Abstract

WAGLE, LEENA V. Incorporating Business Policies into Business Protocols (Under the direction of Dr. Munindar P. Singh).

A business process streamlines how different business parties interact with one another in order to achieve a business goal. The modeling and enactment of business processes is notoriously complex, especially in open systems where the participants are autonomous and heterogeneous. Conceptually, a business process consists of two important elements: the business protocols that specify the interactions between the partners and the business policies that drive the partners' behavior during the enactment of the protocols.

Business policies are an integral and important component of business processes. Business policies are often expressed and implemented as business rules. It is common to embed these business rules within the business process implementation. When business practices or policies change, as they often do, it is difficult to dynamically manage these changes without affecting the business process implementation. It is a challenge to correctly reflect the policy changes with minimal or no change to the implementation of the business process. Conventional approaches of representing business policies, reduce the flexibility and reusability of business processes.

It is possible to control the behavior of protocol participants dynamically, without changing the protocol specification by representing the private internal business policies separate from the generic, public business protocols. However, given a separate policy and protocol specification, it is important that when the participants enact the protocol, the behavior dictated by their local policies be congruent with the protocol specifications. The focus of this thesis is to develop a methodology to configure the private policies of a participant such that, its behavior during protocol enactment is always compliant with the given protocol. We propose a methodology that describes how the agent policies should be added to the protocol specification such that the protocol is not violated during enactment.

INCORPORATING BUSINESS POLICIES INTO BUSINESS PROTOCOLS

BY

LEENA V WAGLE

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

DEPARTMENT OF COMPUTER SCIENCE

RALEIGH, NORTH CAROLINA

AUGUST 2005

APPROVED BY:

DR. DENNIS R. BAHLER

DR. JAMES C. LESTER

DR. MUNINDAR P. SINGH
CHAIR OF ADVISORY COMMITTEE

असतो मा सत् गमय
तमसो मा ज्योतिर्गमय
मृत्योर्मा अमृतं गमय
–ब्रीहदर्णयका उपनीषद

Lead me from the unreal to the Real;
Lead me from darkness to the Light;
Lead me from mortality to Immortality.

- Bryhadaranyak Upanishad

Biography

Leena Wagle was born in Mumbai, which was Bombay earlier, on January 24, 1980. She holds a Bachelor's degree in Computer Engineering, awarded in June 2002, from the University of Mumbai. She joined the North Carolina State University in August 2002. She is currently a Masters candidate in the department of Computer Science, North Carolina State University. She has also been working as Software Engineer Intern at IBM, Tivoli in RTP, NC since June 2004. She will be joining FedEx Services in Memphis, TN, as a software engineer on completion of her graduate studies.

Acknowledgement

I would like to thank my advisor Dr. Munindar P. Singh for his constant support, guidance, and patience. I would also like to thank my committee members Dr. Dennis Bahler and Dr. James Lester for their insightful comments on my work. I would like to thank Ashok for helping me with learn Latex. I would like to thank Yathi for sitting through my mock presentation. I am grateful to the people at the Department of Computer Science who helped me with all the paper work and the exam scheduling. I am also grateful to my parents for always being there for me. Finally I would like to thank Girish; I would not have completed this work without his constant support and help.

Contents

List of Figures

Chapter 1

Introduction

A business process specifies how different parties interact with one another in order to achieve a common business goal. Unlike traditional business processes, processes in open , Web-based setting typically involve complex interactions among autonomous business partners. Conventionally, business processes are modeled as centralized flows, which are horizontally complete, specifying the exact steps for each participant. However, due to the exceptions and opportunities that arise in open systems, business relationships cannot be pre-configured to full detail. Conventional methods of business process modeling do not facilitate flexibility and autonomy of the participants. Traditional approaches like centralized workflows do not provide the flexibility and semantics needed to represent the ever-evolving and exception-prone processes in open, Web-based settings.

Business policies are an integral and important component of business processes. Local policies that govern behavior of the participant during public interaction. These interactions can be modeled as business protocols. Business policies are often expressed and implemented as business rules. It is common to embed business rules within the implementation or enactment of a business process. When the business practices or policies change, as they often do, it is difficult to correctly reflect these changes in the applications that implement the business process. Since these policies are hard-coded, the business process becomes rigid. A minor change in a policy can cause a large impact on the business process application.

[Desai et al., 2005] present OWL-P, a language, tool and methodology to specify busi-

ness processes as a combination of business protocols and policies. They propose that a business process can be conceptualized as a cohesive set of protocols, and be enacted by agents playing specified roles in the protocols. OWL-P is an approach to increase process flexibility by separating the public part of the business process, specified as protocol interactions, from the private reasoning of the partner, expressed in the form of policies. The separation of policies from protocols can also cause the policies to be configured such that the behavior of the agent is not compliant with the protocol. In this case, the advantage gained by separating the public and private logic of the agent is lost. Given a protocol specification, it is necessary to have a guideline about writing the business policies that result in a successful protocol enactment.

We propose a methodology to add the private business policies of a participant to the public protocol interactions between the participants. We represent business partners as autonomous agents enacting a business process by interacting with each other. The public aspects of these interactions are called *protocols* and their private agent specific reasoning are called *policies*. The business policies are implemented as rules using Java Expert System Shell (Jess) [Friedman-Hill, 1997]. The business process participants are implemented as agents using the JADE (Java Agent Development Environment) framework.

1.1 Motivation

Business protocols involve multiple participants and address purposes such as ordering, shipping, payment and so on. In order to maximize the autonomy and heterogeneity of the participants and to make the protocols reusable, it is necessary to emphasize the interactions between the participants. We use agents to implement protocol participants. The agent interactions are in the form of messages exchanged between them. During the protocol enactment, the agents often make promises to one another. These promises, between the agents are expressed as *commitments* and actions on those commitments. Each agent uses its internal business logic to decide how to behave when participating in a protocol. The business logic of each agent is expressed as its business policies.

Whenever an agent needs to make a decision about what action to take next or how to

calculate a parameter value for a message, the agent consults its internal business logic.

In order to honor the autonomy of the participant agents, it is necessary to distinguish between the logic internal and external to the agent. By internal logic we mean the business intelligence of the agent playing a particular role. For example, consider an agent participating in the order protocol as a merchant. When the agent receives a request for quote for a particular item, it has to send the price back to the customer agent so that the order protocol can be successfully completed. For the merchant to send the price of the item, it needs to calculate the price of the goods requested. The calculation of the price is completely dependent on the internal business logic of the merchant role. The logic used by the merchant to calculate the price may vary from one agent configuration to another. Thus the main steps underlying the order protocol and the typical messages exchanged between the customer and the merchant remain same from scenario to scenario. However what changes is the agent's internal business policies. For example, for one scenario a regular customer who orders 100 units of a item gets a 10% discount, whereas if the customer is a returning customer then the merchant might offer a free gift instead of a discount. Thus, policies determine the distinction between business processes as they determine how the underlying business protocols run.

Policies control the behavior of participants in a business protocol and hence shape the entire business process. In order to preserve the autonomy of the agents participating in the protocol, it is necessary that the business policies of the agent be kept separate from the specifications of the business protocol.

A business processes can be viewed as combinations of protocols, spliced or merged together and then loaded with the local business reasoning of the participants. Thus a way to model business processes would be to represent business protocols and represent the business policies and then find a methodology to combine and use these two representations.

We present a methodology that can guide a protocol designer or implementor to write the business policies for a protocol participant. Using these guidelines the implementor can ensure that the participant will play a chosen role with maximum flexibility and minimum protocol violations.

1.2 Representing Business Policies using Rules

Business policies can best be described as business rules. Rules are useful to represent the contents of messages exchanged between roles during a business protocol. For example, to describe products and services, delivery dates, quantities, customer service agreements and other surrounding agreements that constitute the content of the business negotiation. Rules are also useful to represent the relevant aspects of business processes like how to place an order, respond to a message that is received, return an item or cancel a delivery. The usefulness of rules in representing business policies is based largely on advantages in terms of abstraction and flexibility [Grosof and Labrou, 1999].

- *High level of abstraction*: Rule expressions offer a high level of abstraction as compared to procedural code. Rules are more easily understood by humans, especially business domain experts who are typically non-programmers.
- *Flexibility*: Rules are easier to modify dynamically. Rules enable business rules to be specified and modified at run-time than procedural code relatively easily by business domain experts.

Several rule representations exist. We represent business rules for OWL-P in Jess [Friedman-Hill, 1997]; a Java implementation of production rules.

1.3 Challenges

Traditionally, the business policies of the agents are programmed into the business process. The business policies differ from scenario to scenario, from agent to agent, and hence from one process implementation to another. Programming business policies within the business process makes the business process rigid and non re-usable, as the implementation becomes highly customized to the business of the implementor.

Since business policies depending upon several external and unpredictable factors like business strategies, market-status, the business policies are ever-changing.

One of the challenges in developing the methodology is to keep the flexibility of the business policies in mind. Business policies are dependent on factors external to a business process, such as the popularity of a product and competitors' strategies, which cannot be

clearly defined in discrete technical parameters. Hence a methodology to write policies should accommodate the volatile nature and flexibility of the policies.

Since business policies are reviewed by management personnel who need not be programmers, they should be written in a human readable form. Given a protocol specification, our methodology helps the process developer to program policies so that they would cause minimum protocol violations. Thus, it becomes necessary to identify potential protocol violations and how can they be addressed. Mallya and Singh [2005b] introduce a model for exceptions in commitment protocols. Our methodology uses this model as a basis for protocol compliance. Since business policies depend on the business strategy, the business policy definition depends upon the business domain that the policies are defined for. For example, though the shipping protocol for an on line book purchase and on line electronic item purchase might be the same, the shipping policies defined for both business processes might be completely different. It is important that the methodology provides guidelines that do not get into the unnecessary details about the policies controlled by the domain.

1.4 Contributions

The methodology developed will help developers of business process to write unambiguous and clean policies. The main contribution is the set of guidelines to recognize potential protocol violations and then to avoid them. The methodology suggests a proactive approach to protocol programming.

Given the protocol specification, the protocol rules as they apply to the role adopted by the agent, using the methodology will help the developer program the protocol with maximum flexibility.

1.5 Organization

Chapter 2 presents the technical background and the architecture for implementing our methodology. Chapter 3 introduces business protocols, the OWL-P methodology to represent business protocols for building business processes. Chapter 4 explains the guidelines

for writing business policies with an example of Order protocol, a two-party protocol and a Payment protocol, a three-party protocol. Chapter 5 we summarize the contributions of this approach and compare them to the literature.

Chapter 2

Background Concepts

In this chapter we introduce the technical aspects of implementing business process using agents. In this chapter we discuss the technical framework required to program a multiagent system, where all agents participate in a business process. The agents act according to business protocols and use rules to represent internal policies. Next, we discuss the use of agent communication languages in multiagent systems. We also give a brief overview of JADE, an agent development framework, used to implement agents. This chapter also introduces forward chaining and backward chaining inference mechanisms. These inference mechanisms are used to infer about policies. We introduce Jess in this chapter. Jess is the rule production system that implements, manages, stores and infers rules.

2.1 Architecture

We describe the notion of agent, multiagent system and the communication mechanisms used by agents to interact in a multiagent system.

2.1.1 Agents

Agents are computer programs which attempt to perform some set of tasks autonomously for their users in a trustworthy and personalized fashion. These programs can be configured manually or can be automated to learn and adapt from situations or tasks that they are

assigned to do. In addition, an agent can perform more than one task. Agents have been around for quite some time but they are getting increasingly popular with more and more web based applications. Intelligent agents are software entities that carry out some set of operations on behalf of a user or another program with some degree of independence or autonomy, and in doing so, employ some knowledge or representation of the users goals or desires. In the context of this thesis, an agent is an entity that can be implemented and it represents a real-world, autonomous business partner with its local business policies. Agents represent the business partners that participate in a business process. When the agents participate in a business process, they interact with other agents to achieve a business goal. In our work, agents are programmed to perform different functions during the business process enactment. The agent are programmed and managed using an agent development environment. The architecture of an agent participating in a protocol is explained in detail in Section 3.3.

2.1.2 Multiagent System

A multiagent system is a network of agents, which work in a collaborative fashion, to achieve certain goals or complete certain predefined tasks. In our work, the agents represent different business partners and communicate with each other to complete certain business goals. To enact a business process, several agents are configured to interact with one another in a collaborative fashion. Thus, a business process enactment can be looked at as a multiagent system, where agents adopting the role of business partners communicate with one another.

A multiagent system consists of individual agents which have the following properties:

- Autonomy: Agents perform without the intervention of humans. It is the goal of an automated business process that all business partners, once configured and programmed should be able to complete the business process without any intervention.
- Social Ability: Agents interact with other agents via some Agent Communication Language. In the context of our work, agents will exchange business messages with one another to achieve business goal.

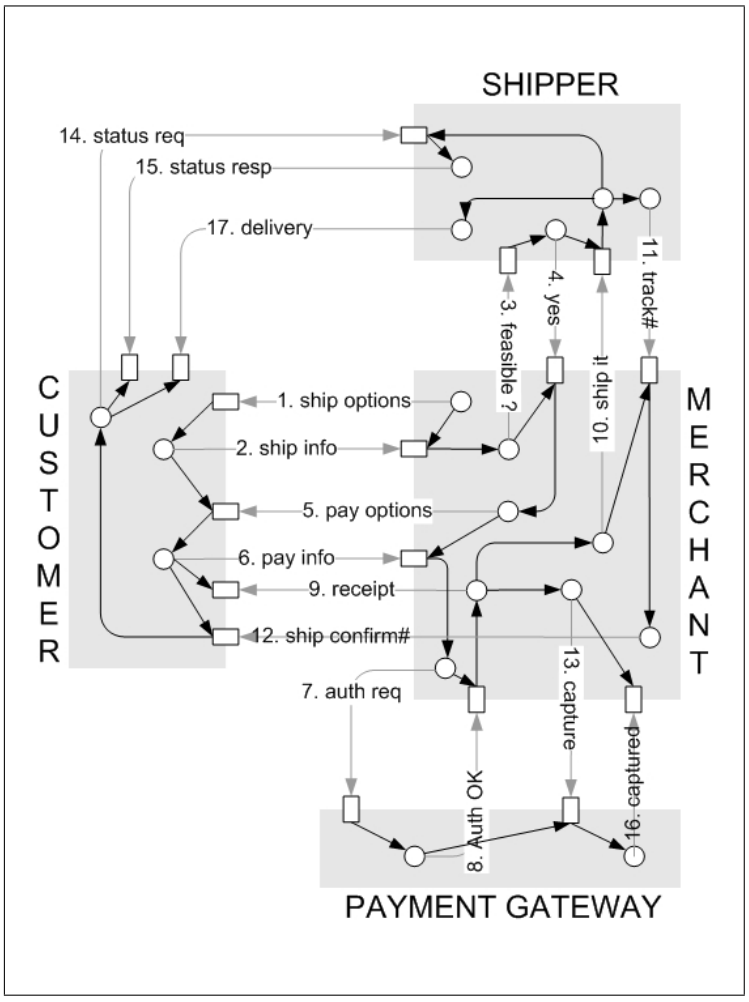

Figure 2.1: Multiagent system enacting a purchase business process

- Reactivity: Agents perceive their environment and respond based on changes in the environment. In the case of business process, these changes are brought about by the progress of the business process.
- Proactiveness: Agents exhibit goal directed behavior by taking an initiative. There will always be an agent that will start the business process.

An example of a multiagent system that enacts a business process is shown in Figure 2.1. The figure depicts a purchase process. The process involves a Customer agent who wants to buy items, a Merchant who sells items, a Shipper agent who ships items and a Payment Gateway who authorizes payments.The payment related interactions are imported

from Secure Electronic Transactions (SET). Each process participant is shown via a separate shaded region. The graph made of dark edges denotes the local flow of each agent. Circular nodes represent internal business logic. Rectangles represent external interfaces through which the agent receives messages. When multiple edges emerge out of a node, all of then can occur concurrently.

2.1.3 Agent communication language (ACL)

Communication plays an important role in multiagent systems. In order that the multiagent system be able to achieve the collaborative goal, it is imperative for the agents to communicate with other agents. Communication supplies agents with valuable information. By setting a common language between various agents language differences between agents can be eliminated. Inter-agent communication is usually defined by the use of an agent communication language (ACL). The ACLs exist in a logical layer above the transport layer. The transport layer is implemented using various protocols such as TCP/IP, HTTP or IIOP and is responsible for the communication at the level of data and message formats whereas ACLs manage the communication on the intent and social levels. One of the most widely implemented and used ACLs in the agent community is Foundation for Intelligent Physical Agents (FIPA) ACL. ACLs are complex structures composed of different sublanguages that specify message content, ontology, and propositions. They are tailored to support the various kinds of collaboration, negotiation and information transfer required in multiagent interaction.

2.2 JADE

JADE (Java Agent Development Framework), [JADE], is a software development framework aimed at developing multiagent systems and applications conforming to FIPA standards for intelligent agents. It includes two main products: a FIPA-compliant agent platform and a package to develop Java agents. JADE has been fully coded in Java language and is made of various Java packages, giving application programmers both ready-made pieces of functionality and abstract interfaces for custom, application dependent tasks. Agents

are implemented as Java threads and live within Agent Containers that provide the runtime support to the agent execution.

JADE is a middleware that facilitates the development of multi-agent systems. It includes:

- A runtime environment where JADE agents can "live" and that must be active on a given host before one or more agents can be executed on that host.
- A library of classes that programmers have to/can use (directly or by specializing them) to develop their agents.
- A suite of graphical tools that allows administrating and monitoring the activity of running agents.

The communication architecture of JADE, offers flexible and efficient messaging, where JADE creates and manages a queue of incoming ACL messages, private to each agent. The agents can access their queue via a combination of several modes: blocking, polling, timeout and pattern matching based. The full FIPA communication model is implemented and its components are clearly distinctive and fully integrated: interaction protocols, envelope, ACL, content languages, encoding schemes, ontologies and transport protocols. JADE has also been integrated with JESS, a Java shell of CLIPS, in order to exploit its reasoning capabilities.

We use JADE to implement the agents that adopt the different roles in protocol enactment. The messaging interface of the agents is also implemented using JADE.

2.3 Rules

As mentioned in Section 1.2, business policies naturally occur in rule form. We choose to represent business policies using Production Rule System. Production rule systems are able to represent the complex relationship between business policies. expressed as rules. An additional advantage of production rule system is that the frequent changes in business policies can be managed well without affecting other aspects of business process that depend upon business policies. Rules in production system consist of a collection of IF-Condition-THEN-Action statements. Each rule has a left-side, corresponding to the IF-

Condition part. The left-side is the Antecedent of the rule. The right-side, THEN-Action part, of the rule is the Consequent of the rule.

Production rules can be inferred by two main inference mechanisms, namely forward-chaining and backward-chaining.

2.3.1 Facts

Facts are assertions that are considered to be true at a given time in the context of the multiagent system. We represent facts as predicates. The facts can be a conjunction(logical AND), disjunction (logical OR) or negation (logical NOT) of a predicate. In the context of this work, negation is interpreted as "negation-as-failure".

As an example, consider the fact represented in Listing 2.1. The $requestForQuote$ is the name of the predicate. The strings with $?$ preceding them represent the variables passed as parameters to the predicate. The agent knowledge base is programmed to infer the predicates using rules or functions. These rules or functions assign values to the variables.

Listing 2.1: Examples of representation of a fact

```
1
2 ;;Single Predicate
3  requestForQuote(?tranID,?itemID)
4
5 ;; Predicate as Conjunction
6
7    quote(?tranID,?itemID,?itemPrice) ∧
8 createCommitment(Merchant,CC(Merchant,Customer,pay(?itemPrice),goods(?
      itemID)))
9
10 ;; Predicate as negation
11 ¬authOKPolicy(?cardNo,?expDate)
```

2.3.2 Forward chaining inference rules

Forward chaining inference engines process initial, known information or facts to infer the final state or goal. Starting from an initial or current data, the forward chaining inference engine makes a chain of inferences till a goal is reached. In forward chaining the slots are matched against the antecedent of rule (namely the IF part). Whenever the antecedent matches the context, the inference engine executes the consequent of the rule. Many a times the consequent changes the facts, thus modifying the context. If the new facts match the antecedents of more rules, the inference engine repeats the match-execute cycle.

The use the forward-chaining to infer the desired goal using the active facts and assertions is already programmed into the agent's knowledge base.

Consider the Listing 2.2 as an example of forward-chaining rule. In this case, the antecedent of the rule is the predicate $requestForQuote$. The symbol $\Rightarrow$ shows that the rule will be inferred by forward-chaining. The consequent of the rule is a conjunction of two predicates, namely, $quote$ and $createCommitment$.

Listing 2.2: Example of a forward chaining rule

```
1
2 requestForQuote(?tranID,?itemID)
3     ⇒
4     quote(?tranID,?itemID,?itemPrice) ∧
5   createCommitment(Merchant,CC(Merchant,Customer,pay(?itemPrice),goods(?
        itemID)))
```

2.3.3 Backward chaining inference rules

Backward chaining inference is drawn by starting with the final goal and then trying to obtain the facts to infer the final desired goal. Starting with the final goal, the inference engine breaks down the goal into simpler sub-goals. The process of goal decomposition continues recursively until additional subgoals or facts can be obtained. The backward chaining inference engine uses rules to facilitate the process of goal decomposition. If the consequents of the rule matches against an active goal, the antecedent if the rule gives

the new subgoals. The process of matching the consequents and decomposing the goal continues until no rules can be found, the consequents of which match any active subgoals. The backward chaining inference engine now matches the set of active goals against the facts. of there is a match then the rule that provided the subgoals executes. The inference engine, thus, uses the new facts to match the new set of subgoals. This process continues till the inference engine can infer the original final goal. Thus backward chaining a goal-oriented approach toward inferencing about the agent context.

The use the goal-oriented reasoning to derive the desired goal from the facts, assertions and rules already programmed into the agent's knowledge base.

Consider the Listing 2.3 as an example of Backward-chaining rule. In this case, the goal of the rule is the predicate $quotePolicy$ predicate. The symbol $\Leftarrow$ shows that the rule will be inferred by backward-chaining. The subgoal of the rule is $calculatePrice$ predicate. In this case, the $calculatePrice$ predicate is programmed to return the value of variable $?itemPrice$. The symbol $\leftarrow$ indicates that the value of variable $?price$ is assigned to variable $?itemPrice$. Another predicate $calculate$ returns a value that is assigned to $?price$ variable. The $calculate$ predicate is programmed to call a function to return a value processed using $?itemID$, $?rate$ and $?discount$ as parameters. The notation $?rate\ RATE$ is to express that the value of constant $RATE$ is assigned to $?rate$ variable.

Listing 2.3: Example of a backward chaining rule

```
1
2 quotePolicy(?itemID,?itemPrice))
3  ⇐
4  calculatePrice(?itemPrice ← ?price&:calculate(?itemID,?rate RATE,?
       discount DISCOUNT))
```

2.4 Jess

We use Jess as the rule system to represent protocol and policy rules.

Jess (Java Expert System Shell) [Friedman-Hill, 1997] is a rule engine and scripting language developed at Sandia National Laboratories. Jess (Java Expert System Shell)

is a representative member of one group of currently commercially important rule systems, namely, production rule systems descended from OPS5 [Cooper and Wogrin, 1988], which in turn are closely related to event-condition-action (ECA) rule systems [Ullman and Widom, 1997]. These systems primarily apply forward-chaining (rather then backward), and their applications heavily rely on their capabilities for procedural attachments. This group of rule systems is sometimes also called as "reactive" rule systems; since the rules often run in response to the arrival of knowledge-base updates consisting of "facts" or "events".

Jess offers seamless interoperability with Java objects. Jess rules can be fired from Java and Java objects can be accessed from Jess environment. The Jess knowledge base can be accessed from Java. These features allow us to easily augment agent behavior programmed in JADE to business policy rules in Jess. RuleML [RULEML] is an emerging industry standard for XML rules and can be used represent business rules. Grosof et al. [2002] show how to translate rules, syntactically encoded in RuleML into Jess. Thus, the business rules can be directly written in Jess or they can be translated from RuleML into Jess rules.

Jess rule definition begins with the *defrule* keyword followed by the rule name. Since Jess is a forward chaining inference rule engine, Jess rules have using the norms similar to those described in Section 2.3.2. Listing 2.4 gives a sample Jess rule. Note that Jess also has a mechanism to add procedural attachments to predicate. Using these procedural attachment, like Jess functions, Jess can assign and bind values to variables. The name of the Jess rule is $send - accept$. This Jess rule describes what happens when a customer sends an accept message to a merchant in an order protocol. The Jess rule also represents how Jess variables can be bound to Java variables using the new keyword.

Listing 2.4: Example of a Jess rule

```
1
2 (defrule send-accept
3     (and (offer (m_nTransID ?t) (m_szQuoteFor ?x)(m_szPrice ?y)) (accept
          -offer ?x ?y))
4           =>
5    ;; Create the conditional commitment ``goods if accept"
6    (bind ?cc-var (new ConditionalCommitment ``merchant" ``customer" ``
          ACCEPT" ``GOODS" ?t))
7    (definstance ccom ?cc-var static)
8    ;;Create the accept message
9    (bind ?accept-var (new Accept ?t))
10   (definstance accept ?accept-var static)
11   (store OUTPUT ?accept-var)
12   ;; Create the conditional commitment ``pay if goods"
13   (bind ?cc-var (new ConditionalCommitment ``customer" ``merchant" ``
          GOODS" ``PAYMENT" ?t))
14   (definstance ccom ?cc-var static))
15   ;(printout t ``ACCEPT sent is" ?accept-var crlf))
```

Chapter 3

Representing Protocols

In this chapter we discuss the mechanism of protocol specification. We introduce business protocols and explain commitments and actions on commitments. Next, we describe the OWL-P methodology for protocol specification. We discuss protocol skeletons with the help of order protocol example. We also explain a usage scenario for OWL-P and then describe how agent specific protocol view is represented.

3.1 Protocols

A business protocol represents the interactions between two or more agents to achieve a goal. Thus the protocol defines the roles played by the agents, the messages exchanged between the roles, the message parameters and, most importantly, the goal that the roles are trying to achieve.

Definition 1 *A business protocol is a modular, public specification of an interaction among different roles that achieves a desired goal.*

Thus a protocol is a description of the steps involved in an interaction between two or more agents. Protocols make interactions coherent and easy to implement. Protocols enable unambiguous and smooth interactions among agents. A protocol needs to have a clear definition and its specification must be clearly understood by all the participants.

A business protocol involves multiple roles and addresses specific purposes such as ordering, payment and shipping. We represent protocols as shown in Figure 3.1 and Figure 3.2. Multiagent systems require protocols to be specified at a high level of abstraction, to accommodate the complexity of agent systems, and not to overwhelm protocol designers with unnecessary details.

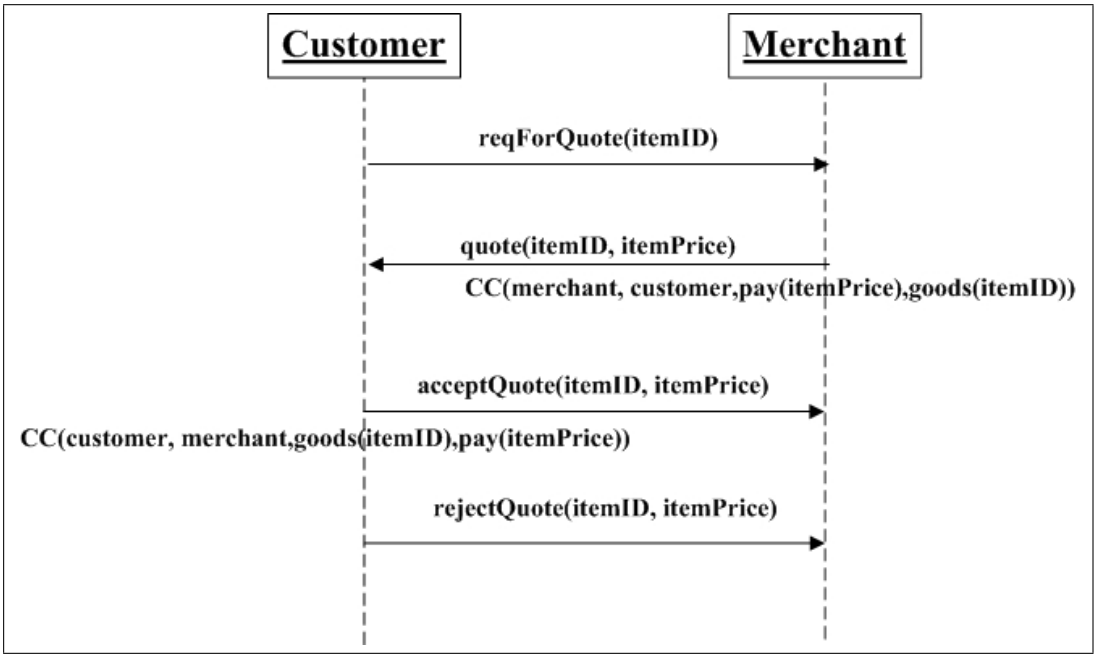

Figure 3.1: Order Protocol

3.1.1 Commitments

The contractual relationships between the protocol roles are expressed through the notion of *commitments* as described in [Singh, 1999]. In order to accommodate protocol flexibility, there is a need to specify the behavior of participants adopting a protocol role and also how the contractual relationships among the participants evolve over the course of an interaction. Commitments capture obligations that one agent, adopting a role, has toward another agent because of a promise. This promise is a result of a contract that is integral to the protocol. Commitments are legally enforced. As is proposed by [Mallya and Singh, 2005a], commitments allow flexible execution of protocols and help agents reason

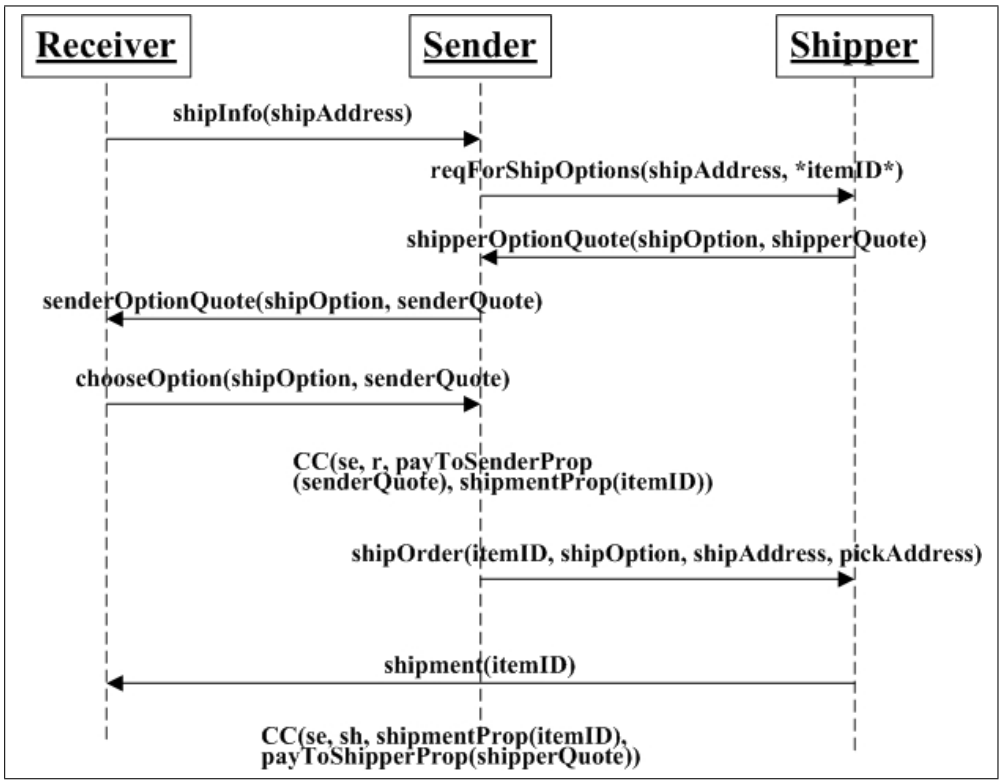

Figure 3.2: Shipping Protocol

about protocols and plan their actions accordingly, at the same time commitments also provide a basis for compliance checking. Violations of commitments are contract violations and hence are potential causes of protocol violations as well.

By defining states that need to be reached to achieve the goal, commitments allow multiple paths to achieve states and hence result in a flexible protocol specification.

Definition 2 *A commitment C(x, y, p) denotes that an agent x is responsible to the agent y for bringing about the condition p.*

Thus for the commitment *C*(x, y, p), x is the *debtor*, y is the *creditor* and p is the *condition*. The condition p is expressed in suitable formal language. Commitments are also conditional.

Definition 3 *A conditional commitment CC(x, y, p, q) denotes that an agent x is responsible to an agent y for bringing about the condition q as long as the condition p holds.*

Condition p is the precondition of the commitment.

For example, a customer c's agreement to pay the price $itemprice$ of the $item$ after it is delivered is a commitment that the customer c has towards the merchant m. This commitment will be expressed as *CC*(c, m,pay($c, m, itemPrice$),deliver($m, c, item$)).

Commitments are created, satisfied, and transformed in certain ways during the protocol execution. The following operations that can be performed on commitments were introduced in [Singh, 1999].

1. CREATE(*x,c*) establishes the commitment *c* in the system *s*. This operation can only be performed by the debtor of the commitment x.

2. DISCHARGE(*x,c*) satisfies the commitment *c*. It can only be performed by the debtor *x*.

3. CANCEL(*x,c*) cancels the commitment *c*. It can only be performed by the debtor *x* of the commitment. Generally, cancellation of a commitment is followed by the creation of another one to compensate for it.

4. RELEASE(*y,c*) releases the debtor in the commitment *c*. This operation can only be performed by the creditor *y* of the commitment.

5. ASSIGN(*y,z,c*) replaces *y* with *z* as the creditor of the commitment *c*. This operation can only be performed by the creditor *y* of the commitment.

6. DELEGATE(*x,z,c*) replaces *x* with *z* as the debtor for the commitment *c*. This operation can only be performed by the debtor *x* of the commitment *c*.

We note that a commitment has to be created to exist. For every operation on commitments listed above, we also define a corresponding predicate. These predicates are represented by the commitment operation name,with lower case characters, augmented with the word *Commitment*. The predicates are true when the corresponding operation succeeds.

Any policy that violates the contractual relationship would cause a protocol violation. This entails that all the contractual relations be respected to determine whether the policies

are complaint with stated protocols. It is the business policies that determine the interactions between the protocol roles. In order to develop a guideline for writing policies, it is important to understand the contractual relationships that result out of the protocol.

3.1.2 OWL-P

Desai et al. [2005] introduce OWL-P, a language, tool and methodology to represent business processes as a combination of business protocols. They propose an agent-based approach for business process modeling and enactment which is centered around the concepts of commitment-based interactions. OWL-P is an ontology based on Web Ontology Language (OWL).

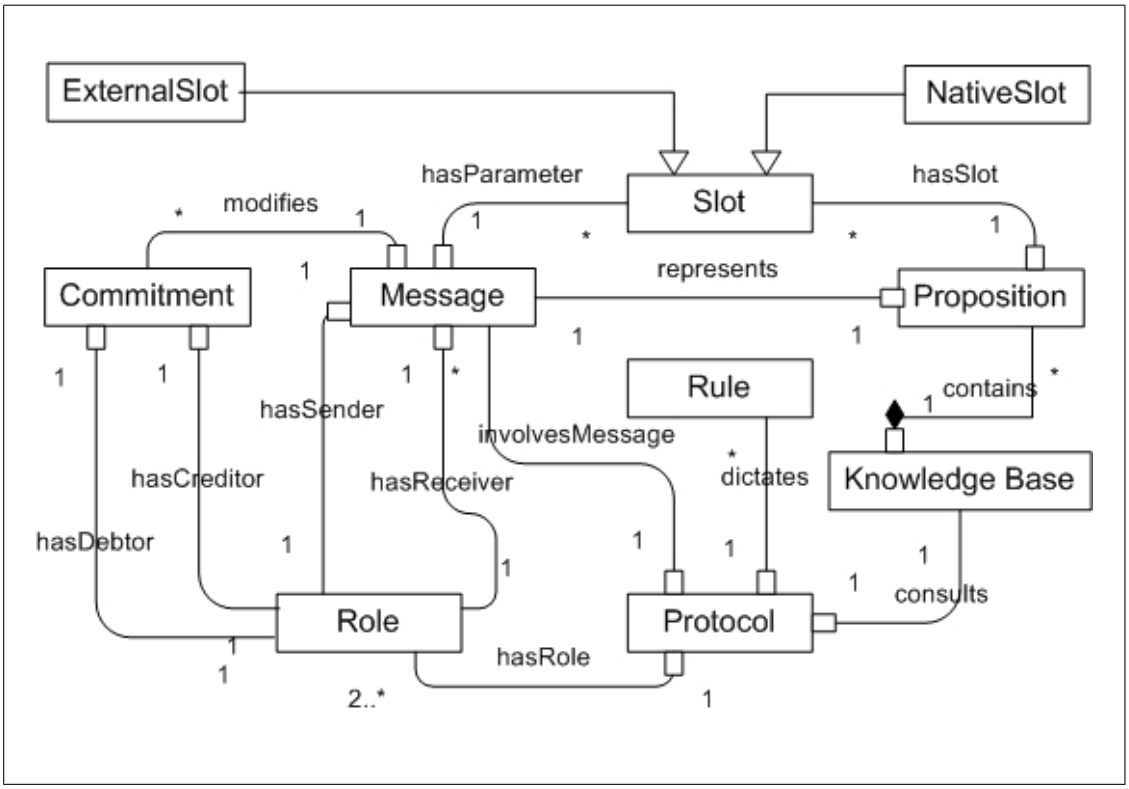

Figure 3.3: Basic OWL-P ontology

In OWL-P Specification the protocol is specified using rules called *protocol logic*. Chopra et al. [2004] employ Semantic Web Rule Language (SWRL) [Horrocks et al., 2005] for defining rules. Grosof et al. [2002] explain how to translate SWRL rules into Jess. We

express protocol rules as Jess rules for the purpose of implementation. To specify rules, we use the notations introduced in Section 2.3. The OWL-P ontology elements and their properties are shown in Figure 3.3. In the figure, an entity with a little rectangle represents the domain of the corresponding property.

A protocol has multiple *roles*. A *Message* has a sender and a receiver and can have several *Slots* as its parameters. The operational semantics for messages is provided in terms of their effects on the *Commitments* among the roles. Messages can be thought of as operators in commitments. Slots are analogous with data variables. A slot is said to be defined when it is assigned a value and it is said to be used when its value is assigned to another slot. A slot in a protocol may be assigned a value produced by another protocol and hence is represented as *External Slot*. An external slot is untyped until it is given the type of the external slot. By contrast a *Native slot* is typed and defined inside the protocol. A *Protocol* dictates several rules and consults a *Knowledge Base*. A knowledge base consists of *Propositions*. A proposition in a knowledge base may correspond to a message, an active commitments other specific propositions. Every message *msg* is presented in the knowledge base as *msgProp*.

3.1.3 Processes as combination of protocols and policies

Figure 3.4 shows the conceptual model for treatment of business processes based on protocols and policies, as discussed in OWL-P [Desai et al., 2004]. Here the boxed rectangles are abstract entities (interfaces), which must be implemented and combined with business policies to yield configurable entities that can be fielded in a running system (rounded rectangles). Abstract entities can be published, shared, and reused among the process developers. In OWL-P a business protocol is specified in terms of a set of rules termed *protocol logic*. Protocol logic encodes the interaction of participating *roles*. Roles are abstract and are adopted by agents when it participates in the protocol.

The protocol logic specifies the protocol from the global perspective, a *protocol skeleton (P-Skel)* specifies the protocol from the perspective of one of the participant roles. Thus, each P-Skel defines the behavior of the respective roles with respect to the given protocol. In other words, the P-Skel gives the role's view of the protocol.

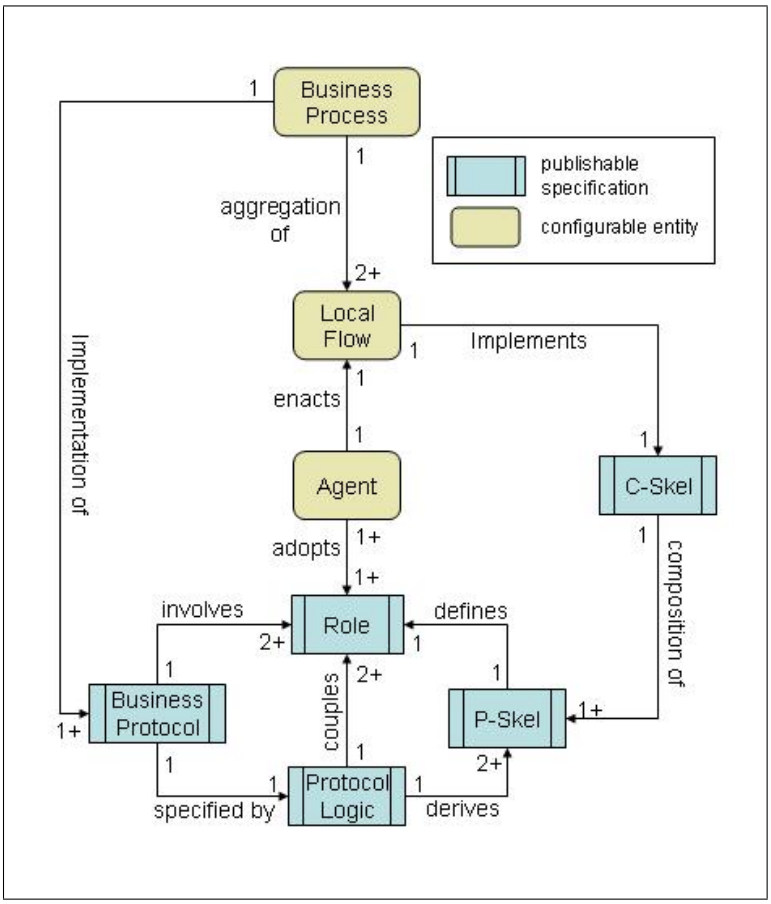

Figure 3.4: Conceptual model for business processes based on protocols and policies

An agent may participate in multiple business protocols by adopting a role in each of them. For example, a bookstore may adopt the role of a seller while interacting with customer and the role of a buyer while interacting with publishers. When an agent needs to participate in multiple protocols, a *composite skeleton (C-Skel)* can be created by combining P-Skels, one of each role that the agent plays in the given protocols. For example, in a supply chain process, a supplier would be merchant when interacting with a retailer in a trading protocol and would be a client in a shipping protocol for sending goods to the retailer. The C-Skel for such a supplier would be composed by splicing the P-Skels for a trading merchant and a shipping customer. This C-Skel specification could be published and then reused for developing other supplied processes.

An agent stipulates its internal business policies in terms of a set of rules termed as

business logic. Thus the *local flow* of an agent is an executable realization of a P-Skel and in case the agent assumes different roles in different protocols then the *business logic* becomes a realization of the C-Skel. The local flow of the agent consults the business logic to make decisions. Thus, the combination of a C-Skel with business logic entails the desired local flow, which may be represented in a flow language. A *business process* thus becomes an aggregation of the local flows that all the agents participating in it. Conversely, a business process can also be looked at as an implementation of the constituent business protocols.

3.2 Skeletons and Policies

The P-Skel provides the agent with the view of the protocol from the perspective of the role that the agent adopts. The P-Skel contains the protocol messages that the agent can receive and send during the protocol enactment. The P-Skel also describes the contractual interactions that the agent engages in, as commitments and actions on those commitments. Generation of the role skeleton is not enough to obtain the local process of an agent.

The local process of the agent combines the policy rules and the protocol rules. This local process defines the agent's actions, commitments and decision making capabilities. Some protocols rules may be *abstract*. Here *abstract* means that some native slot values in the P-Skel rules must be derived from the private business logic of the role. These native slots may be values of decision variables, values of message parameters or actions on commitments like commitment creation, delegation. The business policies of a role are decision makers which are consulted when any decision needs to be made by the role. If the protocol designer can identify the events in the P-Skel that require the agent to consult the business policies to make decisions to evaluate slot values, then the protocol designer can easily separate the business logic from the protocol logic. The distinction between the protocol logic, that is the public interface of the agent, and the business policies, the internal intelligence of the agent, will help to keep the business process development easy to manage.

3.2.1 Representing P-Skels

Figure 3.5 shows the merchant P-Skel in an order protocol. The propositions in the merchant's knowledge base change with incoming and outgoing messages. The rectangles indicate the message propositions that are true for the role at a given time. Thus, the contents of each rectangle can be considered to be the new state of the agent's knowledge base. The transitions from one state to another are labeled by the incoming or outgoing messages that cause new message propositions to be added to the knowledge base. The incoming messages are shown using dotted lines and the outgoing messages are shown using solid lines. For simplicity, only new message propositions that are added to the knowledge base are shown in rectangle. When a requestForQuote message is received, the requestForQuoteProp is true and is indicated as a state labeled as *requestForQuoteProp*. The states are also labeled with any propositions asserted by actions on commitments. In Figure 3.5, the requestForQuoteProp state also contains a createCommitment label. This indicates that when a *quote* message is sent, a Conditional Commitment is created.

We also show the P-Skel for the customer role in Figure 3.6.

3.3 Agent architecture

Figure 3.7 shows the interplay between policy rules and protocol rules of an agent. the agent receives messages from agents which are in the public domain. The messaging interface of the agent is implemented using JADE. These messages cause the message propositions to be asserted in the knowledge base of the agent. The updates in the knowledge base cause the business policy rules to be fired by the rule engine. These business rules invoke the business logic. The policy rules might add more facts to the knowledge base, thus changing the knowledge base state. As a result the agent may send messages to other agents, or engage in commitment actions. We have used Jess to program, manage and store the protocol and policy rules of the agent.

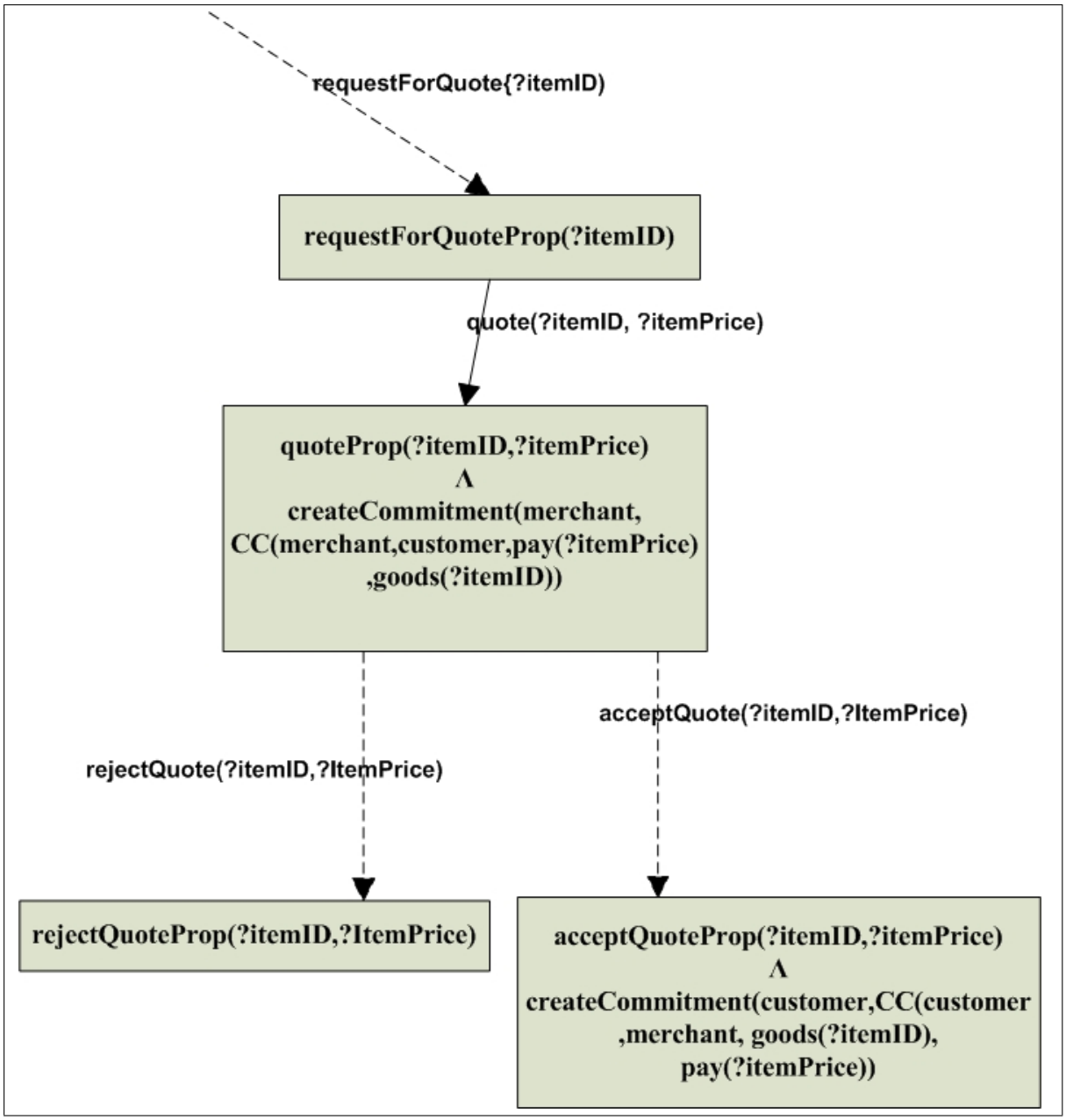

Figure 3.5: P-Skel representation for a merchant role in order protocol

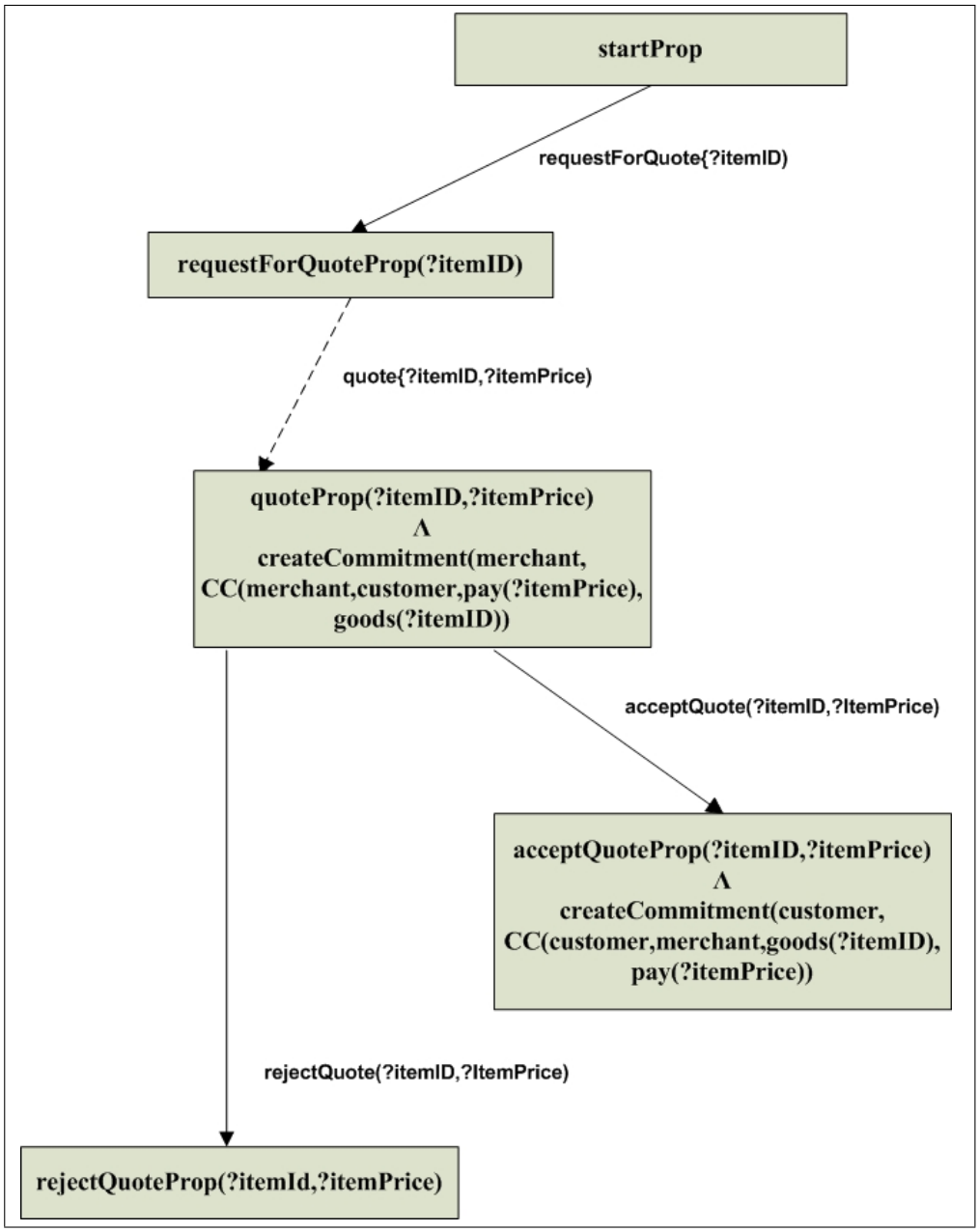

Figure 3.6: P-Skel representation for a customer role in order protocol

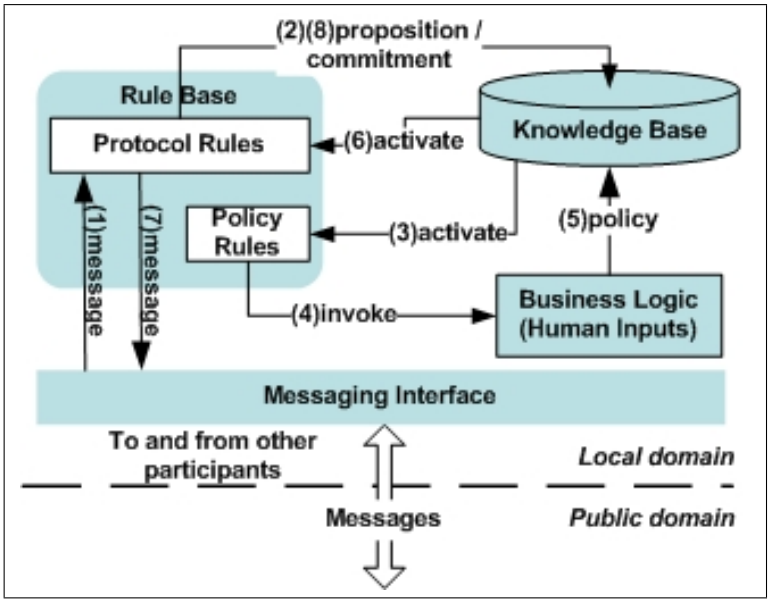

Figure 3.7: Agent architecture: Protocol and policy interplay

Chapter 4

Policies

Now we explain the guidelines to plug in policies into the protocol role skeletons generated using the OWL-P specification of protocols. We describe how to recognize the need to insert business policy rules into a protocol role skeleton. For this, we introduce the notion of policy invokers, which are propositions added to the protocol skeleton rules. The policy invoker propositions invoke the business policy rule base of the agent, that adopts a particular role in a protocol enactment. Having added the policy invoker proposition, there is a need to define a rule or rules that infer the policy invoker; for which we introduce policy checkers. We also present a classification of policy checkers depending upon how they are implemented and show that policy checkers can express different types of business rules.

This chapter also discusses a methodology to add the business policy rules to the protocol skeleton. We start with the OWL-P specification of the protocol and the P-Skels for the roles in the protocol. We then identify the business policy plug-in points, add policy invokers and policy-checkers to each P-Skel. We also show that the resultant local process of each agent is compliant with the individual P-Skel of each role. Thus, we show how to handle the different types of business policies and to incorporate these business policies into business protocols, keeping the policies compliant with the original protocol.

4.1 Representing Policies

In order to represent business policies, it is necessary to develop a conceptual model of business policies. This model also explains how business policies can be broken down into business rules.

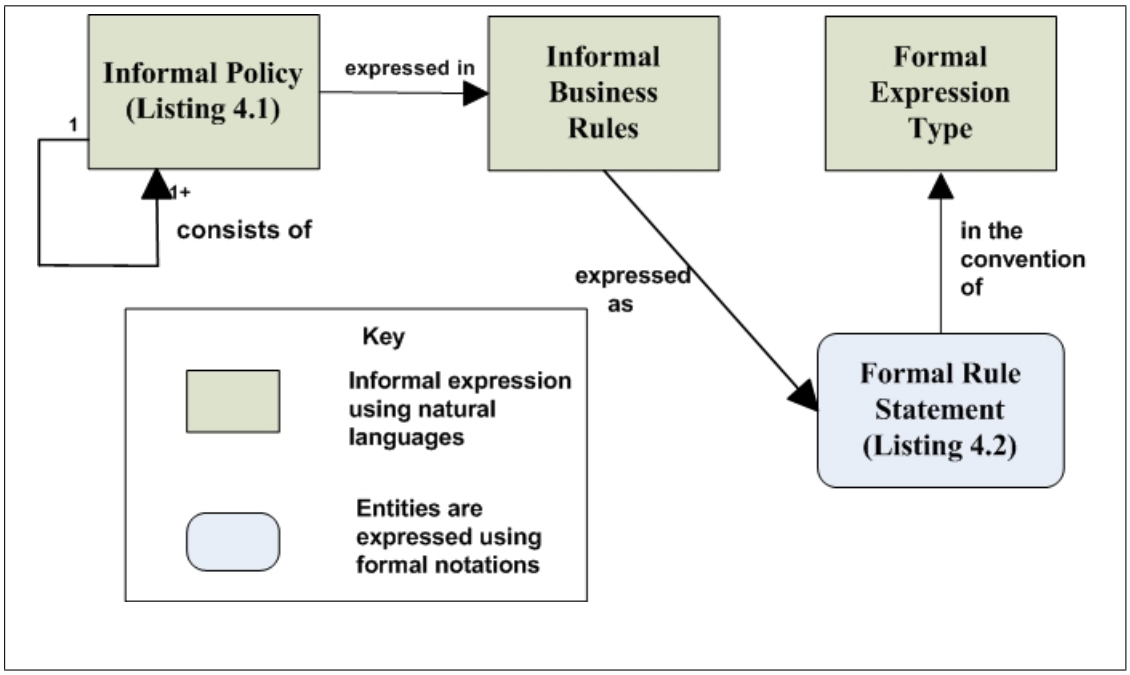

Figure 4.1: Model of business policies

Figure 4.1 explains a conceptual model of business policies and shows the relationships between business policy statements and business rules. A business policy can be considered to be a statement of direction for an agent participating in a business protocol. A generic business policy may be composed of several specific business policies. A business policy is expressed in the form of a business rule statement. The business rule statement can be formally expressed as a business rule, which, is an *atomic* entity. A business rule cannot be decomposed further into more detailed business rules. A business rule can then be expressed in a formal rule statement. The formal rule statement is in convention with a formal expression type.

We describe the conceptual model of business policies using an example. Consider the business policy for calculating the discount given on the price of an item being sold in an

order protocol. When a merchant receives a request for a quote from a customer, it will consult its business policy rule base to calculate the price of an item and will then send the price, as a part of a quote message to the customer. This business policy is called the *Discount Policy*. The *discount policy* is consulted by the merchant to calculate the price of the item being requested. The details of the *discount policy* are explained in Listing 4.1. The *Discount Policy* is a generic business policy that is composed of specific policies like *Returning Customer Discount* policy and *Sale Voids Discount* policy. Each numbered statement in Listing 4.1 is an informal business rule statement. Statement 3, *Sale voids Discount*, cannot be broken down further into business rule statements. But, Statement 2, *Returning Customer*, can be easily broken down into more specific atomic business rules such as *what defines a 'Returning Customer'* and *what defines a 'New Customer'*. Listing 4.2 gives the formal rule statement for Statement 3. This formal rule statement can then be expressed in a standard rule language. We have chosen to represent the business rules using Jess.

As explained in Section 2.3, there are two formal rule inference mechanisms, namely, forward chaining and backward chaining. Jess rule engine is a production rule engine that infers primarily using forward chaining.

Listing 4.1: Example of informal Discount Policy for a merchant

1. *Returning Customer Discount*: A discount of 10% is given to all customers who are returning customers with an order of more than 100 books.
2. *New Customer Discount*: If the customer is a new customer then the customer is given a free gift for placing an order.
3. *Sale Voids Discount*: If there is a sale going on, then all other discounts are void.
4. *Preferred Customer Discount*: For "Preferred Customer"; the price is always 20% off the calculated price.
5. *Maximum Discount*: 20% is the maximum discount a customer can get.

Listing 4.2: Example of informal business rule from the Discount Policy for a merchant

```
IF sale.on THEN
SET discount = 0
```

4.2 Policy Checkers and Policy Invokers

Once the P-Skel for a role is available, the software designer can plug-in the business policies to generate the local process for the agent. In order to formulate the local process for an agent, it is necessary to identify where the agent should refer to its business policies. The next step is to incorporate a functionality to infer from the business policy rules. To identify where to refer to the policies, we define the notion of *Policy Invokers*. A policy invoker helps the designer identify the rules in the P-Skel that need to consult or invoke the business policies. In order to incorporate a functionality to infer the policy invoker from the business rule knowledge base, we define the notion of *Policy Checkers*. A *Policy Checker* is the a rule that checks the policy rules that infer the policy invoker from the business rule knowledge base of the agent. A policy checker triggers the action of firing the business rules. The inferences drawn from these business rules help in deciding the actions that the agent can perform.

In order to add policies to a P-Skel rule, we perform two steps :

1. *Insertion of Policy Invoker into the P-Skel rule.*

   A P-Skel rule is a forward-chaining rule with one or more message propositions as the antecedent (the message proposition is asserted when an incoming message is received). The consequent of the rule is a conjunction or disjunction of outgoing message propositions and commitment action propositions. The P-Skel rule is augmented with a policy invoker. The policy invoker is a proposition that is logically ANDed with the antecedent of the P-Skel rule. The logical AND operation denotes that a business policy is invoked if and only if the corresponding incoming message

is received, i.e., the incoming message proposition is true. To maintain a naming standard for policy invoker, we follow the norms outlined below:

- The policy invoker is denoted as a single atomic proposition.
- The name of the policy invoker consists of two parts: The first part is the name of the outgoing message, that the policy affects; and the second part is the word *Policy*. The outgoing message proposition is found in the consequent of the P-Skel rule.
- The policy invoker proposition may have one or more parameters. These parameters are the P-Skel slots which are undefined by the P-Skel rules. These slots may be native or external.

2. *Addition of Policy Checkers*

   Once a policy invoker is added to the P-Skel, we need a mechanism by which the appropriate policy rules can be checked. This is achieved by adding a backward-chaining rule that checks the policy rule knowledge base, which we call Policy Checker. Policy checkers assign values to undefined native or external slots or make decisions about agent actions. A call to the policy knowledge base returns the truth value of the policy invoker. Thus, the policy checkers infer the policy invokers. When policy checkers are added to the P-Skel, we obtain a *concrete* local process of the agent. The policy checker may be a single rule or several rules that express the business rule statements of the agent.

Consider the example of a Merchant P-Skel rule in the Order Protocol given in Listing 4.3. The protocol designer identifies that this rule is a candidate rule for addition of a policy invoker. We define how to recognize the candidate P-Skel rules for addition of policy invokers in Section 4.2.1. Listing 4.4 shows the same P-Skel rule after addition of policy invoker and policy checker. The policy invoker and policy checker have been highlighted. The policy checker will derive the value of the $?itemPrice$ variable. When the $?itemPrice$ variable gets a value then the $quotePolicy$ policy invoker is asserted and a $quote$ will be sent. The policy checker, contains several rules that define the mechanism to compute the value of the $?itemPrice$ variable. For simplicity we have listed only one such

rule in Listing 4.4.

4.2.1 Identifying Policy Invokers

In order to manage the policy invokers, it is important to identify where to insert the policy invokers in the P-Skel. We identify three P-Skel rules that are candidates for policy invoker augmentation.

1. *Rules with undefined slots*

   When a P-Skel is derived from a protocol specification using the OWL-P methodology, the derived P-Skel outlines the local process for a protocol role. The P-Skel contains external slots or variables that are not defined within the P-Skel specification. The values of these external slots cannot be derived from any of the other P-Skel rules. To assign values to these slots, it is necessary for the agent to do some internal processing or consult some other agent. The P-Skel rules containing such external undefined variables need to be augmented with policy invokers.

   As an example, consider the P-Skel rule in the merchant role of an order protocol given in Listing 4.3.

Listing 4.3: Example of merchant P-Skel rule in order protocol

```
1 ;;The merchant role sends a quote to the Customer on receiving the
     requestForQuote message. Merchant also creates a Conditional
     Commitment to send the goods if the Customer pays for the goods.
2 requestForQuote(?tranID,?itemID)
3    ⇒
4    quote(?tranID,?itemID,?itemPrice) ∧
5   createCommitment(Merchant,CC(Merchant,Customer,pay(?itemPrice),
       goods(?itemID)))
```

   In the Listing 4.3, on Line 4, the $?itemPrice$ variable which is used on the consequent of the rule is not defined in the antecedent of the rule. Since the P-Skel rule is a forward-chaining rule, the value of $?itemPrice$ cannot be derived from other P-Skel rules. This is an external slot and the software designer needs to specify

some way to bind this variable to some value. This rule is a candidate for inserting a policy invoker. The policy invoker in this case, will calculate the value of the slot $?itemPrice$. The same rule augmented with a policy invoker is given in Listing 4.4. The $quotePolicy(?itemID, ?itemPrice)$ on Line 4 is the policy invoker. In addition, there is also a need to write the policy checker. The policy checker for this policy invoker will have all the rules that derive a value for the $?itemPrice$ variable. We give an example of a policy checker in Line 11 of the Listing 4.4. In Listing 4.4, the policy invoker and the policy checker are written in bold letters.

Listing 4.4: Example of merchant P-Skel rule in order protocol with policy checker and policy invoker

```
1
2  ;; P-Skel rule with quote Policy Invoker
3
4  requestForQuote(?tranID,?itemID) ∧ quotePolicy(?itemID,?itemPrice)
5      ⇒
6      quote(?tranID,?itemID,?itemPrice) ∧
7  createCommitment(Merchant,CC(Merchant,Customer,pay(?itemPrice),
       goods(?itemID)))
8
9  ;; Policy Checker for quotePolicy
10 ;; calculatePrice assigns a value to ?itemPrice variable. The
       calculate function calculates and returns the price. This price
       is stored in variable ?price and then assigned to ?itemPrice.
11
12  quotePolicy(?itemID,?itemPrice))
        ⇐calculatePrice(?itemPrice ← ?price&:calculate(?itemID,?rate RATE,?discount
        DISCOUNT))
```

2. *Rules with branching actions*

   Another P-Skel rule that needs insertion of policy checkers is a P-Skel rule that needs to make a choice between different actions. When the P-Skel contains the same incoming message and different outgoing messages, there is a need of some mechanism that will help the role decide which action to take during the protocol enactment. This decision is internal to the agent and not a specified as a part of the protocol role skele-

ton. The business policy of the role will decide which outgoing message to send. This P-Skel rule is a candidate to insert policy invokers.

Consider the example of the authentication rule in the P-Skel for the gateway role in a standard three-party payment protocol. The payment protocol is specified in Figure 4.2.

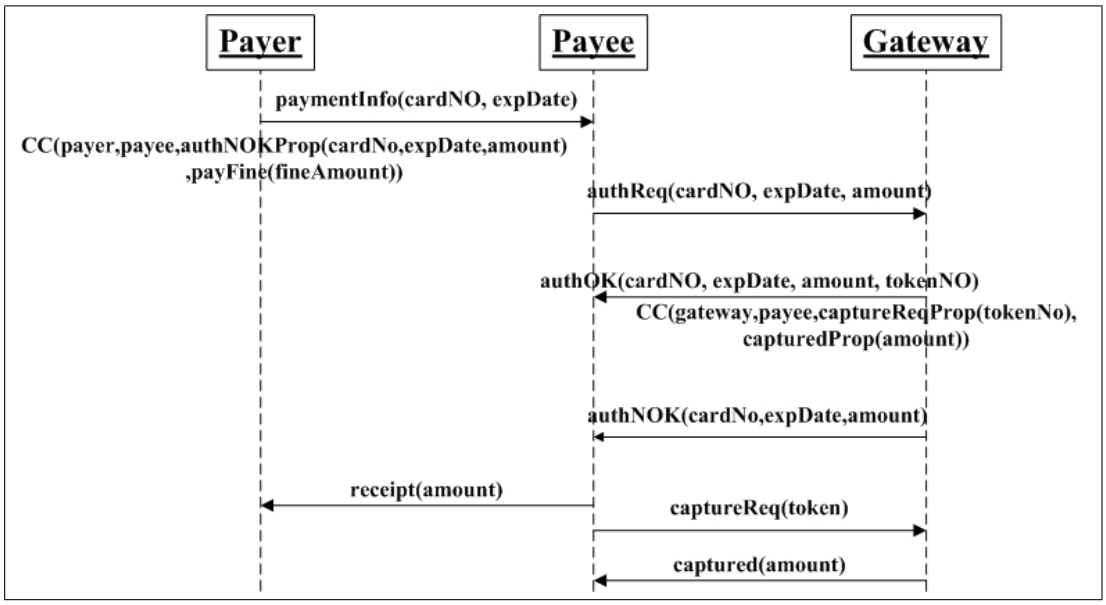

Figure 4.2: Payment Protocol

In this scenario the gateway receives a request from the payee to authenticate a credit-card with credit card number $?cardNo$, expiration date $?expDate$ for an amount $?amount$. If the gateway can successfully authenticate the payer, then the gateway sends an $authOK$ message, saying that the authentication is successful. The gateway also creates a conditional commitment to capture the amount $?amount$ if the payee can provide the token $?token$ for the transaction. But if the gateway cannot authenticate the payer then the gateway sends an $authNOK$ message, saying that the authentication was unsuccessful. Since the authentication failed, there is no commitment between the payee and the gateway. In this scenario, the gateway has to decide whether to send the $authOK$ or the $authNOK$ message. The gateway makes this decision by consulting its policies that are defined for authenticating a

credit card owner. These policies are internal to the gateway and are not a part of the payment protocol specification. Hence, these two rules need to be augmented with policy invokers, namely, the $authPolicy$ proposition and the $authNOKPolicy$ proposition. In the Listing 4.6, the ¬ symbol is used to denote a logical negation. ¬ *authOKProposition* means that the truth value of the authOKPropostion is *false*.

Listing 4.5: Examples of two Gateway P-Skel rules in Payment Protocol

```
1
2  ;; The Gateway gets a message from the Payee asking to authenticate
       the Payer. The Gateway sends the authentication is successful
       message to the Payee. A Conditional Commitment is created from
       the Gateway to the Payee that if the Payee gives a token number
       for the payer then the amount will be deducted by the Gateway
       from the Payer's account.
3
4  authReq(?tranID,?cardNo,?expDate,?amount)
5      ⇒
6      authOK(?tranID,?cardNo,?expDate,?tokenNo) ∧
7  createCommitment(Gateway,CC(Gateway,Payee,captureReqProp(?tokenNo),
       capturedProp(?amount))))
8
9  ;; The Gateway gets a message from the payee asking to authenticate
       the payer. The Gateway sends the authentication is
       unsuccessful message to the payee.
10
11 authReq(?cardNo,?expDate,?amount)
12     ⇒
13     authNOK(?cardNo,?expDate,?tokenNo)
```

Listing 4.6: Example of policy invoker to choose between two actions in the Gateway P-Skel in Payment Protocol

```
1
2  ;; The Gateway gets a message from the Payee asking to authenticate
       the Payer. If the authentication policy invoker is true, i.e.,
       the authentication is successful, the Gateway sends the
       authentication is successful message to the Payee. A
       Conditional Commitment is created from the Gateway to the Payee
       to capture that if the Payee gives a token number for the Payer
       then the amount will be deducted by the Gateway from the Payer's
       account.
3
4  authReq(?tranID,?cardNo,?expDate,?amount) ∧
5      authOKPolicy(?cardNo,?expDate)
6      ⇒
7      authOK(?tranID,?cardNo,?expDate,?tokenNo) ∧
8  createCommitemnt(Gateway,CC(Gateway,Payee,captureReqProp(?tokenNo),
       capturedProp(?amount)))
9
10 ;; The Gateway gets a message from the Payee asking to authenticate
       the Payer. If the authentication policy checker fails, the
       Gateway sends the authentication is unsuccessful message to the
       Payee.
11
12 authReq(?tranID,?cardNo,?expDate,?amount) ∧
13     authNOKPolicy(?cardNo,?expDate)
14     ⇒
15     authNOK(?cardNo,?expDate,?tokenNo)
16
17 ;; The rule given below is a part of the policy checker for the
       authNOKPolicy proposition.
18 ;; If the authOKPolicy predicate returns a false value then
       authNOKPolicy is asserted.
19
20 authNOKPolicy(?cardNo,?expDate)
21    ⇐
22 ¬authOKPolicy(?cardNo,?expDate)
```

```
23
24 ;; The rule given below is a part of the policy checker for the
       authOKPolicy proposition.
25 ;; The authentication is successful if the credit card number is
       valid and the expiration date later than the current date.
26
27 authOKPolicy(?cardNo,?expDate)
28   ⇐
29   valid(?cardNo) ∧ check_expiration(?expDate (?today ←&:(
         get_current_date())))
```

When the role needs to make a decision about which outgoing message to send, or the role has to make a choice between two different actions to perform, the protocol can have several possible runs.

3. *Rules outlining preference structure of protocol runs*

   In order to manage protocol exceptions, [Mallya and Singh, 2005c] introduce the concept of modeling preferences in commitment protocols. The business policies are used as inputs in formulating the protocol preference structure. The preference rules that are used to model the preference structure are also candidates that need to be augmented with policy invokers.

To explain how the P-Skel representation of a role changes with the insertion of policy invokers and checkers, we use the Figure 3.5 of the merchant role in order protocol and the Figure 4.3 which represents the same P-Skel after the addition of policy invokers and policy checkers. In Figure 4.3, the capsules represent the policy invoker propositions. The rounded rectangles represent the policy checkers.

4.3 Types of Policy Checkers

To incorporate business policies into a P-Skel, we use policy invokers. The actual business policy rules of the agent are written as policy-checkers. To better understand policy checkers, we present a classification model of policy checkers. The classification

model helps to identify how to write the policy checkers depending upon what type of business rule the policy checker expresses. So the classification we present is dependent on the types of business rules the policy checkers capture.

Several different business rules classification exist. According to [Hay and Healy, 2000] and [Martin and Odell, 1998], business rules can be divided into *structural assertions* (or *term rules* and *fact rules*), *action rules*, and *derivation rules*. Similarly, Bubenko et al. [1998] categorize business rules into *constraint rules*, *event-action rules*, and *derivation rules*. Herbst [1997] distinguishes between *integrity rules*, that are further divided into *static* and *dynamic* integrity constraints, and *automation rules*. Hay and Healy [2000], further propose another class of business rules, called *authorizations*. Authorizations represent a particular type of deontic assignments. Synonyms for authorizations are rights and permissions. They define the privileges of an agent with respect to certain actions. Complementary to rights there are also rules that capture duties. An authorization policy is the one that defines what actions the agent is allowed to perform given that certain preconditions exist. Obligation policies define what the agent must or must not do. The assignments of powers, rights and duties to the roles. These policies are specified in terms of necessary or sufficient conditions to have the privilege to perform a task.

We classify policy checkers into two basic types and show that the two classes encompass different types of business rules:

1. *Derivation policy checker*: A derivation policy checker is a local policy checker. By local, we mean that the agent can infer about the business policy using the knowledge that is derived from current knowledge, by an inference mechanism, by use of a mathematical calculation, or by referring to a definition or a fact. Derivation rules capture terminological and heuristic domain knowledge that is stored explicitly within the knowledge base of the agent. Derivation policy checkers infer from existing or other derived information and the inference is done on demand. To infer about derivation policy checkers, the agent does not have to interact with other agents or refer to any knowledge-base outside its own.

    - Derivation policy-checkers express the business rules that contain mathematical calculations and logical inferences drawn from the agent knowl-

edge base at run time.

As an example, consider a business policy of the payee in an instance of the payment protocol: *The address of payment gateway to be contacted for a payment transaction, is inferred from the credit card number provided by the payer and the list of gateway addresses maintained in the knowledge base*. When this policy is added to the P-Skel of the Payee Role, it will correspond to a derivation policy checker. In this because the policy rule can be inferred locally by the Payee agent from the payment information provided by the Payer (in earlier messages exchanged between the Payer agent and the Payee agent)and by comparing it against the gateway address table maintained in the local knowledge base.

- Derivation policy checkers also encompass the constraint rules that must be satisfied in all evolving business scenarios of a role.

  An example of such constraint rule is: *To get free shipping, the number of orders should be between 100 and 200*. This policy will be inferred from the number of orders placed and by checking whether that number is between 100 and 200 and then asserting that the shipping cost is 0. These facts are known to the agent based on explicit assertions and inferences drawn from the policy rule knowledge base. For inferring about these policies the agent does not need to interact with other agents.

- Derivation policy checkers capture the definitions of business terms that state facts about the business terms. Derivation policy checkers can express policies authorizations and obligations.

  As an example, consider the authorization policy: *A customer is allowed to return broken or delayed goods*. This policy translates to a derived policy checker since the fact that the goods are broken or delayed will be inferred from the agent's policy knowledge base. Another important policy type that can be captured by derivation policy checker is obligation. For example: *Shipping insurance must be purchased for all electronics goods priced above thousand dollars*. This policy can be inferred from the facts that assert the price, type of the goods and hence can be expressed as

a Derivation policy-checker.

Since derivative policy checkers draw inferences from the local knowledge base of the agent, derivation policy checkers are written as backward-chaining rules.

2. *Subprotocol policy checkers*: A subprotocol policy checker is used when it is necessary for an agent to interact with other agents in order to draw conclusions about the business policies. This type of policy checkers require the agents to follow a certain sequence of message exchanges with other agents to draw conclusions about the business rules that express the agent policies.

   In order to reason about this kind of policy checkers, the information in the agent's local knowledge base is not sufficient. Consider a typical supply-chain business rule in an order protocol: *The delivery date of a product, provided to a customer, is ten days more than the date on which the product is shipped to the warehouse by the raw-material supplier*. For this kind of business policy rule, the merchant agent has to obtain a delivery date from the raw-material supplier before he can calculate the delivery date for a customer date. Hence the merchant agent has to communicate with the raw-material supplier using messages that are not specified in the P-Skel derived from the order protocol. An important thing to note about these kind of policy checkers, is that the the underlying business rule requires an interaction between two roles in the business process and this interaction is not encoded as part of the protocol specification. These policies specify interactions between agents outside the business protocol specification hence, these policies may cause protocol violations. They can also cause commitments to be breached or dishonored [Mallya, 2002]. Hence this class of policy checkers is important for ensuring the flexibility and the correctness of the business protocol being enacted.

   Subprotocol policy checkers need to send and receive messages from other agents in addition to drawing conclusions from the local knowledge base of the agent. These policy-checkers use a combination of forward-chaining and backward-chaining rules to draw inferences. The forward chaining rules are required to send and receive messages from other agents. The use of backward-chaining rules essentially to draw

conclusions from the agent's local knowledge base, much similar to the use of this reasoning in derivation policy checkers.

4.4 Handling Policy Checkers

In this section we describe how to write policy checkers. We also show that our guidelines to write policy checkers can express the two types of policy checker described in the above section. First we identify when protocol violations can occur.

4.4.1 Protocol Violations

A protocol specifies the interactions between two to more roles in the form of messages and commitments. The P-Skel indicates the outgoing message or messages that the agent sends when certain incoming message is received. The P-Skel also specifies the commitments and the actions on commitments produced by the message interchange. Thus the outgoing messages and commitment actions are effects of the incoming messages. In order to obtain a concrete local process, it is important that the incoming and outgoing messages in the local process, obtained after additions of policies, be aligned with the incoming and outgoing messages specified in the P-Skel of the role. When the messages in the local process of the agent are not aligned with the protocol specification, then a protocol violation is said to have occurred. In addition, all the commitments created during the protocol enactment must be discharged when the protocol ends. We identify three scenarios where a protocol may be violated.

1. During protocol enactment, if the sequence of the messages is not honored, then the agent is in violation of the protocol.

2. If a commitment created during the protocol is not discharged at the end of the protocol, then the protocol is violated.

3. If the protocol stops in a state that is not defined as a valid end state.

Next, we describe how to handle policy checkers.

4.4.2 Default Policy

We explain the notion of default policy using the example of Figure 4.4 and the P-Skel for the merchant role in Figure 4.5. The merchant receives a $acceptQuote$ message from the customer. The protocol specification dictates that the merchant should send the goods to discharge the commitment that was created. This is P-Skel rule is specified in Figure 4.5. Consider a scenario that the inventory warehouse in which the merchant stocks its goods burns down. Now the merchant cannot send the goods as it had promised in the commitment. In other words, the merchant cannot bind a value to the slot of $goods$ and hence cannot send the $sendGoods$ outgoing message. Since the P-Skel does not indicate what the merchant should do next, the merchant remains in the $acceptQuote$ state. That is, the purchase protocol ends in a state that is not a valid end state. Thus the purchase protocol is violated. Since the goods were not sent, the commitment created by the merchant to send the goods, if the customer accepts the quote, cannot not be discharged.

We define the notion of *Default Policy* to account for these kind of scenarios. We define a default policy checker for every policy invoker. The default policy checker ensures that, a policy invoker binding is always inferred. The default policy checker is programmed by adding a policy rule that assigns a default inference value to the policy invoker. This default inference value of the policy invoker is always inline with the incoming and outgoing messages for the role specified in the P-Skel.

Default policies ensure that all P-Skel slots are always assigned values. In Listing 4.7, a policy invoker $sendGoodsPolicy$ is added to the merchant P-Skel. The parameter values for the $sendGoodsPolicy$ are obtained by backward-chaining rule to determine whether the goods are in stock, derived by $inStock(?itemID, ?quantity)$, and whether the customer has a valid address, determined by $validAddress(?customer)$. In the scenario where the $inStock$ proposition fails, the ***defaultSendGoods(?tranID, ?customer, ?itemID DEFAULT-ID, ?itemPrice DEFAULT-PRICE, ?quantity DEFAULT-QUANTITY)*** is fired. This is the default policy which will ensure that the agent sends some default goods to the customer. Since the merchant fulfilled the $sendGoodsPolicy$; the merchant can now process the sendGoods message. Subsequently, the commitment ***CC(merchant, customer, acceptQuoteProp, sendGoods)*** can be discharged.

Listing 4.7: Example of merchant P-Skel rule with default policy

```
1
2  ;; P–Skel rule with policy invoker added for the sendGoods message.
3  ;; If the customer has accepted the quote and the the merchant has
       consulted the policies to determine whether the goods can be sent,
       the merchant sends the goods and discharges the conditional
       commitment of sending goods.
4
5  acceptQuoteProp(?tranID,?customer,?itemID,?itemPrice,?quantity) ∧
       sendGoodsPolicy(?tranID,?customer,?itemID,?quantity)
6      ⇒
7      sendGoods(?tranID,?customer,?itemID,?quantity) ∧
8     dischargeCommitment(Merchant,CC(Merchant,Customer,acceptQuoteProp,
          sendGoods))
9
10 ;;Policy checker for the sendGoodsPolicy is calculated by backward–
       chaining.
11 ;; If the particular item is in stock in the desired quantity and the
       address provided by the customer is valid then send the goods.
12
13     sendGoodsPolicy(?tranID,?customer,?itemID,?itemPrice,?quantity)
14       ⇐
15     (inStock(?itemID,?quantity) ∧
16      validAddress(?customer)
17
18 ;;Default policy checker for sendGoodsPolicy
19 ;; The merchant sends a default item instead of the item requested.
20
21     sendGoodsPolicy(?tranID,?customer,?itemID,?itemPrice,?quantity)
22      ⇐
23      defaultSendGoods(?tranID,?customer,?itemID DEFAULT–ID,?itemPrice
            DEFAULT–PRICE,?quantity DEFAULT–QUANTITY)
24      ∧
25      validAddress(?customer)
```

4.4.3 Prioritized Policies

We explain the notion of prioritized policy using an example of a merchant role. This policy is given in Listing 4.8. When the discount policy checker is fired from the P-Skel rule, the policy invoker will be inferred from these policy rules.

Listing 4.8: Example of discount policy for a merchant

1. *Loyal Customer*: Discount is 5% if the customer is a loyal customer, who shops every week.
2. *Platinum Customer* : Discount is 15% if the customer has a platinum card.
3. *Card Customer*:Discount is 10% if the customer has a store card.
4. *Late Payment*: No discount if the buyer has a late payment history.

Listing 4.9: Example of discount policy checker

```
1
2 ;; Policy-checker written to calculate value to be assigned to the ?
      discount variable.
3
4 discountPolicy(?discount 5) ⇐ isLoyal(?customer)
5
6 discountPolicy(?discount 15) ⇐ hasPlatinumCard(?customer)
7
8 discountPolicy(?discount 20) ⇐ hasStoreCard(?customer)
9
10 discountPolicy(?discount 0) ⇐ hasLatePayment(?customer)
```

Consider a scenario where the customer is a “Loyal” Customer and has a “Platinum” card. In this scenario both the rule statements 1 and 3 apply. Thus there is a conflict between the policy checkers. We need to provide some way in which the agent can decide which value to assign to the $?discount$ variable. We need to specify a mechanism by which the policy invoker gets invoked only once.

Conflicts like the one described arise when policy invoker is inferred more than once, yielding different assertion values. The conflicts also arise when different values are assigned to the one variable and all the values are valid. Further, conflicts arise when the same incoming actions result in different outgoing messages depending upon the inference of the policy invoker. If the policy checker does not specify how to resolve the conflict, the agent will not know what to do next: which value to bind to the outgoing message parameter, which outgoing message to send; or which subprotocol policy checker to follow. We propose that these conflicts be resolved by the use of *Prioritized policy checkers*. Prioritized policy checkers help the agent decide which policy checker to use to infer the policy invoker.

Lupu and Sloman [1999] suggest how to manage such conflicts in distributed systems using modality.

Kagal [2002] defines a policy conflict handling mechanism using a specification about the policy language, *Rei*. Our approach in resolving conflicts among policy checkers is similar to the approach of metapolicies proposed by [Lupu and Sloman, 1999]. We propose that the protocol designer assign explicit priorities to individual policy checkers and also define metapolicies. Metapolicies are policies about policy checkers that define policy precedence. We implement the priorities as a number assigned to each policy checker. In addition to this, in order to determine the precedence of priorities, it is important to write metapolicies. The metapolicies reason about the priority precedence. We implement the prioritized policy checkers by assigning a priority variable to each policy checker that directly infers the policy invoker. Thus all derivation policy checkers that derive a value for the policy-invoker using backward-chaining are assigned a priority variable which is passed as a parameter in the policy-proposition.

Jess does provide a conflict resolution mechanism called *salience*. However, salience uses the sequence in which rules are activated to fire and infer from the rules. This is not a good priority mechanism to order policy checkers. We implement the prioritized policy checkers using a numerical policy-implementation. A prioritized policy-invoker is expressed as :

Policy-Proposition-Name(*Policy-Parameters..,..,*?priority NUMBER).

The priority variable is a numerical variable that denotes the relative priority of the policy checker. We have modeled prioritized policy-invoker using the incremental policy technique. This means that the higher the value of priority variable, the greater the priority of the policy-invoker.

Listing 4.10: Example of Discount policy checker with Priority

```
1
2  ;; policy checker written to calculate value to be assigned to the ?
       discount variable.
3
4  discountPolicy(?discount 5,?priority 1) ⇐ isLoyal(?customer)
5
6  discountPolicy(?discount 15,?priority 2) ⇐ hasPlatinumCard(?customer)
7
8  discountPolicy(?discount 20, ?priority 3) ⇐ hasStoreCard(?customer)
9
10 discountPolicy(?discount 0, ?priority 4) ⇐ hasLatePayment(?customer)
11
12 ;; Metapolicy policy checker for Discount policy
13 ;; choosePolicy implements the metapolicy methods to choose the discount
       policy with maximum priority. It returns the value of the ?discount
       variable from the highest priority policy chosen. This value is
       assigned to ?discount-value variable using ?d as an intermediate
       variable.
14
15 calculateDiscount(?discount-value) ⇐ (choose-discount(?discount-value
       ← ?d&:(choosePolicy(?customer))))
```

It is interesting to note that when the default policy checker is added to prioritized policies, the default policy checker always has the least priority. This will ensure that there is no conflict between policy checkers that adhere to the P-Skel messages and commitments by way of original agent policies, as opposed to those that do so by way of default policy checkers.

The techniques of modeling policy checkers as default policies and prioritized policies ensure that the policy invokers will always be inferred. Due to some domain asserted propositions, if the policy checker cannot infer the policy-invoker then the default policy invoker is inferred and the P-Skel slots are defined.

4.5 Methodology to combine P-Skels and Policies

Now we outline the methodology to combine the P-Skel and the policies of the agents in four steps.

1. Get the P-Skel for the Role the agent wants to adopt during the protocol execution. Identify the P-Skel rules. These rules should have all incoming messages on as events and outgoing messages and commitments as effects of those events. In other words, make sure that all P-Skel rules are forward chaining rules with incoming message propositions as assertions and outgoing messages and commitment operations as implications.

2. Identify the P-Skel rules that are candidates for augmentation with policy invokers.

3. Identify the type of policy invokers by consulting the policy checker and the candidate P-Skel rule.

4. Write the policy checkers for each policy invoker.

    - Write a default derivation policy checker for every policy invoker.
    - If a policy invoker needs more than one policy checker, i.e the policy invoker invokes more than one policy checker, provide a priority to every policy checker and also specify the meta-policy that identifies the precedence.

4.6 Proving Protocol Compliance

We have presented a methodology for writing the policies of the agent, such that when added into a desired P-Skel, the resulting local process of the agent, allows the source protocol to execute with maximum flexibility and minimum violations.

In the methodology, we identify the P-Skel rules that need to refer to the policies. These P-Skel rules are augmented with policy-invokers that invoke the policy knowledge base. Thus all P-Skel slots are assigned valid values.

Each policy invoker has a corresponding policy checker. The policy checker is a rule set that infers the policy-invoker. For the local process to be compliant with the source protocol, the local process to be *concrete*, i.e., all the outgoing message parameters should be bound to concrete values and all commitments created should be discharged.

Consider a P-Skel augmented with default policy-invokers. The corresponding default and prioritized policy checkers are written for each policy-invoker. Thus for every incoming message, the corresponding outgoing message will be sent. The outgoing message parameters might be derived from the priority precedence policy checker or may have the default policy checker derivation values. In both these cases, the outgoing message will be sent. In addition, as the P-Skel actions are always performed, all the commitments created during the P-Skel will be discharged. The resulting local process will always have the same states and the same messages in the same order.

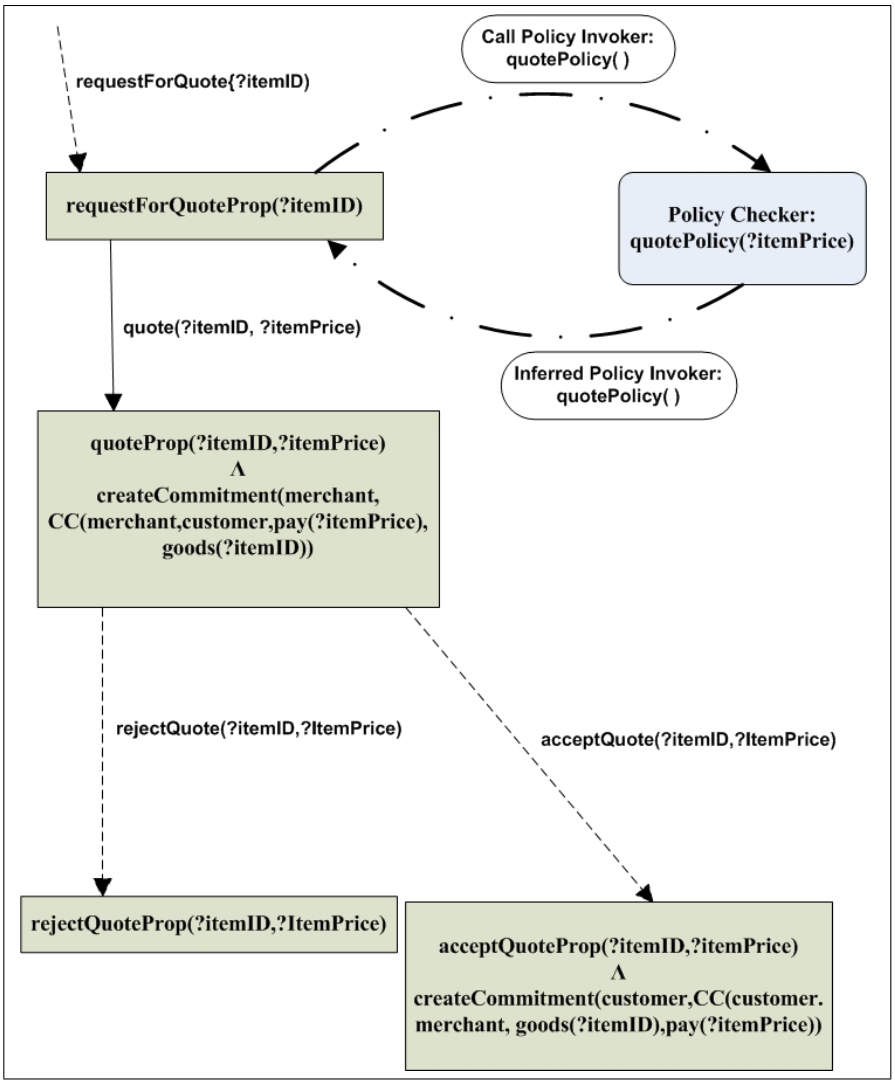

Figure 4.3: P-Skel representation for a merchant role in order protocol with Policy invoker and Policy checkers

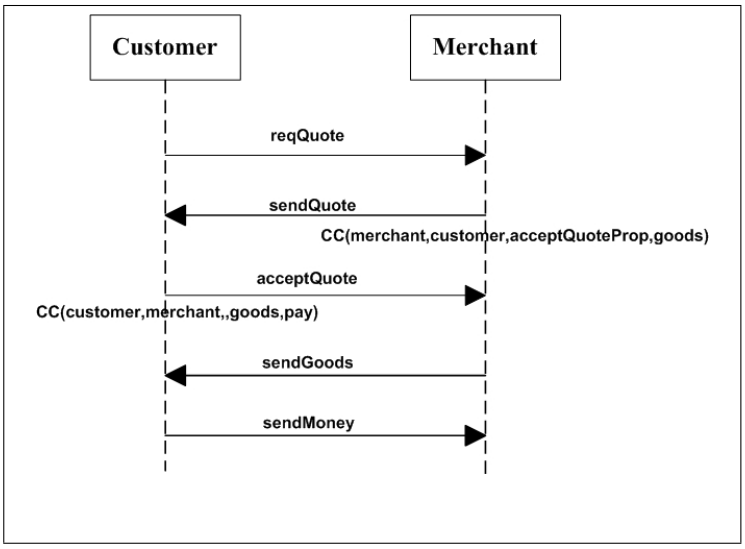

Figure 4.4: Purchase protocol

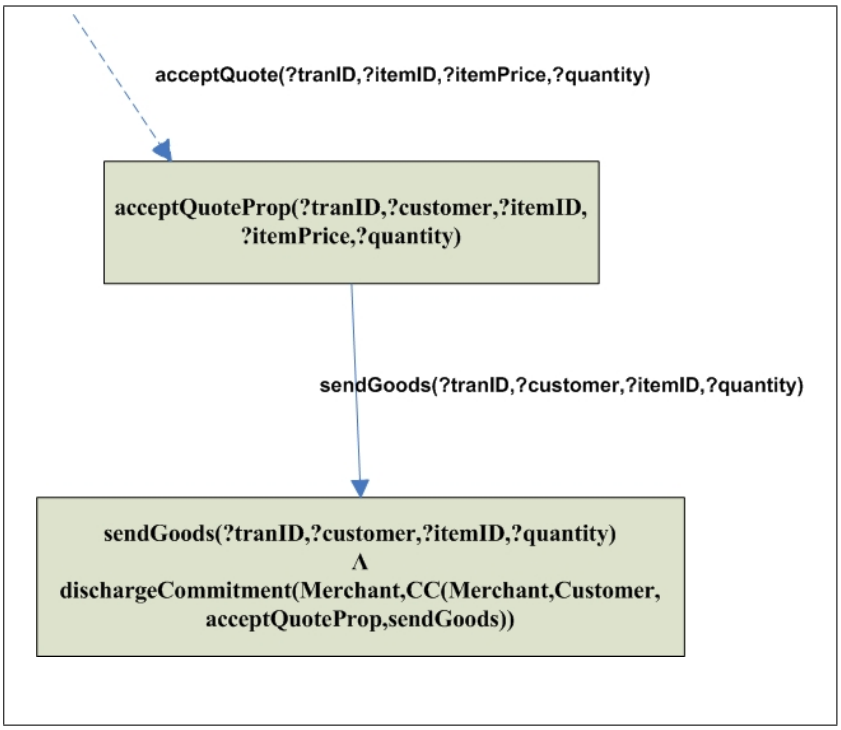

Figure 4.5: Snippet of P-Skel of Merchant Role in Purchase Protocol

Chapter 5

Discussion

Agents can engage in a rich variety of interactions and need rich models to support their autonomy and heterogeneity. Business protocols capture agent interactions and outline the social behavior of agents. Individual agents have their own policies about how they participate in a given protocol. Existing approaches to specify policies in distributed multiagent systems are focused on analysis and classification of policies. An approach to separate agent policies that govern its behavior from the functionality of the protocol is highly desirable. Such an approach can accommodate agent autonomy while allowing the protocol to execute with flexibility. We have outlined a methodology to incorporate the agent policies into protocol specifications. Our approach not only separates the private policies from the public protocols, but also enables each agent to control its behavior dynamically, via the use of rules, such that the protocol executes with flexibility and minimum violations.

Kagal [2002] present Rei, which is, a policy framework that integrates support for policy specification, analysis and reasoning by ability to represent concepts of right, prohibitions, obligations and dispensations. Rei also provides constructs to specify policies, which are defined as rules. Rei also provides constructs to specify role-based access-control policies. Rei provides a policy-conflict resolution mechanism that is very similar to the prioritized policy checkers we introduce. However, Rei framework does not provide a policy enforcement model. The Rei policy engine is not designed to enforce policies but to reason about them and reply to queries. Thus, Rei framework provides limited or no protection against non-compliant agents, which can cause protocol violations.

KAoS is a collection of component agent services compatible with several popular agent frameworks. KAoS *domain services* provide the capability for groups of software components and agents to be structured into the organizations of domains to facilitate inter-agent collaboration. KAoS *policy services* allow for specification, management, conflict resolution and enforcement of policies within domains. KAoS policy ontologies distinguish policies as *authorizations* (policies that permit or forbid actions) and *obligations* (policies are dictate compulsory actions). KAoS allows the policy-administration to accommodate extensible policies and dynamic policy changes at runtime. KAoS is able to detect conflicts at specification time. KAoS implements policy enforcement mechanism by the user of platform specific*Enforcer* class. This requires that in order to support policy enforcement, the *Enforcer* needs to be implemented for every obligation. This requires a deep understanding of the platform. Thus even though KAoS is able to manage the policies dynamically, its platform dependence makes KAoS a less expressive approach to represent agent policies.

Future Directions

We have identified how to write the agent policies, such that, for given a protocol, the agent can be configured to be compliant with that protocol. We identify the following aspects to extend this research further.

Protocol Composition

Our work concentrates on writing agent policies in protocols, where the agent plays one role at a given instance. Our methodology does not specify how to write agent polices when the agent participates in a business process and adopts multiple roles at the same time. In developing the conceptual model of OWL-P, [Desai et al., 2004] introduce the notion of a C-Skel. As defined in Section 3.1.2, a C-Skel is a composite protocol skeleton for an agent that adopts multiple roles. Thus a C-Skel would be a composition of the P-Skel rules of the individual roles derived from different protocols. While our methodology guarantees compliance with a single protocol skeleton, the next step will be to see how this methodology can be extended to guarantee compliance with a composite protocol skeleton, which encompasses multiple roles.

Mallya and Singh [2005a] present an approach for designing commitment-based protocols where the traditional software engineering notions such as refinement and aggregation can be extended to protocols. They present an algebra of protocols that can be used to compose protocols by refining and merging existing protocols. Our approach to incorporate policies into protocols guarantees that agent polices are written to be compliant with an existing protocol. An extension to this work will be to develop guidelines to program policies to be compliant with the refined and aggregated protocols. Using our methodology to start with agent policies which compliant with an existing protocol, further research can help determine how can these policies be modified to be compliant with a refined protocol. This requires reasoning about detection and elimination of policy conflicts as well as developing a software engineering approach to refine and aggregate policies similar to that of protocol algebra. We leave that as future work.

Protocol Exceptions

Protocol exceptions can be expected, like violations, or unexpected,like errors. The flexible and open-source nature of protocols increase the chances of expected and unexpected protocol exceptions. Mallya and Singh [2005c] present an approach to incorporate preferences into protocols. These preferences help identify preferable protocol runs and thus incorporate protocol exception handling. In identifying P-Skel rules to be augmented with policy invokers, we mentioned that the protocol rules that specify preferences are candidates for addition of policy invokers. An extension to our work of policy configuration methodology, is a mechanism to write policy checkers to adhere to this protocol preference structure.

Selection of Most Compliant Protocol

Our approach specifies the thumb-rules for protocol compliance, by allowing a protocol designer to write agent policies with a view to satisfy a specific protocol. As long as the agent wants to be compliant with the protocol specification, policies written using our methodology guarantee protocol compliance. A promising agent configuration mechanism is that, given a set of protocols, the agent is automatically able to select the protocol it is most compliant with. Thus, looking at the protocol specifications and its own policies, the agent can dynamically pick the protocol that is most flexible for its policies. Our approach

can be extended by developing additional analysis of protocol compliance and detection and elimination of policy-protocol conflicts. This approach will facilitate in generation of agent policies that can detect and select protocols at run time.

Bibliography

J. A. Bubenko, D. Brash, and J. Stirna. EKD: Enterprise knowledge development user guide. Technical report, Dept. of Computer and Systems Science, Royal Institute of Technology and Stockholm University, Stockholm, Sweden, 1998.

Amit K. Chopra, Nirmit Desai, Ashok U. Mallya, Leena Wagle, and Munindar P. Singh. A semantic approach for developing commitment protocols. In *Second International Conference on Service-Oriented Computing, IBM Research Report RA221 (WO411-084)*, pages 99 –107, November 2004.

T. Cooper and N. Wogrin. *Rule-Based Programming with OPS5*. Morgan-Kauffman, 1988.

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

Nirmit Desai, Ashok U. Mallya, Amit K. Chopra, and Munindar P. Singh. OWL-P: A methodology for business process developement. In *Proceedings of the 4th International Conference on Autonomous Agents and Multiagent Sytems (AAMAS), Workshop on Agent Orientated Information Systems (AOIS)*, July 2005.

E. J. Friedman-Hill. *Jess, The Java Expert System Shell*. Manning Publishing Co., 1997. URL `http://herzbeg.ca.sandia.gov/jess`.

Benjamin N. Grosof and Yannis Labrou. An approach to using XML and a rule-based content language with an agent communication language. Technical report, IBM Research, RC21491(969650), May 1999. URL `http://www.research.ibm.com/`.

Benjamin N. Grosof, Mahesh D. Gande, and Timothy W. Finn. SweetJess : Inferencing in situated courteous ruleML via translation to and from jess rules. In *Proceedings International Workshop on Rule Markup Languages for Business Rules on Semantic Web*, June 2002.

J. D. Hay and K. A. Healy. Defining business rules - what are they really? Technical report, The Business Rules Group 1.3, July 2000.

H. Herbst. *Business Rule-Oriented Conceptual Modeling to Management Science*. Springer-Verlag, 1997.

I. Horrocks, P.F. Patel-Schneider, H. Boley, M. Dean, and Said Tabet. SWRL: A semantic webrule language combining OWL and ruleml, August 2005. URL `http://www.daml.org/2003/11/swrl/`.

JADE. Java agent development framework, 2005. URL `http://jade.tilab.com/`.

Lalana Kagal. *Rei*: A policy language for the me-centric project. Technical report, HP Labs., 2002.

E. Lupu and M. Sloman. Conflicts in policy-based distributed systems managment. *IEEE Transactions on Software Engieering*, 25(6):852–869, November/December 1999.

Ashok U. Mallya. Specifying and resolving temporal commitments. Master's thesis, North Carolina State Univeristy, December 2002.

Ashok U. Mallya and Munindar P. Singh. A semantic approach for designing commitment protocols. *Developments in Agent Communication. Lecture Notes in Artificial Intelligence*, 3396:37 – 51, 2005a.

Ashok U. Mallya and Munindar P. Singh. Modelling exceptions via commnitment protocols. In *Proceedings of the 4th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, July 2005b.

Ashok U. Mallya and Munindar P. Singh. Introducing preferences into commitment protocols. In *Proceedings of the 4th International Conference on Autonomous Agents*

and Multiagent Systems (AAMAS), Workshop in Agent Oriented Software Engineering (AOSE), July 2005c.

James Martin and James Odell. *Object-Oriented Methods: A Foundation (UML Edition)*. Prentice-Hall, 1998.

RULEML. Rule markup language initiative, 2005. URL `http://www.ruleml.org/`.

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

J. D. Ullman and J. Widom. *A First Course in Database Systems*. Prentice-Hill, 1997.