restricts the autonomy of its participants as compared to a protocol which specifies few steps and allows variations in the length of computations.

The combination of changes along these dimensions result in the following observations.

- Protocols that allow a few long computations are the most restrictive because there are only a few ways to enact these protocols.

- Protocols that allow many long computations are more flexible than the above, because they offer more alternatives.

- Protocols with a few long computations restrict the autonomy of the participants. Such protocols are not necessarily bad, though. They have a distinct advantage in checking compliance. In an extreme case, if a protocol allows exactly one sequence of actions, we will know immediately if it has been violated when one of the expected actions does not occur.

There is thus a tradeoff between protocol flexibility and the complexity of compliance-verification. Figure 2.1 shows this tradeoff schematically.

- General protocols allow more computations; are more flexible.
- Long computations are more specific than shorter ones that achieve the same interaction.
- There is a tradeoff between protocol-flexibility and complexity of compliance-verification.

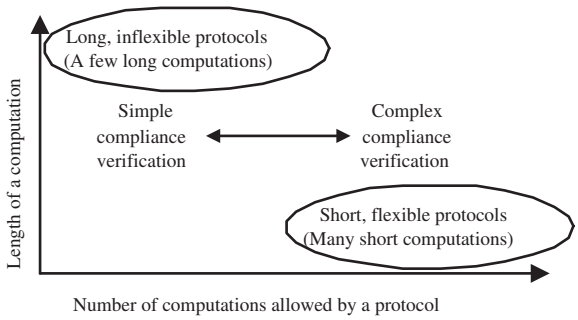


Figure 2.1: The tradeoff between flexibility allowed by a protocol and the ease of verifying compliance with it: Compliance with flexible protocols is harder to verify because of the large number of computations it allows.

2.2 Commitments

Commitments are an important abstraction that we use to build our model of protocols. Commitments are obligations that autonomous agents have towards one another because of promises they make. Commitments are thus akin to contracts and enable predictability in agent interactions to a certain degree. Here, we provide arguments for the use of commitments in business processes.

2.2.1 Commitments Among Agents

To talk about how a protocol specializes or aggregates one or more protocols presupposes that we can characterize the computations allowed by a protocol and the evolving states of those computations so we can consider whether a particular refinement or detour is legitimate. For business protocols, therefore, this means we must represent not just the behaviors of the participants, but also how the *contractual relationships* among the participants evolve over the course of an interaction. Doing so enables us to determine if the interactions are indeed compliant with the stated protocols.

The contractual relationships of interest are naturally represented through *commitments*, which have gained importance in the field of multiagent systems (Castelfranchi, 1993). Commitments capture the obligations of one party to another. For example, the customer's agreement to pay the price for the book after it is delivered is a commitment that the customer has towards the bookstore. Using commitments enables us to model not just a participant actions, but also how they advance the ongoing business interaction, which enables us to more readily detect and accommodate business exceptions and opportunities (Yolum and Singh, 2002).

To apply commitments presupposes that we are modeling the participants as *agents*, which, for our purposes can be regarded as persistent computations capable of forming commitments with other agents. Agents naturally represent parties such as the business partners involved in an e-business scenario, who might collaborate but retain their autonomy. Agent interaction is a lively field of ongoing research (Huget, 2003; Dignum, 2003) which this work contributes to as well as draws from. Commitments lend coherence to the interactions because they enable agents to plan based on the actions of others. In principle, violations of commitments can be detected and, with the right social relationships, commitments can be enforced—by penalizing agents who do not comply with their commitments. Enforceability of contracts is necessary in practical settings where the participants are autonomous and heterogeneous (Singh, 1998).

2.2.2 Commitments in Business Interactions

Commitments have been used to model interaction protocols, especially in e-business settings (Verdicchio and Colombetti, 2002; Yolum and Singh, 2002). Such formulations generally afford more flexibility to the participants in choosing the actions they may take at different stages. However, this line of research does not adequately address the specific need of designing business protocols, particularly with respect to reasoning such as specialization and aggregation. One of the objectives of this dissertation is to develop notions akin to traditional subsumption and aggregation that are applicable to protocols. Doing so presupposes that we have a crisp semantics and can reason formally about protocols. Accordingly, our first task is to develop a semantics that facilitates flexible actions.

The semantics of protocols is orthogonal to the methodology used for specifying protocols. A designer may proceed top-down by first specifying a high-level protocol and then refining it; another designer may proceed bottom-up by first specifying a variety of refined protocols and then generalizing over them. Still another designer may begin with an abstract protocol and merge in specific protocols for some of its steps. In Chapter 6, we describe a group effort that has sought to realize commitment protocols. This chapter describes some protocol design considerations with examples.

- Commitements are contracts between agents.

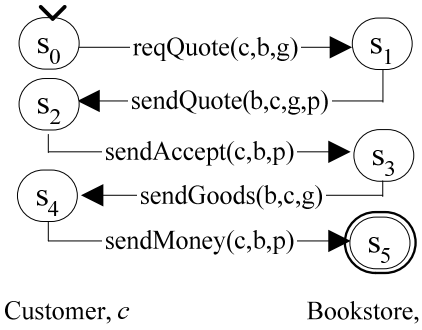


Figure 2.2: *Purchase scenario 1*: Customer agent asks Bookstore agent for a quote and is given one which it accepts. Customer agent is then sent the goods for which it pays.

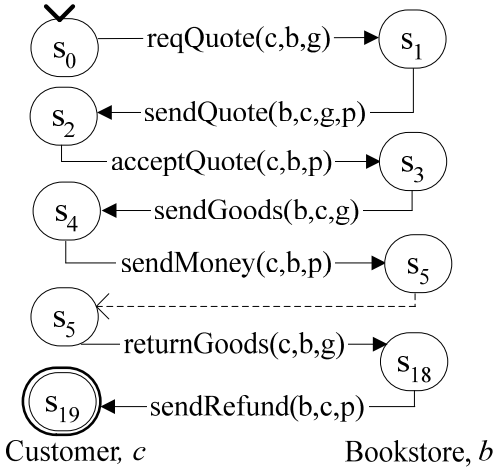


Figure 2.3: *Purchase scenario 2*: Customer agent asks for a quote and receives one which it accepts. Customer agent is then sent the goods for which it pays. Customer agent then returns the goods for a refund.

2.3 The Purchase Example

As a running example, we consider a *purchase* interaction in which a customer wants to buy a book from an online bookstore. The process involved is a *Sell* process as defined in the MIT Process Handbook (Malone et al., 2003), specifically the *Obtain order*, *Deliver product or service*, and the *Receive payment* parts of the *Sell* process. The bookstore obtains the customer's order for a book if the customer accepts the price quoted by the bookstore for that book. The book is then shipped to the customer, and the bookstore is paid for the book. The actual execution of the process, however could involve many different scenarios, as described next.

We identify four distinct, but related, scenarios that can arise during the above book purchase interaction. Each of these scenarios requires a different amount of effort from the participants in terms of protocol execution, planning, and coordination. All participating agents would benefit from being able to compare scenarios to choose the one that best serves their interests. These scenarios are shown in Figures 2.2, 2.3, 2.4, and 3.4. *Customer* refers to the customer's agent and *Bookstore* refers to the bookstore's agent. Ellipses represent states, labeled s_i. Solid arrows are labeled by the messages that are passed between the participating agents. These messages correspond to actions that the agents take. Note that we have displayed states in columns representing roles. This is purely for convenience, since the protocols states are not local to any role.

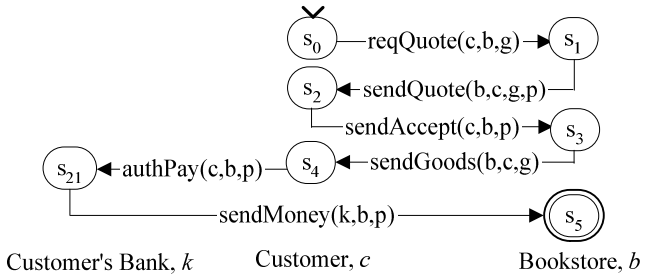


Figure 2.4: *Purchase scenario 3*: Customer agent asks for a quote and receives one which it accepts. After receiving the books sent by the Bookstore agent, the Customer agent authorizes its Bank agent to pay the Bookstore agent on its behalf.

1. Normally, the customer would ask the bookstore for a price quote on the book it wishes to buy, and upon receiving a quote from the bookstore, would accept the bookstore's offer. The bookstore would then send the book, after which the customer would send the payment. This is modeled after the NetBill protocol (Sirbu, 1997). Figure 2.2 shows this interaction.

2. The bookstore might be willing to give a refund if the customer returns the book for some reason. This scenario is similar to the scenario described above till the book is delivered to the customer. However, this scenario is longer, since the customer returns the book, and the interaction terminates only when the bookstore sends the refund to the customer. Figure 2.3 shows this interaction.

3. The customer might delegate the payment to a third party, e.g., a bank. Such a situation is not very different from using a credit card to pay for goods, and is shown in Figure 2.4. The customer, after accepting the bookstore's price quote and later receiving the book, sends a message to both the bookstore and the bank (although only one arrow is shown in the figure, between states s_4 and s_{21}) indicating that the bank will honor the customers commitment to pay.

4. The bookstore might have to negotiate with and contract out the actual shipping to a shipper. This might happen, for example, if the customer wants insured shipping, and the bookstore's existing shipper does not insure goods. Here, the bookstore interacts with the shipper, and gets the books delivered to the customer. The shipper is then paid by the bookstore, but only after the book has been delivered to the customer. To complicate matters, the customer pays via its bank in this scenario, just like in the previous one. This scenario is shown in Figure 3.4 and discussed in detail in Section 3.3.

Table 3.3 explains the meanings of the states that the first scenario runs through. Table 3.1 shows the meanings of the messages passed. In this table, c represents the customer, b represents the bookstore, g represents the book that the customer is interested in buying, and k represents the customer's bank. The *delegate* and *assign* messages relate to corresponding operations that can be performed on commitments. Commitments and the operations that are possible over them are defined in the next section.

2.4 Status and Trends

Current models of processes specify an ordering of steps to be taken to enact the process. These approaches prove inflexible for modeling complex processes that change frequently. Even when such systems allow re-configuration to meet the demands of a dynamic organization, it comes at a high price, that of intensive software re-engineering (Janssens et al., 2000). This has been an important reason for workflows not being very successful. Modern business processes span multiple autonomous entities or business partners. Such processes, therefore, are based on a rich variety of interactions among software components that are independently designed and configured, and which represent independent (sometimes mutually competitive) business interests. Even though technologies for implementing processes have advanced, the fundamental concepts underlying the interaction between business components remain largely unchanged from those used in early centralized systems. We outline here the main trends in technologies that have been employed for business process automation.

2.4.1 Trends in Workflows

Early implementations of processes were tied to specific data and applications leading to limited usability under changing business conditions. With time and a better understanding of business automation practices, there have been advances in abstracting out the data, the applications, the flow, and the process from the systems that implement it (van der Aalst and van Hee, 2002). This has been the trend that workflows, one of the oldest and most mature process automation and modeling technologies, have followed. The Workflow Management Coalition (WfMC) is a standards body that has created a reference model for workflows (Fischer, 2004). This model has two basic parts, a modeling and an enactment part. The model prescribes a workflow engine as the system that executes the workflow. These models specify a rigid sequence of steps. Workflows

require human intervention to handle most exceptions. Because of their inflexibility, workflows have had only limited success. Workflows are executed by a central workflow engine that steps through the flow, invoking applications as required, and supplying the result of one invocation to the next application in the flow. Such engines view these applications as black boxes, and are not equipped to handle high-level exceptions (where the workflow system and its components function correctly but are in disagreement with business policies) in an intelligent manner. The engine can retry a failed application at best, or abort the entire workflow instance, at worst. What processes need are models that better handle semantic exceptions, resolving them by compensating techniques. When something bad happens in a process, retrying the failed components is not always the best thing to do. For example, if a process involves shipment of goods, and the shipment is delayed due to adverse weather,a monetary compensation might be more prudent than sending the goods again after a timeout.

Although advances have been made in workflow technologies (Fischer, 2004) and exception handling in workflows (Casati, 1998; Kamath and Ramamritham, 1998), workflows provide marginal gains over procedural scripts because they still view process enactment as a series of steps to be performed, not as a principled combination of meaningful interactions. Agent-based technologies have been applied to workflows such as agent conversations, where a process is treated as a conversation between partners (Hanson et al., 2002a), and multiagent system-models that enact workflows that adapt easily to changes (Buhler and Vidal, 2003; Hanson et al., 2002b). Our efforts are in line with the agent approach.

2.4.2 Trends in Web Services

A recent trend is to expose services offered by a business as web services and engage a service over the web (Singh and Huhns, 2005). Web services are being promoted as the most suitable medium to enact cross-enterprise processes because web services allow systems to interact in a platform-independent manner. Whereas web services have helped standardize important aspects of interoperation such as advertising a service and invoking a service, our chief concern is with the *interaction* between a service provider and a service consumer, which has only recently been addressed by efforts for web service *choreography*—which deals with the way services interact—and *orchestration*—which deals with the way services are composed using other services (Peltz, 2003).

Web service orchestration refers to one party's view of an interaction. It involves putting

together service components to expose a service interface for consumers to use. For example, a travel agency would put together software to access its internal databases and various airlines' systems as part of orchestration of its service. The agency would reveal an interface for customers to access and utilize the service it offers. Orchestration is therefore analogous to an *implementation* of an interface. Web service choreography, on the other hand, refers to the actual interaction pattern between services. It involves the specification of the ordering requirements among messages and method invocations. In the above travel agency example, choreography of the interaction with the customer would involve the sequence of operations that the customer has to invoke or messages that it has to send to the travel agency service to engage the service normally. Among the emerging standards for web service interaction and composition, the Business Process Execution Language for Web Services (BPEL4WS) (Andrews et al., 2003) seems to be the most popular. BPEL4WS specifies *abstract processes* for choreography and *executable processes* for orchestration of web services. Our effort applies to service choreography more than orchestration.

While web service orchestration and choreography standards facilitate interoperation, they are too low a level of abstraction to help much with the rich variety of interactions that real-life businesses engage in because web services, like workflows do not provide a meaning to interactions. This has led to interest in a range of technologies and standards for coordination and process flows.

Coordination and Process Flow Technologies.

The need for process composition and interoperability has led to the development of standards for orchestration and choreography of web services (Peltz, 2003). Orchestration refers to intra-service planning and choreography to an overall view of inter-service coordination. Notable standards among these are the Web Service Choreography Interface (WSCI) (Arkin et al., 2002) for describing the interface of one service in a choreography, the Web Services-Choreography model (Burdett and Kavantzas, 2004) for describing the format and sequence of globally observable messages sent between services in a choreography, the Business Process Management Language (BPML) (Arkin, 2002) that specifies orchestration, The Web Services Coordination (WS-C) (WS-C, 2003) standard, and the Business Process Execution Language for Web Services (BPEL4WS), which is currently the most widely used web services standard for describing business processes (Andrews et al., 2003). BPEL4WS is based partly on WS-C.

These standards, however, are not significantly different from workflows in terms of their usage. While these approaches have some benefits, they suffer from the major shortcoming that they

mostly take a centralized perspective, akin to workflow technologies, viewing a process as a series of tasks to be performed. This proves too tedious for reliable modeling and too rigid for enactment. The Collaxa BPEL Orchestration Server (Collaxa, 2003) and the IBM BPWS4J (BPWS4J, 2004), for example, are execution engines for BPEL4WS. Both function like workflow engines, invoking services as necessary, treating the BPEL4WS document as a script.

Fu and colleagues (2004) develop methods to verify if a given web service will adhere to a given conversation protocol. Their work develops formal results about verification of protocol compliance. We intend to study their work further. Hamadi and Benatallah (2003) develop a protocol algebra for petri nets and show its applicability to workflows and web services. However, this approach suffers from the same pitfalls as workflows modeled using Petri nets. Buhler *et al.* (2003) and Vidal *et al.* (2004) are working on applying multiagent systems to create agile workflows.

The MIT Process Handbook (Malone et al., 2003) is a project that aims to create a hierarchy of commonly used business processes. The goal of the project to create an extensive classification and systematic organization of business processes based on two dimensions of process hierarchies, one that composes the *uses* of a process out of its constituent *parts*, and another that subclasses *generalizations* of a process into *specializations*. Based on this hierarchy, Grosof and Poon (2003) develop a system to represent and execute business rules.

Distributed Transaction Technologies.

These include such technologies as the web service transaction (WS-T) (WS-T, 2002) standard, which BPEL4WS improved upon, the Tentative Hold Protocol (THP) (Roberts and Srinivasan, 2001), and the Business Transaction Protocol (BTP) (BTP, 2002).

This dissertation relates to both workflow and web services efforts, but concentrates on the semantical aspects of the interactions among business partners. We introduce the concept of commitments, as used in multiagent systems, to accommodate high-level reasoning about processes that involve interaction between business partners. Such reasoning enables business partners handle special situations in a natural manner.

2.4.3 Trends in Agent Communication

One of the most important benefits of multiagent systems is the social structure that they model, i.e., the the autonomy and heterogeneity of the constituent agents. To maximise these benefits, agent interaction and agent communication have been extensively studied.

Yolum and Singh (2002) give one of the first accounts of the use of commitments in modeling agent interaction protocols and the flexibility that it affords the participating agents. Fornara and Colombetti (2003) describe how commitments relate to FIPA-ACL messages and demonstrate with an example. Both approaches highlight the benefits of a commitment-based approach to interaction protocol design.

Johnson *et al.* (2003) develop a scheme for identifying when two commitment-based protocols are equivalent. Their scheme, however, is simplistic, classifying protocols based solely on their syntactic structure. Our work provides stronger results about the relationships between protocols from an application point of view and relates better to the Web Services approach.

Bussmann *et al.* (2003) present a design methodology to aid in the selection of a protocol from a library of existing protocols to apply to agent-based control applications. They identify criteria like the number of agents, the number of roles, and the number and kind of commitments and use these to select a protocol from an existing pool of interaction protocols. This approach is quantitative, and lacks a formal semantics to base the methodology on.

Pitt and Mamdani (2000) describe a semantics for agent interaction protocols using the Belief-Desire-Intention (BDI) theory. Using this semantics, they outline the design of a system of agent plans that are instantiated by agents to carry on conversations with other BDI agents. In our work, an agents beliefs, desires, and intentions are private to that agent. We work with social commitments which are observable by all agents and whose breach is easier to verify.

In more recent work, Vitteau and Huget (2004) describe an approach for designing agent interaction protocols using modular *micro-protocols*. This scheme is similar to our protocol design proposal in spirit. However, Vitteau and Huget do not provide a formal basis for putting protocols together.

The Foundation for Intelligent Physical Agents (FIPA) has produced many standards specifications for different parts of multiagent communication and interaction. The FIPA interaction protocols, communicative acts, and content language specifications form the FIPA-ACL (Agent Communication Language) (FIPA-ACL, 2000). The FIPA-ACL is a clear specification because it is based on formal semantics, but can be computationally demanding on agents which have to reason when protocols are being executed. The FIPA content language specifications deal with the representation of the content of ACL messages. The communicative acts give meanings to messages of the ACL based on modal logic. The interaction protocols describe the sequence of message exchanges between FIPA compliant agents.

Chapter 3

A Formalization of Interaction Protocols

In this chapter, we develop our formalism for modeling commitment protocols and an algebra for protocols. Our formalism is developed so as to capture the flexibility of execution that commitment protocols provide. We demonstrate the use of this algebra in combining protocols with an example. We also outline the technical developments and applications of this algebra that will be presented in the rest of the dissertation.

3.1 Technical Framework

We represent protocols as transition systems similar in spirit to finite state machines. These protocols generate computations or *runs*, which are sequences of *states* that a valid protocol computation (execution) goes through. State changes are caused by *actions* that the participants perform. We devise a hierarchical classification based on the runs generated by protocols. This classification forms the basis of our work. The remainder of this section introduces commitments, our running example, and the basic technical concepts of propositions, states, actions, runs, and protocols.

3.1.1 Commitments

Much of our technical development is based on established concepts of distributed artificial intelligence. We first introduce commitments, a concept used in the agent realm.

Definition 1. *A commitment* $\mathsf{C}(x, y, p)$ *denotes that the agent* x *is responsible to the agent* y *for bringing about the condition* p*.*

Here x is called the *debtor*, y the *creditor*, and p the *condition* of the commitment. The condition is expressed in a suitable formal language.

Commitments can also be *conditional*, denoted by $\mathsf{CC}(x, y, p, q)$, meaning that x is committed to y to bring about p if q holds.

3.1.2 Commitment Operations

Commitments are created, satisfied, and transformed in certain ways. The following operations are conventionally defined for commitments (Singh, 1999).

1. *create*(x, C) establishes the commitment C in the system. This can only be performed by C's debtor x.

2. *cancel*(x, C) cancels the commitment C. This can only be performed by C's debtor x. Generally, cancellation is compensated by making another commitment.

3. *release*(y, C) releases C's debtor x from commitment C. This only can be performed by the creditor y.

4. *assign*(y, z, C) replaces y with z as C's creditor.

5. *delegate*(x, z, C) replaces x with z as the C's debtor.

6. *discharge*(x, C) C's debtor x fulfills the commitment.

A commitment is said to be *active* if it has been created, but not yet been operated upon by a *discharge*, *delegate*, *assign*, *cancel*, or *release*. We adopt this definition from our previous work on the temporal aspect of commitments (Mallya and Singh, 2005) to maintain consistency with the commitment semantics defined therein. Although this definition suffices for our study of contracts among businesses, models that require a concept of *responsibility* might necesitate the use of other definitions in which a commitment is tracked as being active even after it has been delegated or assigned.

3.1.3 Propositions

Propositions capture facts about what conditions hold, what commitments have been made, and whether these commitments have been fulfilled. To motivate the comparison of states and runs, we differentiate propositions into three pairwise disjoint sets.

Table 3.1: Messages used in the purchase example

Message	**Meaning**
reqQuote(c, b, g)	c asks b what the price of g is.
sendQuote(b, c, g, p)	b quotes price p to the c, for g.
sendAccept(c, b, g, p)	c accepts the price p quoted by b for g. c is now committed to pay if the book is sent to it.
sendGoods(b, c, g)	b sends g to c.
sendMoney(c, b, p)	c sends the money p to b.
delegate(c, k, C)	c delegates the commitment C to k.
returnGoods(c, b, g)	c returns g to b.
sendRefund(b, c, p)	b refunds the money p to c.
authPay(c, b, p)	c authorizes its bank to pay the amount p to b. Essentially c delegates $\mathsf{C}(c, b, p)$ to k.

1. *Domain propositions.* The set Dom of protocol-specific atomic propositions. A domain proposition used in the purchase example is *goods*(b, c, g), which means that agent b has delivered g to agent c. Table 3.2 describes some domain propositions for the purchase example.

2. *Commitment propositions.* The set Com of propositions which represent active commitments. An example of such a proposition is $\mathsf{C}(c, b,$ *pay*$(c, b, p))$, which means that agent c is committed to agent b to pay the amount p, as in Definition 1.

3. *Commitment operation propositions.* The set ComOp of propositions that represent commitment operations that were described in Section 3.1.2, and form the set ComOp. An example of such a proposition is *assign*(c, x, C), which means that agent c assigns its commitment C (which is of the form $\mathsf{C}(c, \cdot, \cdot)$ to agent k. In addition to the six commitment operations, this set of propositions also contains propositions that represent whether a commitment has been fulfilled (*satisfied*$(\cdot)$) or violated (*breached*$(\cdot)$).

The propositions used in a protocol are assumed to be understood by parties involved in the protocol. In the purchase example, we use the domain propositions given in Table 3.2. These propositions belong to the set Dom, but do not form an exhaustive listing of the set, since propositions given here are parameterized by g and p. Here, c represents the customer, b the bookstore, g the book the customer wants to buy, and p the price quoted for g by b.

The set of propositions $\mathbb{P}$ is given by $\mathbb{P} = \mathsf{Dom} \cup \mathsf{Com} \cup \mathsf{ComOp}$. A proposition is generated by the nonterminal P in the following Backus-Naur Form (BNF) grammar. Here, terminal symbols begin with a lowercase letter, the exceptions being C and CC. The symbols $\longrightarrow$ and $|$ are

Table 3.2: Propositions in the purchase example

Proposition	**Meaning**
reqQuote(c, b, g)	c has requested a quote for g from b
quote(b, c, g, p)	b quotes to c price p for g, i.e., b will deliver if c commits to pay upon delivery. This is represented by $\mathsf{CC}(b, c,$ *goods*$(b, c, g))$, *acceptQuote*(c, b, g, p)
acceptQuote(c, b, g, p)	c has accepted the price p that b quoted for g, i.e, c commits to pay if the goods are delivered. This is represented by $\mathsf{CC}(c, b,$ *pay*$(c, b, p),$ *goods*$(b, c, g))$
goods(b, c, g)	g has been delivered to c by b.
pay(c, b, p)	The amount p has been paid to b by c.
return(c, b, g)	g has been returned to b by c.
refund(b, c, p)	The amount p has been refunded to c by b.

a meta-symbol of the BNF, a and b belong to the set of *roles*, or participants $\mathbb{R}$ of the protocol, and *atomic_prop* belongs to Dom.

The nonterminal *Commit_prop* generates commitment propositions and *Commit_oper_prop* generates commitment operation propositions.

1. $P \longrightarrow$ *atomic_prop* | *Commit_prop* | *Commit_op_prop*

2. *Commit_prop* $\longrightarrow \mathsf{C}(a, b,$ *atomic_prop*$) \mid \mathsf{CC}(a, b,$ *atomic_prop*, *atomic_prop*$)$

3. *Commit_op_prop* $\longrightarrow$
   create$(a,$ *Commit_prop*$)$
   | *cancel*$(a,$ *Commit_prop*$)$
   | *delegate*$(a,$ *Commit_prop*$)$
   | *assign*$(a,$ *Commit_prop*$)$
   | *discharge*$(a,$ *Commit_prop*$)$
   | *satisfied*$($*Commit_prop*$)$
   | *breached*$($*Commit_prop*$)$

3.1.4 States

Definition 2. *A state is an assignment of truth values to members of* $\mathbb{P}$.

Equivalently, a state is labeled with the set of propositions which hold at that state. For example, state s_1 of the purchase example is labeled by the set $\{$*reqQuote*$(c, b, g)\}$ and state s_2 by

$\{\mathit{quote}(b, c, g, p)\}$. We denote the label of a state s by $[s]$. Table 3.4 gives the labels that are assigned to states in the purchase example. The set of states is denoted by $\mathbb{S}$.

Table 3.3: Meanings of some states in the purchase example

State	Meaning
s_1	Customer has asked the bookstore the price of the goods. No commitments made.
s_2	Bookstore has quoted a price for the said goods. The bookstore is now willing to send the goods if the customer promises to pay for them
s_3	Customer has agreed to the bookstores price. The customer is willing to pay the price if the books are delivered
s_4	Bookstore has delivered the book
s_5	Customer has paid for the book

Table 3.4: State labels in the purchase example

State	Associated Label
s_0	$\{\mathsf{true}\}$
s_1	$\{\mathit{reqQuote}(c, b, g)\}$
s_2	$\{\mathit{quote}(b, c, g, p)\}$
s_3	$\{\mathsf{C}(b, c, \mathit{goods}(b, c, g)),$ $\mathsf{CC}(c, b, \mathit{pay}(c, b, p), \mathit{goods}(b, c, g))\}$
s_4	$\{\mathit{goods}(b, c, g), \mathsf{C}(c, b, \mathit{pay}(c, b, p))\}$
s_5	$\{\mathit{goods}(b, c, g), \mathit{pay}(c, b, p)\}$
s_{21}	$\{\mathit{goods}(b, c, g), \mathsf{C}(k, b, \mathit{pay}(k, b, p))\}$
s_{17}	$\{\mathit{goods}(b, c, g), \mathit{pay}(c, b, p)\}$
s_{18}	$\{\mathit{goods}(b, c, g), \mathit{return}(c, b, g),$ $\mathsf{C}(b, c, \mathit{refund}(b, c, p))\}$
s_{19}	$\{\mathit{goods}(b, c, g), \mathit{return}(c, b, g),$ $\mathit{refund}(b, c, p)\}$

3.1.5 Actions

Agents perform actions to bring about changes in the world. The communicative actions of the agents correspond to messages sent by the agents to others. A communicative action of an agent affects the state of a protocol in which it participates. For simplicity, we use messages to model even interactions that take place in the real world. For example, a $\mathit{sendMoney}(c, b, p)$ message sent during the execution of Scenario 1 of the purchase example models the action of the customer c paying the bookstore b an amount p, and a $\mathit{returnGoods}(c, b, g)$ message models the action of the customer c shipping the book g back to the bookstore b. Messages model actions just like in the real world where, for example, filling a form and submitting it over the web can cause a transfer of funds.

The set of actions is denoted by $\mathbb{A}$. The meanings of actions used in our purchase example are given in Table 3.1. In addition to the actions shown, $\mathbb{A}$ also contains an action for each proposition formed from the commitment operations described in Section 3.1.2. For example, $\mathbb{A}$ contains an action $\mathit{delegate}(c, b, p)$ since $\mathbb{P}$ contains the proposition $\mathit{delegate}(c, b, p)$.

3.1.6 Runs

Definition 3. *A run is a sequence of states* $\langle s_0 \ldots s_{|\tau|} \rangle$.

A run is one possible execution sequence of a protocol. A protocol can allow many computations, or *runs*. A run is a sequence of states $\langle s_0 \ldots s_i \ldots \rangle$. We use $\in$ to indicate the occurrence of a state on in a run. For example, $s_i \in \langle s_0 \ldots s_i \ldots \rangle$. In this paper, we consider only finite runs. The empty run is allowed.

The operator $\prec_\tau$ orders states temporally with respect to a run τ, so that $s_i \prec_\tau s_j$ implies that s_i occurs before s_j in the run τ. The concatenation of a state s_n to a run $\tau = \langle s_0 s_1 \ldots s_k \rangle$, written as $\tau \circ s_n$, is given by appending the state to the run, i.e., by $\langle s_0 s_1 \ldots s_k s_n \rangle$. The sequence of states in a *subrun* of a run $\tau = \langle s_0 \ldots s_n \rangle$ is a possibly noncontiguous subsequence of $s_0 \ldots s_n$. We denote the first state of a run τ by $[\tau]^0$, the nth state by $[\tau]^n$, and the last state by $[\tau]^\top$.

Computationally, a protocol corresponds to a set of computations that it allows. These can be captured as a set of runs where any of the runs that *subsume* the given runs may be realized. That is, each run in a protocol defines a sequence of steps that must be performed in the same order relative to each other. The subsumption of runs is defined below. A protocol is represented as a transition system as defined by a tuple $\langle \mathbb{A}, \mathbb{S}, S_0, \Delta, \mathbb{F}, \mathbb{R} \rangle$ where $\mathbb{A}$ is a set of actions, $\mathbb{S}$ is a set of states, S_0 is the set of initial states ($S_0 \subseteq \mathbb{S}$), Δ is a set of transitions ($\Delta \subseteq \mathbb{S} \times \mathbb{A} \times \mathbb{S}$), $\mathbb{F}$ is a set of final states, ($\mathbb{F} \subseteq \mathbb{S}$), and $\mathbb{R}$ is a set of roles (or participants).

Δ contains transitions of the form $\langle s_i, a, s_j \rangle$, where $s_i, s_j \in \mathbb{S}$ and $a \in \mathbb{A}$. Here s_i is the source of the transition and s_j its destination. Such a transition advances a computation from state s_i to state s_j when an action a is performed, i.e., when the message corresponding to a is sent (and received, assuming synchronous message passing, for convenience). In other words, a run can be generated from a protocol by the successive concatenation of transitions beginning from the initial state of the transition system. The concatenation of a transition to a run appends the destination of transition to the run if the source of the transition matches the last state of the run. Consequently, a run $\langle s_0 s_1 s_2 \ldots s_n \rangle$ can be generated by a protocol whose initial state is s_0, and whose transition set contains the elements $\langle s_0, \_, s_1 \rangle$, $\langle s_1, \_, s_2 \rangle$ and so on till $\langle s_{n-1}, \_, s_n \rangle$, where $s_0 \in S_0$ and $s_n \in \mathbb{F}$. The set of all such runs is denoted by $[P]$.

Protocols are specified by propositions and actions that cause states to change. The semantics of actions are given in terms of commitments such as those shown in Table 3.1. Given the actions and their semantics, the formalization of a protocol is straightforward. The transition function of a protocol can be specified explicitly as state-action-state triples or as a set of rules that are compiled into such triples for runtime efficiency. Two example transition mechanisms for commitment-based protocols are commitment machines (Yolum and Singh, 2002) and nonmonotonic commitment machines (Chopra and Singh, 2003). For example, Tables 3.2, 3.1, and 3.4, along

with a set of rules for determining the new state given the old state and the action taken would define the purchase protocol.

- Commitments can be operated upon by a fixed set of operations.
- Propositions capture the universe of discourse of a commitment protocol.
- States of a commitment protocol are labeled by propositions that hold true.
- Runs are finite sequences of states.

3.2 Reasoning about Protocols

This section presents the development of relationships among runs and protocols based on the notion of *state similarity*. We develop our results based on the theoretical work presented above. We categorize based on our definition of similarity of states, similarity and *subsumption* relationships between runs and protocols. Our theory captures our intuition that generic protocols specify little and allow a lot, whereas specific protocols specify in detail and allow little freedom.

3.2.1 Similarity of States

States form the fundamental components of runs, and are identified (and labeled) by sets of propositions. Any comparison of states, therefore, must be based on comparing propositions. This section introduces three state-similarity functions ι, σ, and $\alpha_{A,P}$, all based on commitment propositions, and shows how these help relate different runs.

A *state-similarity function* f is a mapping from a state to a set of states, i.e., $f : \mathbb{S} \mapsto 2^{\mathbb{S}}$. From such a function, we can induce a binary relation $\approx_f \subseteq \mathbb{S} \times \mathbb{S}$, which is defined as $\approx_f = \{(s, f(s)) : s \in \mathbb{S}\}$. That is,

$$s_i \approx_f s_j \quad \Longleftrightarrow \quad s_j \in f(s_i) \tag{3.1}$$

Definition 4. *A state similarity function f is well formed if $\approx_f$ is reflexive, symmetric, and transitive, i.e., $\approx_f$ is an equivalence relation.*

We denote by $\mathbb{F}$ the set of all well-formed state-similarity functions.

We constrain all runs to prevent *stuttering*, i.e., to not have consecutive states that are similar, i.e., for every run $\tau = \langle s_0 \ldots s_n \rangle$, s_i is not similar under f to s_{i+1}, for all $0 \leq i < n$.

Identity State-Similarity. ι is the identity state-similarity function. That is, $s_i \approx_\iota s_j$ if and only if s_i and s_j are labeled by same set of propositions. $\iota(s_i) = \{s_j | [s_i] = [s_j]\}$. $\approx_\iota$ is an equivalence relation.

What other kinds of similarity functions can one develop? We have developed our semantics to support reasoning about protocols on commitments among autonomous agents. States are labeled by the propositions that hold in them. Whereas domain propositions can have different semantics in different applications, commitments are an abstraction that can be used across domains with uniform semantics. Since a commitment has a well-defined life cycle and is a *directed* obligation, comparison of commitments can be performed based on two criteria.

- A comparison based on which agent is committed to which other agent.
- A comparison based on which commitment operations were performed.

As examples of usage of both the above criteria, we introduce two state similarity functions, the creditor state-similarity function σ and the role-and-commitment state-similarity function α.

Creditor State-Similarity. As another state-similarity function, consider σ. Under σ, a state s_i is similar to a state s_j if in the two states all the participants of the protocol have the same commitments being made towards them, regardless of which agent makes it. Since the creditor of a commitment is immaterial under σ and a *delegate*$(\cdot, \cdot, \cdot)$ action changes the creditor of a commitment, σ can be defined as $\sigma(s_i) = \{s_j | s_j$ can be reached by finite number of *delegate*$(\cdot, \cdot, \cdot)$ actions from $s_i\}$

As an example, consider states s_4 and s_{21} from the of the example scenarios.These states are similar under σ because, as described in Table 3.4, these states have propositions representing commitments that differ only in their creditors. $\approx_\sigma$ is an equivalence relation.

Role-and-Commitment State-Similarity. From the point of view of a participant of a protocol, two states can be said to be similar if they involve the same commitments. Based on this intuition, we describe role-and-commitment state-similarity function α. A state s_i is similar to a state s_j under $\alpha_{A,P}$, where A is a set of roles and P is a set of propositions, if the commitments made by any role in A to any other role in A, and the propositions in P that hold at s_i, also hold at s_j. If, for example, A represents all the roles in a protocol and P represents all the proposition used by that protocol, then $\alpha_{A,P}$ can be used as a similarity function to detect cases where the protocol has been merged with other protocols.

3.2.2 Subsumption and Similarity of Runs

Comparisons among protocols are based on a notion of *subsumption* of runs. $[\![f]\!\rangle$ denotes subsumption operator over runs. The operator $[\![f]\!\rangle$ is an order-preserving mapping from one run to another, and depends on the function f.

Definition 5. *A run τ_j subsumes a run τ_i under function f if and only if, for every state s_i that occurs in τ_i, there occurs a state s_j in τ_j that is similar under f, and s_j has the same temporal order relative to other states in τ_j as s_i does with states in τ_i.*

$$\begin{aligned} \tau_j [\![f]\!\rangle \tau_i \quad \Longleftrightarrow \quad & (\forall s_i, s_i' \in \tau_i : (\exists s_j, s_j' \in \tau_j \\ & \text{and } s_i \approx_f s_j \text{ and } s_i' \approx_f s_j' \\ & \text{and } (s_i \prec_{\tau_i} s_i' \Rightarrow s_j \prec_{\tau_j} s_j'))) \end{aligned} \tag{3.2}$$

That is, longer runs subsume shorter ones, provided they have similar states occurring in the same order. Before we describe properties of run subsumption, we define *run-similarity*.

Definition 6. *A run τ_i is similar to a run τ_j under a well-formed state-similarity function f if and only if the two runs are of equal length and every kth state of τ_i is similar under f to the kth state in τ_j. In notation, $[\tau_i]^k \approx_f [\tau_j]^k, 0 \leq k < |\tau_i|$ and $|\tau_i| = |\tau_j|$.*

Lemma 1. *If a run τ_j subsumes a run τ_i under f, then, for all subruns β_i of τ_i, there exists a subrun β_j of τ_j such that $\beta_i \approx_f \beta_j$. That is, $\tau_j [\![f]\!\rangle \tau_i \Rightarrow (\forall \beta_i$ subrun of $\tau_i, \exists \beta_j$ subrun of $\tau_j : \beta_i \approx_f \beta_j)$.*

Proof. This lemma holds trivially for subruns of lengths 0 or 1.

We prove this lemma by induction on the length of subruns.

Base Case: We note that by the definition of run subsumption, there must be a pair of states in both runs for which the above property holds, that is, $(\forall \beta_i$ subrun of $\tau_i : |\beta_i| = 2 \Rightarrow (\exists \beta_j$ subrun of τ_j and $\beta_i \approx_f \beta_j))$.

Induction Hypothesis: Assume this property holds for subruns of a particular length $k > 2$.

Induction Step: Consider $|\beta_i'| = k + 1$, where $[\beta_i']^k = s_i'$. The first k states of β_i' form a subrun of length k. Therefore, there exists a subrun β_j of τ_j such that $\beta_i \approx_f \beta_j$ and β_j is the earliest such subrun in τ_j. Consider the k pairwise temporal precedence relationships (in τ_i) between the first k states of β_i' and s_i'. By the definition of run subsumption, we know that since $\tau_j [\![f]\!\rangle \tau_i$, $\exists s_j' \in \tau_j$

such that $[\beta_j]^\top \prec_{\tau_j} s'_j$ and $s'_j \approx_f s'_i$. Let s'_j be the earliest such state in τ_j, so that $\beta'_j = \beta_j \circ s'_j$. Thus, the inductive hypothesis holds for subruns of length $k+1$.

Conclusion: By induction, the lemma holds. □

Theorem 1. *Under any well-formed state similarity function, run subsumption is reflexive, transitive, and antisymmetric up to state similarity.*

Proof. Reflexivity: The reflexivity of run subsumption follows from the reflexivity of state-similarity functions. Since every state is similar to itself, every run subsumes itself.

Transitivity: The transitivity of run subsumption follow from the definition. If $\tau_i [\![f]\!\rangle \tau_j$, then τ_i contains states similar to those in τ_j, and in the same temporal order relative to each other. If $\tau_j [\![f]\!\rangle \tau_k$, then τ_j contains states similar to those in τ_k, and in the same temporal order relative to each other. Therefore, for every state in τ_k, there is a corresponding similar state in τ_j, which, in turn, has a corresponding similar state in τ_i. Since state similarity is transitive, the transitivity of run subsumption holds.

Antisymmetry: The antisymmetry of run subsumption means that if two runs subsume each other under some well-formed state similarity function f, then those runs are similar under f. For proof, consider Lemma 1. By Lemma 1 we know that if $\tau_j [\![f]\!\rangle \tau_i$, then for each subrun of τ_i, there is a similar subrun of τ_j. Let the entire run τ_i be treated as a subrun of itself. Then, there is a similar subrun in τ_j, i.e., $|\tau_j| \geq |\tau_i|$. In the same way, from $\tau_i [\![f]\!\rangle \tau_j$, we infer that $|\tau_i| \geq |\tau_j|$. Hence, $|\tau_i| = |\tau_j|$. Since the runs are of equal length and all the subruns of one have similar subruns in the other, $\tau_i \approx_f \tau_j$. □

Let us consider an example to better explain the above concepts. Let τ_1, τ_2, and τ_3 be the runs shown in Figures 2.2, 2.3, and 2.4, respectively. We then have $\tau_2 [\![\iota]\!\rangle \tau_1$. Also, $\tau_3 [\![\iota]\!\rangle \tau_1$. However, τ_1 subsumes neither τ_2 nor τ_3, because τ_3 has a state s_{21}, whose label does not match any state label in τ_1, and τ_2 has states s_{18} and s_{19}, whose labels do not match any state label in τ_1. Run τ_3 does not subsume τ_2 because of s_{21}, and τ_2 does not subsume τ_3 because of s_{18} and s_{19}.

Next, we consider run subsumption under the creditor state-similarity function σ. Here, $\tau_2 [\![\sigma]\!\rangle \tau_1$. Also, since $s_{21} \in \sigma(s_4)$, $\tau_1 [\![\sigma]\!\rangle \tau_2$. Run τ_3 subsumes both τ_1 and τ_2 for the same reason, but neither τ_1 nor τ_2 subsumes τ_3, since neither of them have states that are σ-similar to s_{18} and s_{19}.

Figure 3.1 shows the subsumption relation between the runs τ_1, τ_2, and τ_3 under the identity function ι and Figure 3.2, under σ.

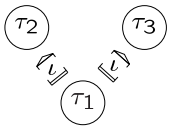


Figure 3.1: Subsumption of runs τ_1, τ_2, and τ_3 (shown in Figures 2.2, 2.3, and 2.4 respectively) under the identity state-similarity function ι.

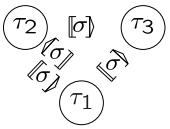


Figure 3.2: Subsumption of runs τ_1, τ_2, and τ_3 (shown in Figures 2.2, 2.3, and 2.4 respectively) under the creditor state-similarity function σ.

- State-similarity compares propositions that label states.
- A run subsumes another if its states are similar to all of the latter's states, and in the same relative temporal order.
- A run is similar to another if both are of the same length and their k^{th} states are similar.
- Run subsumption is reflexive, transitive, and antisymmetric.

3.2.3 Subsumption and Similarity of Protocols

Closure of Protocol Runs. In line with our intuitions that generic protocols allow many variations, we desscribe the semantics of a protocol to have a set of runs that is closed under run subsumption. That is, if any run occurs in the set of runs of a protocol, all the runs that subsume it (under a well-formed state-similarity function) are also in that set. Given a well-formed state-similarity function f, a protocol P contains all runs that subsume any run in $[P]$. We define this new set of runs as the set of runs allowed by the protocol

Observation 1. *From the definition of closure of the set of allowed runs of a protocol $[\![P]\!]$ under a well-formed state-similarity function f, we have that $[\![P]\!] = \{\tau | \forall \tau' \in [P] : \tau [\![f]\!\rangle \tau'\}$.*

We refer to $[\![P]\!]$ also as the *closure* of the protocol P under a well-formed state-similarity function.

Operationally, the runs allowed by a protocol completely characterize that protocol by naturally illustrating the key tradeoff in protocol design, that of *flexibility versus compliance:* A protocol that allows many runs is better than one that allows a few runs, since the many-run protocol affords more choice and flexibility in protocol execution to the participants. However, checking for compliance can potentially be more demanding when protocols have more runs that do not subsume a common run. The definition of the subsumption of protocols reflects these intuitions.

Definition 7. *A protocol P_j subsumes a protocol P_i under a state-similarity function f if and only if, every run in $[\![P_i]\!]$ subsumes, under f, a run in $[\![P_j]\!]$.*

$$P_j [\![f]\!\rangle P_i \iff \forall \tau_i \in [\![P_i]\!] \, \exists \tau_j \in [\![P_j]\!] : \tau_i [\![f]\!\rangle \tau_j \tag{3.3}$$

If P_j is a protocol that has short runs (and consequently all runs that subsume them) and P_i is a protocol that has long runs only, then P_j subsumes P_i. Since long runs subsume shorter ones, protocols with long runs only are subsumed by protocols with short runs as well.

Before we describe the properties of protocol subsumption, we define protocol similarity as follows:

Definition 8. *Two protocols are similar under a state similarity function f if and only if for every run in one protocol, there exists at least one similar (under f) run in the other protocol, and vice versa.*

$$\begin{aligned} P_j \approx_f P_i \quad \Leftrightarrow \quad & \forall \tau_i \in [\![P_i]\!], \exists \tau_j \in [\![P_j]\!] : \tau_i \approx_f \tau_j \\ & \text{and } \forall \tau_j \in [\![P_j]\!], \exists \tau_i \in [\![P_i]\!] : \tau_j \approx_f \tau_i \end{aligned} \tag{3.4}$$

Theorem 2. *For any well-formed state-similarity function f, the protocol subsumption relation $[\![f]\!\rangle$ is a partial order, i.e., reflexive, transitive, and antisymmetric (up to state similarity).*

Proof. *Reflexivity:* Follows from the definition of protocol subsumption.

Transitivity: Transitivity also follows from the definition.

Antisymmetry: Because of the property of closure of the set of runs of a protocol under run subsumption and the definition of protocol subsumption, we know that if $P_i [\![f]\!\rangle P_j$, then $\forall \tau_j \in [\![P_j]\!] : \exists \tau_i \in [\![P_i]\!] : \tau_j \approx_f \tau_i$. Conversely, if $P_j [\![f]\!\rangle P_i$, then $\forall \tau_i \in [\![P_i]\!] : \exists \tau_j \in [\![P_j]\!] : \tau_i \approx_f \tau_j$. Together, these are equivalent to $P_j \approx_f P_i$. □

3.2.4 The Protocol Algebra

We now introduce our protocol algebra as consisting of two operators (*merge* and *choice*), their identity elements ($\mathbb{1}$ and $\mathbb{0}$, respectively), and an ordering relationship (protocol subsumption, as defined above).

Merge. The merge operator, denoted by $\otimes_f$, splices two protocols (under a state-similarity function f) to produce a new protocol. A merge of two protocols involves interleaving their runs.

Definition 9. *The merge of two protocols P and Q under a well-formed state-similarity function f creates a protocol whose closure consists exactly of runs that subsume some run from $[\![P]\!]$ and some run from $[\![Q]\!]$.*

$$[\![P \otimes_f Q]\!] \quad = \quad \{r | \exists r_p \in [\![P]\!], \exists r_q \in [\![Q]\!] : r [\![f]\!\rangle r_p \text{ and } r [\![f]\!\rangle r_q\} \tag{3.5}$$

Choice. The choice operator, denoted by $\oplus_f$, combines two protocols P and Q (under a state similarity function f) to create a protocol R, whose set of runs $[\![R]\!]$ contains exactly those runs that exist either in the set $[\![P]\!]$ or in the set $[\![Q]\!]$.

Definition 10. *The choice of two protocols P and Q under a well-formed state-similarity function f creates a protocol whose closure consists exactly of runs that subsume either a run from $[\![P]\!]$ or a run from $[\![Q]\!]$.*

$$[\![P \oplus_f Q]\!] \quad = \quad \{r | \exists r_p \in [\![P]\!], \exists r_q \in [\![Q]\!] : r [\![f]\!\rangle r_p \text{ or } r [\![f]\!\rangle r_q\} \tag{3.6}$$

Choosing a run from $[\![P \oplus_f Q]\!]$ is equivalent to choosing a run from either $[\![P]\!]$ or a run from $[\![Q]\!]$.

Constants. The properties of the merge operator lead us to define two protocols, $\mathbb{O}$ and $\mathbb{1}$. The $\mathbb{O}$ protocol is an "impossible" protocol, which does not have any runs. $[\![\mathbb{O}]\!] = \{\}$. The $\mathbb{1}$ protocol is a "trivial" protocol which contains the zero-length run in its semantics, and consequently contains all possible runs. $[\![\mathbb{1}]\!] = \{\tau : \tau [\![f]\!\rangle \tau_\phi\}$, where $\tau_\phi = \langle\rangle$, and f is any well-formed state-similarity function. The $\mathbb{O}$ is the bottom element and the $\mathbb{1}$, the top element of a hierarchy of protocols based on subsumption. That is, all protocols are subsumed by the $\mathbb{1}$ protocol and all protocols subsume the $\mathbb{O}$ protocol.

Formal Results

We briefly present some formal results, which simplify reasoning about protocols using our algebra. First, we present a formulation for the operators in terms of protocol subsumption.

Since Definitions 9 and 10 state the properties of the operators only in terms of the runs of the protocols, this new formulation simplifies the understanding of the behavior of the merge and choice operators, and likens them to set intersection and union, respectively.

Theorem 3. *Given the run-subsumption relation $[f\rangle$ under a well-formed state-similarity function f, the merge $\otimes_f$ of two protocols P and Q under f is the greatest lower bound of P and Q under the protocol subsumption relation $[f\rangle$.*

Proof. Let $R = P \otimes_f Q$. That is,

$$[\![R]\!] = \{r | \exists p \in [\![P]\!], \exists q \in [\![Q]\!] : r[f\rangle p \text{ and } r[f\rangle q\} \tag{3.7}$$

From Observation 1, it is easy to see that $P[\![f\rangle R$ and $Q[\![f\rangle R$. That is, R is a lower bound of P and Q.

Consider some protocol R' such that $P[f\rangle R'$ and $Q[f\rangle R'$. By definition of protocol subsumption,

$$\forall r'_p \in [\![R']\!], \exists p \in [\![P]\!] : r'_p[f\rangle p \tag{3.8}$$

$$\forall r'_q \in [\![R']\!], \exists q \in [\![Q]\!] : r'_q[f\rangle q \tag{3.9}$$

From Equation 3.7 we know that all runs that subsume a run each from $[\![P]\!]$ and $[\![Q]\!]$ are in $[\![R]\!]$. From Equations 3.8 and 3.9, we know that every run in $[\![R']\!]$ is just such a run. Hence, $\forall r' \in [\![R']\!] : r' \in [\![R]\!]$. From the closure property of protocol run-sets, we further infer that $\forall r' \in [\![R']\!], \exists r \in [\![R]\!] : r[f\rangle r'$, which means that $R[f\rangle R'$. Since any R' is subsumed by $P \otimes_f Q$ under f, this means $\otimes_f$ gives the greatest lower bound of P and Q under f. □

Theorem 4. *Given the run-subsumption relation $[f\rangle$under a well-formed state-similarity function f, the choice $\oplus_f$ of two protocols P and Q under f is the least upper bound of P and Q under the protocol subsumption relation $[f\rangle$.*

Proof. Let $R = P \oplus_f Q$. This implies that

$$[\![R]\!] = \{r | \exists p \in [\![P]\!] : r[f\rangle p \text{ or } \exists q \in [\![Q]\!] : r[f\rangle q\} \tag{3.10}$$

Consider some protocol R' such that $R'[f\rangle P$ and $R'[f\rangle Q$. By definition of protocol subsumption,

$$\forall p \in [\![P]\!], \exists r'_p \in [\![R']\!] : p[f\rangle r'_p \tag{3.11}$$

$$\forall q \in [\![Q]\!], \exists r'_q \in [\![R']\!] : q[f\rangle r'_q \tag{3.12}$$

Let $r \in [\![R]\!]$. From equation 3.10, without loss of generality, we can assume that $\exists p \in [\![P]\!] : r[f\rangle p$. From this, equation 3.11, and the transitivity of run subsumption, we infer that $\forall r \in [\![R]\!], \exists r'_p \in [\![R']\!] : r[f\rangle r'_p$, which is the definition of protocol subsumption under f. Hence $R'[f\rangle R$. Since any R' subsumes $P \oplus_f Q$ under f, $\oplus_f$ gives the least upper bound of P and Q under protocol subsumption under f. □

Essentially, the merge is analogous to the intersection of the sets of runs of the protocols being merged, where run similarity is used to determine equality of elements of the sets. Formally,

$$[\![P \otimes_f Q]\!] \quad = \quad [\![P]\!] \cap_f [\![Q]\!]$$

Where $\cap_f$ is the *run intersection* operator, defined over sets of runs A and B as

$$A \cap_f B \quad = \quad \{r | \exists p \in A : r \approx_f p \text{ and } \exists q \in B : r \approx_f q\}$$

We could alternatively define $\cap_f$ as the greatest lower bound of P and Q under f.

Lemma 2. *Given protocols P and Q and a well-formed state-similarity function f, $r \in [\![P \otimes_f Q]\!] \Rightarrow r \in [\![P]\!]$ and $r \in [\![Q]\!]$*

Proof. Consider $r \in [\![P \otimes_f Q]\!]$. From Definition 9, we know that $r[\![f\rangle r_p$ and $r[\![f\rangle r_q$ for some $r_p \in [\![P]\!]$ and $r_q \in [\![Q]\!]$. From this and Observation 1 (the closure of sets of protocol runs), we infer that $r \in [\![P]\!]$ and $r \in [\![Q]\!]$. □

Lemma 3. *Given protocols P and Q and a well-formed state-similarity function f, $r \in [\![P]\!]$ and $r \in [\![Q]\!] \Rightarrow r \in [\![P \otimes_f Q]\!]$*

Proof. Consider $r \in [\![P]\!]$. From Observation 1 and the reflexivity of run subsumption under any well-formed state-similarity function, we can infer $\exists r_p \in [\![P]\!]$ such that $r[\![f\rangle r_p$. Similarly, from $r \in [\![Q]\!]$, we infer $\exists r_q \in [\![Q]\!]$ such that $r[\![f\rangle r_q$. From Definition 9, we conclude that $r \in [\![P \otimes_f Q]\!]$. □

Choice is analogous to the union of the sets of the runs, using run similarity to compare elements of the sets. Formally,

$$[\![P \oplus_f Q]\!] \quad = \quad [\![P]\!] \cup_f [\![Q]\!]$$

Where $\cup_f$ is the *run union* operator, defined over sets of runs A and B as

$$A \cup_f B \quad = \quad \{r | \exists p \in A : r \approx_f p \text{ or } \exists q \in B : r \approx_f q\}$$

We could alternatively define $\cup_f$ as the least upper bound of P and Q under f.

Lemma 4. *Given protocols P and Q and a well-formed state-similarity function f, $r \in [\![P \oplus_f Q]\!] \Rightarrow r \in [\![P]\!]$ or $r \in [\![Q]\!]$*

Proof. Consider $r \in [\![P \oplus_f Q]\!]$. From Definition 10, we know that $r[\![f]\!\rangle r_p$ or $r[\![f]\!\rangle r_q$ for some $r_p \in [\![P]\!]$ and $r_q \in [\![Q]\!]$. From this and Observation 1 (the closure of sets of protocol runs), we infer that $r \in [\![P]\!]$ or $r \in [\![Q]\!]$. □

Lemma 5. *Given protocols P and Q and a well-formed state-similarity function f, $r \in [\![P]\!]$ or $r \in [\![Q]\!] \Rightarrow r \in [\![P \oplus_f Q]\!]$*

Proof. If $r \in [\![P]\!]$, then, from Observation 1 and the reflexivity of run subsumption under any well-formed state-similarity function, we can infer $\exists r_p \in [\![P]\!]$ such that $r[\![f]\!\rangle r_p$. Similarly, if $\exists r_q \in [\![Q]\!]$, then $r[\![f]\!\rangle r_q$. From Definition 10, we conclude that $r \in [\![P \oplus_f Q]\!]$. □

The following results are then obvious

1. The merge of a protocol with itself yields the same protocol (idempotence).
   $(P \otimes_f P) = P$

2. The merge operator is commutative and associative.
   $P \otimes_f Q = Q \otimes_f P$
   $P \otimes_f (Q \otimes_f R) = (P \otimes_f Q) \otimes_f R$

3. Merge distributes over choice.
   $P \otimes_f (Q \oplus_f M) = (P \otimes_f Q) \oplus_f (P \otimes_f M)$

   Proof. From Lemmas 2, 3, 4, 5, and the distributivity of "and" over "or",

$$
\begin{aligned}
r \in [\![P \otimes_f (Q \oplus_f M)]\!] \quad &\Rightarrow \quad r \in [\![P]\!] \text{ and } r \in [\![Q \oplus_f M]\!] \\
&\qquad \text{by Lemma 2} \\
&\Rightarrow \quad r \in [\![P]\!] \text{ and } (r \in [\![Q]\!] \text{ or } r \in [\![M]\!]) \\
&\qquad \text{by Lemma 4} \\
&\Rightarrow \quad (r \in [\![P]\!] \text{ or } r \in [\![Q]\!]) \\
&\qquad \text{and } (r \in [\![P]\!] \text{ or } r \in [\![M]\!]) \\
&\qquad \text{by distributivity of "and" over "or"} \\
r \in [\![P \otimes_f (Q \oplus_f M)]\!] \quad &\Rightarrow \quad r \in [\![(P \otimes_f Q) \oplus_f (P \otimes_f M)]\!] \qquad (3.13) \\
&\qquad \text{by Lemmas 3 and 5}
\end{aligned}
$$

Similarly, we can prove

$$r \in [\![(P \otimes_f Q) \oplus_f (P \otimes_f M)]\!] \Rightarrow r \in [\![P \otimes_f (Q \oplus_f M)]\!] \tag{3.14}$$

From Equations 3.13, 3.14, and the definition of protocol-similarity (Definition 8), we conclude that merge distributes over choice. □

4. The merge of any protocol with $\mathbb{1}$ gives that protocol and the merge of any protocol with $\mathbb{O}$ gives $\mathbb{O}$. In this way, the $\mathbb{1}$ protocol is the identity element and the $\mathbb{O}$ is the nil element for merge.
   $P \otimes_f \mathbb{1} = P$
   $P \otimes_f \mathbb{O} = \mathbb{O}$

Choice also supports idempotence, commutativity, and associativity. The choice of a protocol with $\mathbb{1}$ yields $\mathbb{1}$ and the choice of a protocol with $\mathbb{O}$ yields that protocol itself.

1. Idempotence.
   $(P \oplus_f P) = P$
2. Commutativity.
   $(P \oplus_f Q) = (Q \oplus_f P)$
3. Associativity.
   $P \oplus_f (Q \oplus_f R) = (P \oplus_f Q) \oplus_f R$
4. Choice distributes over merge.
   $P \oplus_f (Q \otimes_f R) = (P \oplus_f Q) \otimes_f (P \oplus_f R)$
5. Choice with $\mathbb{1}$ and $\mathbb{O}$
   $P \oplus_f \mathbb{1} = \mathbb{1}$
   $P \oplus_f \mathbb{O} = P$

3.3 Applying the Algebra

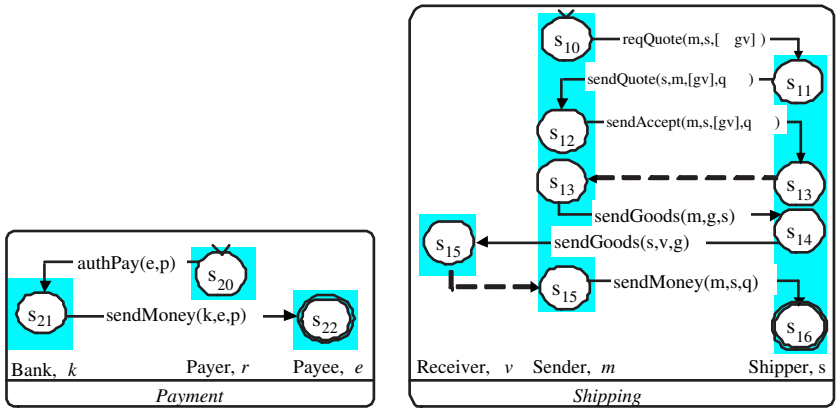
The previous chapter introduced our algebra of protocols. We suggested that this algebra can be used to compose hybrid protocols using existing ones by merging them. This chapter

discusses a methodology for bottom-up design of hybrid protocols using existing ones. The methodology has not yet been fully developed and is part of ongoing research. The basic principle of this methodology is that designers can use the merge operator to splice protocols together to create new protocols whose behavior will satisfy both the merged protocols. We envision that tools for protocol composition will be developed based on our algebra that will enable quick and convenient development of new protocols to suit ever-changing business needs. In this regard, state-similarity functions will be used to guide the composition so as to yield hybrid protocols whose properties are well understood.

3.3.1 Designing Protocols via Aggregation

Protocols can be developed by aggregating smaller protocols. Consider the purchase example. At a high level of abstraction, this example is implemented by a protocol that has three stages: an initial *negotiation* stage, followed by a *shipment* stage, and finally a *payment* stage. A stage is a logical set of interactions within a protocol. We do not formally define stages of a protocol but use the concept only as an aid to understanding our intuitions about protocol composition using our algebra. In the purchase scenario shown in Figure 2.2, states s_0, s_1, s_2, and s_3 belong to the negotiation stage, states s_3 and s_4 belong to the shipment stage, and states s_4 and s_5 belong to the payment stage. Even at this level of abstraction, we see that a protocol can be flexible in the runs it allows. That is, a protocol that allows the payment stage to precede the shipping stage essentially achieves the same goal, that of exchanging goods for money. For simplicity, let us adhere to the negotiation, shipment, payment order. This three-stage purchase protocol can be refined, for example, by substituting, or *splicing into* it, any one of the numerous shipping protocols available. One can ship via regular mail or use return-receipt mail. Whatever the shipping protocol used, the splicing works because the purchase protocol is specified as an interface, and the shipping protocol adheres to the interface. The content of the states and the set of generated runs is the interface of a protocol. As an example, consider the shipping protocol in Figure 3.3(b). This protocol can be spliced into *Purchase* as shown in Figure 3.4. Similarly, the payment protocol shown in Figure 3.3(a) can be spliced into the purchase protocol.

An important observation is that when a protocol p_2 is spliced into a protocol p_1, p_2 itself might be spliced into by parts of p_1. As a particular case, consider the shipping protocol. This protocol splices the purchase protocol by taking a circuitous run to achieve the delivery of the goods to the customer. However, the shipper is not paid by the bookstore until the customer has paid the



(a) Payment protocol (b) Shipping protocol

Figure 3.3: A payment and a shipping protocol

bookstore. Therefore, the shipping protocol has essentially been spliced in between states s_{14} and s_{15}. From the point of view of the shipping protocol, the bookstore waits at state s_{15}. From the point of view of the purchase protocol, the customer receives the goods, and performs the payment via the bank.

3.3.2 Refining Protocols via Merge

To see how protocol refinement via splicing is achieved using our protocol algebra, consider the state-similarity function $\alpha_{A,P}$, where A denotes a set of agents, and P a set of propositions. A state s_i is similar to a state s_j under $\alpha_{A,P}$ if the commitments made by any agent in A to any other agent in A, and the propositions in P that hold at s_i, hold at s_j. That is, $\alpha_{A,P}(s_i) = \{s_j | \forall e, e' \in A :$ all commitments made by e to e', by e' to e, and all propositions in P that hold at s_i also hold at $s_j\}$.

Under $\alpha_{A,P}$, where A denotes the set containing the participants of shipping, i.e., $\{b, x\}$, and P denotes the set of all propositions that are used in shipping, we see that the shipping run shown in Figure 3.3(b) is subsumed by the refined purchase run shown in Figure 3.4. Specifically, with $A = \{Sender, Receiver, Shipper\}$ and $P = \{$The propositions used in states s_{10} through$s_{16}\}$ the states $s_3, s_{11}, s_{12}, s_{13}, s_{14}, s_5$, and s_{16} of refined purchase are similar under $\alpha_{A,P}$ to the states $s_{10}, s_{11}, s_{12}, s_{13}, s_{14}, s_{15}$, and s_{16} of shipping, respectively. Similarly, the states s_4 and s_{21} of the refined purchase are similar to states s_{20} and s_{21} of payment under $\alpha_{A,P}$, when A denotes

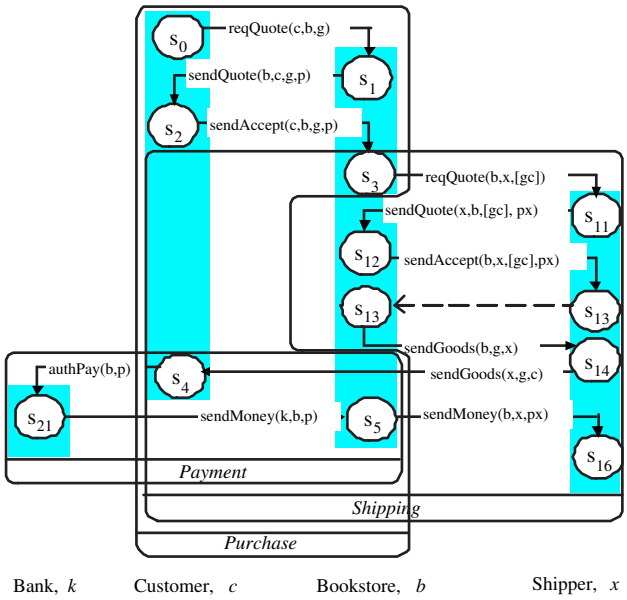


Figure 3.4: A refinement of the purchase protocol by splicing in the shipping protocol shown in Figure 3.3(b) and the payment protocol shown in Figure 3.3(a) into the purchase protocol shown in Figure 2.2.

$\{Payer, Payee, Bank\}$ and P denotes the set of all propositions used in payment. Consequently, the refined purchase protocol subsumes payment.

The refined purchase example given here is achieved by applying the merge operator on the purchase, shipping, and the payment protocols. Since merge is associative, the order in which these protocols are merged is unimportant. What is important is how the merge generates runs in which the shipping protocol and the payment protocol are spliced in at the correct states. A typical usage scenario would be for a protocol designer to run a tool to perform the merge of the shipping protocol with the purchase protocol. Given that the propositions used by both protocols have the same meaning, the tool would generate all possible interleavings of the states of both protocols such that each run of the resulting protocol subsumes a complete run of purchase and a complete run of the shipping protocol. The set of possible interleavings will be restricted by the states in the protocols that are similar. In Figure 3.4, only one run of the resulting protocol is shown, which is a run where the shipping protocol is begun after the bookstore has received an acceptance of the quote it had sent to the customer. Note that this is not the only possible run. The shipping protocol can be initiated right after the customer asks the bookstore for a quote.

Based on the above observations, we note that the runs generated by a protocol that is the

result of merging two protocols are constrained by two main factors

1. The description of the state of the protocol as given by the propositions.

   This pertains to the detail captured by the propositions labeling the states of the merged protocols . For example, the shipping protocol might require that the customer has committed (at least conditionally) to buying an item before the shipping protocol is initiated. This might be done, say, because protocol initiations prove to be expensive for the shipper in cases where the shipping is not required later. Such a shipping protocol has to begin at state s_3 as shown in Figure 3.4, and cannot begin at any state earlier than s_3. The propositions labeling the states of the shipping protocol encode this requirement. The start state of the shipping protocol shipping protocol requires that the role to which the goods are to be delivered be committed to purchasing (or otherwise being the recipient of) the goods.

2. The similarity function applied to the merge.

   This pertains to how different similarity functions can yield different sets of runs in the merged protocol, as explained in Section 3.2.1.

More extensive examples of merging protocols to create hybrid ones have been studied and demonstrated as part of the OWL-P (Web Ontology Language for Protocols and Processes) project. This effort has developed an operational framework for specifying and enacting commitment protocols among agents. OWL-P uses a rule-based specification for protocols using OWL. It is explained in greater detail in Section 6. Below, we summarize the content of succeeding chapters, to provide an overview of the applications and scope of our results.

3.3.3 Exception Modeling and Handling

Exceptions are abnormal conditions that occur during the enactment of a protocol. They are an important aspect of real-life processes, since a process does not always encode all possibilities. We have therefore developed methodologies for modeling and handling exceptions for commitment protocols. In addition to the challenges that face traditional distributed systems, we face the added tension between the flexibility of a commitment protocols and the need to constrain behavior to avoid exceptions. We have taken a two-pronged approach

- Modeling foreseeable exceptions in a hierarchy of exceptions so that a notion of *severity* can be deduced, enabling a realistic model where one can distinguish between which runs are normal and which are not.

- Handling unforeseen exceptions when they occur at runtime by splicing exception handlers from a library of handlers into the protocol.

Chapter 4 describes these concepts in greater detail.

3.3.4 Designing Interaction Among Components

To complete our treatment of commitment protocols, we examine the first step in the life of a protocol–its creation. We develop a principled methodology for designing protocols from requirements. To this end, we develop upon two independent strands of research.

- We augment an established agent-oriented software engineering methodology, Tropos, by introducing commitment protocols into every stage of software development. In doing so, Tropos benefits from the modeling of interactions at design time and our approach finds a context in which commitment protocols can be developed.

- We develop upon the theory of discourses among agents and describe how commitment operation patterns and reusable commitment protocols can be identified from agent discourses, which can be seen as use-cases.

Chapter 5 describes these concepts in greater detail.

- Chapter 4 describes our methodology for modeling (foreseen) and handling (unforeseen) exceptions in commitment protocols.
- Chapter 5 describes how we have incorporated software engineering and agent-discourse principles to develop a methodology for designing protocols.

Chapter 4

Modeling and Handling Exceptions in Commitment Protocols

A protocol is an agreement between participants to engage in an interaction in a predetermined way. Protocols lend determinism to an interaction, so that agents can plan their actions. Sometimes, abnormal or unforeseen conditions occur during the enactment of a protocol. Such conditions are termed *exceptions*. Although exceptions have a negative connotation in the literature such as in the study of programming languages, we take the view that exceptions can be desirable in certain circumstances. In a business interaction, one partner might wish to deviate from the protocol being enacted to avail of an opportunity that would not necessarily subvert the protocol they are following. For example, a merchant in a purchase interaction might wish to proactively send quotes to potential customers, if the customers do not object to it. Regardless of whether an exception is desirable or not, it requires special processing so that participants of the protocol can benefit from opportunities that arise and minimize the losses they could incur due to exceptions. Exceptions are an essential feature of real-life processes. Many businesses, for example, entertain exceptional requests from customers in the interest of better customer service. Whereas agent interaction is not immune to exceptions caused due to lost or garbled messages between agents, we concentrate on high-level exceptions, i.e., exceptions that arise even in the face of reliable messaging guarantees.

- Exceptions are abnormal conditions.
- Exceptions can sometimes be desirable, i.e., opportunities.

4.1 Motivation

As described before, open business processes such as those conducted among independent business partners (or interactions between autonomous agents in general) face challenges quite different from those faced by processes in closed systems.

1. Open systems require flexible processes that can adapt as the environment evolves.

2. Flexible processes require complex models to capture interactions that are rich in semantics.

Models for flexible processes prove notoriously intractable for human process designers. None but the most trivial of business processes can be designed with every possibility explicitly specified. Most times, processes automation yields diminishing returns as increasing effort is expended to encode cases of deviations from the "normal" execution of the process. Exception modeling is therefore expensive. Further, verifying the compliance of participants with such models is difficult since what is considered a valid detour from the normal enactment of a process in one situation may be considered noncompliant behavior in another. Exceptions also lead to poor user satisfaction if processes interrupt human operators to handle frequently-occurring exceptions.

Decentralization of process enactment frameworks also poses significant challenges. Workflows, the most popular method of automating business processes, employ a *workflow engine* to enact flows. The engine synchronizes all activities involved in the workflow as desired. With such a framework, the engine can detect abnormal conditions easily, although such models become quite complicated (Casati, 1998). In contrast to such centralized modeling and enactment, interactions among businesses are not usually governed by a central engine. Inter-business coordination is increasingly enacted in an open, distributed environment such as the Internet, among autonomous partners who have differing views on what constitutes desirable or even normal behavior in a particular process. Simply put, the key features of open systems lead to the possibility of exceptions. Traditional ways of modeling processes become intolerably complex in the face of exceptions. Traditional ways to enact processes simply fail in the face of exceptions or throw the entire reasoning burden on humans.

There has been an increasing recognition of the importance of agent-based, semantically rich approaches for modelling processes in open systems. However, existing approaches prove to be limited in that they do not provide a principled means for modeling exceptions ahead of time or handling exceptions that have not been explicitly modeled ahead of time. The basic insight that agent-based process models agree upon is the use of a suitable rule-based mechanism and sometimes

of organizational structure, but there are key aspects of the rules and the organizations that are not considered by current approaches.

A strength of commitment protocols is that they enable agents to act flexibly, thereby enabling them to accommodate varying local policies and respond to exceptions. A consequent weakness is that commitment protocols thus fail to distinguish between possible executions that are normal and those that may be allowed but are not desirable. Whereas protocols regulate the externally observable behavior of business partners, distinguishing what is allowed from what is not, they do not make any finer distinctions about what is *normal* and what is not. What is required is a knowledge engineering stance toward commitment protocols.

- To accommodate agent autonomy, protocols must enable the participating agents to choose their actions and responses to different conditions as they see fit. For example, a purchase protocol should allow the possibilities that negotiations may fail, that the payment may be made via a third party, that an agent awaiting missing goods may send a reminder for them.

- To be realistic, the protocols themselves must be based on a study of different usage scenarios, wherein not all possible executions are considered equal. For example, as a practical matter, we would all recognize that it is more normal for a purchase protocol to lead to an exchange of goods and money than for a refund to be issued or for the goods to be repossessed by the merchant because of lack of payment.

In other words, protocols should be flexible enough to accommodate exceptions, but should still be described in such a manner that the exceptions are distinguished from the normal executions. Moreover, there must be a hierarchy of exceptions, some being more acute than others.

- Traditional process models
  - cannot be changed during execution.
  - become unwieldy when exceptions are numerous.
  - distinguish what is allowed from what is not; not what is *normal* and what is not.
- Realistic models require notions of
  - preferences among allowed executions.
  - how *severe* an exception is.

4.2 Challenges

Exception handling is challenging for the following reasons.

- *Exception handling requires domain knowledge.* The risks taken by the participants in a protocol are not the same across domains. Whereas exceptions may cause some participants to suffer a loss, the magnitude of the loss suffered by participants depends on the importance of the protocol to the participants in that domain. For example, delayed payment might be acceptable to a merchant in a purchase protocol, whereas delayed shipment of life-saving drugs might not be acceptable to customers who have ordered the drug. Therefore, exceptions cannot be handled in the same manner across different domains and situations.

- *Exceptions must be foreseen to be handled.* Eder and Liebhart (1995) classify exceptions that can occur during the enactment of workflows into four categories.

    1. *Basic failures*, which are system level failures handled by the underlying software systems such as networks and databases.
    2. *Application failures*, which are handled by trying to execute the failed application again or compensating for the failure by executing a compensating application.
    3. *Expected exceptions*, which are deviations from the normal flow that occur infrequently but often enough to be incorporated into the workflow model. Expected failures are handled by invoking predetermined exception handlers.
    4. *Unexpected exceptions*, which are rare and require a change of the workflow model itself. Such exceptions cannot be handled by systems that cannot change their flow schema at runtime.

- *Handling exceptions requires changing the process model.* Traditional process frameworks such as workflows cannot handle unexpected exceptions since the process model cannot be changed at runtime. That is, an abstraction for reasoning about pragmatic exceptions is required. Pragmatic exceptions are those that require commonsense type of reasoning to determine whether to retry a failed action or try a different action to achieve the same results. For instance, when a payment is not received by the deadline that was set for it, the payee can choose to wait if it wishes to give the payer some more time. If the payment is critical, however, the payee might want the payer to arrange for an alternative means of payment.

- *Not all exceptions are equally bad.* Traditionally, there have been coarse-grained classification schemes to assign severity to exceptions. For example, an end-of-file exception is less severe than a divide-by-zero exception in terms of how a processor handles them. For applicability to business processes, however, what is required is a way to specify a finer grained classification of exceptions. In addition, reusable protocols add another dimension to the problem by requiring a modification of an exception-classification when the protocol is deployed in different environments.

- Exception handling requires expert knowledge about the process.
- Unforeseen exceptions cannot be handled by process models that cannot change during execution.
- A hierarchy of exceptions based on their severity needs to be specified in a suitable language with a clear semantics.

4.3 Contributions

Current implementations of business processes, based on traditional workflows or on web service compositions, do not address the above challenges well. Exception handling in these approaches involves no more than the invocation of an exception handler. Further, current workflow technologies cannot handle unexpected exceptions at runtime, since workflow models are immutable during execution. We propose a novel approach to exception handling in which commitments provide meaning to the interactions, as a result of which cross-domain exception handling patterns can be developed. The study and development of interaction patterns to handle changes made to commitments is not new. Xing *et al.* (2001) have developed a set of interaction patterns that agents can use to handle common exceptions in workflows such as revisions, inconsistencies, and incompleteness of results generated by some component of the workflow. Wan and Singh (2003) have also studied how changing commitments can be handled in agent interaction. Our approach explains how exceptions affect a protocol as a whole, and develops a methodology to handle exceptions. We base our method for andling unexpected exceptions on our framework for protocol composition, which enables changes to the underlying model of a protocol at runtime.

Our methodology can be used for the following

Modeling expected exceptions. In traditional process models, exceptions are modeled in an ad hoc manner, when the designer demarcates blocks of the process and assigns exception handlers

to those blocks. In our approach, exceptions can be defined separately from the specification of the process or the protocol. As a result, different exception conditions can be assigned to the same protocol based on the context of the protocol execution. This is similar in spirit to aspect-oriented software. Preferences are modeled in an analogous fashion.

Selecting protocols. When multiple protocols are available for an agent to realize a certain interaction, that agent can negotiate with the other parties involved in the protocol and, based on its preferences, find out if an execution of that protocol with those participants would be acceptable to it. For example, a customer agent that does not wish to enact a hotel booking protocol in which the hotel can cancel the room and award a refund can choose which hotel to interact with if its preferences are made clear before enactment and if the hotel and the customer negotiate the protocol to enact based on their preferences.

Handling unexpected exceptions. Our framework for modeling exceptions can be applied generically across domains using commitment protocols. Further, the protocol algebra we have developed can be used to handle unforeseen exceptions by splicing in exception handlers, thus changing the protocol model at runtime.

- The semantically rich abstraction based on commitments enable effective exception modeling and handling while allowing flexibility in execution.

4.4 Approach

We propose an approach to exception handling based on commitments that exist between the participants. Our approach has the following features:

- For expected exceptions, we define a preference structure over runs, i.e., a notion that some runs are more desirable than others, which can be used to model expected exceptions.
- For unexpected exceptions, we introduce the use of protocol splicing using the merge operator to handle unexpected exceptions in business processes. Essentially, exception handlers are formalized as sets of runs, formally on par with protocols, and these handlers are spliced into protocols.

4.4.1 Preferences Specification and Resolution

Preferences are denotationally characterized using sets of runs (i.e., computations), and are operationalized via translation into executable rulesets. With the above background, our approach proceeds as follows to enable augmenting protocol specifications with preferences.

- We augment an existing protocol specification language with an ability to specify preferences among different executions of protocols. For simplicity of specification, we use a simple linear temporal logic based language to specify expressions over protocol events to capture executions that are of interest.
- We show how preferences based on such expressions can be operationalized into executable rulesets for incorporation into OWL-P, which is a language and a framework for specifying and enacting rule-based, modular protocols.
- We show how the above preferences are mapped into a graph whose points are sets of runs, and characterized via branching time models.
- We show that the model-theoretic and the operational characterizations coincide.

4.4.2 Exception Modeling and Handling

Expected exceptions are modeled at design time using a preference structure and unexpected exceptions are handled at runtime by splicing in appropriate exception handler protocols from a library of common exception handling patterns.

- We use a preference structure such as the one described above, to specify expected exceptions.
- Nodes (which represent sets of runs) towards the bottom of the preference graph are marked as exceptions by designers who deploy the protocol within the processes of interest. In this manner, we obtain a hierarchy of exceptions based on their severity as identified by the designer.
- We show how exception handlers can be spliced into the protocol at runtime to handle exceptions that were not envisioned in the model at design time.

- We specify preferences based on temporal logic.
- We incorporate preferences into a rule-based protocol enactment system.
- We specify exception severity by means of a preference directed graph.
- We splice in exception handlers from a library at runtime to handle unexpected exceptions.

4.5 Room Reservation Example

To motivate our results, let us consider a protocol used to reserve a hotel room. This protocol consists of two roles, a customer and a hotel. The essence of the interaction is the customer booking a room at the hotel and paying for it. This protocol is similar to the room reservation workflow example used by Casati *et al.* (1999). We have modified that example to involve a confirmation step. The steps taken in this interaction are listed below. Here, h represents the hotel and c the customer. Figure 4.1 shows this process. Table 4.1 describes the meanings of the messages used in

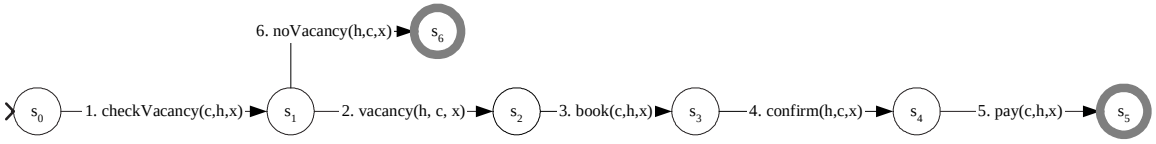


Figure 4.1: The room reservation process model using traditional systems. Thick circles represent terminating states.

this protocol.

- The customer checks if there is a vacancy at the hotel on the dates that it wishes to stay in the room. This is done by the customer sending the *checkVacancy*(c, h, x) message to the hotel. The object x encodes the dates for which the room is required and other possible room preferences that the customer might have.

- If there is a vacancy, the hotel informs the customer of the price of the room by sending the *vacancy*(h, c, x) message, else it informs the customer that there is no vacancy by sending the *noVacancy*(h, c, x) message. The protocol ends if there is no vacancy. By sending a *vacancy*(h, c, x) message, the hotel promises the customer that the room is available for the dates and for the price indicated, i.e., the hotel makes the (temporal) commitment $\mathsf{CC}(h, c,$ *book*$(c, h, x),$ *confirm*$(h, c, x))$, stating that if the customer books before a certain deadline, it will confirm the room for that customer.

- If the quote for the room is acceptable to the customer, the customer asks the hotel to book the room by sending the *book*(c, h, x) message. By booking, the customer promises the hotel that he will pay for the room if the hotel sends a confirmation, i.e., the customer makes the commitment $\mathsf{CC}(c, h, \textit{confirm}(h, c, x), pay(c, h, x))$.
- The hotel sends a confirmation of the room booking to the customer using the *confirm*(h, c, x) message. The customer is now obliged to pay the price agreed upon.
- The customer pays the hotel, using the *pay*(c, h, x) message.

Our introduction of a confirmation step is to clearly demonstrate the flexibility of commitment protocols. For example, the vacancy that the hotel promised might not be available when the customer sends the *book*(c, h, x) message, as is common in large online travel booking systems. In that case, the customer times out waiting for the *confirm*(h, c, x) message from the hotel (or receives a message from the hotel about the inavailability of a room). Since no unconditional commitment are violated, a commitment-based model such as ours naturally handles cases such as the above. In

Table 4.1: Messages used in the room reservation example, with their associated meanings.

Message	Meaning
checkVacancy(c, h, x)	c asks h if there is a vacancy for a certain room for certain dates, encoded by x.
vacancy(h, c, x)	h informs c that the room-date-price combination x is available. Now, h is committed to confirming the room reservation if c books the room for the given dates, i.e., h makes the commitment $\mathsf{CC}(h, c, \textit{book}(c, h, x), \textit{confirm}(h, c, x))$.
book(c, h, x)	c asks h to book the room for the given dates. By this message, c is committed to paying h when the room is confirmed, i.e., c makes the commitment $\mathsf{CC}(c, h, \textit{confirm}(h, c, x), \textit{pay}(c, h, x))$
confirm(h, c, x)	h confirms that the room has been booked. c is now committed to paying h.
pay(c, h, x)	c pays the amount specified in x to h.
timeout$(h, c, \textit{pay})$	h detects that c has not sent the *pay*(c, h, x) message within the specified deadline.

practice, commitment protocols are specified declaratively, for example, using the OWL-P language proposed by Desai *et al.* (2006). The specification identifies roles that participate in the protocol, the messages that are exchanged (with the meanings of the messages in terms of commitments that are created), and a set of rules that constrain the set of runs of the protocol by defining ordering, data

flow, and other constraints. For a formal treatment, however, we represent protocols as transition systems similar in spirit to commitment machines (Winikoff et al., 2004), as introduced in Chapter 3. To briefly summarize, protocols generate computations or *runs*, which are sequences of *states* that a valid protocol computation (execution) goes through. States are labeled by *propositions* that hold true. Propositions represent facts about the universe of discourse of the protocol such as commitments that are active and messages that have been sent. State changes are caused by *messages* that the participants send to each other.

Table 4.2 is a snippet of the specification of the room reservation protocol. The *policy(x,y)* term checks if x wishes to send the message y, and *start* is a special proposition indicating the start state. The derivation of local flows for each role and their binding with the policies of each participant is described by Desai *et al.* (2006). Figure 4.2 shows some runs of this protocol. All run begins at the start state s_0 and end at thick circles, which represent terminating states. Arrows with shaded text show exceptions and filled circles represent exception states.

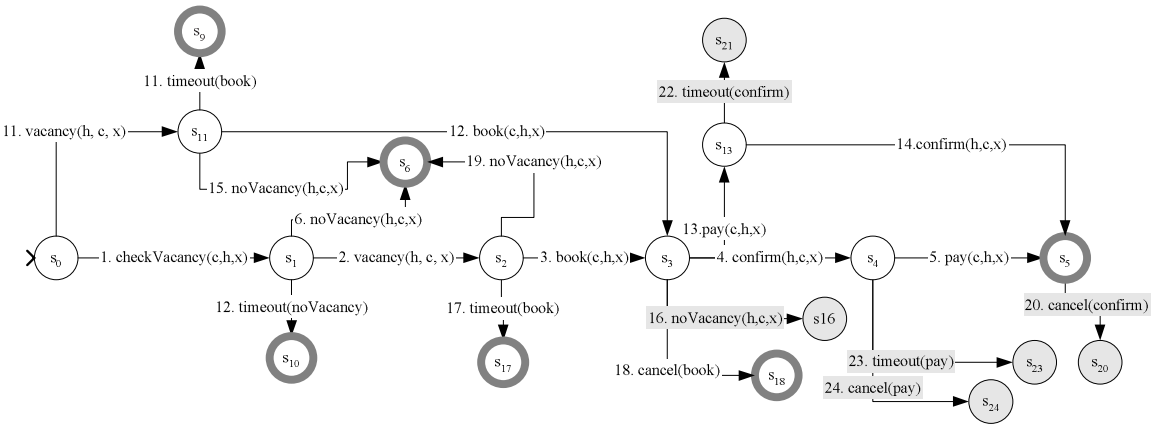


Figure 4.2: Some runs of the room reservation protocol. Filled circles are exceptions; thick circles are terminating states. s_0 is the unique start state.

We have used the room reservation example in this chapter to demonstrate the effectiveness of our methodology on an existing example, developed by others. The principles involved, however, can be related in a straightforward manner to our original purchase example. Table 4.3 lists the mapping from the messages used in the room reservation example (4.2) to the messages used in the purchase example (3.1).

Table 4.2: The room reservation protocol in OWL-P

Role 1: Customer, c **Role 2:** Hotel, h
Rule 1: *start* $\&\&$ *policy*$(c, \textit{checkVacancy})(h, ?x) \Rightarrow$ *checkVacancy*$(c, h, ?x)$ **Rule 2:** *start* $\&\&$ *policy*$(h, \textit{vacancy})(c, ?x) \Rightarrow$ *vacancy*$(h, c, ?x)$ **Rule 3:** *vacancy*$\Rightarrow$ *policy(c, book)* . . .
Message 1:*checkVacancy*(c, h, x) c asks h if there is a vacancy for a certain room for certain dates, encoded by x. This message does not create any commitments.
Message 2:*vacancy*(h, c, x) h informs c that the room-date-price combination x is available. Now, h is committed to confirming the room reservation if c pays the price quoted. This message creates the commitment $\mathsf{CC}(h, c, \textit{confirm}(h, c, x), \textit{pay}(c, h, x))$.
Message 3:*book*(c, h, x) c asks h to book the room for the given dates. By sending this message, c commits to paying h if the room is confirmed. This message creates the commitment $\mathsf{CC}(c, h, \textit{pay}(c, h, x), \textit{confirm}(h, c, x))$.
Message 4:*confirm*(h, c, x) h confirms that the room has been booked. h is now committed to c to allow the use of the room as indicated earlier in the vacancy dates. This message creates the commitment $\mathsf{C}(h, c, \textit{use}(x))$.
Message 5:*pay*(c, h, x) c pays the amount specified in x to h. This message does not create any commitments. If c is committed to h to pay for x, this message fulfills that commitment.
Message 6:*timeout*$(h, c, \textit{pay})$ h detects that c has not sent the *pay*(c, h, x) message within the specified deadline. This message does not create any commitments; it is an indication of the breach of any commitment that c might have made to h to pay for x.
Message 7:*cancel*$(h, c, \textit{confirm})$ h cancels its commitment to c to allow the use of the room. This message cancels the commitment $\mathsf{C}(h, c, \textit{use}(x))$.
. . .

4.6 Modeling and Handling Exceptions

This sections describes our treatment of exceptions in commitment protocols. Handling unexpected exceptions at runtime is described in Section 4.6.2. Modeling expected exceptions via a preference graph is outlined in Section 4.6.1, but a detailed discussion is deferred to Section 4.7.

In the room reservation example, the challenges to exception handling that we outlined

Table 4.3: A mapping between the messages of the room reservation example and the messages of the purchase example. c_r denotes the customer for the room, r the room details, h the hotel, c_p the customer in the purchase, g the goods sold in the purchase, p the price of the good sold, and b denotes the merchant in the purchase example.

Room Reservation	**Purchase**
checkVacancy(c_r, h, x)	*reqQuote*(c_p, b, g)
vacancy(h, c_r, x)	*sendQuote*(b, c_p, g, p)
book(c_r, h, x)	*sendAccept*(c, b, g, p)
confirm(h, c_r, x)	*sendGoods*(b, c_p, g)
pay(c_r, h, x)	*sendMoney*(c_p, b, p)
cancel$(h, c_r,$ *confirm*$)$	*cancel*$(b, c_p,$ *sendAccept*$)$

earlier manifest themselves as follows.

1. *Domain-independent reasoning.* The normal process dictates confirmation of the hotel room before payment. A customer who sends in the payment before the confirmation is in violation of the protocol under such a model, even though practical processes should allow for it, since the essence of the process (reserving a room) is not violated.

2. *Unforeseen exceptions*. The hotel might be forced to revoke the confirmation for the room because of a fire that destroys the room. In traditional process enactment mechanisms such as workflow engines, such exceptions cannot be handled because they require a change in the workflow model. Realistically, the customer should be compensated by some means such as a refund or an offer for an alternative room or hotel.

In the following sections, we describe how our approach handles these situations.

4.6.1 Expected Exceptions

A protocol allows multiple runs, which is what gives its participants flexibility in enacting the protocol. We propose that each protocol specify a graph of preferred runs using a (possibly partial) preference relation over sets of runs. Such preferences prove valuable in reasoning about why a particular state is an exception state. For example, a preference structure can be induced over the process model shown in Figure 4.1 by stating that runs with no *noVacancy*$(h, c, \cdot)$ states are preferred over others.

Preferences between runs can be used to define exceptions independent of the protocol specification. Run sets that are lower in the preference graph can be declared exceptions by de-

ployers of protocols. In this manner, different deployments of the same protocol can have different exception conditions, without having to change the protocol specification.

Consider the room reservation protocol. The protocol might specify that runs in which the hotel does not recant its *vacancy*(h, c, x) message (by sending a *noVacancy*(h, c, x) message) are preferred over runs that do, and the latter set of runs signal an exception. This protocol will consider the runs $\langle s_0 s_1 s_2 s_3 s_{16} \rangle$, $\langle s_0 s_1 s_6 \rangle$, and $\langle s_0 s_1 s_2 s_6 \rangle$ as exception-causing runs (See Figure 4.2). The same protocol without such a preference structure and exception definition would allow the the runs listed above and would thus be better for the hotel. Notice that commitments enable the reasoning that s_{16} is a valid state, i.e., the hotel can send the *noVacancy*(h, c, x) message even after it receives the *book*(c, h, x) message, since no unconditional commitments exist at s_3.

Run preferences may be based on different metrics. For example, short runs might be preferred over longer ones and runs that do not involve the delegation or cancellation of a particular commitment may be preferred over runs that do. The benefit of inducing a preference structure over runs are

1. A preference structure relates an exception to the protocol as a whole rather than modeling an exception as a trigger for specific exception handler code.
2. Participants can reason about what kind of preference structures and related exception definitions they want to entertain. For example, customers might like to choose a room reservation protocol with a preference structure that terms the *noVacancy*(h, c, x) sent after a *vacancy*(h, c, x) message an exception.
3. Preference structures decouple the specification of a protocol from the environment in which it is enacted. A situation that is normal in a protocol in one context might be deemed an exception in the same protocol when it is being executed in a different context. For example, the room reservation protocol allows the hotel to send a *vacancy*(h, c, x) message to the customer to start the protocol. Starting the protocol in this manner is equivalent to the hotel advertising the availability of a room. In some situations, however, such an unsolicited advertisement might be deemed illegal, as in the case of spam.

- A preference relation between sets of runs creates a preference graph.
- If the nodes of the preference graph are interpreted as sets of exception runs, we obtain a graph of exceptions.
- Protocol specification is decoupled from specification of exceptions.

4.6.2 Unexpected Exceptions

Unexpected exceptions are deviations from the modeled interaction detected at runtime but not part of the process model in traditional systems. In a commitment protocol, however, a model is a declarative specification of the meanings of messages, which, combined with the semantics of commitment operations, drives the protocol. In this section, we show how our protocol algebra can be used to combine exception handlers with protocols at runtime so that new situations can be handled when they arise.

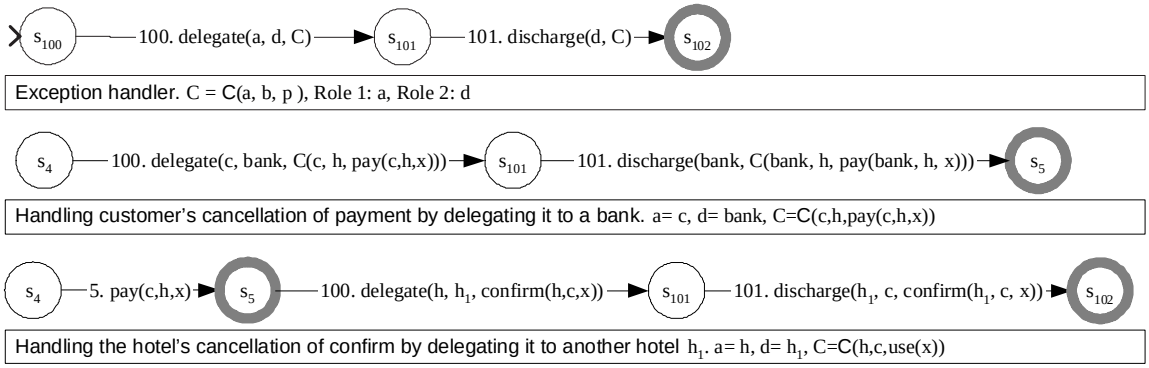


Figure 4.3: An exception handler and its usage by splicing. Thick circles denote terminating states.

Splicing Exception Handlers

As introduced in Section 3, protocols can be *spliced* using the merge operator, with additional constraints. For example, if a complicated payment protocol existed, where a payer transfers funds to a payee via a bank, that payment protocol could be spliced into the room reservation protocol between states s_4 and s_5 or between states s_3 and s_{13}. The new protocol thus obtained would be a *refinement* of the room reservation protocol, since it dictates a finer grained protocol to its participants than the plain room reservation protocol.

We treat exception handlers as sets of runs, just as we do protocols. This enables handlers to be spliced into protocols. As an example, consider the exception state s_{24} in Figure 4.2. This exception is caused by the customer cancelling its commitment to pay for the room after the hotel has confirmed it. Normally, this would be a violation of a commitment. The customer might wish to cancel the commitment to pay, say, because of financial hardship. Consider now an exception handler as shown in Figure 4.3, at the top. This handler can be spliced into the room reservation protocol after appropriately binding the role *a* to customer, role *b* to hotel, and role *d* to the cus-

tomer's bank. This splicing lets the customer keep the hotel room while not having to pay the hotel. States s_{100} and s_{102} of the exception handler correspond to states s_4 and s_5 of the room reservation protocol, respectively. The splicing is shown in Figure 4.3, in the middle.

The same handler can also be used to delegate commitments, when the need arises, instead of cancelling them. For example, this handler can be used by the hotel to delegate the reservation of a room to another hotel in case of fire. In this case, the handler would be spliced at state s_5 of the room reservation protocol, with the role *a* corresponding to the hotel, role *b* to the customer, and role *d* to another hotel which can provide the room in lieu of the former hotel. State s_{100} now corresponds to s_5 and states s_{101} and s_{102} are added to the protocol, as shown at the bottom of Figure 4.3.

Splicing Exceptions Handlers in Practice

We envision that exception handlers will be selected for splicing based on state-similarity functions from a library of handlers or, more generally, a library of commitment protocols. In this manner, exceptions that the protocol designer did not foresee can be handled if an extensive library of exception handler protocols is available. Handlers would be selected based on state-similarity functions. This requirement is not as far-fetched as one might imagine. Projects such as Rosetta Net (RosettaNet) and the FIX protocol (Twi, 2005), have already developed libraries of business protocols.

- Exception handlers are treated as protocols.
- At runtime, when an unexpected exception occurs, a handler can be spliced with the protocol.
- Handlers for splicing are derived from a library of exception handler-protocols.

4.7 Specifying and Resolving Preferences

We propose that protocol specifications be enhanced with modular, pluggable descriptions of the *preferences* among executions of the protocol. These preferences are the protocol designer's view of what are the most normal or most desired executions. In this sense, they have a normative force. Individual agents would have their own *policies* for how they participate in a given protocol. The policies should generally be in line with the protocol preferences. (Verifying compliance with protocol preferences, however, is nontrivial—we return to this point in Section 4.7.1.)

Protocols can be refined to yield protocols that serve the same goal but impose additional requirements. For example, payment by cash is a refinement of payment (in general). Agents

contemplating interacting can negotiate about the refinement of the protocol that they will enact. For example, if a merchant accepts only cash, payment by check or charge card is ruled out and payment by cash is the only kind of payment that will work.

More interestingly, agents who are participating in a protocol may negotiate about the specific actions that each would take (from among those that are allowed by the given protocol). For example, given that two parties agree to participate as seller and buyer in a purchase protocol, they might then negotiate about whether the buyer can have a third party pay on his behalf. Whereas this dissertation does not study negotiation of preferences, it provide a basis for argumentation among the parties.

For compatibility with our protocol model described in earlier chapters, we use the linear temporal logic-based event ordering language proposed by Singh (2003) to specify preferences between sets of executions in that transition system. Although Singh's language is given semantics based on linear models, the models are related in an incremental manner to other possibilities: thus the spirit of it is arguably branching. We use expressions over events in this language to specify sets of runs of a protocol over which preferences can be expressed. This section briefly introduces the semantics of commitments to lay the groundwork for preference specification and introduces the language.

OWL-P

Our work on preferences builds on OWL-P (Desai et al., 2005, 2006), which is a practical framework and an associated language for specifying, combining, and enacting commitment protocols. OWL-P (OWL for Protocols and Processes), uses OWL (Web Ontology Language) to specify protocol and protocol composition constructs. An OWL-P specification identifies roles that participate in the protocol, the messages that are exchanged (with the meanings of the messages in terms of commitments that are created), and a set of rules that constrain the set of runs of the protocol by specifying ordering, data flow, and other constraints. The key features of the OWL-P framework are listed below. A more detailed exposition of OWL-P appears in Chapter 6.

- An OWL-P specification includes the roles, the meanings of messages, and the rules that dictate valid executions.
- An OWL-P specification assigns meanings to messages by specifying how messages assert and retract various propositions and how they create and manipulate commitments.

- In the OWL-P framework, skeletons are derived for each role. A programmer can specify the policies for each agent that plays a given role to determine whether and how to act in enacting the given protocol. Desai *et al.* (2004) discuss this aspect at length.

- Commitment protocols allow numerous runs; we enhance them with a modular, pluggable specification of preferences among the runs.
- We will demonstrate how preferences can be incorporated into OWL-P, a framework for specifying and enacting rule-based commitment protocols.

4.7.1 Specifying Preferences

As explained before, OWL-P specifies protocols declaratively and leverages the semantics of commitments and their operations to make protocols flexible. However, OWL-P does not distinguish between the normal and abnormal executions of a protocol. For example, executions in a purchase protocol that involve returns and refunds or missing shipments are treated on par with the normal executions involving no missing shipments or delayed payments. For this reason, we extend OWL-P to capture preferences among sets of executions. We first devise a language over events to concisely specify sets of protocol executions. Next, we specify preferences among such sets using this language. Preferences serve as a rough-and-ready means to capture design goals wherein the normal executions are preferred, yet abnormal executions arising from exceptions or unexpected actions by some of the agents are allowed.

The Constraint Specification Language

We use the event-based linear temporal logic $\mathcal{I}$ as introduced by Singh (2003) for specifying sets of runs. We repeat here the syntax and semantics of $\mathcal{I}$ from (Singh, 2003). I is the start symbol of the BNF for the language of $\mathcal{I}$. In this BNF, *slant* indicates nonterminals, $\longrightarrow$ and $|$ are meta-symbols of the BNF, /* and */ begin and end comments, respectively; all other symbols are terminals.

L_1. $I \longrightarrow dep \mid dep \wedge I$ /*conjunction: interleaving*/

L_2. $dep \longrightarrow seq \mid seq \vee dep$ /* disjunction: choice*/

L_3. $seq \longrightarrow bool \mid event \mid event \cdot seq$ /* before: ordering*/

L_4. $bool \longrightarrow 0 \mid \top$

Dependency. A dependency is an expression generated by I. It specifies constraints on the occurrence and ordering of events.

Event Literal Set. $\Gamma \neq \{\}$ is the set of event literals used in I. Γ_D is the set of literals mentioned in a dependency D and their complements, for example, $\Gamma_e = \{e, \overline{e}\}$. For a set of dependencies $\mathbb{D}$, we define $\Gamma_{\mathbb{D}}$ as $\Gamma_{\mathbb{D}} = \bigcup_{D \in \mathbb{D}} \Gamma_D$.

Since state labels in OWL-P capture the history of events in the system, the event-based language (and semantics) that we use here can be applied to the state-based execution semantics of OWL-P.

Legal runs satisfy the following requirements:

1. Event instances and their complements are mutually exclusive.
2. An event instance occurs at most once in a computation.

Universe of Runs. $\mathbb{U}_{\mathcal{I}}$ is the universe of runs; it contains all legal runs involving event instances from Γ.

Constraint Language Semantics

The formal semantics of $\mathcal{I}$ is based on runs, i.e., sequences of events. For a run $\tau \in \mathbb{U}_{\mathcal{I}}$ and $I \in \mathcal{I}$, $\tau \models I$ means that I is satisfied over the run τ . This notion can be formalized as follows. Here, τ_i refers to the ith item in τ and $\tau_{[i,j]}$ refers to the subrun of τ consisting of its elements from index i to index j, both inclusive. $|\tau|$ is the last index of τ and may be ω for an infinite run. We use the following conventions in the specification of semantics below: $e, f, \overline{e}, \overline{f}$, etc. are literals; D, E, etc. are dependencies; i, j, k, etc. are temporal indices; and τ, etc. are runs. The semantics of $\mathcal{I}$ is

M_1. $\tau \models e$ iff $(\exists i : \tau_i = e)$

M_2. $\tau \models I_1 \vee I_2$ iff $\tau \models I_1$ or $\tau \models I_2$

M_3. $\tau \models I_1 \wedge I_2$ iff $\tau \models I_1$ and $\tau \models I_2$

M_4. $\tau \models I_1 \cdot I_2$ iff $(\exists i : \tau_{[0,i]} \models I_1$ and $\tau_{[i+1,|\tau|]} I_2)$

Denotation. The denotation $[\![D]\!]$ of a dependency D is the set of runs that satisfy D, i.e., $[\![D]\!] = \{\tau : \tau \models D\}$.

Preferences

The specification language $\mathcal{I}$ supports dependencies to succinctly specify sets of runs. To induce a preference structure over such sets, i.e., to specify if one set is preferred over another, we introduce the *preference relation*.

Preference Relation. Let $\mathbb{R} : 2^{\mathbb{U}_\mathcal{I}} \mapsto 2^{\mathbb{U}_\mathcal{I}}$ be the preference relation between sets of runs. $\mathbb{R}$ is irreflexive, transitive, and antisymmetric. $(D_i, D_j) \in \mathbb{R}$ means that any run that satisfies the constraints in D_i is preferred over any run that satisfies the constraints in D_j, i.e., $\forall \tau_i, \tau_j$: $\tau_i \in [\![D_i]\!]$ and $\tau_j \in [\![D_j]\!], \tau_i$ is preferred over τ_j.

Commitment protocols can be expressed as a set of dependencies. For a protocol P specified in terms of a set of dependencies D_P, the denotation of the protocol is the set of runs that the protocol allows, and is given by $[\![P]\!] = \bigcup_{D \in D_P} [\![D]\!]$.

Preference Graph. A preference graph specifies preferences among a set of dependencies. Each dependency labels one node of the graph, and a preference relation specifies preferences among these dependencies, and consequently among the nodes they label. A preference graph $L = \langle D_x, \mathbb{R} \rangle$ specifies preferences among the elements of the set of dependencies D_x using the partial order induced by the preference relation $\mathbb{R}$ over D_x. For the above L, we define its event literal set to be the set of all events that are mentioned in the dependencies in D_x, and their complements. $\Gamma_L = \bigcup_{D \in D_x} \Gamma_D$.

Figure 4.4 shows our schematic representation of a preference graph.

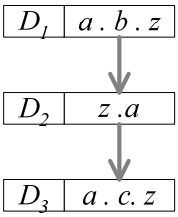


Figure 4.4: A preference graph $L = \langle \{D_1, D_2, D_3\}, \{(D_1, D_2), (D_2, D_3)\} \rangle$, where $D_1 = a \cdot b \cdot z$, $D_2 = z \cdot a$, and $D_3 = a \cdot c \cdot z$.

Preference Node Denotation. The denotation of a preference node N_D labeled by D, with respect to a protocol P and a preference graph $L = \langle D_P, \mathbb{R} \rangle$, where $D \in D_P$, is the set of runs allowed by the protocol that are also allowed by D, but not by nodes in L that are preferred

over N_D. The motivation for the above is that each run can occur in at most one node in the graph. Thus if a given run is allowed by two sets of runs, one preferred over the other, the less set would not "get credit" for this run. On each path from a top node (i.e., a node with no ancestors) to the bottom node (i.e., a node that is not preferred over any other) of the graph, a given run can occur at most once. Formally, $[\![N_D]\!] = [\![P]\!] \cap ([\![D]\!] - \bigcup_{(D_i,D)\in\mathbb{R}} [\![D_i]\!])$.

Consider the room reservation protocol. In some cases, the designer might want to prevent cancellation of an unconditional commitment, e.g., to prevent cancellation of the customer's commitment to pay after the room has been confirmed or the hotel's commitment to confirm after a payment has been made by the customer. The preferences for this can be encoded as follows

- $P_1 = \mathit{confirm}(h, c, x) \cdot \mathit{pay}(c, h, x)$
- $P_2 = \mathit{pay}(c, h, x) \cdot \mathit{confirm}(h, c, x)$
- $P_3 = \mathit{confirm}(h, c, x) \cdot \mathit{cancel}(\mathit{confirm}(h, c, x))$
- $P_4 = \mathit{pay}(c, h, x) \cdot \mathit{refund}(h, c, x)$
- $P_5 = \mathit{pay}(c, h, x) \cdot \mathit{cancel}(m, \mathsf{C}(m, c, \mathit{confirm}(h, c, x)))$
- $P_6 = \mathit{confirm}(h, c, x) \cdot \mathit{cancel}(c, \mathsf{C}(c, m, \mathit{pay}(c, h, x))$
- P_1 and P_2 are each preferred over each of P_3, P_4, P_5, and P_6
- P_3 and P_4 are each preferred over P_5 and P_6

This preference graph is shown in Figure 4.5. There is no preference between P_1 and P_2, no preference between P_3 and P_4, and no preference between P_5 and P_6. This preference structure restricts the runs of the purchase protocol by disallowing cancellation of unconditional commitments. Based on our interpretation of the denotation of the preference graph nodes, we obtain the following sets of runs:

- $[\![N_{P_1}]\!] = [\![\mathit{confirm} \cdot \mathit{pay}]\!]$. All runs in which the hotel confirms the room and a payment is made afterwards.
- $[\![N_{P_2}]\!] = [\![\mathit{pay} \cdot \mathit{confirm}]\!]$. All runs in which the customer pays, after which the hotel sends a confirmation.

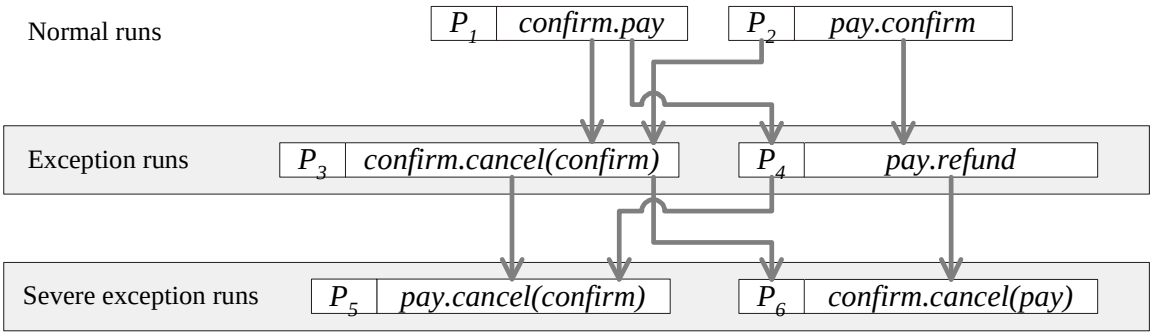


Figure 4.5: A preference graph for the purchase protocol. Grey arrows depict preferences. Events are represented without their parameters, to avoid clutter. *refund* represents payment from the hotel h to customer c, and *cancel* represents the commitment operation.

- $[\![N_{P_3}]\!] = [\![\mathit{confirm} \cdot \mathit{cancel}(\mathit{confirm}) \wedge \overline{\mathit{pay}}]\!]$. All runs in which the hotel cancels a confirmation it has made before the customer pays, and the payment is never done. This run cancels an unconditional commitment, hence it valid, although undesirable.

- $[\![N_{P_4}]\!] = [\![\mathit{pay} \cdot \mathit{refund} \wedge \overline{\mathit{confirm}}]\!]$. All runs in which the hotel refunds the customers payment and never sends a confirmation to the customer. This run cancels an unconditional commitment, hence it is valid, although undesirable.

- $[\![N_{P_5}]\!] = [\![\mathit{pay} \cdot \mathit{cancel}(\mathit{confirm}) \wedge \overline{\mathit{refund}} \wedge \overline{\mathit{confirm}}]\!]$. All runs in which the hotel cancels the confirmation after the customer has paid, never confirms, and never sends a refund. Note that this set contains runs that will never occur because they violate the commitment life cycle. Specifically, if *confirm* never happens, then $\mathit{cancel}(\mathit{confirm})$ has no meaning, since commitments cannot be operated upon unless they are created. Therefore, we induce constrains based on commitment operations, to obtain $[\![N_{P_5}]\!] = [\![(\mathit{pay} \cdot \mathit{confirm} \cdot \mathit{cancel}(\mathit{confirm}) \wedge \overline{\mathit{refund}}) \vee (\mathit{pay} \wedge \overline{\mathit{confirm}} \wedge \overline{\mathit{refund}})]\!]$. That is, all runs in which the hotel cancels its confirmation or never makes a confirmation, but does not refund the payment made by the customer.

- $[\![N_{P_6}]\!] = [\![(\mathit{confirm} \cdot \mathit{cancel}(\mathit{pay}) \wedge \overline{\mathit{cancel}(\mathit{confirm})} \wedge \overline{\mathit{pay}}) \vee (\mathit{confirm} \wedge \overline{\mathit{pay}} \wedge \overline{\mathit{cancel}(\mathit{confirm})})]\!]$ This is obtained similar to $[\![N_{P_5}]\!]$.

- We use Singh's linear-temporal logic specification for concisely expressing sets of runs.
- Each node in the preference graph describes one set of runs, specified by an expression using the above logic.
- Every run occurs in at most one such set if we trace a path from amost-preferred node to a least-preferred node of the preference graph.

4.7.2 Resolving Preferences

How can an agent use a preference graph protocol enactment? We describe here the operationalization of a preference graph, and a methodology for inclusion of preferences into a rule-based commitment protocol framework, OWL-P.

Evolution of Preference Graphs

A preference graph evolves as events occur and cause the state of the protocol to change. The evolution changes the denotation of each node of the preference graph because events residuate dependencies that label preference graph nodes.

Residuation

The residual of a dependency D by an event e is denoted by D/e and corresponds to the largest set of runs satisfying the given dependency. Formally, $\nu \in [\![D/e]\!]$ iff $(\forall v : v \in [\![e]\!] \Rightarrow (v\nu \in \boldsymbol{U}_{\mathcal{I}} \Rightarrow v\nu \in [\![D]\!]))$

The residual represents what remains in the dependency after a certain event has occurred for the dependency to be satisfied on any run. The following are residuation rules as given by Singh:

R_1. $0/e \doteq 0$

R_2. $\top/e \doteq \top$

R_3. $(E_1 \wedge E_2)/e \doteq ((E_1/e) \wedge (E_2/e))$

R_4. $(E_1 \vee E_2)/e \doteq ((E_1/e) \vee (E_2/e))$

R_5. $(e.E)/e \doteq E$ if $e \notin \Gamma_E$

R_6. $D/e \doteq D$ if $e \notin \Gamma_D$

R_7. $(e' \cdot E)/e \doteq 0$ if $e \in \Gamma_E$ where e' is any event literal

R_8. $(\overline{e} \cdot E)/e \doteq 0$

Below, we describe the evolution of a (preference) preference graph, and give an algorithm for the same. Here, $\tau + e$ concatenates event e to run τ.

Evolution Node. An evolution node is a node that represents the evolving state of a preference graph. The event literal set Γ_V of an evolution node V is the event literal set Γ_L of the preference graph L it represents. An evolution node represents a *path* in the evolution tree. This path is a legal run and is a concatenation of the events that took place for the evolution node to be generated, starting from the root of the evolution tree. Evolution nodes are described $V_\tau = \langle L, \tau \rangle$, where L is a preference graph and τ is the path to V_τ from the root node of the evolution tree.

Evolution Tree. An evolution tree $\mathbb{T}_L$ of a preference graph L represents, all possible states that L can evolve through as events happen in Γ_L. An evolution tree is a 3-tuple, $\mathbb{T}_L = \langle \mathbb{V}, \mathbb{E}, L \rangle$, where $\mathbb{V}$ is the set of evolution nodes in the tree, $\mathbb{E}$ is the set of edges between evolution nodes, called *evolution edges*, of the form $\langle V_\tau, e, V_{\tau+e} \rangle$. The root node of an evolution tree T_L is $V_{\tau_0} = \langle L, \tau_0 \rangle$, where τ_0 is the zero-length run.

Given a preference graph L, we construct the evolution tree $\mathbb{T}_L$ for it by first creating the root evolution node $V_{\tau_0} = \langle L, \tau_0 \rangle$. From V_{τ_0}, we draw an arc for each event in Γ_L that is not in τ_0. The arcs are labeled by the event they represent. For each arc e, we create a new evolution node $V_{\tau_0+e} = \langle L', \tau_0 + e \rangle$. For this new node, we set $L' = \langle D' \mathbb{R}' \rangle$, where D' is the set of non-zero dependencies obtained by residuating each dependency in L by e, and $\mathbb{R}'$ relates two dependencies in D' if the dependencies corresponding to them in D were related by $\mathbb{R}$. That is, we preserve the preferences among the residuated dependencies in the evolution nodes. Figure 4.7 lists the algorithm for generating an evolution tree of a specified finite depth, given a preference graph.

Figure 4.6 shows parts of the evolution tree generated using the algorithm in Figure 4.7 on the preference graph shown in Figure 4.4.

Choosing Between Runs

Consider an agent X that operates within the preference preference graph specified for a protocol in which it is participating. For such an agent to determine during enactment the action

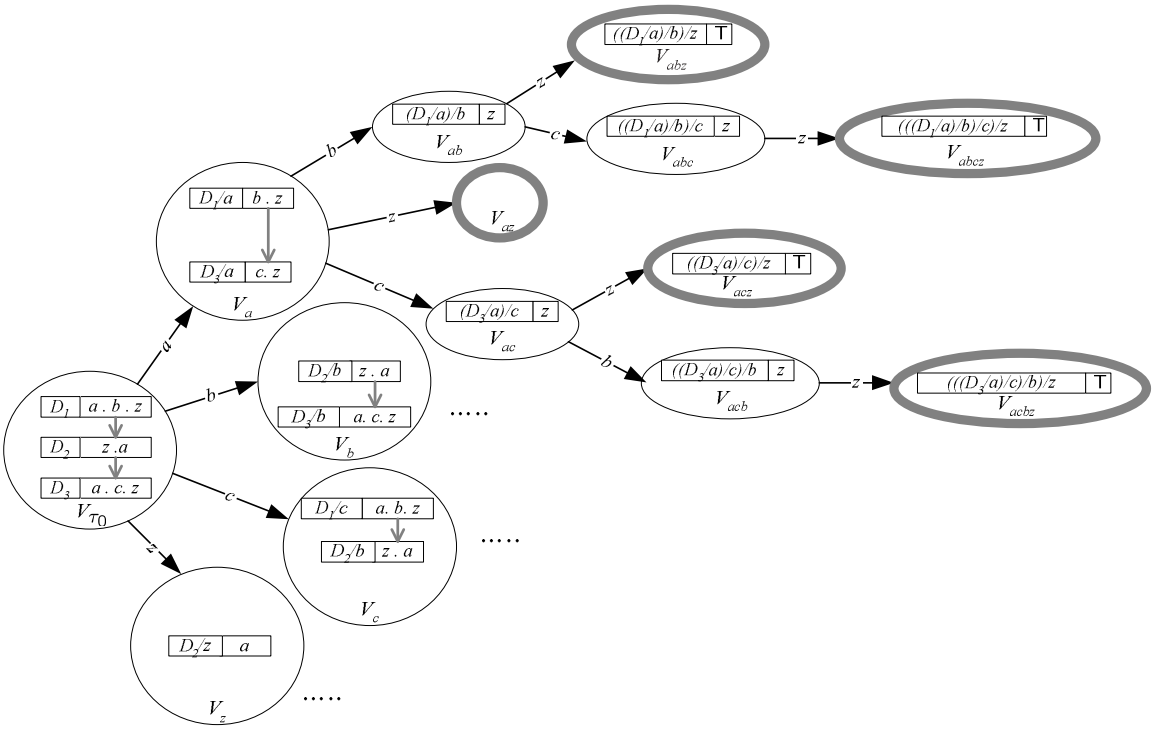


Figure 4.6: The partial evolution tree for the preference graph shown in Figure 4.4, using residuation.

to take (we assume there is an event corresponding to every action instance), each evolution node is a decision point. Logically, X chooses the event that takes the protocol to the most preferred preference graph node or, for a protocol whose state is already in the most preferred preference node, keeps it there. Since preference nodes are distributed among the evolution nodes, and X cannot change the state of the protocol at a finer granularity than that of evolution nodes, X will have to evaluate which evolution node is the most preferred. To determine this, a preference graph of evolution nodes is constructed as follows: At any evolution node, to decide the next action to take, induce a preference graph among the child evolution nodes using the original preference structure, i.e., the preference structure specified in the root of the evolution tree. Once this is done, induce a preference graph on the evolution nodes based on the preference arcs that originate from one evolution node and terminate in another.

Figure 4.8 shows the algorithm for constructing a preference graph between evolution nodes.

Applying this algorithm to the children of the root evolution node V_{τ_0} in Figure 4.6, we obtain the evolution node preference graph shown in Figure 4.9. To explain this preference graph,

input : A preference graph $L = \langle D_x, R \rangle$, and a limit MAX_DEPTH for the depth of the tree generated
output: An evolution tree $\mathbb{T}_L = \langle \mathbb{V}, \mathbb{E}, L \rangle$ of L

```
1  Create the root node V_{τ0} = ⟨L, τ0⟩ ;
2  depth ← 1 ;
3  V ← V_{τ0} ;
4  T_L.V ← {V} ;
5  T_L.E ← {} ;
6  siblings ← {V};
7  while depth < MAX_DEPTH && siblings ≠ {} do
8      foreach Evolution Node V ∈ siblings do
9          siblings ← siblings −{V} ;
10         foreach event e ∈ Γ_L such that e ∉ V.τ do
11             V_{V.τ+e} ← createChildNode (V, e) ;
12             T_L.V ← T_L.V ∪ {V_{V.τ+e}} ;
13             T_L.E ← T_L.E ∪ E ;
14             if V_{V.τ+e}.L.D ≠{} then
15                 siblings ← siblings ∪{V_{V.τ+e}} ;
16     depth ← depth +1 ;
17 return T_L ;
```

18 **Procedure** `createChildNode` (V, e): Create a child evolution node for the node V on event e

input : An evolution node $V = \langle L, \tau \rangle$, where $L = \langle D, R \rangle$, and an event e.
output: An evolution node that represents the state of L on event e.

```
19 D' ← {} ;
20 R' ← {} ;
21 foreach dependency d ∈ V.L.D do
22     if d/e ≠ 0 then
23         D' ← D' ∪ (d/e) ;
24         foreach preference (d_i, d_j) ∈ V.L.R do
25             if d == d_i or d == d_j then
26                 R' ← R' ∪ (d_i, d_j) ;
27 L' = ⟨D', R'⟩ ;
28 V_{V.τ+e} = ⟨L, V.τ + e⟩ ;
29 return V_{V.τ+e}
```

Figure 4.7: GenerateEvolutionTree(L, MAX_DEPTH): Generate the evolution tree for a preference graph $L = \langle D_x, R \rangle$, where MAX_DEPTH is a bound on the depth of the tree to be generated.

we show the preference relationship among the dependencies within the evolution nodes, in Figure 4.10.

input : An evolution node V_τ, and a root preference graph $L_r = \langle D_r, R_r \rangle$
output: A preference graph $L_e = \langle D_e, R_e \rangle$ representing the preferences among the child nodes of V_τ.

```
1  D_e ← {} ;
2  R_e ← {} ;
3  foreach child node V_{τ+x} of V_τ do
4      D_e ← D_e ∪ V_{τ+x} ;
5      foreach child node V_{τ+y} of V_τ do
6          D_e ← D_e ∪ V_{τ+y} ;
7          foreach d_x ∈ V_{τ+x} do
8              foreach d_y ∈ V_{τ+y} do
9                  if (getRootDep (d_x, V_{τ+x}, D_r), getRootDep (d_y, V_{τ+y},
                   D_r)) ∈ R_r then
10                     R_e ← R_e ∪ (V_{τ+x}, V_{τ+y});
11                 if (getRootDep (d_y, V_{τ+y}, D_r), getRootDep (d_x, V_{τ+y},
                   D_r)) ∈ R_r then
12                     R_e ← R_e ∪ (V_{τ+y}, V_{τ+x});
13 L_e ← ⟨D_e, R_e⟩ return L_e ;
```

14 **Procedure** `getRootDep` $(d, V_{\tau'}, D_0)$: Return the dependency in D_0 that was residuated to d along a path τ'.
input : A dependency d, its evolution node $V_{\tau'}$,and a set of dependencies D_0.
output: The dependency in D_0 that was residuated to d along the path τ'; **null** if no such dependency exists.

```
15 foreach d_0 ∈ D_0 do
16     d_t ← d_0 ;
17     for i ← 0 to |τ'| − 1 do
18         d_t ← d_t/τ'_[i] ;
19     if d_t == d then
20         return d_0 ;
21     else
22         return null ;
```

Figure 4.8: GenerateEvolutionPreferenceGraph(V_τ, L_r): Generate a preference graph of preferences among the child evolution nodes of an evolution node V_τ, given the root preference graph L_r.

Selecting the Best Path

Given the preference graph of preferences among the child (evolution) nodes, the best action to take (event to bring about) is the event that leads to any one of the top elements of the preference graph, i.e., to a node in the evolution node preference graph over which no other node is preferred. However, the evolution node preference graph might have no top element. This happens

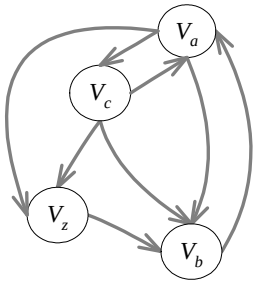


Figure 4.9: The preference graph for the evolution nodes V_a, V_b, V_c, and V_z shown in Figure 4.6, based on the preference graph L shown in Figure 4.4

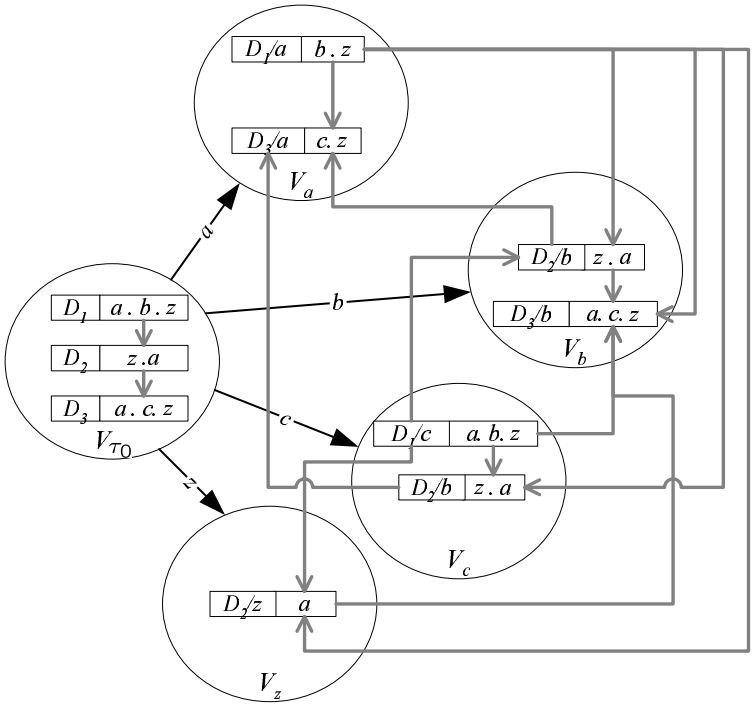


Figure 4.10: A detailed view of the evolution node preference graph shown in Figure 4.9. Grey arrows indicate the preference relation.

when all nodes are involved in a cycle, as in shown in Figure 4.9, which signals an unsound original preference specifications. The agent can then choose any of the paths available. Path selection is discussed in greater detail in Section 4.7.3.

Types of Events

So far, we have discussed the single agent case, where the choice among events is for a single agent to make. In agent interaction, however, events are brought about by different agents during the enactment of a protocol. Choosing the best path, if one is available, is therefore not up to a single agent. We therefore identify two types of events, *observed* and *controlled*.

Controlled Events. For a given agent, an event that it can bring about is called a controlled event. We introduce a distinguished controlled event, Noop, which means "do nothing." An agent can perform a Noop to wait.

Observed Events. For a given agent X, an event which X can observe, but not bring about, is called an observed event. We distinguish two unique observed events, start and end, which

are observed by agents participating in a protocol when the protocol begins and ends respectively.

Whether an event is observed or controlled is decided when the events are described for the system being modeled.

4.7.3 Patterns for Path Selection

Given that an agent will make a choice at every node of the evolution tree, and that events can be controlled or observed, we identify typical patterns of decision making that agents can adopt in common situations. Before describing these patterns, we explain rule-based protocols of the type found in the OWL-P framework.

Incorporating Preferences into Rule-Based Protocols

We envision the application of our algorithms in rule-based protocol enactment frameworks such as OWL-P. OWL-P enables agents to specify their *local policies* independent of the protocol specification. The general structure of rules in such protocols is the Event-Condition-Action (ECA) structure,

$$\begin{aligned} \textsf{On} \quad & e \\ \textsf{if} \quad & \textit{localPolicy}(e, x) \\ \textsf{then do} \quad & a(x) \end{aligned}$$

Here, e is an event and $a(\cdot)$ is an action (with a corresponding event e_a), which depends on the local policy for its parameters.

As an example, consider a purchase protocol that requires that the receiver of a *rfq* $(c, m,$ *itemID*$)$ message (which is a request for a price quote for an item)–the merchant m–respond the the sender–the customer c–with a *quote*$(m, c,$ *itemID*, *price*$)$ message (which is a price quote). The rule for this requirement will be

$$\begin{aligned} \textsf{On} \quad & \textit{rfq}(c, m, \textit{itemID}) \\ \textsf{if} \quad & \textit{localPolicy}(\textit{rfq}(c, m, \textit{itemID}), \textit{price}) \\ \textsf{then do} \quad & \textit{quote}(m, c, \textit{itemID}, \textit{price}) \end{aligned}$$

Where multiple choices of action are afforded by the protocol on the same triggering event, there will be one rule for each action, with the same event and condition. In such a case, the local policy

decides which rule to enable and which to disable. In case the policy enables multiple rules, the preference specification of the protocol is used to decide which action to take. Based on these principles, we describe patterns of decision making that agents can use. Henceforth, we consider the preferences from the point of view of an agent X unless otherwise specified.

Clear Myopic Choice

If an agent has to choose between paths in an evolution tree, and a top node exists in the evolution node preference graph, the agent takes the path to the top node. This situation is shown in Figure 4.11.

Noop Usage

Sometimes, even when there is a clear myopic choice, an agent's local policy might force the agent to not take any action. Consider once again Figure 4.11. At the decision point V_τ, agent X can take action a or z. According to preference specifications, taking a is clearly the better choice. What will happen if X's local policy suggests that a is not currently the best choice? X could perform z, since it is the next best choice. However, if the protocol does not require that X act immediately (most realistic protocols have a notion of timeouts and deadlines), then X can perform a Noop, waiting till the policy allows a to be performed or till the protocol forces X to perform z (which will also be a policy decision, if the policy has been designed to not violate the protocol). A Noop, when used, can introduce cycles in the evolution node preference graph. In the example given here, however, the introduction of a V_{Noop} node as a child of V_τ will not introduce a cycle between V_a, V_z, and V_{Noop}, as shown in Figure 4.12.

Unclear Myopic Choice

Agent X cannot choose a path to follow if the evolution node preference graph has no topmost element, i.e., the preference graph has a cycle over all its nodes. This pattern was illustrated in Figure 4.6, among the child nodes of V_{τ_0}.

Unclear Non-Myopic Choice

If there is a cycle in the evolution node preference graph but some events are only observed by X, then we have an unclear non-myopic choice. X has to make an arbitrary choice or wait for one of the other agents to perform one of the observed events. For example, if a and b were observed

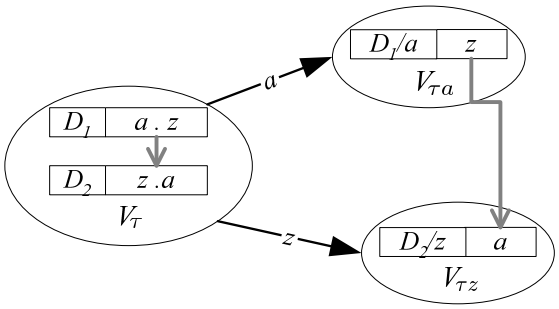


Figure 4.11: Clear myopic choice: An agent can unambiguously decide which path to choose, since the evolution node preference graph has a top element, $V_{\tau a}$.

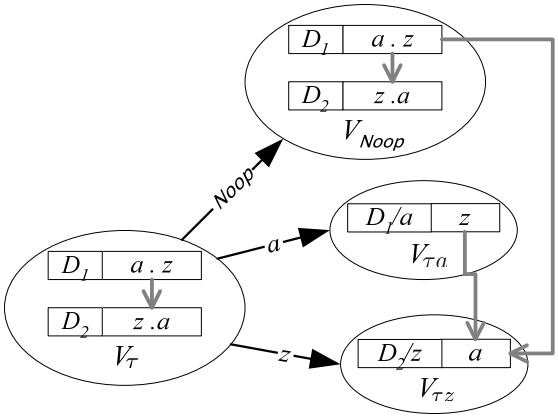


Figure 4.12: Using a Noop when local policy does not allow the best choice and the protocol does not impose immediate deadlines on the other choice(s).

events for X, the evolution node preference graph shown in Figure 4.10 would have been a case of unclear non-myopic choice for X.

- The "computation tree" of a preference due to residuation represents its evolution.
- An agent chooses the action that moves the run to the topmost node in the preference graph.
- An agent cannot choose unambiguously if all nodes of the preference graph are involved in a cycle.
- Local policies can make an agent wait, performing a Noop even when a topmost element exists in the preference graph, if the protocol allows waiting.

4.8 Discussion

Exceptions are numerous in real-life systems since the universe of discourse of a business process has ill-defined boundaries. Real-world processes need pragmatic exception handling, i.e., reasoning about meaning in context. We have proposed a methodology that utilizes commitments among participants to handle exceptions in a protocol. Although, protocols help us capture agent interactions perspicuously, the very strength of protocols, namely, their flexibility, can become a problem. Executions allowed so as to make a protocol flexible under exceptional conditions can begin to be selected over superior executions even when the exceptional conditions do not obtain.

In this chapter, we have shown that protocols be specified generically, and preference conditions among different sets of runs can be stated separately, depending on the context of usage

of the protocol. Preferences serve as a rough and ready means to capture design goals wherein the normal executions are preferred, yet abnormal executions arising from exceptions or unexpected actions by some of the agents are allowed. We have also identified a denotational semantics for the preference conditions understood as a preference graph and showed how such a preference graph evolves as events and actions take place. We have also proposed a simple method by which an agent can decide upon its next action, describing an anomalous condition and giving an algorithm for identifying it. Preferences are often understood solely in decision-theoretic terms. Decision theory is clearly important for modeling preferences and coming up with strategies for acting according to them. However, a denotational understanding of preferences can support the proper application of decision theory.

Further, we have also demonstrated how an algebra of protocols can be employed to handle exceptions that are not originally in the process model. With this approach, we have demonstrated how commitment protocols used in specific contexts can create agile processes. These contributions are a significant step forward from the closed, rigid process models provided by traditional formalisms. Next, we discuss literature relevant to the content presented in this chapter.

4.8.1 Related Literature

Exception handling has been studied for programming languages, multiagent systems, and workflows, among others.

Exceptions in Programming

Miller and Tripathi (1997) identify four different reasons for exceptions in object-oriented systems: *errors*, which are illegal conditions, valid *deviations*, *notifications* that invalidate certain assumptions, and *idioms*, which are legal conditions that are rare, such an end-of-file encounter. They study errors in detail and deviations and notifications to a limited extent whereas our research deals mainly with deviations, notifications, and idioms.

Exceptions in Multiagent Systems

Klein *et al.*(2003) study exception handling in multiagent systems in detail. We agree with their premise that open systems require fundamentally different exception handling techniques than those required by closed systems. They develop an architectural framework for multiagent systems that uses a directory of agents to keep track of the agent population so that situations such

as the death of an agent can be handled with minimal resource wastage. In another work, Klein (1999) develops a library of generic exception handlers and proposes the use of specialized agents that handle exceptions. This work could be combined with our approach to enable the specialized agents to reason about exceptions.

Exceptions in Workflows

Kamath and Ramamritham (1998) develop a rich workflow model that enables *opportunistic* rollback of tasks that failed, so that the effect of the failure is contained. Their work represents a significant improvement over traditional workflow engines. Chiu *et al.* (2001) describe several exception resolution techniques in ADOME-WFMS, a workflow system. They identify tasks in the workflow as critical, optional, replaceable, or repeatable. Exceptions are handled by modifying task assignments, or skipping a task. Casati *et al.* (1999) develop a library of exception patterns for workflow systems. In related work, Casati and colleagues (1998) architect an environment for designing exceptions. Our approach improves upon these ideas by introducing commitments.

Preferences in Commitment Protocol

One of the first operationalization of commitments for agent interaction was done by Yolum and Singh (2002). In their work, protocols were specified by listing legal states in terms of the commitments and domain propositions that hold at that state, and using an event calculus planner to generate the set of runs that were allowed. Winikoff *et al.* (2004) have advanced this line of research. Fornara and Colombetti (2003) have also proposed a commitment-based interaction protocol framework. However, none of the above approaches specify or operationalize a notion of preferences among the various execution sequences allowed by a protocol. Our work, therefore, is a significant step in this direction.

Preferences in Business Processes

Grosof and Poon (2003) have implemented rule based agent interaction systems where rules are prioritized. Grosof and colleagues propose what are known as *courteous Logic Program*s, or CLPs. In a CLP, when there is ambiguity regarding which rule to fire, i.e., a conflict arising because multiple rules can be fired at a particular state of the world, the priorities assigned to the rules are used to resolve the conflict. Our work is similar to CLPs in this respect, but different in that we propose a scheme in which preferences among *runs* are specified, independent of a protocol

specification. Further, we also present a methodology for translating these preferences into rules that can be embedded into the (rule-based) protocol specification.

Preferences in Agent Interaction

Our work is based on the concept of social interaction among agents, which gives importance only to the observable behavior of agents. We describe how preferences among runs can exist. However, we do not study how agents can reason about the benefits of using one set of protocol runs over another. Pasquier and Chaib-Draa (2003) introduce the cognitive dissonance theory into multiagent communication by incorporating the theory and dialogue game protocols into agent interactions. Their theory explores ways in which agents can decide when to start dialogues with other agents and what kind of dialogues to initiate, among other things. This line of research is complementary to and would strengthen the interaction framework we have presented here. Preferences among the available runs of a protocol have also been studied from the game theoretic point of view by Otterloo *et al.* (2004). They describe a logic that can be used for reasoning about a strategy to adopt in a game when the preferences of other agents in the game are known. The work differs from ours because of the use of games instead of commitment protocols. Also, preferences of agents are assumed to be known by other agents, which does not always apply in real-world applications such as the business interaction we have outlined.

Chapter 5

Protocol Engineering: Developing Commitment Protocols

We developed an algebra for commitment protocols and described its use in creating refinements of existing protocols and in a methodology for modeling and handling exceptions. However, an important question has remained unanswered: *How are commitment protocols created?* In this chapter, we present two novel approaches for creating commitment protocols.

1. By gathering requirements and identifying dependencies among the interacting entities.
2. By identifying reusable components from discourses (which are use-cases) among the interacting entities.

The former approach is motivated by the following observation: modern agent-oriented methodologies deal with the key aspects of software development including requirements acquisition, architecture, and design, but can benefit from a stronger treatment of flexible interactions. On the other hand, commitment protocols declaratively capture interactions among business partners, thus facilitating flexible behavior and a sophisticated notion of compliance. However, they lack support for engineering concerns such as inducing the desired roles and selecting the right protocols. We therefore seek to combine these two directions. For concreteness, we choose the Tropos methodology, which is strong in its requirements analysis, but our results can be ported to other agent-oriented methodologies.

For identifying reusable components from discourses among independent agents, we start from a discourse in which every utterance is annotated by its conversational relationship with other

utterances. Our approach extends Parunak's work on deriving Dooley graphs of discourses annotated by well defined relationships (Parunak, 1996), Singh's work on generating skeletons for individual agents in such discourses (Singh, 2000), and Wan and Singh's work in introducing commitments into annotated discourses to model business processes (Wan and Singh, 2003).

- Commitment protocols are developed by enhancing an existing agent-oriented software engineering methodology, namely, Tropos.
- Commitment protocols are identified from discourses among agents.

5.1 Contributions

In conjunction with the protocol algebra and exception handling schemes we have proposed, the developments presented in this chapter provide a comprehensive treatment of commitment protocols. In addition, the following are our contributions.

- *Engineering from requirements:* With regard to software engineering, our work contributes to both Tropos and the theory of commitment protocols. Through protocols, our approach gives interactions the same status as goals in Tropos. Interactions among independent parties can be captured early and successively refined based on a theory of protocol subsumption. Because of its identification of stakeholders and their goals, dependencies, and plans, Tropos provides a valuable approach in which to identify and refine commitment protocols. We illustrate our approach via an example of a large software system that was developed using Tropos.

- *Engineering from use-cases:* Our contributions include (1) a simplified conceptual model of communicative and other acts including an enhanced set of causal relationships among the acts; (2) a robust, reusable characterization of patterns of communication corresponding to the various commitment operations; (3) an algorithm to generate a graphical representation of a discourse based on commitment operations; and (4) a methodology to synthesize specifications of commitment protocols. The ultimate benefit of this approach is improved automation in protocol design based on a deeper understanding of commitments and communications.

5.2 Commitment Protocols and Tropos

Tropos is an agent-based software methodology that uses the notions of goals, plans to achieve goals, and dependencies among the goals and plans of agents (Bresciani et al., 2004). The

dependencies help capture the relationships between the various stakeholders in the system being engineered. Following i^* (Yu, 1995), Tropos gives prominence to identifying stakeholders and their goals early. Tropos models dependencies among stakeholders well and accommodates their evolution as the goals and plans of the stakeholders are refined. The requirements serve as reminders and guards throughout the development process. However, Tropos does not capture agent interaction requirements in the early stages. Protocols are not identified until the penultimate (detailed design) stage whereas dependencies are defined early. Protocols evolve as the design progresses. Tropos can benefit from an interaction model that allows interactions to be refined with each successive stage of software development. On the other hand, the theory of commitment protocols does not address how interaction protocols and the contexts of their application can be identified in a multiagent system. Tropos can provide cues for identifying protocols because it identifies actors, their goals, their plans to achieve goals and dependencies. Thus, both approaches have their strengths, but also have some limitations, which can be overcome by a synthesized approach.

Our approach is as follows. First, using Tropos, analyze requirements based on dependencies between actors. Second, identify application points for protocols based on dependencies among actors, while respecting the logical boundaries of their interactions. Third, refine protocols based on the actors' goals. Consequently, Tropos provides a rigorous basis for modeling and composing protocols whereas the protocols help produce perspicuous designs that respect the participants' autonomy. We evaluate our approach using a large existing case.

- Tropos identifies goals and dependencies in early stages and refines these in successive stages.
- Tropos can benefit from identifying generic interaction protocols in early stages and successively refining them in later stages.

5.2.1 Background: Tropos by Example

Tropos uses the following key concepts:

Actor. An actor models an entity that has goals or plays a part in the software being developed. Actors are similar to agents or roles, in traditional terminology.

Resource. A physical entity or a piece of information.

Goal. A goal corresponds to an actor's desire. *Hardgoals* are measurable, like functional requirements, whereas *softgoals* are subjective, like non-functional requirements.

Plan. A plan is an abstract description of steps to be taken to achieve a goal.

Dependency. An actor (*depender*) can depend on another (*dependee*) for acquiring a resource, satisfying a goal, or executing a plan. The resource, goal, or plan is the *dependum*. The *reason* for a dependency is a plan, goal, softgoal, or resource (belonging to the depender) for which the depender depends on the dependee.

Tropos uses three methods, all from an actor's perspective, for refining goals and identifying plans to achieve them.

Means-end analysis. identifies plans, resources, or goals (*means*) to satisfy a specified goal or plan (*end*). When a plan is the end, the means can be another plan or a resource, but not a goal.

AND-OR decomposition. breaks up plans into subplans. AND requires all subplans; OR requires one. Likewise for goals and subgoals.

Contribution analysis. identifies the positive and negative impact that a plan, a goal, or a resource may have on the achievement of a goal.

Table 5.1 summarizes the stages of Tropos and how they use the concepts of actor, goal, plan, and dependency.

The eCulture Example

Tropos was used to develop the *eCulture System* for the Trentino provincial government (called *PAT*) (Bresciani et al., 2004). This system provides information about cultural services such as museums to citizens and tourists.

Early Requirements

Figure 5.1 identifies four stakeholders, (top-level actors) in the *eCulture System*. They are, Citizen, PAT, Visitor, and Museum. The figure also shows goals and dependencies these stakeholders have. Citizen has a hardgoal get cultural info and a softgoal taxes well spent, PAT has the hardgoal increase Internet use, Visitor has the softgoal enjoy visit, and Museum has the softgoal provide cultural services. Citizen depends on PAT for the softgoal taxes well spent.

Next, the model of Figure 5.1 is refined via goal and plan analyses. During goal analysis, each goal is either *expanded* into subgoals using AND-OR decomposition, *delegated* to a new or existing actor, or *accepted* by an actor as its own. Tropos performs goal and plan modeling for different actors using label propagation to check that all the root goals, i.e., goals that the modeling

Table 5.1: Tasks performed in modeling actors, dependencies, goals, plans, and capabilities in different stages of Tropos. Within each stage, the different modeling techniques are not ordered.

	1. Early Requirements	2. Late Requirements	3. Architectural Design	4. Detailed Design
Actor Modeling	Identify "top-level" actors actors or stakeholders in domain.	Introduce system as an actor called system-actor.	Decompose system-actor into subactors. Identify all dependencies.	Define agents to model capabilities of system-actor and its subactors.
Goal Modeling	Refine goals using means-end analysis, AND-OR decomposition, and contribution analysis. Find new dependencies.			
Plan Modeling	Refine plans using the three plan analysis methods analogous to goal analysis.			
Dependency Modeling	Identify dependencies between stakeholders using goal modeling.	Model dependencies between system-actor and other actors.	Model dependencies between subactors of the system-actor to identify capabilities.	
Capability Modeling			Identify capabilities of subactors required to handle dependencies with all others.	

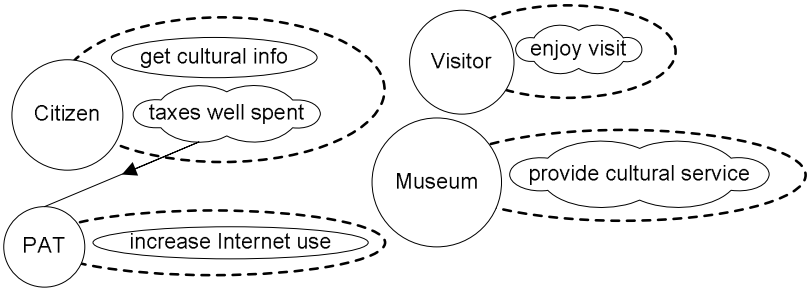


Figure 5.1: Actors identified in early requirements. Actors are circles, their scopes demarcated by dotted ovals. Hardgoals are solid ovals; softgoals are clouds. Dependencies are lines with arrowheads at their center, going from the depender (or from the reason) to the dependee (or to the dependum).

began with, are accepted by some actor. Figure 5.2 shows the partial result of such a goal and plan analysis. The get cultural info hardgoal, which is a root goal for the actor Citizen, is OR-

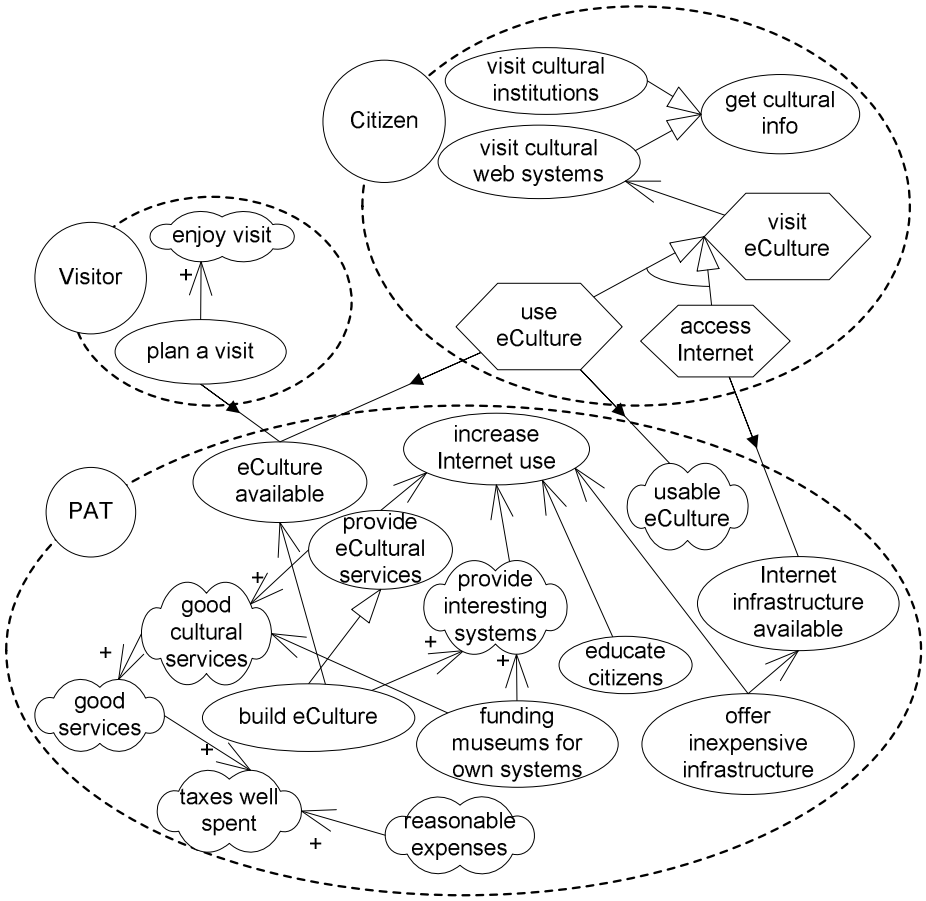


Figure 5.2: Actor model after early requirements. Plans are hexagons; AND decompositions are arrows with empty triangles as arrowheads, with an arc spanning over all the arrows; OR decompositions are similar, but without the spanning arc. Contributions are a $+$ or a $-$ next to an arrowhead; means-end relationships are similar, but without the $+$ or the $-$.

decomposed into two subgoals—visit cultural institutions and visit cultural web systems. Under means-end analysis, the latter subgoal yields the plan visit eCulture as a means. This plan is AND-decomposed into two subplans, namely, use eCulture and access Internet. The softgoal taxes well-spent—the reason for Citizen's dependency on PAT—is delegated to PAT, which accepts it.

Late Requirements

During late requirements, the software system is introduced as an actor, called the *system-actor*. Dependencies between existing actors (stakeholders) and the system-actor are identified, and

goal and plan analyses are performed. Figure 5.3 shows part of the actor model for PAT, Citizen, Museum, and the system-actor eCulture. This figure also shows a part of the goal model for eCulture. For example, PAT depends on eCulture for the softgoal usable eCulture and for the hardgoal provide eCultural services, among others. Goal analysis performed on these goals from the point of view of eCulture results in both goals being adopted by eCulture and decomposed as shown in the goal diagram (within the dotted oval) in Figure 5.3.

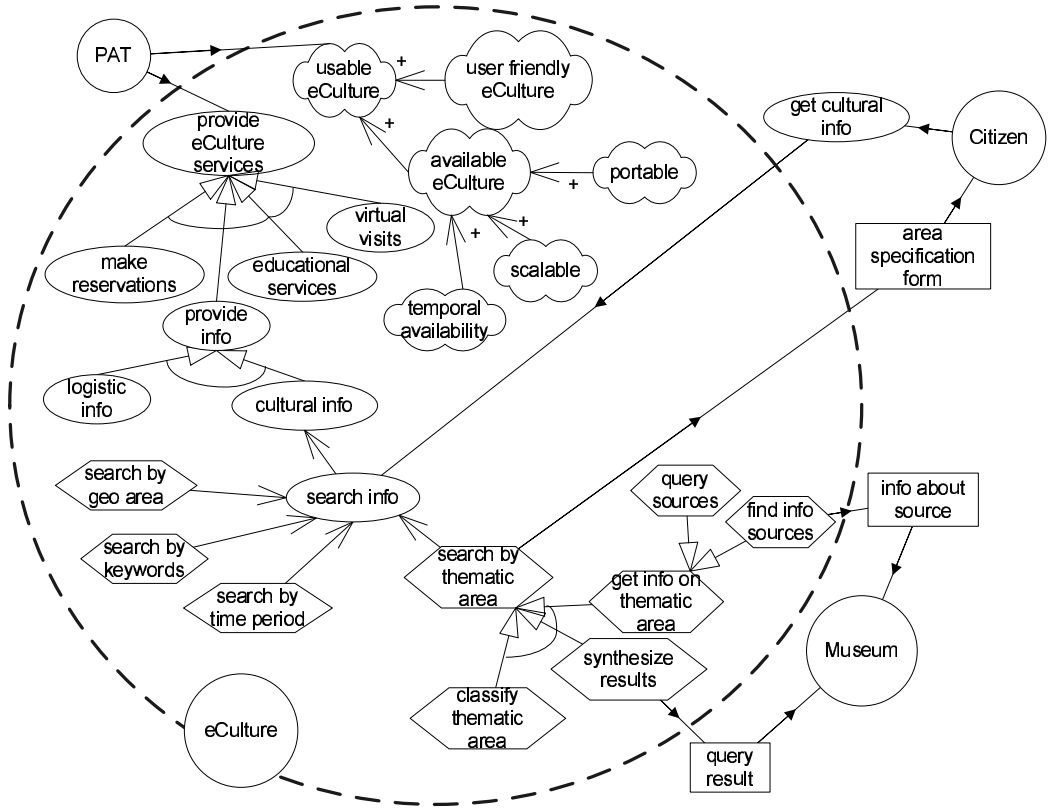


Figure 5.3: Partial actor and goal models showing dependencies of PAT, Citizen, and Museum on the system-actor eCulture. Resources are rectangles.

Architectural Design

During architectural design, eCulture is decomposed into several subactors, including an actor Info Broker introduced to satisfy the goal provide info. Goal and plan analyses are performed after identifying the dependencies between the new subactors and the other actors.

5.2.2 Dependencies in Tropos

We propose the use of commitment protocols in Tropos with actors as agents, and dependencies between actors as the bases of application of these protocols. This section describes intuitions about dependencies in Tropos that are used when developing and applying protocols.

In Tropos, a plan is a sequence of steps that an actor may take in order to achieve a certain goal, and a goal is a state which the actor wants to bring about. Plans are *means* to achieve goals. Plans are *executed*, goals are *achieved*, and resources are *made available*. Nine types of dependencies can exist between actors in Tropos, since dependums on the dependee's side and reasons on the depender's side can be either a plan, a hardgoal, or a resource. These dependency types are shown in Figure 5.4, leading to the following observation about the operational behavior of the dependencies.

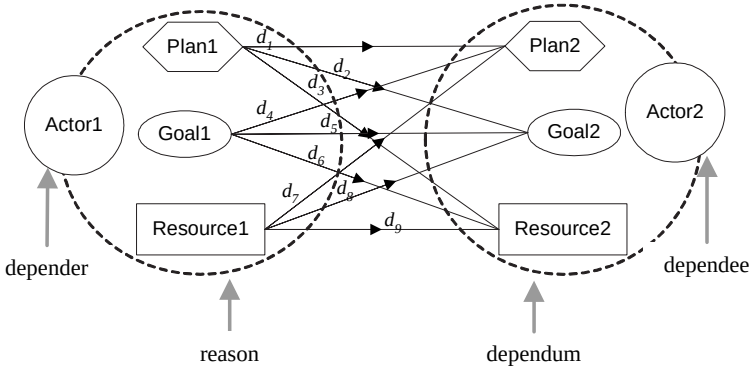


Figure 5.4: Types of dependencies in Tropos

Observation 2. *The reason of a dependency cannot be executed to completion, achieved, or made available till its dependum is executed (at least partially), achieved, or made available.* ▌

For instance, a plan cannot be executed to completion if the goal that it depends on is not achieved. Dependencies can be fulfilled multiple times. For example, the dependency on the resource query result between the eCulture and the Museum is fulfilled every time eCulture makes a query result available to the museum.

Observation 3. *A dependency's reason is an actor's local view of an interaction protocol.* ▌

For example, the access Internet plan of Citizen is the citizen's view of the interaction it has with PAT on the dependum Internet infrastructure available. If a dependency is one of several

dependencies realized by a single protocol, then that dependency is only part of the actor's view of that protocol.

Observation 4. *Outgoing dependencies can be propagated up the hierarchy in AND-decomposition trees.* ▌

Generally, outgoing dependencies from all non-root nodes of an AND tree can be propagated to the root. In essence, a tree can be captured with just its root node as the reason for all its outgoing dependencies. Consider PAT's plan search by thematic area and its AND decomposition tree, as in Figure 5.3. This plan is the reason for the dependency on the resource query result, because synthesize results, a non-root node in the AND tree, depends on query result. The outgoing dependency has therefore been propagated up the tree.

With means-end trees and OR-decomposition trees, since only one of the non-root nodes need to be achieved, executed, or made available, the dependencies cannot always be propagated to the root. Consider PAT's plan find info sources in Figure 5.3. This plan is part of the OR tree with the plan get info on thematic area as its root. find info sources has a dependency on Museum for info about source. This dependency cannot be propagated up to the root plan get info on thematic area because there is an alternative way—query sources—of executing the root plan without involving any dependency. Designers propagate dependencies down the hierarchy as part of Tropos, when goals and plans are refined.

- Actors interact using protocols only if they have dependencies.

5.2.3 Protocols Based on Dependencies

This section provides guidelines for introducing protocols into Tropos using dependencies among actors as the basis.

Guideline 1. *A protocol is required between two actors if and only if at least one dependency exists between them.* ▪

A single protocol might *realize* all associated dependencies between actors. This protocol would be coherent only if the dependencies were somehow related. For example, both the dependencies between eCulture and Citizen shown in Figure 5.2 can be realized by a single protocol since the dependencies are part of a coherent interaction in which Citizen queries and receives information from eCulture. System designers can thus state the relationships between dependencies in terms of interactions between actors.

Conversely, consider actors that have multiple, unrelated dependencies realized by a single protocol. Such a protocol would not be the best design because it combines independent interactions. OWL-P is a framework for describing, composing, and enacting protocols (Singh et al., 2004). The composition makes use of a designer-specified *profile*, which includes axioms specifying correspondences between roles, messages, and data in the protocols being combined. As an example, consider Figure 5.5, which is a part of Figure 5.3. Let the dependency between eCulture and Citizen on get cultural info be realized by an *information transfer* protocol with two roles: *information provider* and *information consumer*. Let the dependency on area specification form be realized by a *form filling* protocol with two roles: *form creator* and *form filler*. These two protocols

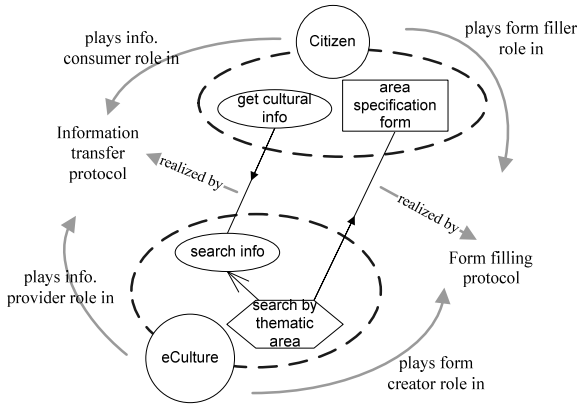


Figure 5.5: Realizing dependencies using one protocol per dependency. Actors play multiple roles.

can be combined by specifying in the composition profile that Citizen plays the roles information consumer and form filler, and eCulture plays the roles information provider and form creator. The composition profile would also specify that the form data be filled before the cultural information is provided. Under such a scheme, a protocol that realizes unrelated dependencies between two actors would not have any composition axioms other than the ones required to bind roles between the protocols. That is, protocols group related dependencies, defining interactions in coherent units rather than as unrelated dependencies.

Guideline 2. *Protocols cannot realize dependencies that have softgoals as dependums or reasons.*

▪

Whereas softgoals can be used by designers to refine protocols, they cannot be realized using protocols since the achievement of softgoals is not objectively verifiable.

Identifying Related Dependencies

Relationships between dependencies can be identified only by the system designer, based on expert knowledge about the stakeholders and actors. However, additional information about potential relationships between dependencies can be obtained from the structure of the AND-OR decomposition and the means-end analysis. Detecting sets of related dependencies corresponds to identifying and demarcating the scope of a protocol. Identifying relationships between dependencies also indicates how a protocol should be designed. Here, we describe the guidelines for identifying related dependencies and how they correspond to protocols.

Guideline 3. *If the means for an end are reasons for dependencies, those means should either be parts of local views of different runs of the same protocol or parts of local views of different protocols that achieve the same interaction. This guideline applies to OR decompositions as well.* ∎

The means for an end are possible ways to achieve or execute the end. If a plan or a goal has many means, any one of them is a way for executing the plan or achieving the goal. If means are reasons for dependencies, then they are an actor's view of a protocol. Therefore, multiple means for a common end provide different views of an actor's involvement in an interaction whose essence is the same: to achieve the end. As an example, consider PAT's goal search info, as shown in Figure 5.3. This goal can be achieved by four means, search by geo area, search by time period, search by keywords, and search by thematic area. All these means are different plans for PAT's view of an information-searching interaction with Citizen. Hence, all these means can be designed as local views of different runs of an information-searching protocol or as local views of runs of different protocols to search for information.

Guideline 4. *If the non-root elements of an AND decomposition are reasons for dependencies, those elements should be parts of the local view of the same protocol.* ∎

Again, the reasoning is that in an AND-decomposition, all non-root elements must be executed, achieved, or made available for the root to be executed or achieved.

Identifying 3-Party Protocols

A protocol is used to realize dependencies, and dependencies in Tropos exist only between two parties. For realistic situations, however, we need to be able to identify 3-party protocols or n-party protocols in general, where $n > 2$. We first note that any n-party protocol can be viewed as

a set of at most $\frac{n(n-1)}{2}$ 2-party protocols with the appropriate composition profile. Therefore, we need an operational definition of what constitutes a *true* n-party protocol. For the purposes of this discussion, we define a true n-party protocol as a protocol which cannot be broken into constituent protocols without any data dependency or temporal ordering among them.

Guideline 5. *If the AND decomposition tree has dependencies, either incoming or outgoing, with two different actors, a 3-party protocol exists between them.* ∎

This guideline is based partly on Observations 2 and 3. Consider the dependencies shown in Figure 5.6 for example. Actor A_0 has an AND tree, shown partially to ignore unnecessary detail. The root of this tree is plan p_1, which has been AND decomposed. Actor A_1 depends on plan p_1 via the dependency d_1. Further, there exist a non-root node p_2 which depends on actor A_2 via the dependency d_2. From Observation 2, we know that p_1 will not be executed to completion until p_2 is. Also, from Observation 3, we know that p_1 and p_2 are local views of some interaction protocol. Therefore, we infer that the protocol that realizes d_1 depends on the protocol that realizes d_2. Therefore, based on our operational definition of a true n-party protocol, the model shown in Figure 5.6 warrants the use of a 3-party protocol. As a more realistic example, albeit a variation

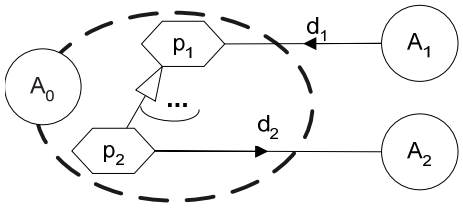


Figure 5.6: Identifying 3-party protocols based on plan dependencies among 3 actors A_1, A_2, and A_3.

of the above, consider the plan search by thematic area belonging to PAT in Figure 5.3. This plan depends on Citizen, and has an AND descendant synthesize results, which depends on Museum. Therefore, this plan cannot be executed to completion without the help of both Citizen and Museum. Therefore, a 3-party protocol can be used here.

Guideline 6. *If a resource belonging to one actor is the dependum for a dependency with a second actor and a reason for a dependency with a third actor, a 3-party protocol exists between the the actors.* ∎

For example, PAT depends on Citizen for the resource area specification form, as shown in Figure 5.3. If Citizen depended on some actor other than PAT for this resource, then a 3-party

protocol would be required because of the data-dependency between the protocols realizing these dependencies.

Refining Protocols

We have shown how protocols can be applied in Tropos models. An advantage of using commitment protocols in Tropos is that protocols can be refined with successive stages of software development. We proposed a protocol-design methodology based on hardgoals and the plans that achieve them. Refinement of these protocols should be based on the softgoals of the participants. In this regard, softgoals are analogous to the private policies of a protocol-participant.

- We proposed guidelines for identifying where to apply commitment protocols in Tropos.

5.3 Protocols from Discourses

Although commitment protocols have been proposed as a flexible model of interactions among agents and the operationalization of commitment protocols has been studied via commitment machines (Winikoff et al., 2004), communicative acts (Kremer and Flores, 2005), and rule-based business process automation frameworks such as OWL-P (Desai et al., 2006), no principled approach exists for developing commitment protocols that are suitable for an application domain. This section provides an approach for deriving reusable commitment protocols from annotated discourses. A discourse is a sequence of utterances between agents, which makes it a use-cases of a multiagent interaction. We base our work on speech-act theory and commitments.

5.3.1 Objectives and Foundations

The concepts of speech-act theory were proposed in Austin's seminal work on the nature of communication (Austin, 1962). Speech-act theory treats utterances as actions on par with physical acts. Further, communications viewed as performatives are acts that are completed merely by an agent uttering them under suitable conditions. For example, a duly authorized official may declare a holiday just by saying so. In the situations of interest to this chapter, communicative acts enable agents to request something of another, commit themselves to bringing about some state of the world, delegate commitments, and so on.

Speech act theory is naturally applied in designing agent interaction models. To this end, sets of communicative and physical acts are designed along with predicates capturing the state of the

world in the domain of interest. The various communicative and physical acts are given a semantics using an appropriate mathematical model. The semantics assigns a meaning to the communicative and physical acts performed by interacting agents by relating the acts to the states of the world and the states of the agents in it. A semantics that associates actions to publicly observable elements of an interaction is verifiable; a semantics that relies upon internal, externally invisible states of an agent is not verifiable.

This chapter takes as its point of departure the research initiated by Parunak on examining and annotating discourses among agents as a means to model the interactions among agents as Dooley graphs (Parunak, 1996). Parunak's work dealt with the semantics of utterances at a conceptual level along with the relationships between them. Based on Longacre's work (Longacre, 1976) on the relationships between communicative acts in a discourse, Parunak motivated additional relationships between communicative acts. Parunak outlined an algorithm to generate a Dooley graph from a discourse annotated with the relationships between utterances. A Dooley graph has the nice property that it places closely related acts structurally close to each other, and places unrelated acts in disconnected components. Thus, a Dooley graph provides a visual representation of a discourse based on the selected relationships between utterances that a modeler wishes to capture.

Parunak's work was subsequently enhanced by Singh (2000), who developed a methodology for extracting the execution skeletons for individual agents participating in a given discourse. Singh's methodology uses domain expertise to generalize agent skeletons, potentially from different parts of a single discourse, thus deriving reusable components from agent discourses.

Huhns *et al.* (2002) developed on Singh's methodology by applying it in a business setting, where state-based skeletons were used by organizations to interact with other independent businesses. Huhns *et al.*also suggested the addition of exception handling techniques in such interaction models. Along similar lines, Wan and Singh (2003) motivated the use of agent conversations and their Dooley graphs in business process modeling and enactment. Their work, in particular, is relevant here since they introduced commitments and relationships among them based on an analysis of a Dooley graph and associated knowledge. Wan and Singh related the theory of communicative acts in agent discourses to a framework for manipulating social commitments among the agents. They described how business processes involving independent organizations can benefit from the flexible, yet verifiable nature of discourses using commitments. Wan and Singh developed Commitment Causality Diagrams (CCDs), which depict the causal relationships between operations on various commitments that are created in the discourse. Their treatment, however, creates processes for specific discourses, not reusable commitment protocols that can be re-combined to demonstrate

valuable alternatives.

Objectives

With respect to the above approaches, we aim to achieve the following:

1. Allow a large variety of conversations to be annotated using a small, principled classification of communicative acts, without introducing additional complexity or structure in the classification of communicative acts or their relationships, and
2. Develop a methodology to extract commitment-level conversations from a discourse. Such conversations can later be re-combined as modular components to generate possible variations of the original discourse, which leads to a generic protocol specification.

Along the way of addressing its main objective, this section revisits previous research on incorporating agent communications into software engineering. Further, it develops an alternative conceptual model of communicative and other acts including an enhanced set of causal relationships among the acts. The proposed model is simpler and enables greater flexibility in combining communicative acts. It provides the basis for a robust, reusable characterization of patterns of communication corresponding to the various commitment operations.

5.3.2 Motivation

Our chief concern is how interaction protocols can be developed for real-world deployments such as for interactions between autonomous businesses. Designing a model for interaction between autonomous agents is an exercise in tradeoffs along several dimensions, such as the degree of autonomy of each agent, the computational complexity of enacting an interaction given a model, and the ease of specification of the model. Therefore, such a design effort requires human expertise in determining the relationships, dependencies, and the organizational structure of the entities in question. Clearly, like other software specifications, the design of protocols ought to follow a sound methodology and general principles, so that the protocols produced are not ad hoc or fragile.

Design methodologies have been studied extensively in software engineering. Requirements gathering for software involves understanding the interrelationships among the various entities being modeled. However, traditional approaches do not consider interactions themselves as first class entities. Our approach differs fundamentally from such approaches in that we think of interactions as worthy of modeling in their own right. Protocols are encapsulations of interactions. Just as

traditional approaches design objects (or rather classes from which objects can be instantiated), our challenge is to design the protocols from which correct interactions can be instantiated at runtime.

Specifically, this section focuses on developing a methodology for designing a commitment-based interaction model given a sample discourse between the interacting entities. This chapter answers the following question: *Given an annotated discourse, how can we identify reusable commitment protocols from it?* Our approach combines automated reasoning about the structure of a given discourse with human expertise to generalize discourses and extract protocols from them.

Contributions

Our major contribution is in developing a methodology for extracting reusable protocols from annotated discourses. Whereas older work has focused on deriving skeletons for individual agents and introducing commitments into discourses, our work enables the generalization of discourses by identifying their constituent protocols.

5.3.3 Communicative Acts and Relationships

Many important terms of interest to this section have been used in different ways in the literature. To make this chapter easy to understand, it is helpful to define the terms as used here.

Act. Actions, both communicative and physical, that have an interactional import. The local hidden actions of an agent are not reflected in the representations and reasoning pursued here.

Discourse. A specific sequence of acts performed by interacting agents. A discourse is annotated based on a designer's knowledge to capture the relationships among the various acts in it. A discourse is not to be confused with *conversation*, which has a technical meaning and usually refers to a part of a discourse.

Role. A role is one participant of the discourse. Each role is played by a single agent.

Character. A part played by one role in a conversation. A character can be involved in multiple conversations.

Conversation. A sequence of acts performed by exactly two characters. Conversations are identified from the Dooley graph generated from an annotated discourse, and are therefore a product of the nature of the algorithm that generates the Dooley graph.

Dooley Graph. A graph representing the conversational relationship among the utterances in a discourse.

Communicative Acts

Figure 5.7 shows the classification of communicative acts as given by Parunak. The CANCEL act was added by Wan and Singh.

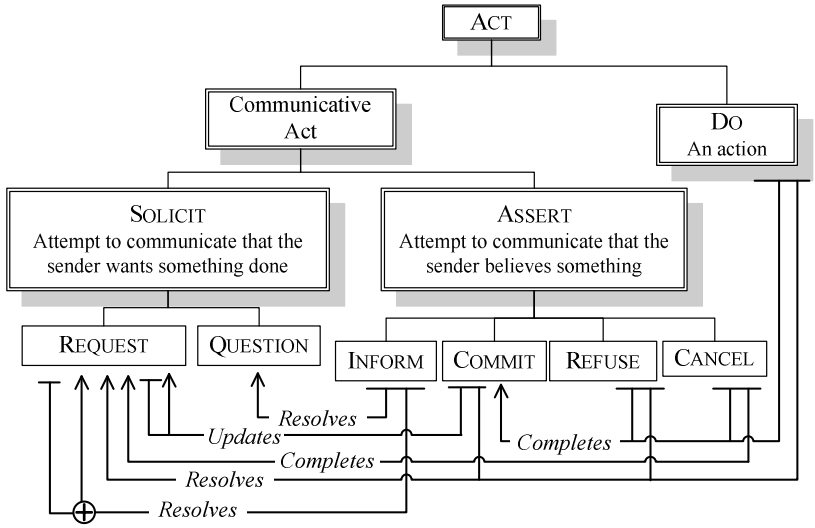


Figure 5.7: Parunak's classification of communicative acts, arrows (beginning at a horizontal line and ending at an arrowhead) showing their possible sequential relationships due to Longacre (*Reply*, *Resolve*), Parunak (*Respond*, *Complete*), and Wan and Singh (*Update*). *Reply* and *Respond* are omitted for clarity; they may relate any act to any other. CANCEL was introduced by Wan and Singh. The $\oplus$ symbol indicates that an utterance of type REQUEST and INFORM resolves an utterance of type REQUEST.

- To SOLICIT is to ask the receiver something.
  1. To REQUEST is to solicit the receiver to commit to the sender to do something.
  2. To QUESTION is to solicit the receiver to inform the sender about some proposition.
- To ASSERT is to attempt to convey some proposition to the receiver.
  1. To INFORM is to tell the receiver something.
  2. To COMMIT is to be committed to do something that the receiver wanted.
  3. To REFUSE is an tell the receiver that the sender does not commit to something that the receiver wanted.

4. To CANCEL is to tell the receiver that the sender is rescinding a particular commitment that it had made to the receiver. Parunak used the REFUSE act to represent utterances agents use to cancel commitments they have already made, unlike Wan and Singh, who introduced the CANCEL act for the same purpose, but to aid in the design of commitment causality diagrams.

- To PROPOSE is to propose to the receiver that the sender wishes to commit to it. PROPOSE is a combination of an INFORM (about its willingness to commit) and a REQUEST (that it be allowed to commit).

We modify this classification, and use only a subset of it. Our changes are motivated and justified by the concerns described below.

Agent Autonomy

In Wan and Singh's algorithm for generating commitments (and consequently the commitment causality diagram) a REQUEST act creates a conditional commitment with the receiver of the utterance as the debtor. For instance, a passenger's request to a travel agent to book an itinerary creates a commitment from the travel agent to the passenger. For an agent to unilaterally create a commitment in which another agent is the debtor violates the autonomy of the debtor. If the commitment created is conditional, the debtor's autonomy might still be preserved, but Wan and Singh's approach unilaterally creates unconditional commitments as well. We therefore want to structure conversations in which a commitment is proposed and the would-be debtor agrees to take on the commitment. We therefore utilize the PROPOSE act type for proposing commitment to an agent, commitment from an agent, or any other commitment operation. In keeping with our aim of enabling the most general interactions (so that they can be constrained to fit specific domains), we want to have each role confirm whether it agrees with a proposal, be it a proposal for a commitment to be made to it or by it or a proposal to execute a commitment operation. Therefore, a PROPOSE proposes a commitment or a commitment operation, and an ACCEPT (which we shall introduce shortly) or a REFUSE creates the commitment(or agrees to the operation) or refuses the commitment (or the operation). A commitment holds only after an ACCEPT utterance has been sent in reply to its proposal. From this point of view, both the ACCEPT and the REFUSE are kinds of INFORM, where the sender is communicating some fact to the receiver.

Separation of Concerns

We wish to separate the communicative acts from the concepts of commitments. Commitment operations such as creation and cancellation should be brought about by a well-defined sequence of communicative acts and the conversational relationships among them. Therefore, we remove the CANCEL and the COMMIT act type from the classification.

Roles and Peers

The term *role* has been used in different contexts in (Singh, 2000) and (Wan and Singh, 2003). Singh follows Parunak's convention, defining each distinct (i.e., distinctly named) sender or receiver in an annotated discourse as a role, which is the definition we adopt. In (Wan and Singh, 2003), however, a role is defined as " *The abstraction of capabilities used by characters who deal with same type of transactions or are involved in context related conversations.*"

This definition puts senders and receivers with different names into a single role. For instance, consider a customer C in a discourse in which it converses with two different bookstores B_1 and B_2. C rejects B_1s quote and accepts B_2s quote. Wan and Singh assume implicitly that an annotated discourse identifies B_1 and B_2 as being agents that offer the same services, and club them under the same role. Since their motivation for using Dooley graphs of discourses is different from ours, Wan and Singh's assumption that the identification of roles (under their definition) has been done, possibly by the annotator of the discourse, is justified. Our approach,however, aims to derive *reusable* protocols from a discourse in which only the utterances of the participants and the conversational relationships among these utterances are known. Reusability here means the ability to apply the same protocol with a different set of participants. To reuse a protocol therefore requires that agents playing the same role have the same capabilities, offer the same services, and be able interact similarly. We define such agents as *Peers*, and describe in Section 5.3.6 how peers can be identified.

With the above in mind, we propose a classification of communicative acts as shown in Figure 5.8. Here, we have removed ASSERT, COMMIT, and CANCEL. Communicative acts are of two general types, SOLICIT and INFORM. A QUESTION is a kind of solicitation, and so is a PROPOSE. However, the proposal is also a kind of INFORM, i.e., a proposal is an inform in that it states what is being proposed, and it is a solicit in that it asks if the proposal will be accepted. Under the INFORM act type, we introduce ACCEPT and keep the older REFUSE. The conversational relationships that exist in Parunak's scheme and the ones we have introcuded are explained in the

next section.

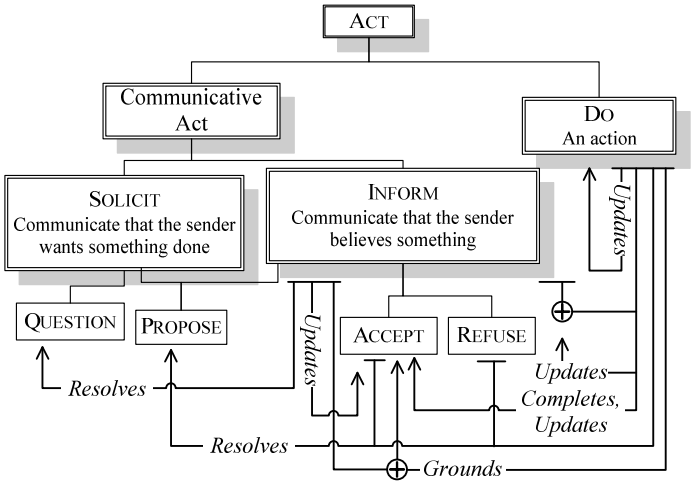


Figure 5.8: Our proposal for a classification of communicative acts as related to agent interaction based on social commitments. Arrows show sequential relationships that we propose.

Conversational Relationships

Longacre (1976) identified a couple of sequential relationships that can exist between communicative acts in a discourse, *Reply* and *Respond*, to which Parunak added *Resolve* and *Complete*, and Wan and Singh added *Update*. These are called *sequential conversational relationships*. Figure 5.7 shows such relationships by means of arrows, indicating which act types *can* be related to which others. Whether utterances in a discourse *are* related depends on the annotator of the discourse.

- *Respond*: Indicates a causal relationship between messages. If a message i is sent by a role a as an effect of a message j that it received, and i was the first such message, then i is said to *Respond* to j. Any type of communicative act can *Respond* to any other type.

- *Reply*: Indicates a causal and reciprocal relationship between two messages exchanged between two agents. If j *Respond*s to i, and j is sent to the sender of i, then j is said to *Reply* to i. Any type of communicative act can *Reply* to any other type. A *Reply* is also a *Respond*.

- *Resolve*: Indicates a desired response to certain communicative acts. A desired response is one that carries the conversation on to its completion, successful or otherwise. In our scheme, an explicit ACCEPT, a REFUSE, or an implicit accept by performing an action resolves a PROPOSE. An INFORM resolves a QUESTION. In Parunak's scheme, a REQUEST is resolved by

a COMMIT (since requests made by the would-be creditor), a REFUSE or by another counter proposal. A *Resolve* is also a *Reply* and, consequently, a *Respond* as well.

- *Complete*: Indicate the conclusion or closing of an act and its effects. In our scheme, an action that performs what was promised in a commitment or what was proposed and accepted completes the proposal. In Parunak's scheme, a DO or REFUSE can complete a COMMIT by fulfilling it or cancelling it respectively. Further, an utterance i cannot both *Complete* and *Resolve* another utterance j because i and j are required to have the same sender (and the same receiver) if i completes j, and their sender and receiver are required to be exchanged if i resolves j (because a resolve is also a reply). In our scheme, we drop the restriction on *Complete*, i.e., an utterance can complete and resolve another. Wan and Singh use CANCEL to cancel the commitment, thereby completing the REQUEST that led to the creation of the commitment.

- *Update*: Indicates that the sender agent has modified a REQUEST or a COMMITment that it made, but the commitment or the request still stands. An *Update* does not change who is committed to whom.

We introduce a new relationship, *Ground*, defined as follows: An utterance i grounds j if i converts the conditional commitment created by j (by accepting a proposal for creation of a commitment) into an unconditional commitment. This new relationship is needed to capture the relationship between an utterance that does not resolve or complete a commitment, but changes a conditional commitment to an unconditional one.

Commitments Revisited

A very important characteristic of our definition of a commitment is that the fulfillment condition, antecedent and consequent, are not directed, i.e., nothing is specified about who performs actions or sends utterances. However, basing interaction on communicative acts requires that the debtor of the commitment utter or do the fulfillment condition and this utterance (or action) is directed towards the creditor. For conditional commitments, the creditor has to utter the antecedent and the debtor, the consequent.

Commitment Operation Patterns

We describe an operationalization of commitment operations using the communicative acts described above. In the following, x, y, z, and a are roles, $c_1 \equiv \mathsf{CC}(x, y, p \rightsquigarrow q)$, $c_2 \equiv \mathsf{CC}(y, x, q \rightsquigarrow p)$, $c_3 \equiv \mathsf{CC}(x, a, p \rightsquigarrow q)$, $c_4 \equiv \mathsf{CC}(a, x, p \rightsquigarrow q)$, and $c' \equiv \mathsf{CC}(z, y, p \rightsquigarrow q)$.

create(c_1)

A commitment can be created, i.e., made active, by an explicit *create*$(\cdot)$ act, or because of the delegation or assignment of another commitment. For an explicit *create*$(\cdot)$, either the creditor y or the debtor x of c_1 can initiate the creation by proposing the create, as shown in the CreateP pattern in Figure 5.9. The pattern for a creditor-proposed commitment is shown in Figure 5.10 for a debtor-proposed commitment in Figure 5.11. Once a create is proposed, the proposal can be accepted, thereby activating the commitment, or refused, thereby completing it. To create an unconditional commitment, it is sufficient to set p to mean that y need not perform any action, i.e., to set $p = \mathsf{true}$.

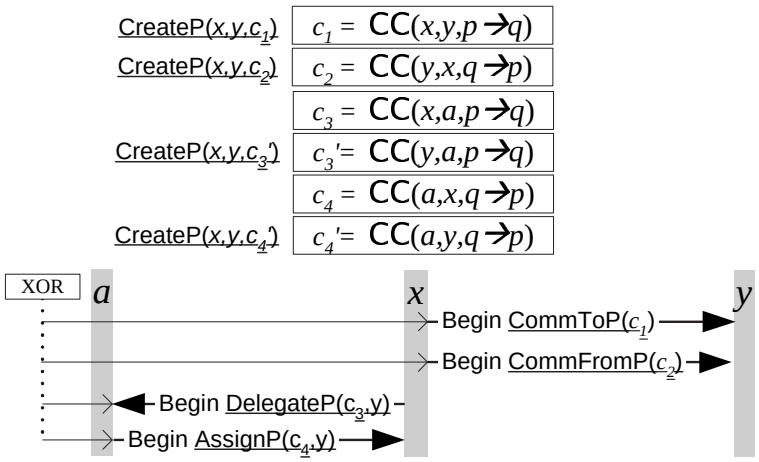


Figure 5.9: The pattern to create a commitment, either by proposing the create or by the delegation or assignment of another commitment. The dotted vertical line under the XOR box denotes that exactly one of the operations linked by horizontal arrows to the dotted line is performed.

discharge(c_1)

A conditional commitment such as c_1 can be discharged by its debtor x by performing q after y performs p. However, in some cases, x may not wait for p and perform q. This is shown in Figure 5.12. Note that *create*$(\cdot)$ and *discharge*$(\cdot)$ are not proposed and performed like other

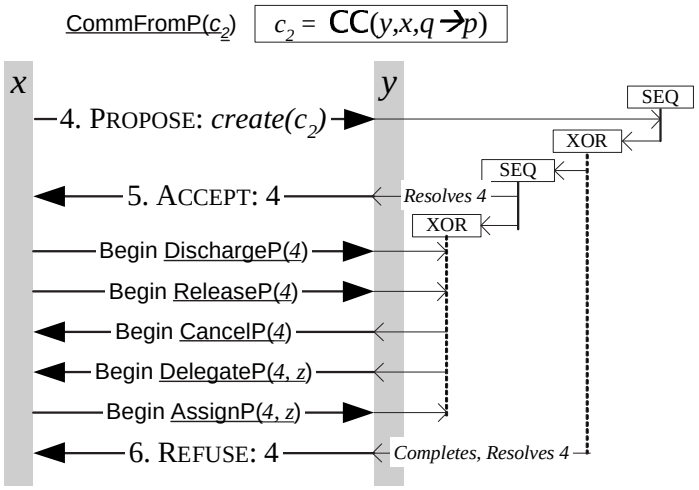


Figure 5.10: The pattern for a creditor to propose creation of a commitment. The vertical line under the SEQ box denotes that all the operations linked by horizontal arrows to the vertical line are performed in sequence, starting with the topmost operation.

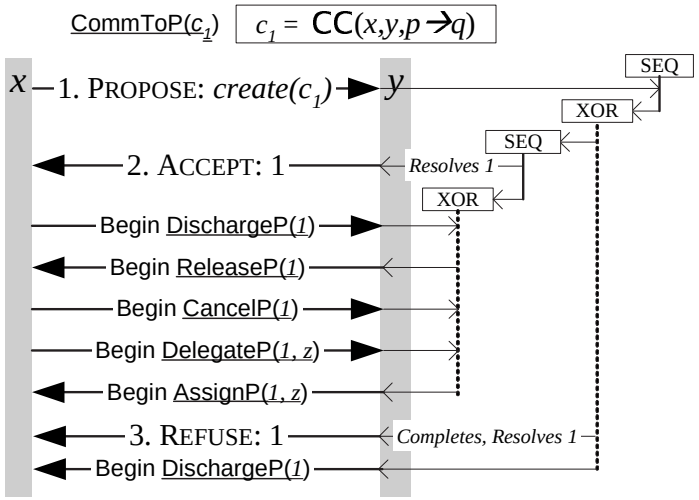


Figure 5.11: The pattern for a debtor to propose creation of a commitment.

commitment operations. The debtor does not propose to discharge a commitment and wait for an acceptance to the proposal before performing it. Also, the debtor does not perform a *create*(c_1) action; acceptance to a proposal to create suffices. Only active commitments can be discharged.

cancel(c)

In real-world settings, unilateral cancellations of commitments can lead to unpredictable results, subverting commitments. Metacommitments can be employed to enforce restrictions on

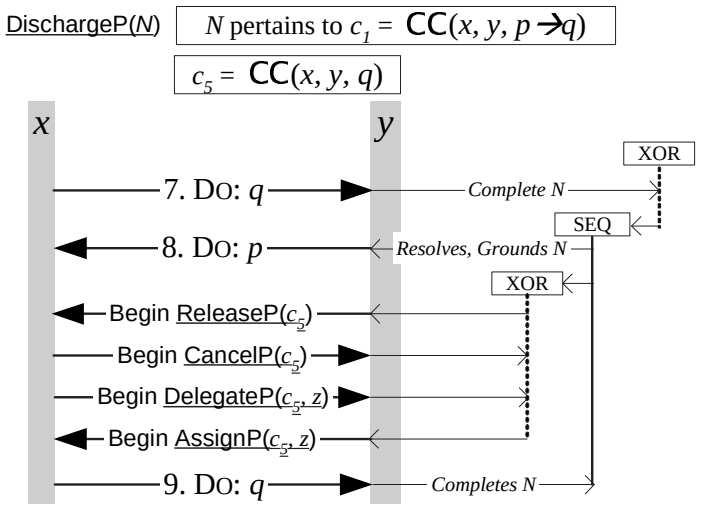


Figure 5.12: The pattern for discharging a commitment.

when cancellations can occur, but such specifications tend to be rigid, since they are determined prior to an actual discourse. We wish to allow the creditor to decide at runtime if a it will allow a cancel. Therefore, the cancellation of a commitment c_1 is proposed, usually (but not necessarily) by the debtor x to the creditor y. If y agrees, x cancels the commitment, and c_1 becomes an inactive commitment. Alternatively, y can refuse x's proposal, thereby keeping c_1 unchanged. This pattern is shown in Figure 5.13. If the creditor y proposes the cancel to the debtor x, x can accept the proposal and then cancel the commitment, cancel the commitment without an explicit acceptance, or refuse the proposal. Only active commitments can be cancelled.

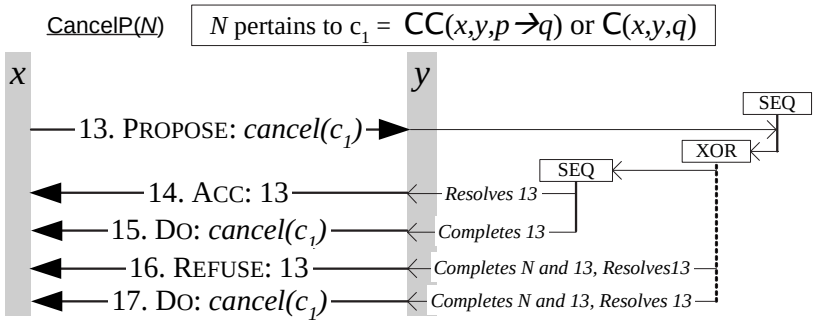


Figure 5.13: The pattern for cancelling a commitment.

release(c_1)

A release pattern in its most generic form is similar to a cancel pattern. The creditor y proposes the release to the debtor x, who can accept or refuse it. If the debtor accepts, the creditor will release him, otherwise the commitment is unchanged. However, the unilateral release of a commitment by the creditor is desirable in a large number of real settings. Therefore, a creditor-initiated release pattern will involve just one utterance releasing the debtor. However, the debtor x can propose the release to the creditor y, in which case y replies with an accept and then releases x, or refuses x's proposal. This case is analogous to the cancel pattern. Only the debtors of active commitments can be released.

delegate(c_1, z)

Typically, the debtor x initiates a delegate by proposing it to both the creditor y and the new, proposed debtor z. If both of them accept the proposal, x does the delegate, which is the act which activates c' and inactivates c_1. This is the typical delegate pattern, as shown in Figure 5.14. Many alternative scenarios are possible, based on the fact that x has to confirm that both y and z agree, and all utterances are not sent to all roles:

D_1. If the creditor y proposes the delegate to the debtor x, then x has to confirm only with z (by proposing the delegate to z and receiving an acceptance from z) before accepting y's proposal (or simply delegating the commitment).

D_2. If z proposes the delegate to x, the pattern is similar to the one in D_1. above, with y and z exchanging their roles.

D_3. If z proposes the delegate to y, y can send a proposal for a delegate to x and send proof of z's proposal along with it. In this case, if x is willing to delegate, it will, sending utterances to both y and z about the delegate. If y proposed the delegate to x without proof of z's proposal, x will confirm with z by either questioning z or by proposing the delegate to z.

D_4. If y proposes the delegate to z, the pattern is similar to the one in D_3. above, with y and z exchanging their roles.

Only active commitments can be delegated.

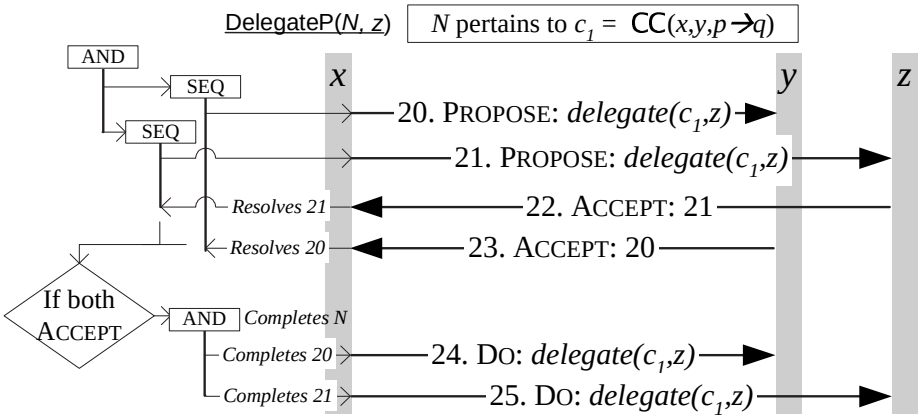


Figure 5.14: The pattern for delegating a commitment. AND denotes all actions lined to its by horizontal arrows have to be performed, but in no particular order.

assign(c_1, z)

An *assign* pattern is analogous to a delegate pattern, with y, the creditor having the responsibility of verifying that both x and z agree with the proposal to assign the commitment. Only active commitments can be assigned.

update(x, c_1, p, p')

Wan and Singh introduced the notion of *updating* a commitment by changing either its antecedent or consequent, keeping the creditor and debtor unchanged. We cast update as a pattern in which either the creditor x of c_1 or the debtor y proposes the update to the other. The proposal may be for changing either c_1's antecedent p to p' or for changing its consequent q to q'. Considering that only an active commitment can be updated, the following cases are possible

U_1. x proposes to change p to p'. y can refuse the proposal, resolving it. Alternatively, if p has not been performed yet, i.e., if c_1 has not be grounded, y accepts the change, resolving the proposal. If y has already performed p, then y can explicitly accept the proposal (resolving it), and perform p' (updating the earlier performance of p), or just perform p' without explicit acceptance.

U_2. x proposes to change q to q'. y can refuse and thereby resolve the proposal. If y accepts the proposal, it updates the utterance that made c_1 active.

The design considerations for designing commitment protocols broadly fall into two levels:

- The design of commitment operations using the relationships among communicative acts. This commitment level is considered the lower level, since the behavior of commitments is uniform across domains, which may add upper level details to tailor commitment operations.
- The design of protocols incorporating domain knowledge and expertise to constrain or otherwise structure the behavior of commitment operations. For example, the travel agent might not entertain requests to change an itinerary once it is made.

Our approach aims to automate as much as possible the derivation of protocols from annotated discourses using commitments as an abstraction to infer details about the discourse that may be domain specific. To illustrate our approach, we apply our methodology to an example developed by Wan and Singh.

5.3.4 The Travel Planning Example

We consider the travel planning example discourse to illustrate the concepts and involved in a Dooley graph generated from annotated discourses and the advantages of our contributions. This discourse is shown in Table 5.2. In this table, S is the sender and V the receiver of a message which is identified by a number, given in the column #. In this discourse, a passenger P asks travel agent T to create an itinerary with a flight, a car rental, and a hotel room. T contacts airline A_1, hotel H_1 and rental car company R. Airline A_2 is contacted when A_1 informs that it has no tickets. Once A_2, R, and H_1 confirm reservations, T informs P of the itinerary. P asks for the rental car to be removed. T get this done via R, after which H_1 withdraws the room reservation. T asks P if an alternative room will do. When P agrees, T books a room with hotel H_2. P then pays T what was promised for the itinerary and T pays A_2 for the airline ticket.

Table 5.2: The annotated discourse for the trip planning example. The sequential conversational relationships used to annotate utterances are *Respond*(Rsp), *Reply*(Rpl), *Resolve*(Rsl), *Complete*(Cml), and *Update*(Upd).

#	S	V	Act Type	Content	Rsp	Rpl	Rsl	Cml	Upd
1	P	T	REQ	I'll pay you x if you send me itinerary i					
2	T	A_1	REQ	I'll pay you x_1 if you send me ticket t for P	1				

Continued on next page

Table 5.2 – continued from previous page

#	S	R	Act Type	Content	Rsp	Rpl	Rsl	Cml	Upd
3	T	H_1	REQ	P will pay you x_2 if you make a room reservation r for P	1				
4	A_1	T	REF	I have no tickets	2	2	2		
5	T	A_2	REQ	I'll pay you x_1 if you send me ticket t for P	4				
6	A_2	T	COM	Here's ticket t for P	5	5	5		
7	T	R	REQ	P will pay you x_4 if you make a car reservation c for P	1				
8	H_1	T	COM	Reservation r done for P	3	3	3		
9	R	T	COM	Reservation c done for P	7	7	7		
10	T	P	COM	Here's itinerary i	6, 8, 9	1	1		
11	P	***T***	REQ	Please remove the car reservation from i					1
12	T	R	CAN	Please cancel reservation c	11			7	
13	T	P	COM	Here's the updated itinerary i	11	11	11		10
14	H_1	T	CAN	Room r cancelled				8	
15	T	P	QUE	Will an alternative room do?	14				
16	P	T	INF	Yes	15	15	15		
17	T	H_2	REQ	P will pay you x_2 if you make a room reservation r' for P	16				
18	H_2	T	COM	Reservation r' done for P	17	17	17		
19	T	P	COM	Here's the updated itinerary i	18	16	16		13
20	P	T	DO	Here's x I owe you for itinerary i	19	19		1	
21	T	A_2	DO	Here's x_1 I owe you for ticket t	6, 20	6		5	

5.3.5 Generating Dooley Graphs

The first step in the identification of protocols from discourses is to represent the discourses in a suitable representation, i.e., Dooley graphs. The Dooley graphs for the travel planning discourse is shown in Figure 5.17. This graph was obtained by following the algorithm described by Singh (2000).

Singh's Algorithm

Singh's algorithm creates one vertex for every utterance sender and one for every utterance receiver. The former belongs to a *sender set* and the latter vertex to the *receiver set*. For instance,

since utterance 5 is sent by T, a and received by A_2, vertex $T5$ is created in the sender set and a vertex $5A_2$ is created in the receiver set. Next, a vertex s from the sender set is connected to a vertex r from the receiver set if any of the following 4 conditions hold: 1) s replies to r, 2) r resolves s 3) r is the last utterance in the discourse and replies to s, or 4) r completes t, and t resolves s. The connected components in the resultant graph relate characters. The algorithm is listed in Figure 5.15.

For example, since 6 replies to 5, the sender set vertex A_26 is connected to the receiver set vertex $5A_2$. Further, since 21 is the last utterance and it replies to 6, the receiver set vertex $21A_2$ is connected to the sender set vertex A_26. This construction makes A_26, $5A_2$, and $21A_2$ a connected component, which is seen in the Dooley graph in Figure 5.17 as a character A_2. In a similar fashion, $T21$ replies to $6T$, and $6T$ resolves $T5$, creating a character for T named T_6. The graph construction is shown in Figure 5.16. To illustrate the changes we propose to Singh's algorithm, we present in Table 5.3, the travel planning example annotated with the new classification of acts. This table introduces three utterances, 12a, 12b, and 14a. Using 12a, the travel agent T proposes that the rental car company R cancel its car reservation, rather that cancelling it via one utterance. This illustrates our propose-accept pattern for commitment operations. Using 12b, R responds to T's request for cancellation by performing the cancellation. Using 14a, H proposes to T the cancellation of its hotel room reservation commitment, which T accepts in 14b, thereby resolving 14a. When H cancels 14c, it completes 14a.

Table 5.3: The trip planning example annotated under the new communicative act classification. The sequential conversational relationships used to annotate utterances are *Respond*(Rsp), *Reply*(Rpl), *Resolve*(Rsl), *Ground*(Gnd), *Complete*(Cml), and *Update*(Upd).

#	S	V	Act Type	Content	Rsp	Rpl	Rsl	Gnd	Cml	Upd
1	P	T	PRO	I'll pay x if you send me itinerary i						
2	T	A_1	PRO	I'll pay x_1 if you send me ticket t for P	1					
3	T	H_1	PRO	P will pay you x_2 if you make a room reservation r for P	1					
4	A_1	T	REF	I have no tickets	2	2	2		2	
5	T	A_2	PRO	I'll pay x_1 if you send me ticket t for P	4					
6	A_2	T	DO + INF	Here's ticket t for P	5	5	5	5		
7	T	R	PRO	P will pay you x_4 if you make a car reservation c for P	1					

Continued on next page

Table 5.3 – continued from previous page

#	S	V	Act Type	Content	Rsp	Rpl	Rsl	Gnd	Cml	Upd
8	H_1	T	DO + INF	Reservation r done for P	3	3	3	3		
9	R	T	DO + INF	Reservation c done for P	7	7	7	7		
10	T	P	DO + INF	Here's itinerary i	1, 6, 8, 9	1	1	1		
11	P	T	PRO	Please remove car reservation from i	10	10				
12a	T	R	PRO	Please cancel reservation c	11					
12b	R	T	DO + INF	Reservation c cancelled	12a	12a	12a		12a	
13	T	P	INF	Here's the updated itinerary i	11	11	11			10
14a	H_1	T	PRO	Can I cancel the room reservation?	8					
15	T	P	QUE	Will an alternative room do?	14a					
16	P	T	INF	yes	15	15	15			
14b	T	H_1	ACC	Ok to cancel room reservation	14a	14a	14a			
14c	H_1	T	DO + INF	Room r cancelled	14b	14b	14b		14b	
17	T	H_2	PRO	P will pay you x_2 if you reserve room r' for P	16					
18	H_2	T	ACC	Reservation r' done for P	17	17	17			
19	T	P	DO + INF	Here's the updated itinerary i	16, 18	16	16		16	13
20	P	T	DO	Here is x I owe you for itinerary i	19	19			10	
21	T	A_2	DO	Here is x_1 I owe you for ticket t	6, 20	6			6	

Our Algorithm

We modify Singh's algorithm as follows: After constructing the sender and receiver sets, we connect vertices if they relate to the same role, and one resolves, grounds, or completes the other. This relates utterances within a commitment operation to one character. Our algorithm therefore, changes only the ConnectVertices function, as shown in Figure 5.18. Using this algorithm, we

input : A discourse $D = \langle u_1, u_2, \ldots u_n \rangle$
output: A Dooley graph of D

$g \leftarrow$ `generateVertices` (D) ;
$g \leftarrow$`connectVertices` (g) ;
$C \leftarrow$ `generateCharacters` (g) ;
$g \leftarrow$ `connectCharacters` (C) ;
return g ;

Procedure `generateVertices` (D): Create the sender and receiver vertices for a discourse $D = \langle u_1, u_2 \ldots u_n \rangle$
$g \leftarrow \{\}$;
foreach *utterance* u *in* D **do**
    create a sender vertex v_s and receiver vertex v_r ;
    v_s.uttr $\leftarrow u$;
    v_r.uttr $\leftarrow u$;
    add v_s and v_r to g ;
return g ;

Procedure `connectVertices` (g): Connect sender-receiver vertices in g
foreach *sender vertex* v_s *in* g **do**
    foreach *receiver vertex* v_r *in* g **do**
        if $v_r == v_s$.*uttr.RplTo* $\vee$ $v_s \in v_r$.*uttr.Rslv* $\vee$ ($v_r == u_n \wedge v_s$.*uttr* $\in v_r$.*uttr.RplTo*) **then**
            connect v_s and v_r ;
        foreach v_t.*uttr* $\in v_r$.*Cmplt* **do**
            if v_s.*uttr* $\in v_t$.*uttr.Rslv* **then**
                connect v_s and v_r ;
return g ;

Procedure `generateCharacters` (g): Generate characters from a sender-receiver vertex graph g
$C \leftarrow \{\}$;
foreach *connected component* g' *in* g **do**
    foreach *role* r *involved in* g' **do**
        Create a new character r_x pertaining to r ;
        Add vertices in g' pertaining to r to r_x ;
return C

Figure 5.15: GenerateDooleyOld(D): Singh's algorithm (Singh, 2000) to generate the Dooley graph for a discourse $D = \langle u_1, u_2, \ldots u_n \rangle$, where u_is are utterances.

obtain a Dooley graph as shown in Figure 5.19.

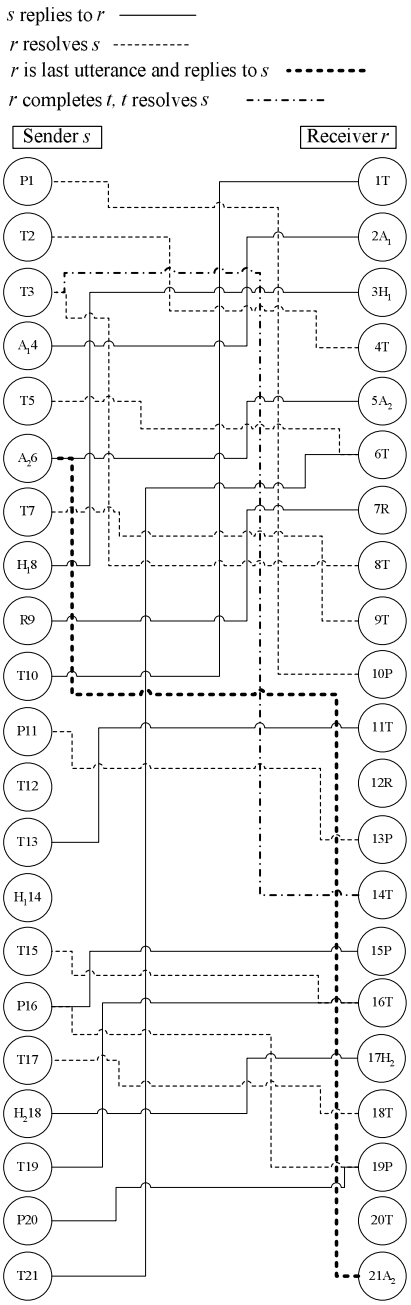


Figure 5.16: The bipartite graph used in the construction of the Dooley graph of the travel planning discourse shown in Table 5.2, using Singh's algorithm.

5.3.6 Reasoning About Conversations

Generating the Dooley graph of a discourse was the first step in identifying reusable protocols. In this section, we describe how we can create protocols from the Dooley graph. Singh's

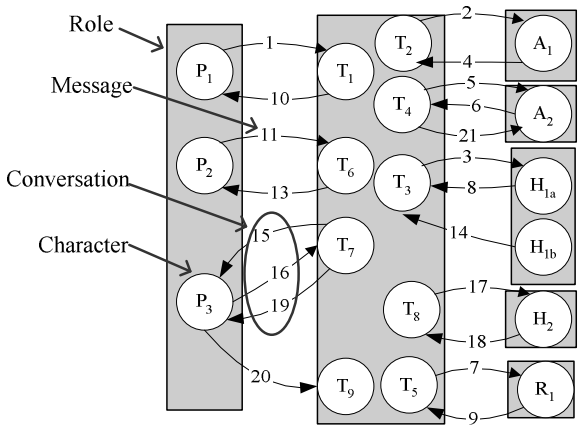


Figure 5.17: The Dooley graph of the travel planning example shown in Table 5.2 generated using Singh's algorithm.

```
1 Procedure connectVertices (g): Connect sender-receiver vertices in g
  based on Resolve, Ground, and Complete;
2 foreach sender vertex v_s in g do
3     foreach receiver vertex v_r in g do
4         if v_r ∈ v_s.uttr.Rslv ∨ v_s ∈ v_r.uttr.Rslv ∨ v_r ∈ v_s.uttr.Grnd ∨
          v_r ∈ v_s.uttr.Grnd ∨ v_r ∈ v_s.uttr.Cmplt ∨ v_s ∈ v_r.uttr.Cmplt then
5             connect v_s and v_r ;
6 return g ;
```

Figure 5.18: ConnectVertices(g): Connect vertices in the sender and receiver sets

work on deriving agent skeletons from Dooley graphs was analogous to our effort here. However, there are fundamental differences in our approaches and motivations. Whereas Singh's approach is based on expertise and knowledge of the discourse, we want to automate as much of the protocol identification process as possible. We therefore use commitments as they are a domain independent abstraction. Specifically, our effort automates the identification of peers, which was either assumed or ignored by older work.

Every conversation is a protocol that can be reused in other situations. However, due to our construction, each conversation will describe one commitment operation. This does not capture larger, richer structures in a conversation. We therefore have to leverage our knowledge of commitments and annotations to detect richer protocols. Briefly, our approach is as follows:

- Aggregate characters in the Dooley graph based on the commitments they operate upon.
- Identify peers in the Dooley graph.

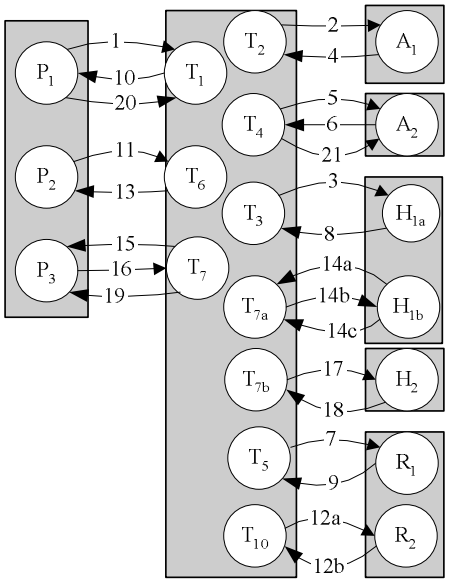


Figure 5.19: The Dooley graph of the travel planning example in Table 5.3, generated using our algorithm.

- Based on the lifecycle of commitments, identify required and optional parts of conversations.
- Using the information from the above three steps, identify protocols.

Character Aggregation via Commitment Operations

To provide an informative view of conversations in a Dooley graph, we can combine characters based on the commitments that their conversations relate to. We *aggregate* two characters belonging to a role if those characters relate to the same commitment, one of them participated in the conversation that made the commitment active, and the other character initiated a commitment operation on the commitment. For example, P begins a new conversation with T by uttering 11 as P_2. Since this conversation relates to the commitment created by character P_1, we aggregate the two characters. Similarly, since the hotel H_1 begins a conversation as H_{1b} with T, and this conversation relates to the commitment created by H_{1a}, we combine H_{1a} and H_{1b}. Similarly, T_7 can be combined with T_1. Figure 5.20 shows the Dooley graph after performing character aggregations.

Peer Identification via Conversation Similarity

Once characters are aggregated, we need to identify *peers*, i.e., roles in the discourse that are conversationally similar to each other and different from other roles. We identify peers based on

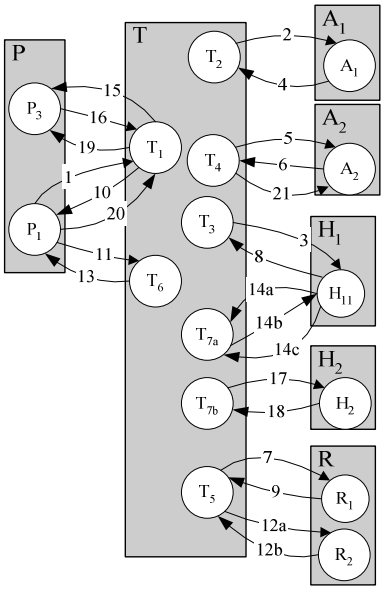


Figure 5.20: The Dooley graph of the travel planning example in Table 5.3, after character aggregation.

the "similarity" between the conversations that they have with a particular role. For instance, both A_1 and A_2 have a conversation with the role T. Let us follow the convention that all conversations are labeled by χ_i, where i is the first message in that conversation. Hence, χ_2 and χ_5 denote the conversations that take place between A_1 and T_2 and A_2 and T_4 respectively. To identify that A_1 and A_2 are peers, we need to find the similarity between conversations χ_2 and χ_5. We therefore define similarity of two conversations as follows

Definition 11. Two conversations χ_i and χ_j are similar if and only if all utterances u_i in χ_i that propose the creation of a commitment $\mathsf{C}(\cdot, \cdot, q \rightarrow p)$ by proposing either a create, a delegate or an assign, there exists an utterance u_j in χ_j that proposes a commitment $\mathsf{C}(\cdot, \cdot, g(q) \rightarrow g(p))$ in a similar manner (via a create, a delegate, or an assign). ∎

Here g is a function from the set of valid strings in the language used to express p and q to the same set. That is, $g : 2^{\mathbb{P}} \mapsto 2^{\mathbb{P}}$, where $\mathbb{P}$ is the set of propositions under consideration. Since our development is at the level of abstraction of commitments, we do not study languages used to express commitment conditions. We can therefore assume without loss of generality that $g(q) = q$. A more realistic function might compare the parse trees of the propositional expressions, ignoring variables (leaves) in the tree.

Definition 12. Two roles R_i and R_j are peers if and only if

1. R_i has a character R_{ix} that participates in a conversation χ_i
2. R_{jy} has a character R_{jy} that participates in a conversation χ_j
3. χ_i and χ_j are similar
4. χ_i and χ_j involve a common character.

∎

This definition operationalizes our intuition that two roles are peers if they can potentially take on the same commitments. Note that if two roles are peers, they need not have identical sets of peers. That is, if a role provides multiple services, such as rental car and hotel and airline ticket (like Orbitz.com or Priceline.com), it will be a peer not just to other rental car services, but also to other airline services and hotel services.

Using the above definition, we can infer that A_1 and A_2 are peers

- χ_2 and χ_5 are similar because the commitments proposed in them have the same precondition and fulfilment condition.
- χ_2 and χ_5 both involve role T.

Similarly, H_1 and H_2 are peers because of the similarity of χ_3 and χ_{17} (via the role T)

Conversation Rearrangement via Commitment Lifecycle

Based on the commitment operation patterns we introduced, all commitment have the following lifecycle: A commitment is created first, after which it can be updated multiple times, it can also be canceled, delegated, released, assigned, or discharged, after which it becomes inactive and no operations can be performed on it. Further, we note that the "normal" lifecycle of a commitment is creation followed by a discharge. Therefore, all other commitment operations (including an update) are optional after the creation of a commitment. Further, updates of the same commitment have no ordering restrictions among them in a reusable protocols. Consider the update of the commitment between P and T. This commitment is updated twice, first for cancelling the rental car and a second time for changing the hotel room. Were these updates performed in the reverse (or any other) order, the interaction would still achieve the same result: that of changing the hotel room and cancelling the rental car, if each update was completed before any other was begun. We can therefore generalize the independence of conversations that update the same commitment, even

if those conversations are ordered by the *Responds* relationship. In our example, χ_{15} can be moved before χ_{11}.

Improvements

The directed nature of all utterances in discourses prevents the modeling of events to which no sender or receiver can be assigned. We therefore require the creditor to perform the antecedent and the debtor to perform the consequent of a conditional commitment. In realistic situations, however, commitment antecedents, consequents, and fulfillment conditions might describe a state of the world to be brought about or an action that is not performed by either the creditor or the debtor. Whereas events external to the multiagent system cannot be ascribed to a performer, all agent actions and utterances can be placed in causal chains (a sequence of events with a complete temporal ordering among them) via to *Respond* relation, thus enabling verification of whether an agent caused a certain action Incorporating a framework for processing events, such as the RAPIDE Complex Event Processing (CEP) framework (Luckham, 2002) with a framework for developing and enacting commitment protocols, such as the OWL-P framework (Desai et al., 2006), would provide a grounding for realizing our commitment-based protocol enactment framework.

5.4 Discussion

This chapter demonstrated how protocols can be introduced into an agent-based software engineering methodology. Tropos benefits from our approach, because (1) protocols capture the dynamic, or runtime behavior of the software system being developed before the implementation stage in addition to the static dependencies between actors; (2) protocols decouple the meaning of an interaction by treating them as entities in their own right, which can be tailored to suit the needs of their participants and local policies at runtime; (3) Treating protocols as coherent units captures realistic interactions among autonomous entities. This is an advantage over a client-server model in which protocols are part of the logic embedded in the server.

Likewise, we contribute to commitment protocols by describing guidelines for designing them from requirements. Specifically, dependencies, means-end models, and AND-OR decomposition models in Tropos provide points of reference for using protocols between actors. Tropos provides the scope, i.e., boundaries, for the protocols.

In the second part of this chapter, we enhanced existing work on the derivation of graphs

from agent conversations to create a methodology for identifying reusable protocol. The benefit of our approach is that protocols can be combined in various ways, easing the effort in designing a process as compared to designing a monolithic flow. The various ways in which protocols can be put together also extracts the most value out of the creation and annotation of a discourse.

5.4.1 Related Literature

We next describe related work.

Commitment Protocols using Conversations

A closely allied work is Flores and Kremer's use of the *protocol for proposals* to propose every commitment operation (2004), thus making commitment operations non-atomic, as we have. They formalize the protocol using the Z notation, and use obligations to denote which agents' turn it is to perform some action in response to some another action. In a more recent effort, Kremer and Flores have developed a classification of the FIPA communicative acts to create, satisfy, or remove commitments as a conversation using these communicative acts progresses (Kremer and Flores, 2005). There are two fundamental differences between this work and theirs: One, Flores and Kremer's work is not a methodology for identifying protocols that can be used in other situations. Both their frameworks are different approaches to runtime state maintainence. Second, both their approaches use only two-party commitment operations, unlike our treatment of the three-party delegate and assign operations.

Verdicchio and Colombetti propose a commitment-based semantics for FIPA communicative acts (2005), which could be used as a basis for applying the methodology we have developed. Our methodology could also be enhanced by the use of verification mechanisms on the derived protocols, using a the correctness and completeness requirements for commitment protocols proposed by Yolum (2005).

Commitment Protocols using Tropos

Our work relates to software engineering and multiagent systems. Yu and Mylopolous (1993) explain the importance of identifying dependencies among autonomous entities in organizational settings, e.g., for business processes. They describe how dependencies based on resources and goals can be used to re-engineer business processes, since dependencies help answer "what-if?" and "why?" questions about changes in business processes. Whereas Yu and Mylopolous describe how

to introduce a dependency model into an existing business processes, we describe how protocols, which can be used to construct business processes, can be developed based on the requirements.

Giorgini *et al.*(2005) present a rigorous analysis of goal decomposition in Tropos. They develop algorithms to identify contributions among goals and possible conflicts among goals. This work would help our research in identifying valid refinements of protocols based on goals.

Gaia, KAOS, MaSE, and SADDE are important agent-oriented methodologies (Bergenti et al., 2004). Tropos differs from these in including an early requirements stage. Besides the early requirements gathering stage, Gaia differs from Tropos in that Gaia describes roles in the software system being developed and identifies processes that they are involved in as well as safety and liveness conditions for the processes (Zambonelli et al., 2003). Gaia incorporates protocols under the *interactions model* and can be used with commitment protocols. However, the lack of a reasoning scheme based on early requirements—to answer "why?" questions—limits the flexibility of Gaia's protocols.

Chapter 6

OWL-P: Realization of Commitment Protocols

This section describes an how commitment protocols based on our protocol model have been realized using a protocol specification language and a protocol enactment framework called OWL-P (Web Ontology Language for Protocols and Processes). The development here is a result of collaborative work (Desai et al., 2006), with significant contributions by Nirmit Desai, Amit Chopra, and Payal Chakravarthy.

6.1 Motivation: Why OWL-P?

The OWL-P project aims to demonstrate the applicability of commitment-protocol concepts to software engineering and process enactment in open environments among autonomous agents. To this end, the prime requirements for OWL-P are

Component Reuse. Protocols are first-class entities. Therefore, just as objects and components are reused, OWL-P should reuse protocols and processes across agents.

Commitment Semantics. Commitment protocols have a precise semantics based on the operations defined on commitments. OWL-P should employ these semantics.

Distributed Enactment. A client-server model of interaction forms a poor models for many business processes, since it violate the clients' autonomy. OWL-P should therefore adopt a peer-to-peer stance in which the role of each participant with respect to others in a protocol is

determined by the commitments that exist between them and enactment is not dependent on a centralized server.

Based on the above requirements, the OWL-P effort aims to

- Develop a language for the succinct specification of reusable protocols,
- Describe the operationalization of the above language into executable runs by mapping the language into a declarative framework,
- Develop a language and a semantics for the language to specify the composition of protocols, i.e., to specify restrictions applicable during a merge of protocols,
- Develop an algorithm that correctly identifies the involvement of each role in the protocol, since a protocol specification is a global view,
- Develop a system for modularly incorporating preferences of agent's and businesses into the protocol to generate enactable processes, and
- Demonstrate the enactment of the above processes

6.2 The OWL-P Language and Ontology

Figure 6.1 shows the conceptual model of OWL-P which was first introduced by Desai and Singh (2004). We repeat here the notation scheme used in the figure and definitions of the entities shown

Abstract Entities. Rectangles with sharp corners denote abstract entities or interfaces, which, when combined with business policies, yield concrete entities. Abstract entities are published and used by process developers.

Concrete Entities. Rectangles with rounded corners denote concrete entities. Only concrete entities can be use in a running system.

Protocol Logic. (Business) Processes in OWL-P are specified declaratively, using a set of rules. This set is called the protocol logic.

Role. A role represents an abstract entity representing one participant of a protocol. During enactment, an agent (which is an entity representing a real-world business partner with its local

business logic) adopts one or more roles to enable concrete computations. An agent may participate in multiple business protocols by playing a role in each of them. For example, a bookstore may play the role of a seller while interacting with customers and the role of a buyer while interacting with publishers.

Role Skeleton. A role skeleton specifies the protocol from the perspective of the corresponding participant role, in contrast with the protocol logic, which specifies the protocol from the global perspective. Thus, each role skeleton defines the behavior of the respective role in the given protocol.

Composite Skeleton. When an agent needs to participate in multiple protocols, a *composite skeleton* can be constructed by combining the protocols according to some composition constraints and deriving the role skeleton. For example, in a supply chain process, a supplier would be a merchant when interacting with a retailer in a trading protocol and would be an item-sender in a shipping protocol for sending goods to the retailer. A composite skeleton for such a supplier could be composed by combining trading and shipping protocols and then deriving the role skeleton for item-sender/merchant role. The resultant composite skeleton can also be published and then reused for developing local processes of other suppliers.

Business Logic. The business logic of an agent represents its policies as dictated by various factors such as business goals, objectives, and market situations.

Local Process. The *local process* of an agent is an executable realization of a composite skeleton obtained by integration of the protocol logic of the composite skeleton and the business logic of the agent.

Business Process. A *business process* is the aggregation of the local processes of all the agents participating in it. Conversely, a business process is an implementation of the constituent business protocols.

The language we have developed for specifying protocols is called OWL-P. In the following discussion, we specify whether we are referring to the framework or to the language where it is unclear from the context. OWL-P is an ontology based on OWL for specifying protocols; it functions as a schema for protocols. The main computational aspects of protocols are specified using rules. OWL-P employs the Semantic Web Rule Language (SWRL) for defining rules. SWRL

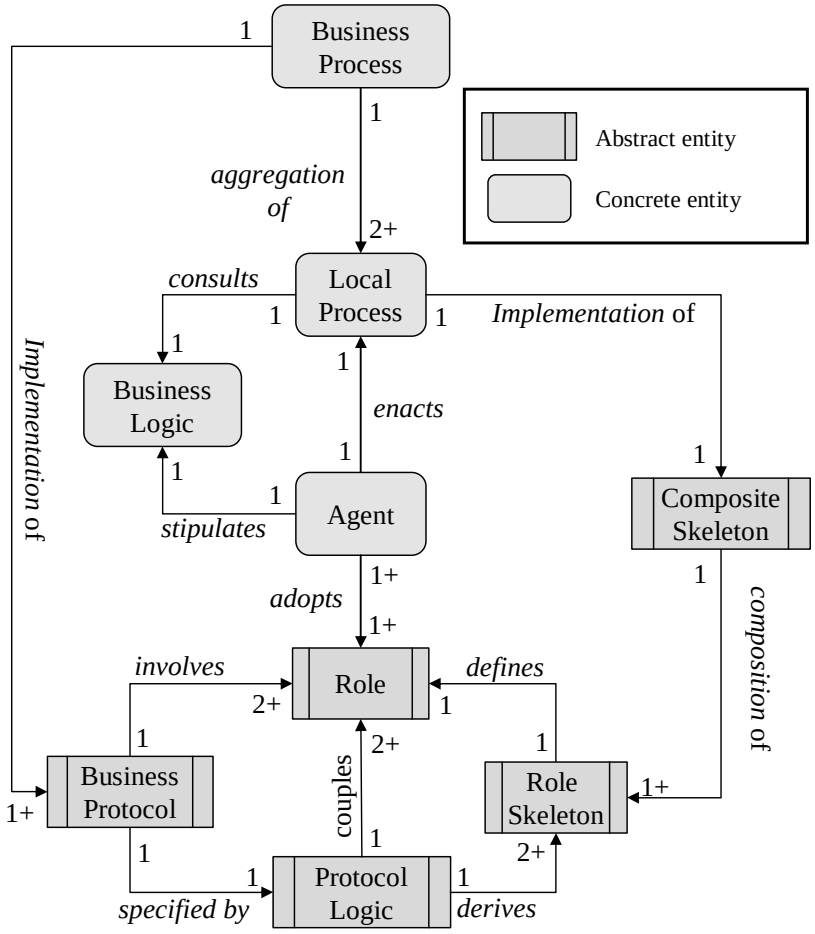


Figure 6.1: The conceptual model of the OWL-P framework as described by Desai and Singh (2004)

allows the specification of implication rules over entities defined as OWL-P instances. The availability of tools such as Protegé is a motivation for grounding the protocol ontology in OWL. With respect to Figure 6.1, we describe the following concepts: the specification of the protocol logic, the composition of protocols, the derivation of role skeletons, and the incorporation of an agents' business logic to create a local process.

6.2.1 The Protocol Model

The important OWL-P elements and their properties are shown in Figure 6.2. Lines between entities denote properties, the end with a little rectangle representing the domain of the property. The properties reflect the OWL-P conceptual model. Slots are analogous to data variables. A slot is said to be *defined* when it is assigned a value and it said to be *used* when its value is assigned

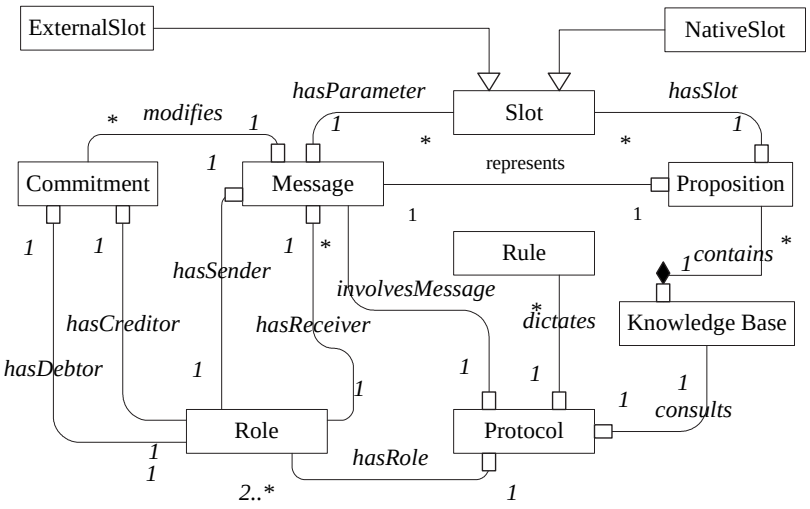


Figure 6.2: The OWL-P ontology for describing protocols.

to another slot. A slot in a protocol may be assigned a value produced by another protocol and hence be represented as an *External Slot*. An external slot is untyped until it is given the type of the external value to which it is bound. By contrast, a *Native Slot* is typed and defined inside the protocol.

A *Protocol* dictates rules that govern the interaction and consults a *Knowledge Base*, which consists of a set of propositions. Propositions are represented as slotted facts in a rule-based system. Propositions capture the state of the protocol by recording sent and received messages, active commitments, commitment operations, and domain facts.

As an example, consider three protocols, one for ordering an item, one for shipping an item, and one for payment. Figure 6.3 is a pictorial representation of the detail contained in OWL-P protocol specifications. The figure also shows a *Composition Profile*, which is a specification of constraints that have to be satisfied when composing these three protocols. Leading and trailing asterisks identify an external slot, as in *amount* in the Payment protocol. The amount to be paid has to be determined elsewhere.

6.2.2 The OWL-P Agent Architecture

OWL-P assumes an agent based architecture for enactment of protocols and the processes built using them. Each role in an OWL-P protocol is played by an agent. The generic architecture of an agent which participates in OWL-P protocols is shown in Figure 6.4. Each agent interacts with others by sending and receiving messages whose format and semantics is described by the OWL-P

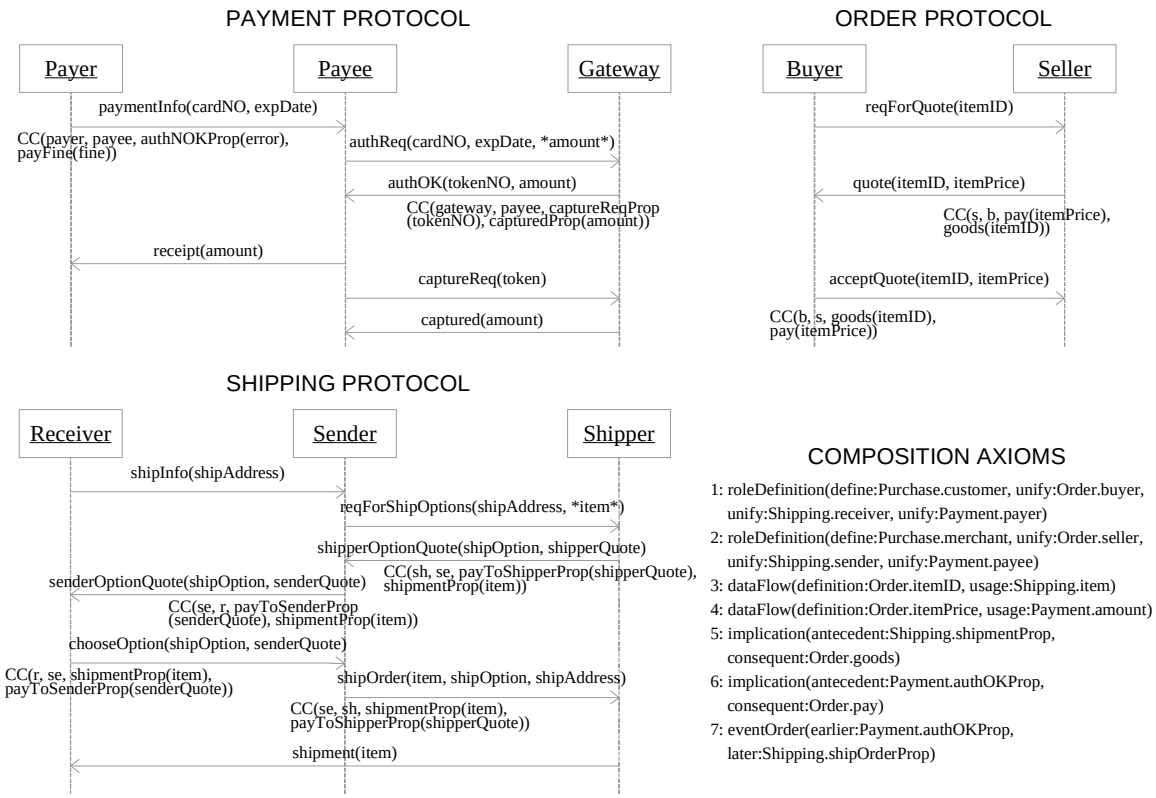


Figure 6.3: A pictorial representation of the OWL-P specification of an order, a shipping, and a payment protocol and a composition profile specifying constraints to be satisfied when combining them.

protocol specification. Each agent follows the protocol be computing, before enactment, its *role skeleton*, which is the part of the protocol it is concerned with because of the role it plays in that protocol. Further, each agent can modularly incorporate its *local policy*, or policy rules into the protocol. It is this modular addition of agent-specific rules to generic, flexible protocols that enables OWL-P to allow agent autonomy. An agent also maintains a knowledge base, which keeps a history of its interactions with other agents. The rules in the role skeleton operate on this knowledge base. The numbers in parentheses in the figure represent the typical order in which the various internal components of an agent interact.

1. The agent receives a message.

2. The agent modifies its knowledge base to record the message received and its effects as prescribed by the protocol.

3. The change in the knowledge base triggers some policy rules (which are stubs in the OWL-P

protocol logic) to fire.

4. Policy rules consult the business logic of the agent.
5. The business logic in turn changes the knowledge base due to its invocation.
6. The change in the knowledge base activates a protocol rule.
7. The protocol rule causes a message to be sent.
8. The agent records the message sent in its knowledge base and asserts propositions as dictated by the protocol.

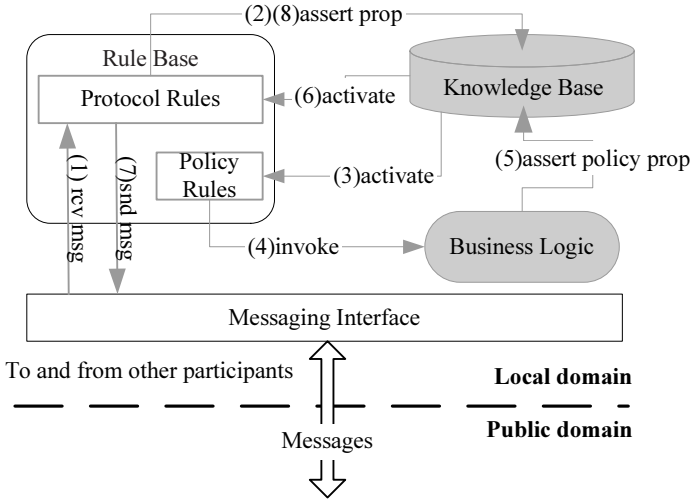


Figure 6.4: The architecture of an OWL-P agent. Numbers is parentheses represent the typical order of activities during the enactment of an OWL-P protocol.

6.3 The Composition Profile

A composition profile describes the relationships among protocols that have to be preserved when composing protocols. These relationships are called *axioms* in OWL-P terminology. OWL-P composition profiles have four kinds of axioms, which follow the syntax: $axiom(property_1 : range_1, property_2 : range_2 \ldots property_n : range_n)$, where $axiom_1$ denotes the type of the axiom, and $property_i$ are properties of axioms as described below.

Role Definition. A role definition axiom relates a role in the composed protocol to roles in the protocols being composed. Properties of a role definition axiom are *define* and *unify*, the

former specifying the name of the role in the composed protocol and the latter specifying a role in one of the composed protocols. Each role definition axiom has exactly one value for the *define* property. In Figure 6.3, for example, the second role definition axiom states that the *merchant* role in the *Purchase* protocol is a unification of the *seller* role in the *Order* protocol, the *sender* role in the *Shipping* protocol, and the *payee* role in the *Payment* protocol.

Data Flow. A data flow axiom provides the bindings for external slots in protocols, since an external slot's value has to be bound outside the scope of the protocol in which it is declared to be external. Properties of a data flow axiom are *definition* and *usage*, the former stating where the value of a slot will be bound and the latter stating where it will be used (as an external variable). Each data flow axiom has exactly one value for the *definition* property. In Figure 6.3, for example, the third axiom is a data flow axiom which specifies that the *item ID* slot from the *Order* protocol is used as the *item* slot in the *Shipping* protocol. Note that data flow axioms induce a temporal dependency as well, since the OWL-P framework requires that a rule using a slot cannot fire till that slot is bound to a value.

Implication. An implication axiom is used to denote propositions in different protocols that have the same meaning. The properties of an implication axiom are *antecedent* and *consequent*, the value of the former property logically implies the value of the latter property. In Figure 6.3, axiom 6 specifies that the *authOKProp* proposition from the *Payment* protocol implies the *pay* proposition of the protocol.

Event Order. An event order axiom specifies temporal ordering among messages in the protocols being composed. In Figure 6.3, axiom 7 specifies that an *authOK* message from the payment gateway must be received before a *shipOrder* message is sent to the shipper. This can be ensured by making the rule for the later event depend on the rule for the earlier event.

A composition profile is developed by a protocol designer. A composed protocol can be composed further like any other protocol. Desai *et al.* (Desai et al., 2006)

6.4 Rules in OWL-P

OWL-P specifies protocols declaratively, using rules of the form *antecedent* $\Rightarrow$ *consequent*. To be compatible with OWL syntax, the Semantic Web Rule Language (SWRL) has been adopted in OWL-P to specify rules. Antecedents check the *knowledge base* for the existence or absence of

propositions. Consequents effect message transfers and add related propositions to the knowledge base. In effect, rules fire based on the history of the interaction as captured by propositions in the knowledge base. One of the most important features of OWL-P in contrast with traditional process models such as workflows is the distributed nature of OWL-P, where each participant of a protocol is a peer, unlike in a client-server system. The knowledge base, therefore cannot be centralized. Further, a distributed enactment framework implies that no participant is guaranteed to be able to observe all messages (and significant events) in the protocol. The OWL-P algorithm for generating role skeletons incorporates this feature of distributed systems and is sound and complete with respect to the set of runs it generates.

A protocol may not be enactable when there is a *nonlocal choice*, i.e., when an participant's decision (of what message to send) depends on information it does not have (Ladkin and Leue, 1995). For example, protocols with rules of the form $a \Rightarrow b$, where proposition a is known only to roles ρ_1 and ρ_2, and message b is sent by ρ_3, are not enactable, as ρ_3 cannot observe a. Such a case arises due to the extraction of local role skeletons from a global protocol specification. As described by Desai *et al.*, the following properties of a protocol P avoid this problem and ensure that P is enactable. Here, r denotes a rule, a, b, p, and q are propositions or messages as applicable, and P is interpreted as a set of rules.

Prop$_1$. $\forall r{\in}P$, $\forall a{\in}r.$***body***, $a{\neq}start \rightarrow \exists r'{\in}P\ a{\in}r'.$***head***. The propositions in the antecedent of a protocol rule must be asserted by some protocol rule.

Prop$_2$. $\forall r{\in}P$, $\forall a{\in}r.$***head***, $(start \rightsquigarrow a){\in}P$. All asserted propositions are reachable from $start$. Here, $p \rightsquigarrow q$ means q is reachable from p. Formally, $p \rightsquigarrow q \stackrel{def}{=} p \Rightarrow q \vee (p \Rightarrow q' \wedge q' \rightsquigarrow q)$.

Prop$_3$. $\forall r{\in}P$, $\forall a{\in}r.$***body***, $(\exists b{\in}r.$***head*** $\wedge\ \exists \rho{=}sender(b) \rightarrow \rho{=}sender(a) \vee \rho{=}receiver(a))$. If a message exchange b has an antecedent a then, the sender ρ of b must be able to verify a. That is, ρ must receive a message corresponding to a or must send a message corresponding to a, i.e., ρ observes a or causes a.

A protocol designer can obtain an enactable protocol by repeated applications of composition. For instance, in Figure 6.3, the rules of *Order* do not assert the propositions *pay* and *goods*, both of which are necessary for discharging the commitments created in the protocol. The rules of *Payment* do not assert the proposition *payFine(fine)*. The rules of *Shipping* do not assert *payToSenderProp* and *payToShipperProp*. The rules of the composite *Purchase* do not assert the propositions

pay(fineAmount), *payToSenderProp*, and *payToShipperProp*. A designer chooses protocols that assert the missing propositions and compose them with *Purchase* to obtain an enactable protocol.

The algorithm for computing the role skeleton of a participant from the protocol specification and the proof of its correctness and soundness is described in detail in (Desai et al., 2006).

Local Policies

A fundamental benefit of raising protocols to first-clas status is to promote their reuse. Reusing protocols, however should not come at the cost of agent autonomy. That is, designers should be able to customize a protocol to suit their requirements while operating within valid protocol runs. OWL-P achieves this objective by enabling agents to have their own *local policies* that can be modularly plugged into the role skeleton derived from the protocol, as described here. We recall that a rule in a protocol may be abstract, meaning that values of some of its native slots must be produced via the role's business logic. That is, a role skeleton must be augmented with business logic to obtain a local process. To capture this intuition, a skeleton is said to be *concrete* if all of its native slots are defined, *abstract* otherwise.

An agent's business logic is specified as local policy rules. The skeleton of the seller in *Order* augmented with the policy rules of the seller agent as shown below. The policy rule invokes a business logic operation to decide what price to quote. The operation asserts *quotePolicy*, which activates the second skeleton rule.

```
Seller skeleton rules:
```

1: startProp ⇒ receive(reqForQuote(?itemID))

2: reqForQuoteProp(?itemID) ∧ **quotePolicy(?itemPrice)** ⇒
   send(quote(?itemID, ?itemPrice)) ∧
   CC(S, B, pay(?itemPrice), goods(?itemID))

3: quoteProp(?itemID, ?itemPrice) ⇒
   receive(acceptQuote(?itemID, ?itemPrice)) ∧
   CC(B, S, goods(?itemID), pay(?itemPrice))

4: quoteProp(?itemID, ?itemPrice) ⇒
   receive(rejectQuote(?itemID, ?itemPrice))

```
Seller's policy for deciding quote:
```

1: reqForQuoteProp(?itemID) ⇒

call(policyDecider, quotePolicy(?itemID))

This pattern of augmenting policy rules applies to rules where the agent has to make a decision and respond. It also assigns a value to native slots that are not defined. Figure 6.4 shows the interplay between the protocol rules and the policy rules of an agent. Steps 3, 4, and 5 show policy rules in action. The business logic could involve looking up a legacy database or waiting for human input.

6.5 Business Process Development using OWL-P

Figure 6.5 summarizes our methodology with a scenario involving a customer interested in purchasing goods online. Software designers design protocols and register them with protocol repositories. They may also construct composite protocols by specifying composition axioms and reusing the existing component protocols from the repository. *Composer* is our tool for generating the composite protocol. A merchant wishing to supply goods online looks up *Purchase* in a repository. It generates the skeleton for the merchant role, augments it with its local policies, and deploys the result as a service. The profile for this service contains an OWL-P description of *Purchase*, and may be published to a UDDI registry. If a customer wishes to procure goods online, it searches the UDDI registry, finds the merchant, and acquires the OWL-P skeleton for the customer role from the merchant. The customer enacts its local process by augmenting the skeleton with its local policies and starts interacting with the merchant. Our tools support these development scenarios and a prototype implementation based on the agent architecture of Figure 6.4.

The OWL-P project is in its early stages and is being actively improved and expanded by the introduction of new tools to support various stages of the design nad development of business protocols.

6.6 Related Work

The uniqueness of OWL-P arises from

1. A commitment-based semantics for interactions,
2. A focus on a methodology that enables reuse, refinement, and aggregation, and
3. A practical grounding of concepts along with their theoretical foundations.

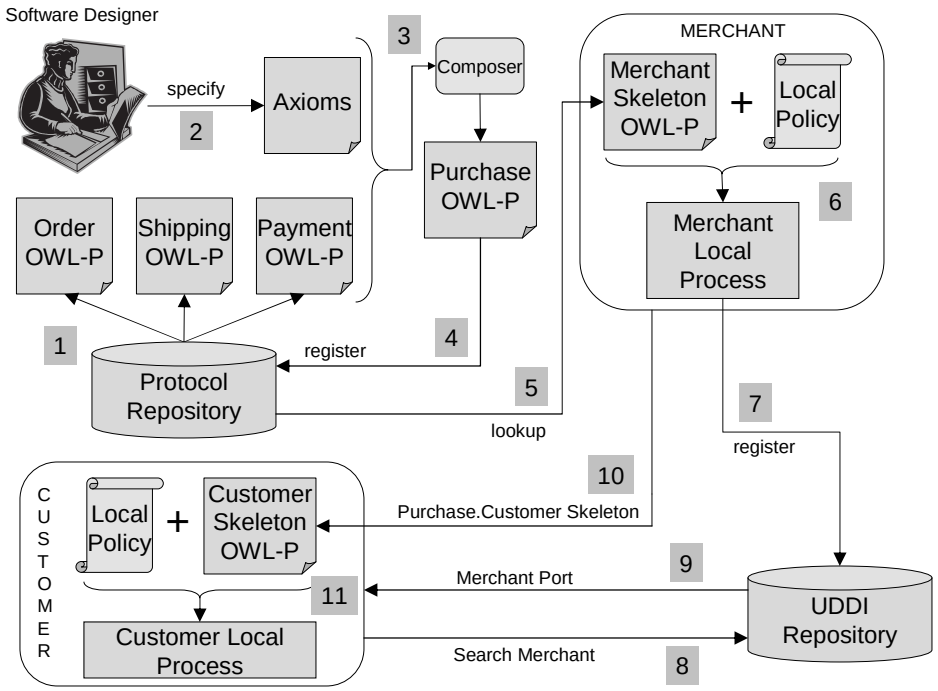


Figure 6.5: Typical usage of OWL-P in the development and enactment of business processes.

This section summarizes several relevant research efforts.

Component behavior modeling Plasil *et al.*(2002) propose a language based on regular expressions for describing the behavior of software components using protocols. They define software components as agents and the proposed language allows expressions of method invocations over components. This behavior protocols are similar to OWL-P protocols but do not support commitments. Similarly, Canal *et al.*(2003) use π-calculus to describe protocols over the methods provided of CORBA objects. They reason about interaction compatibility and object substitutability, i.e., if an object can be successfully substituted by another. Studying architectural connectors, Allen *et al.*(1997) express similar intuitions about interactions being a fundamental abstraction for complex systems. Their components correspond to OWL-P agents, connectors to protocols, and attachments to role adoption. They formalize components and connectors in CSP (Hoare, 1985), and have abstractions of roles and connector composition analogous to those in OWL-P.

These approaches seek to verify interaction specifications for correctness, compatibility, deadlock freedom, and substitutability for components, whereas in OWL-P, the thrust is to exploit protocols to develop processes, thus improving the quality of modeling and reuse of interactions.

Previous approaches view interactions as sequences of send and receive actions without considering commitment-based semantics.

Scenario-based behavior modeling This area develops visual formalisms for modeling interaction scenarios of entities (agents, components, or processes). Popular formalisms include AUML (Bauer et al., 2001), Message Sequence Charts (ITU, 1996), and UML sequence diagrams (Jacobson et al., 1999).

Whittle *et al.*(2000) present an approach for creating a design model (specified as a statechart) from a collection of interaction scenarios using a state-identification approach to merge scenarios. Uchitel *et al.*(2003) present an MSC language to explicitly specify the assumptions under which scenarios are merged. Whereas the above approaches merge behaviors to produce one protocol, OWL-P merges several protocols to produce a composite protocol. Further, the OWL-P methodology incorporates commitments to better capture business requirements.

Uchitel *et al.* (Uchitel et al., 2004) elaborate behavior models by deriving implied scenarios from an existing model, categorizing them as desired or not, and updating the model. Implied scenarios arise as a result of the partial nature of scenario-based specifications. Protocols are naturally extensible, especially open protocols, which necessarily need to be fleshed out to become enactable.

Service composition BPEL (Andrews et al., 2003) is a language for specifying the static composition of Web services. However, it mixes interaction activities with business logic, thus reducing reusability. WS-CDL (Kavantzas et al., 2004) specifies the conversational behavior of a service using control flow constructs. However, these specifications lack a flexible semantics, which makes them difficult to compose and reuse. OWL-S (DAML-S, 2002), which includes a process model for Web services, uses semantic annotations to enable dynamic composition. A composed service is produced at runtime based on stated constraints. Dynamic service composition assumes a perfect markup of the services being composed, and involves ontological matching between inputs and outputs. OWL-S formalizes process models of the ilk of BPEL, but doesn't address protocols as such. An end to end service creation approach separates a logical composition (service type) from a physical one (service instances) (Agarwal et al., 2005). This approach combines a service matchmaking similar to OWL-S and a BPEL based service implementation. Due to its close ties with OWL-S and BPEL, it inherits their limitations mentioned above.

Some approaches treat service composition via formal specifications to support verifiabil-

ity. Solanki *et al.*(2004) employ interval temporal logic to specify and verify ongoing behavior of a composed service. Their use of "assumption" and "commitment" (different meaning than here) assertions yields better compositionality than other approaches. Gerede *et al.*(2004) model services as activity-based finite automata to study the decidability of composability and existence of a lookahead delegator for a set of services. However, these approaches consider neither the autonomy of the partners, nor the flexibility of composition.

Agent-oriented software engineering Agent-oriented software methodologies, e.g., Tropos, Gaia, KAOS, MaSE, and SADDE (Bergenti et al., 2004), apply software engineering principles to agent-based systems and were discussed in detail in Chapter 5. Baïna *et al.*(2004) advocate a model-driven Web service development approach to ensure compliance between a service's implementation and its external protocol specifications. By contrast, the objective of OWL-P is achieving protocol reusability by separating protocols and policies, and by composing protocols.

Component catalogs The OWL-P methodology enables using protocols as building blocks of business processes. Protocols are reused by refining them to yield improved protocols. RosettaNet has (RosettaNet) put protocols in practice. About 100 published protocols, termed Partner Interface Processes (PIPs), account for billions of dollars of business. However, RosettaNet is limited to two-party request-response PIPs, which lack a formal semantics. The MIT Process Handbook (Malone et al., 2003) classifies business processes. For example, *sell* is a generic business process, which can be qualified by *sell what*, *sell to whom*, and so on.

6.7 Summary

In this dissertation, we have developed a lifecycle for engineering protocols. The basic challenge faced was that of managing interactions among autonomous entities. We have described how an agent-based approach grounded in the concept of social commitments forms a solid foundation for modeling and enacting protocols among independent entities such as businesses. Our approach and the problems that we have addressed are summarized in Figure 6.6. Below the grey arrowheads are problem statements that we have answered and the rectanges above the arrowheads specify specific sections of this dissertation that have provided solutions.

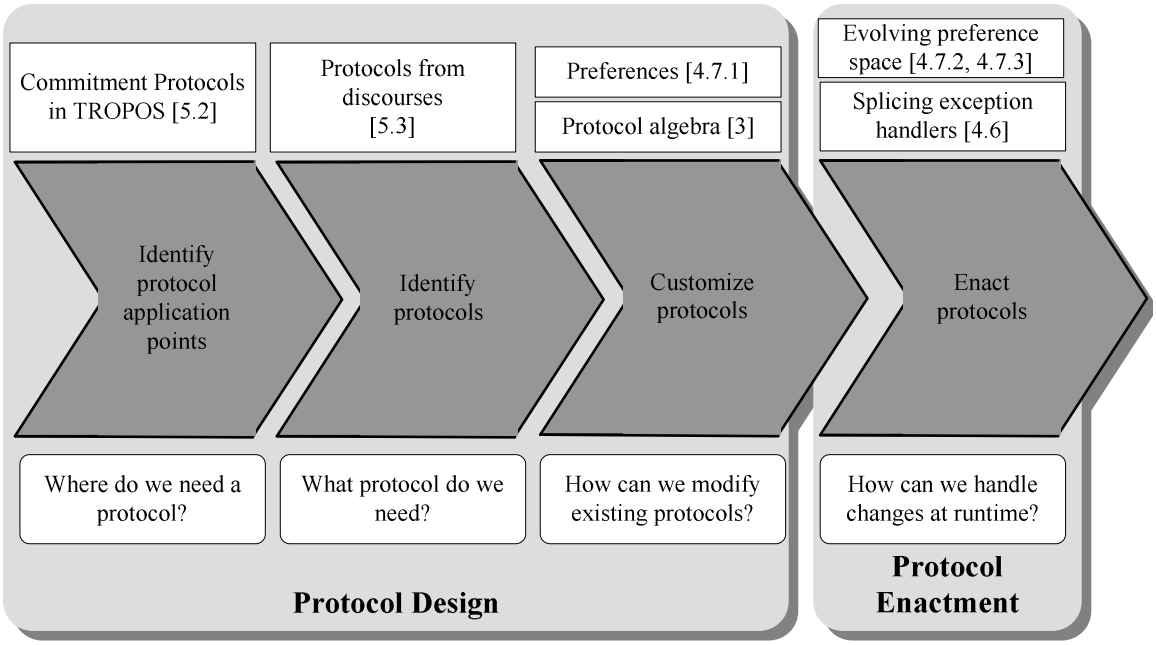


Figure 6.6: An overview of our contributions to the engineering lifecycle of commitment protocols.

Bibliography

TWIST standards, 2005. URL http://www.twiststandards.org/tiki-index.php.

Vikas Agarwal, Koustav Dasgupta, Neeran Karnik, Arun Kumar, Ashish Kundu, Sumit Mittal, and Biplav Srivastava. A service creation environment based on end to end composition of web services. In *Proceedings of the International World Wide Web Conference*, pages 128–137. ACM Press, 2005.

Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business Process Execution Language for Web Services, (BPEL 1.1), May 2003. URL www-106.ibm.com/developerworks/webservices/library/ws-bpel.

Assaf Arkin. Business Process Modeling Language, 2002. URL http://www.bpmi.org/bpml-spec.esp.

Assaf Arkin, Sid Askary, Scott Fordin, Wolfgang Jekeli, Kohsuke Kawaguchi, David Orchard, Stefano Pogliani, Karsten Riemer, Susan Struble, Pal Takacsi-Nagy, Ivana Trickovic, and Sinisa Zimek. Web Service Choreography Interface (WSCI 1.0), 2002. URL http://www.w3.org/TR/wsci.

John L. Austin. *How to Do Things with Words*. Clarendon Press, Oxford, 1962.

Karim Baïna, Boualem Benatallah, Fabio Casati, and Farouk Toumani. Model-driven web service development. In *Proceedings of Advanced Information Systems Engineering: 16th International Conference, CAiSE*, 2004.

Bernhard Bauer, Jörg Müller, and James Odell. *Agent UML: A Formalism for Specifying Multiagent Interaction*. Springer-Verlag, 2001.

Federico Bergenti, Marie-Pierre Gleizes, and Franco Zambonelli, editors. *Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook*. Kluwer Academic, 2004.

Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 501, 2001.

BPWS4J. Business process execution language for web services Java runtime, 2004. URL http://alphaworks.ibm.com/tech/bpws4j.

Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Guinchiglia, and John Mylopolous. Tropos: An agent-oriented software development methodology. *Journal of Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, May 2004.

BTP. Business transaction protocol, 2002. URL http://www.oasis-open.org/committees/download.php/1184/2002-06-03.btp_cttee-spec-1.0.pdf.

Paul Buhler and José Vidal. Towards adaptive workflow enactment using multiagent systems. *Information Technology and Management Journal*, 2003. To Appear.

David Burdett and Nickolas Kavantzas. WS coreography model overview, 2004. URL http://www.w3.org/TR/ws-chor-model/.

Stefan Bussmann, Nicholas R. Jennings, and Michael Wooldridge. Re-use of interaction protocols for agent-based applications. In Fausto Giunchiglia, James Odell, and Gerhard Weiß, editors, *Third International Workshop on Agent-Oriented Software Engineering (AOSE 2002)*, volume 2585 of *Lecture Notes in Computer Science*, pages 73–87. Springer, 2003.

Carlos Canal, Lidia Fuentes, Ernesto Pimentel, Jose M. Troya, and Antonio Vallecillo. Adding roles to corba objects. *IEEE Transactions on Software Engineering*, 29(3):242–259, March 2003.

Fabio Casati. A discussion on approaches to handling exceptions in workflows. In *Proceedings of the CSCW Workshop: Towards Adaptive Workflow Systems*, Seattle, 1998.

Fabio Casati, Mariagrazia Fugini, and Isabelle Mirbel. An environment for designing exceptions in workflows. *Information Systems*, 24(3):255–273, 1999.

Cristiano Castelfranchi. Commitments: From individual intentions to groups and organizations. In *Proceedings of the AAAI-93 Workshop on AI and Theories of Groups and Organizations: Conceptual and Empirical Research*, 1993.

Dickson K. W. Chiu, Qing Li, and Kamalakar Karlapalem. ADOME-WFMS: towards cooperative handling of workflow exceptions. *Advances in exception handling techniques*, pages 271–288, 2001.

Amit Chopra and Munindar P. Singh. Nonmonotonic commitment machines. In Marc-Philippe Huget and Frank Dignum, editors, *Proceedings of the AAMAS Workshop on Agent Communication Languages and Conversation Policies*, 2003.

Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL 1.1), March 2001. URL http://www.w3.org/TR/wsdl.

Luc Clement, Andrew Hately, Claus von Riegen, and Tony Rogers. UDDI version 3.0.2, 2004. URL http://uddi.org/pubs/uddi_v3.htm.

Collaxa. Collaxa BPEL orchestration server, 2003. URL http://www.collaxa.com/home.index.jsp.

DAML-S. DAML-S: Web service description for the semantic Web. In *Proceedings of the 1st International Semantic Web Conference (ISWC)*, July 2002. Authored by the DAML Services Coalition, which consists of (alphabetically) Anupriya Ankolekar, Mark Burstein, Jerry R. Hobbs, Ora Lassila, David L. Martin, Drew McDermott, Sheila A. McIlraith, Srini Narayanan, Massimo Paolucci, Terry R. Payne and Katia Sycara.

Nirmit Desai, Ashok U. Mallya, Amit K. Chopra, and Munindar P. Singh. Processes = Protocols + Policies: A methodology for business process development. Technical report, North Carolina State University, 2004. TR2004-34.

Nirmit Desai, Ashok U. Mallya, Amit K. Chopra, and Munindar P. Singh. OWL-P: A methodology for business process modeling and enactment. In *The AAMAS Workshop on Agent Oriented Information Systems (AOIS)*, July 2005.

Nirmit Desai, Ashok U. Mallya, Amit K. Chopra, and Munindar P. Singh. Interaction protocols as design abstractions for business processes. *IEEE Transactions on Software Engineering*, 2006. URL http://www4.ncsu.edu/~nvdesai/tse05.pdf. To appear.

Nirmit Desai and Munindar P. Singh. Protocol-based business process modeling and enactment. In *Proceedings of the IEEE International Conference on Web Services*, pages 35–42, 2004.

Frank Dignum, editor. *Advances in Agent Communication*, Berlin, 2003. Springer-Verlag.

Johann Eder and Walter Leibhart. The workflow activity model WAMO. In Steve Laufmann, Stefano Spaccapietra, and Toshio Yokoi, editors, *Proceedings of the Third International Conference on Cooperative Information Systems (CoopIS)*, pages 87–98. University of Toronto Press, May 1995.

Tim Finin, Li Ding, Rong Pan, Anupam Joshi, Pranam Kolari, Akshay Java, and Yun Peng. Swoogle: Searching for knowledge on the semantic web. In *AAAI 05 (intelligent systems demo)*, 2005. URL http://ebiquity.umbc.edu/get/a/publication/161.pdf.

FIPA-ACL. The FIPA agent communication specifications, 2000. http://www.fipa.org/repository/aclspecs.php3.

Layna Fischer, editor. *The Workflow Handbook 2004*. Future Strategies Inc., Lighthouse Point, FL, 2004. Compendium of standards and articles by the Workflow Management Coalition (WfMC).

Roberto A. Flores and Robert C. Kremer. A principled modular approach to construct flexible conversation protocols. In *Advances in Artificial Intelligence*, volume 3060 of *Lecture Notes in Artificial Intelligence*, pages 1–15. Springer-Verlag, 2004.

Nicoletta Fornara and Marco Colombetti. Defining interaction protocols using a commitment-based agent communication language. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 520–527. ACM Press, July 2003.

Xiang Fu, Tevfik Bultan, and Jianwen Su. Realizability of conversation protocols with message contents. In *Proceedings of the 2004 International Conference on Web Services*, pages 96–105. IEEE Computer Press, 2004.

Çağdaş Evren Gerede, Richard Hull, Oscar Ibarra, and Jianwen Su. Automated composition of e-services: Lookaheads. In *Proceedings of the International Conference on Service Oriented Computing*, 2004.

Paolo Giorgini, John Mylopolous, and Roberto Sebastiani. Goal-oriented requirements analysis and reasoning in the tropos methodology. *Engineering Application of Atrificial Intelligence Journal*, 18(2), 2005. To Appear.

Benjamin N. Grosof and Terrence C. Poon. SweetDeal: Representing agent contracts with exceptions using XML rules, ontologies, and process descriptions. In *Proceedings 12th International Conference on the World Wide Web*, pages 340–349, 2003.

Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP version 1.2 part 1: Messaging framework, June 2003. URL `http://www.w3.org/TR/soap12-part1/`.

Rachid Hamadi and Boualem Benatallah. Proceedings of the 14th Australasian Database Conference. In Xiaofang Zhou and Klaus-Dieter Schewe, editors, *ADC*, volume 17 of *Conferences in Research and Practice in Information Technology*, pages 191–200. Australian Computer Society, February 2003.

James E. Hanson, Prabir Nandi, and Santosh Kumaran. Conversation support for business process integration. In *Proceedings 6th IEEE International Enterprise Distributed Object Computing Conference*, pages 65–74. IEEE Press, 2002a.

James E. Hanson, Prabir Nandi, and David W. Levine. Conversation-enabled web services for agents and e-business. In *Proceedings of the International Conference on Internet Computing*, pages 791–796. CSREA Press, 2002b.

Charles Antony Richard Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

Marc-Philippe Huget, editor. *Communication in Multiagent Systems: Agent Communication Languages and Conversation Policies*, Berlin, 2003. Springer-Verlag.

Michael N. Huhns, Larry M. Stephens, and Nenad Ivezic. Automating supply-chain management. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 1017–1024. ACM Press, July 2002.

ITU. *Message Sequence Charts, Recommendation z.120*. International Telecommunication Union, Telecomminication Standardization Sector, 1996.

Ian Jacobson, James Rumbaugh, and Grady Booch. *The Unified Software Development Process*. Addison Wesley, 1999.

Gerrit K. Janssens, Jan Verelst, and Bart Weyn. Techniques for modeling workflows and their support of reuse. In Wil van der Aalst, Jörg Desel, and Andreas Oberwis, editors, *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, Berlin, 2000.

Mark W. Johnson, Peter McBurney, and Simon Parsons. When are two protocols the same? In Marc-Philippe Huget, editor, *Communication in Multiagent Systems: Agent Communication Languages and Conversation Policies*, volume 2650 of *Lecture Notes in Artificial Intelligence*, pages 253–268. Springer-Verlag, Berlin, 2003.

Mohan U. Kamath and Krithi Ramamritham. Failure handling and coordinated execution of concurrent workflows. In *Proceedings of 14th International Conference on Data Engineering*, pages 334–341. IEEE Computer Society, 1998.

Nickolas Kavantzas, David Burdett, Gregory Ritzinger, Tony Fletcher, and Yves Lafon. Web Service Choreography Description Language (WS-CDL 1.0), December 2004. URL http://www.w3.org/TR/ws-cdl-10/.

Mark Klein. Exception handling in agent systems. In *Proceedings of the 3rd International Conference on Autonomous Agents*, 1999.

Mark Klein, Juan Antonio Rodriguez-Augilar, and Chrysanthos Dellarocas. Using domain-independent exception handling services to enable robust open multi-agent systems: The case of agent death. *Journal of Autonomous Agents and Multiagent Systems*, 7(1/2), 2003.

Tom Krazit. Intel conducts $5b in transactions via RosettaNet, December 2002. URL http://archive.infoworld.com/articles/hn/xml/02/12/10/021210hnintelrose.xml.

Robert C. Kremer and Roberto A. Flores. Using a performative subsumption lattice to support commitment-based conversations. In *Proceedings of the 4th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 114 – 121. ACM Press, July 2005.

Peter B. Ladkin and Stefan Leue. Interpreting message flow graphs. *Formal Aspects of Computing*, 7(5):473–509, 1995.

Robert E. Longacre. *An Anatomy of Speech Notions*. Peter de Ridder, Lisse, Holland, 1976.

David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002.

Ashok U. Mallya and Munindar P. Singh. A semantic approach for designing commitment protocols. In Rogier Van Eijk, editor, *Developments in Agent Communication*, volume 3396 of *Lecture Notes in Artificial Intelligence*, pages 37–51. Springer, Berlin, 2005.

Thomas W. Malone, Kevin Crowston, and George A. Herman, editors. *Organizing Business Knowledge: The MIT Process Handbook*. MIT Press, Cambridge, MA, 2003.

David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. OWL-S: Semantic markup for web services. W3C Member Submission, November 2004. URL http://www.w3.org/Submission/OWL-S/.

Robert Miller and Anand Tripathi. Issues with exception handling in object-oriented systems. In *European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 85–103, 1997.

H. Van Dyke Parunak. Visualizing agent conversations: Using enhanced Dooley graphs for agent design and analysis. In *Proceedings of the 2nd International Conference on Multiagent Systems*, pages 275–282. AAAI Press, 1996.

Philippe Pasquier and Brahim Chaib-Draa. The cognitive coherence approach for agent communication pragmatics. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 544–551. ACM Press, July 2003.

Paypal. Paypal, 2003. URL http://www.paypal.com.

Chris Peltz. Web service orchestration and choreography. *IEEE Computer*, 36(10):46–52, October 2003.

Charles Petrie and Akhil Sahai. Business processes on the web. *IEEE Internet Computing*, 8(1): 28–29, January 2004.

Jeremy Pitt and Abe Mamdani. Communication protocols in multi-agent systems. In Frank Dignum and Mark Greaves, editors, *Issues in Agent Communication*, volume 1916 of *Lecture Notes in Artificial Intelligence*, pages 160–177. Springer, 2000.

Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, 2002.

Jerry Roberts and Krishnamurthy Srinivasan. Tentative hold protocol, 2001. URL http://www.w3.org/TR/tenthold-1/.

J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session initiation protocol, June 2002. URL http://www.ietf.org/rfc/rfc3261.txt. IETF Proposed Standard.

RosettaNet. Rosettanet, 1998. URL www.rosettanet.org.

Munindar P. Singh. Agent communication languages: Rethinking the principles. *IEEE Computer*, 31(12):40–47, December 1998.

Munindar P. Singh. An ontology for commitments in multiagent systems: Toward a unification of normative concepts. *Artificial Intelligence and Law*, 7:97–113, 1999.

Munindar P. Singh. Synthesizing coordination requirements for heterogeneous autonomous agents. *Autonomous Agents and Multi-Agent Systems*, 3(2):107–132, June 2000.

Munindar P. Singh. Distributed enactment of multiagent workflows: Temporal logic for web service composition. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 907–914. ACM Press, July 2003.

Munindar P. Singh, Amit K. Chopra, Nirmit V. Desai, and Ashok U. Mallya. Protocols for processes: programming in the large for open systems (extended abstract). In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 120–123, New York, NY, USA, 2004. ACM Press.

Munindar P. Singh and Michael N. Huhns. *Service-Oriented Computing: Semantics, Processes, Agents*. John Wiley & Sons, 2005.

Marvin A. Sirbu. Credits and debits on the Internet. *IEEE Spectrum*, 34(2):23–29, February 1997.

Monika Solanki, Antonio Cau, and Hussein Zedan. Augmenting semantic web service descriptions with compositional specification. In *Proceedings of the International World Wide Web Conference*, pages 544–552, 2004.

Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Synthesis of behavioral models from scenarios. *IEEE Transactions on Software Engineering*, 29(2):99–115, February 2003.

Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Transactions on Software Engineering and Methodology*, 13(1):37–85, January 2004.

Wil van der Aalst and Kees van Hee, editors. *Workflow Management Models, Methods, and Systems*. MIT Press, Cambridge, MA, 2002.

Sieuwert van Otterloo, Wiebe van der Hoek, and Michael Wooldridge. Preferences in game logics. In *Proceedings of the 3rd International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 152–159. ACM Press, July 2004.

Mario Verdicchio and Marco Colombetti. Commitments for agent-based supply chain management. *ACM SIGecom Exchanges*, 3(1):13–23, 2002.

Mario Verdicchio and Marco Colombetti. A commitment-based communicative act library. In *Proceedings of the 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 755–761. ACM Press, July 2005.

José Vidal, Paul Buhler, and Christian Stahl. Multiagent systems with workflows. *IEEE Internet Computing*, 8(1):76–82, January 2004.

Benjamin Vitteau and Marc-Philippe Huget. Modularity in interaction protocols. In Frank Dignum, editor, *Advances in Agent Communication*, volume 2922 of *Lecture Notes in Artificial Intelligence*, pages 291–309. Springer, 2004.

Feng Wan and Munindar P. Singh. Commitments and causality for multiagent design. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 749–756. ACM Press, July 2003.

Jon Whittle and Johann Schumann. Generating statechart designs from scenarios. In *Proceedings of International Conference on Software Engineering*, pages 314–323, 2000.

Michael Winikoff, Wei Liu, and James Harland. Enhancing commitment machines. In *Proceedings of the AAMAS Workshop on Declarative Agent Languages and Technologies*, 2004.

WS-C. Web services coordination, 2003. URL http://www-106.ibm.com/developerworks/webservices/library/ws-coor/.

WS-T. Web services transaction, 2002. URL http://www-106.ibm.com/developerworks/library/ws-transpec/.

Jie Xing, Feng Wan, Sudhir K. Rustogi, and Munindar P. Singh. A commitment-based approach for business process interoperation. *IEICE Transactions on Information and Systems*, E84-D(10): 1324–1332, October 2001. Special issue on *Autonomous Decentralized Systems and Systems Assurance*.

Pınar Yolum. Towards design tools for protocol development. In *Proceedings of the 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 99–105. ACM Press, July 2005.

Pınar Yolum and Munindar P. Singh. Flexible protocol specification and execution: Applying event calculus planning using commitments. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 527–534. ACM Press, July 2002.

Eric Yu. *Modeling Strategic Relationships for Process Reengineering*. PhD thesis, University of Toronto, 1995.

Eric S. K. Yu and John Mylopoulos. An actor dependency model of organizational work: with application to business process reengineering. In *COOCS '93: Proceedings of the conference on Organizational computing systems*, pages 258–268. ACM Press, 1993.

Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Developing multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering Methodology*, 12(3): 317–370, 2003.