

## **Abstract**

CHAKRAVARTY, PAYAL . An Event-Driven approach to Agent-Based Business Process Enactment. (Under the direction of Dr. Munindar P. Singh).

Agents enacting business processes in large open environments need to adaptively accommodate exceptions and opportunities. Multiagent approaches can help flexibly model business processes. This thesis proposes an event-driven architecture that enriches such models with events to support the agile enactment of processes.

Specifically, we place this architecture in a business process framework based on protocols and policies, where agents' behaviors are specified via rules. The agents interact via messages, and agreements between them are modeled by commitments. These messages and commitments provide only a high-level view of the interactions and fail to capture fine-grained details of how the interactions were carried out and whether they were carried out smoothly. There might be hindrances due to internal and external influences, resulting in anomalies in the business process enactment. Handling such exceptions or capturing opportunities deviate the protocol from its routine execution but would restore the process to an acceptable state. Our approach introduces fine-grained event monitoring at specific points of the process enactment that require special attention. Detected exceptions are handled by the policies of the involved agents. Monitoring processes and thereby recovering from errors spontaneously, results in a more reliable and proactive distributed system.

The contributions of this thesis include (1) an event-driven agent architecture for process enactment, (2) a specification language that combines event logic with rules, (3) a methodology to incorporate events into a protocol for fine-grained monitoring, (4) an algorithm to help a designer derive high-level (complex) exception patterns, (5) an algorithm to manage subscriptions to low-level events, and (6) policy-driven exception handling. This approach is applied on a well-known business scenario. A proof-of-concept prototype has been implemented to demonstrate the feasibility of the architecture. It illustrates the different perspectives of commitments and different scenarios under which event monitoring proves useful.

**AN EVENT-DRIVEN APPROACH TO AGENT-BASED BUSINESS PROCESS  
ENACTMENT**

by

**PAYAL CHAKRAVARTY**

A thesis submitted to the Graduate Faculty of  
North Carolina State University  
in partial fulfillment of the  
requirements for the Degree of  
Master of Science

**COMPUTER SCIENCE**

Raleigh, NC

2007

**APPROVED BY:**

---

DR. JAMES C. LESTER

---

DR. XIASONG MA

---

DR. MUNINDAR P. SINGH  
Chair of Advisory Committee

*To Ma and Ashish.*

## **Biography**

Payal Chakravarty was born in Kolkata, India on December 5, 1980. She holds a Bachelor's degree in Electronics and Communication Engineering, awarded in July 2003, from the Rashtriya Vidyalyaya College of Engineering, Bangalore, India. She worked as a Software Engineer at IBM Software Labs, Bangalore, India till December 2004. Payal joined the North Carolina State University in January 2005, and is currently a Masters candidate in the Department of Computer Science, North Carolina State University. Payal has also worked as Software Engineer Intern at IBM, Tivoli in RTP and Microsoft Corporation in Redmond. She will be joining IBM, RTP as a software engineer on completion of her graduate studies.

## Acknowledgments

It is a pleasure to thank the many people who made this thesis possible.

First and foremost, I thank my advisor, Dr. Singh, for his attention, guidance, insight and intellectual support during this research and preparation of the thesis. Right from my first semester here at school, he has inspired me in many ways. I thank my committee members, Drs. Xiaosong Ma, and James Lester, for their support.

I thank my current and former colleagues at the MultiAgent Systems lab, Ashok Mallya, Amit Chopra, Nirmal Desai and Yathiraj Udipi for their comments. They have helped me learn the basics of research, and to improve the quality of my thesis and my knowledge of the field. I am grateful to the Department of Computer Science and International Office at the university.

I am grateful to my parents and my sister for all their love and support, for making me what I am today. I would also like to thank all my friends and roommates here in Raleigh for their constant encouragement, for the fun-filled moments through the hectic semesters and for all the support they provided me during the tough times I faced with a fractured leg.

I am eternally grateful to my husband, Ashish, for his love, encouragement, and patience through the entire period I spent here working towards my Masters. He played an invaluable role in making this possible.

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

# Contents

<b>List of Figures</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Business Processes . . . . .	1
1.2.1 Agent-Based Business Processes . . . . .	2
Business Protocols . . . . .	2
1.3 Challenges . . . . .	2
1.4 Events . . . . .	4
1.5 Our Approach . . . . .	5
1.6 Contributions . . . . .	6
1.7 Organization . . . . .	6
<b>2 Background: OWL-P, Commitments and The Specification Language</b>	<b>8</b>
2.1 The OWL-P Framework . . . . .	8
2.2 Commitments . . . . .	10
2.3 The Specification Language . . . . .	11
2.3.1 Constraint Language Semantics . . . . .	12
2.3.2 Residuation . . . . .	12
2.4 Running Example . . . . .	13
<b>3 Events in Business Processes</b>	<b>15</b>
3.1 Event Processing Language . . . . .	16
3.1.1 Event Type . . . . .	16
3.1.2 Event Pattern Language . . . . .	17
3.1.3 Event Incorporated Rules . . . . .	18
3.2 Monitoring Decisions . . . . .	19
3.2.1 Receiver's Perspective of the Shipping Protocol . . . . .	20
3.2.2 Shipper's Perspective of the Shipping Protocol . . . . .	22
3.2.3 Sender's Perspective of the Shipping Protocol . . . . .	22

<b>4</b>	<b>Architecture and Process Monitoring</b>	<b>25</b>
4.1	Architecture	25
4.2	Event Configuration	27
4.2.1	Deriving Exception Patterns	28
4.2.2	Generating Exception Handling Rules	30
4.3	Process Monitoring	31
4.3.1	Event Processing	32
	Transformation	32
	Pattern Matching	32
	Action	33
4.3.2	Selective Subscription	33
	Test Cases to Demonstrate Selective Subscription	35
4.3.3	Exception Detection and Recovery	37
4.3.4	Nesting of Protocols	39
<b>5</b>	<b>Evaluation</b>	<b>42</b>
5.1	Prototype	42
5.2	Use Case 1: Shipping Protocol	44
5.2.1	Handling Conditions and Violations: Shipping protocol under special conditions for perishable items	44
5.2.2	Exploiting an Opportunity: Combining Orders to be Shipped	45
5.3	Use Case 2: Money Wiring Protocol	45
5.3.1	Detecting an Unexpected or Unanticipated Exception	47
5.3.2	Detecting an Expected or Anticipated Exception	47
5.4	Use Case 3: Repair Protocol	48
<b>6</b>	<b>Discussion</b>	<b>55</b>
6.1	Related Work	56
6.2	Future Directions	57
	Bibliography	58
<b>A</b>	<b>Appendix A</b>	<b>63</b>
A.1	Rules for Shipper in Running Example	63
A.2	Complex Events for Shipper	65
A.3	Exception Handling Policy for Shipper	65
A.4	Log Records for Shipper	65

## List of Figures

2.1	Shipping protocol scenario . . . . .	9
2.2	Shipping protocol refined to include tracking by sensors at two checkpoints . . . . .	14
3.1	Receiver's perspective of shipment in the shipping protocol . . . . .	21
3.2	Shipper's perspective of shipment in the shipping protocol . . . . .	23
3.3	Sender's perspective of shipment in the shipping protocol . . . . .	24
4.1	Event-driven agent architecture . . . . .	26
4.2	Event configuration . . . . .	28
4.3	Selective subscription for a complex event pattern . . . . .	36
4.4	Selective subscription . . . . .	38
4.5	Tracking protocol nested within the shipping protocol . . . . .	40
4.6	Shipping protocol nested within another instance of the shipping protocol . . . . .	41
5.1	Tool aided agent configuration . . . . .	43
5.2	Handling conditions and violations: Shipping Protocol under Special Conditions for Perishable Items . . . . .	50
5.3	Exploiting an opportunity: Combining orders to be shipped . . . . .	51
5.4	Money wiring protocol . . . . .	51
5.5	Handling an unexpected exception from <i>Bank's</i> perspective . . . . .	52
5.6	Handling an expected exception in money wiring protocol . . . . .	53
5.7	Repair protocol . . . . .	54



# Chapter 1

## Introduction

### 1.1 Motivation

Business process modeling and enactment in service-oriented computing environments is extremely complex due to the requirement of supporting autonomy, heterogeneity and dynamism [[Singh and Huhns, 2005](#)]. Multiagent approaches to modeling and enacting business processes [[Jennings et al., 1996](#), [O'Brien and Wiegand, 1998](#), [Desai and Singh, 2004](#)] that offer flexibility, have been around for some time. However, such processes may not be completely reliable and robust. They are often challenged by unexpected errors and anomalies from their routine behavior. In order to ensure that a smooth process is enacted, it is important to monitor the business events taking place during the process. This thesis applies an event-driven agent architecture for fine-grained monitoring in specific business situations during a normal business process enactment. By incorporating events into business rules, the agility and proactiveness of existing models are enriched. By forward error recovery, a more reliable distributed system is achieved.

### 1.2 Business Processes

A business process is a way an organization conducts some component of its business. This encompasses its businesses dealings with its customers, suppliers, and all other external entities that it interacts with. A business can have defined several processes, or no

documented process at all. A cross-enterprise business process specifies how different parties interact with each other in order to achieve a common business goal [Mallya, 2006].

### 1.2.1 Agent-Based Business Processes

Agents are situated, persistent, and autonomous software entities, that have been successfully applied to solve problems in which collaboration, cooperation, or communication is required between independent partners, for example, in online auctions, distributed information processing, and human-computer interaction [Huhns and Singh, 1998]. One of the most important application areas of intelligent agents has been in automating businesses and developing newer, more productive ways of conducting business. Agents and multiagent systems have been applied in business process modeling and enactment extensively as monolithic workflows [Jennings et al., 1996] and later as protocols [Desai and Singh, 2004]. For our purposes, an agent-based business process can, be described as an interaction-oriented multiagent system where the agents act in accordance with some rules and commitments to conduct business.

#### Business Protocols

A business protocol is a public specification of the interactions that take place to enact a business process. A protocol is specified in terms of the commitments or contractual obligations between the different parties enacting the business process. Protocols enable flexibility in process enactment due to their ability to accommodate changes that might take place due to external and internal factors during execution, including change in local policies, government regulations or market preferences.

## 1.3 Challenges

The behavior of an agent is specified by business rules of the protocol it is enacting and its policies. The interactions between the agents typically take place via exchange of messages. However, these messages and commitments provide only a high-level view of the interactions and fail to capture fine-grained details of how an interaction is carried out,

for example, whether it is carried out smoothly. There might be hindrances due to internal and external factors during the process, which may result in exceptions and anomalies. For example, consider a scenario where some goods are being shipped. Sending a shipment does not ensure that the shipment actually reached its destination under the agreed upon circumstances. The factors that could be a hindrance in the process are an accident that leads to the damage of the shipped item, a change in export regulations that prevents the item from being delivered to the destination country, the breakdown of the transport carrying the item thereby causing a delay, and the item being a perishable one getting damaged due to weather conditions.

Exceptions can be defined as unusual situations during protocol execution. The different types of exceptions are [Eder and Liebhart, 1995]:

- Basic failures: Associated with failures on the systems, e.g., system crash, network failure.
- Application failures: Failures of the applications invoked to execute tasks, e.g., unexpected data input.
- Expected exceptions: Events that can be predicted during the modeling phase but do not correspond to the normal behavior of the process. These exceptions can happen frequently and can cause a considerable amount of work to process. e.g., a flight cannot be booked because it is already booked up, a late payment.
- Unexpected exceptions: When the semantics of the process is not accurately modeled by the system. This type of exception cannot be predicted at the modeling stage and may require human intervention or even force the system to stop, e.g., changes in rules, a structural change in the organizational environment, or a change in the order processing of a very important client.

Opportunities are (desirable) exceptions. e.g., If merchant wants to ship goods before payment as a trial offer, the model should allow it if the customer agrees.

Unhandled exceptions might lead to the process enactment reaching a standstill or an unknown state. In order to recover from exceptions, the process enactment needs to be

restored to a stable state that might be a deviation from the routine path of the process. Handling exceptions can prove to be challenging since it requires domain knowledge. For instance, while delayed payment might be acceptable in some circumstances, delayed shipment of food might not be acceptable. Unexpected exceptions are not part of the process model, hence difficult to handle. Effects of exceptions must be bounded. For example, merchant not paying the shipper must not affect the customer [Mallya, 2006].

Since protocols are generic abstractions of business processes, the high-level interactions specified by the protocol are not sufficient in detecting such exceptions and opportunities. Moreover, handling such exceptions will deviate the protocol from its routine path. In order to detect early indications of an impending failure or an unexpected opportunity, it is important to have a microscopic view of the process enactment, to take appropriate action beforehand instead of relying completely on the high-level interactions.

## 1.4 Events

What are events? As Albek *et al.* [2005] describe, an event is a significant change in state. An event could be normal, an anticipated event, or an unanticipated event. An example of a normal event is the delivery of a package on time or the arrival of a flight on schedule. An anticipated or abnormal event is a slight deviation from the expected that can be prepared for. An example would be a delayed shipment or the disruption in flight schedule due to weather disturbance. A completely unanticipated event is something like a system crash or network attack. Anticipated events are, of course, easier to handle since we can specify a pattern for such an event and hence devise an appropriate response to it. Handling an unanticipated event would require estimating the amount of deviation from normal.

Event-Driven Architecture (EDA) handles events by managing and executing rules of the form [Chandy, 2005]:

```
WHEN reality deviates from expectations
THEN update expectations and initiate response
```

The primary characteristic of a system built using this architecture is to sense, analyze, and

respond to events.

In a service-oriented business process management scenario enacted by distributed agents, sensing involves detecting events across an extended environment from multiple sources or agents in real time. Analyzing involves aggregating such events and comparing the results with expected patterns. Responding involves updating expectations and invoking other processes and interacting with other agents in real time in order to bring the system to a stable and correct state.

Intelligent agents working together to monitor processes and workflows have been proposed earlier by Wang *et al.* [2002, 2005]. While in previous approaches a multiagent system works towards monitoring events in workflows or processes enacted by other agents, in our approach every agent enacting a business process monitors business events in the process it is enacting itself. We understand the high-level interactions of a process enactment as complex events defined by patterns. The low-level steps that need to be followed in order to accomplish a complex event are simple events.

## 1.5 Our Approach

We have employed the advantage of events to enrich agent-based business process enactment models. The challenges mentioned in Section 1.3 are addressed by an event-driven agent architecture that introduces fine-grained monitoring of business events at specific points of the process enactment that require special attention.

The most significant points are those which involve a contractual agreement or commitment since the violation of such commitments is not desirable. Every commitment can potentially be monitored. In our design we leave this decision to the agent since the perspective of a commitment varies from role to role and instance to instance. For example, a commitment between a merchant and a customer that states that the merchant must ship goods to the customer once a payment has been made, can be operationalized in different ways by the two agents. The customer might wait for a specific amount of time before he sends out a reminder or a failure notice. The merchant might send a shipment and track it from point to point to ensure the goods are delivered on time. In another instance of the

same protocol, the customer might track the shipment instead and could be responsible for sending an acknowledgment to the merchant once the shipment is received. Thus the same commitment can be viewed differently by the merchant and customer under different circumstances. Monitoring the commitments results in a more detailed view of the interaction, thereby providing the opportunity to detect errors and anomalies and recover from them. In order to perform such event monitoring the agent might need to interact with other agents by instantiating a new protocol or by subscribing to sensors. By incorporating events into agents' rules, all types of exceptions and opportunities are capable of being detected and handled.

## 1.6 Contributions

We place our event-driven agent architecture in a business process framework based on protocols called the OWL-P framework introduced by Desai *et al.* [2006b]. In this framework, agents' behaviors are described via rules. By incorporating events into agents' rules a more stable, robust and proactive business process is enacted.

Our contributions specifically include (1) an event-driven agent architecture, (2) a specification language that combines event logic with rules, (3) a methodology to incorporate events for fine-grained monitoring, (4) an algorithm to help a designer derive high-level (complex) event patterns, (5) an algorithm to manage subscriptions to low-level events, and (6) exception handling with policies. This approach is applied on some well-known business scenarios. A proof-of-concept prototype has been implemented to demonstrate the feasibility of the architecture. Some experiments have been carried out to demonstrate the different perspectives of commitments and different scenarios under which event monitoring proves to be useful.

## 1.7 Organization

Chapter 2 discusses the background of protocol driven business process modeling and enactment, the specification language used for formalizing event patterns and events. This

section also introduces a running example that we use to demonstrate our approach. Chapter 3 introduces the event processing language, and describes how complex event patterns can be designed on the basis of the perspectives on a commitment. Chapter 4 describes the agent architecture and the steps by which event configuration and process monitoring takes place in a typical business process scenario. Chapter 5 evaluates the architecture with a few use cases. Chapter 6 concludes the work and discusses some related work.

## Chapter 2

# Background: OWL-P, Commitments and The Specification Language

This chapter introduces some of the background concepts used as a basis for this work, including the OWL-P framework, the specification language used to define complex event patterns and describe commitments. It then proceeds to introduce the running example used for this thesis.

### 2.1 The OWL-P Framework

OWL-P, proposed by Desai *et al.* [2005], is a framework for specifying and enacting business processes based on protocols. A *business protocol* is a public specification of an interaction. It is an abstract, modular and publishable specification of *rules* that govern a business interaction between two or more business partners or *roles*. A protocol typically addresses a well-defined business purpose such as payment, shipping, or order placement. It is specified in terms of the *commitments* among the different parties.

The enactment of a protocol-based business process is accomplished via an interaction-oriented multiagent system which is enacted as follows. The protocol dictates a set of *rules* which specify the business logic. Each agent that plays a particular role in a protocol, fleshes out the skeleton from the public protocol, which specifies its perspective of the interaction. The skeleton in combination with the private business logic of the agent, also



known as its *policies*, produces the agent's *local process*. Each agent hence programmatically consists of a knowledge base it consults, a rule base that defines its local process, and a queue to receive messages from other participating agents. Every commitment or message that is received or sent is asserted to the knowledge base as a proposition. This activates the rules and action is taken accordingly. In a rule-based system if the LHS of any rule holds, then the rule is fired and the action specified in the RHS are enacted. A logical combination of the propositions in the knowledge base and policies ideally constitute the LHS of a rule. The business process is thus enacted by interaction among the agents following the rules of the protocol.

Figure 2.1 shows a scenario for the Shipping Protocol:

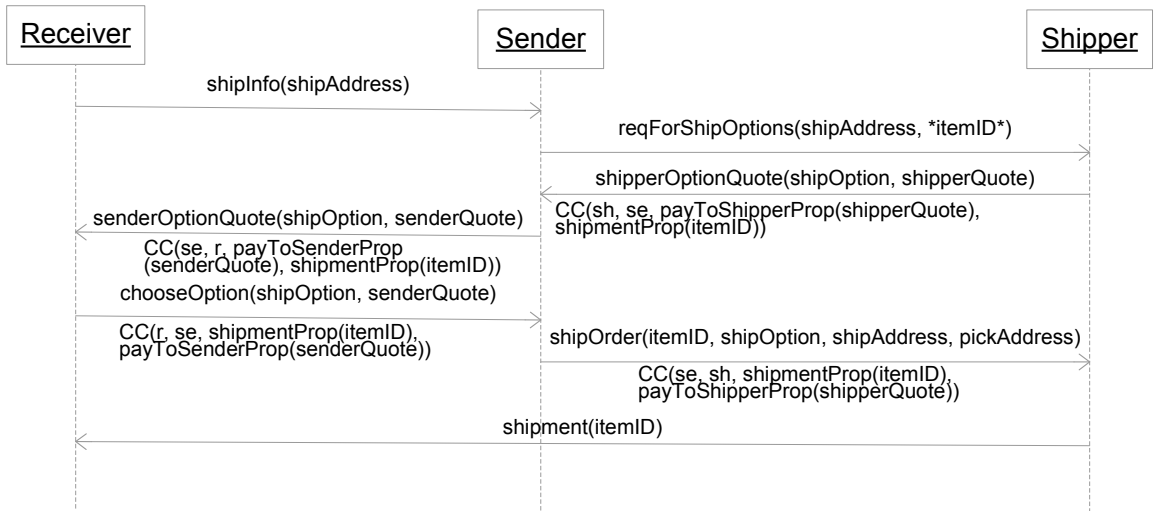


Figure 2.1: Shipping protocol scenario

There are three roles involved in this protocol namely, the receiver, the sender and the shipper. The receiver is the customer who purchased some item from the merchant or the sender. The sender uses a third party, i.e., the shipper, to ship the item to the receiver. The sequence of messages exchanged to enact this protocol are shown in Figure 2.1. The terms in the parantheses following the message name indicate the parameters or slots that uniquely identify the messages. The commitments created are represented as `CC(..)`. Commitments are explained in Section 2.2.

Below are some of the rules for the Shipping protocol in the Shipper's skeleton. The

first rule indicates that if the shipper's knowledge base (KB) contains a senderOptionQuote message and its shipOrderPolicy is also asserted to the KB, then the shipper should send the receiver a shipOrder message and create a commitment which indicates that the sender is committed to pay to the shipper if the shipper sends the shipment to the receiver.

```
contains (KB, senderOptionQuoteProp (? senderQuote , ? shipOption ))
 $\wedge$  contains (KB, shipOrder_Policy )
 $\Rightarrow$  send ( Receiver , shipOrder (? pickAddress , ? shipAddress ,
? itemID , ? shipOption ))
 $\wedge$  createCommitment ( Sender , Shipper , shipmentProp (? itemID ) ,
payToShipperProp (? shipperQuote ))

contains (KB, shipOrderMsgProp (? pickAddress , ? shipAddress , ? itemID ,
? shipOption ))  $\wedge$  contains (KB, shipment_Policy (? itemID ))
 $\Rightarrow$  send ( Receiver , shipment (? itemID ))
```

## 2.2 Commitments

Commitments among agents represent the contractual obligations that exist among them. An agent can commit to bringing about a certain state in the environment in which it exists or to carry out a certain action. Commitments always exist within a well-defined social structure, which forms their context [Singh, 1999]. Commitments provide a powerful formalism for modeling and understanding interactive behavior in multiagent systems. For example, a customer's agreement to pay the price for the item after it is delivered is a commitment that the customer has towards the merchant. Violations of commitments can be detected; in some important circumstances, violators can be penalized. Such enforceability of contracts is necessary in practical settings where the participants are autonomous and heterogeneous [Venkatraman and Singh, 1999].

Commitments are formalized as follows: A commitment  $C(x, y, p)$  denotes that the agent  $x$  is responsible to the agent  $y$  for bringing about the condition  $p$ . Here  $x$  is called the *debtor*,  $y$  the *creditor*, and  $p$  the *condition* of the commitment. The condition can be expressed in a suitable formal language. Commitments can also be conditional, denoted by  $CC(x, y, p, q)$ , meaning that  $x$  is committed to  $y$  to bring about  $q$  if  $p$  holds where,  $q$

is called the *precondition* of the commitment. For example, the conditional commitment  $CC(c, b, \text{goods}(g), \text{pay}(p))$  means that the customer  $c$  is committed to pay the bookstore  $b$  an amount  $p$  if the bookstore delivers the book  $g$  to the customer. When the bookstore delivers the goods, i.e., when the  $\text{goods}(g)$  proposition holds, the conditional commitment  $CC(c, b, \text{goods}(g), \text{pay}(p))$  is automatically converted into the base-level commitment  $C(c, b, \text{pay}(p))$  [Wan and Singh, 2005].

## 2.3 The Specification Language

We use the event-based linear temporal logic  $\mathcal{I}$  as introduced by Singh [2003] for formalizing complex event patterns.  $I$  is the start symbol of the BNF for the language of  $\mathcal{I}$ . In this BNF, *slant* indicates nonterminals,  $\longrightarrow$  and  $|$  are meta-symbols of the BNF,  $/*$  and  $*/$  delimit comments, and all other symbols are terminals.

$$L_1 I \longrightarrow dep \mid dep \wedge I \text{ /*conjunction: interleaving*/}$$

$$L_2 dep \longrightarrow seq \mid seq \vee dep \text{ /* disjunction: choice*/}$$

$$L_3 seq \longrightarrow bool \mid event \mid event \cdot seq \text{ /* before: ordering*/}$$

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

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

**Event Literal Set.**  $\Gamma \neq \{\}$  is the set of event literals as generated by the nonterminal *event*. Each event  $e$  literal has a *complement*  $\bar{e}$ . Intuitively, initially, neither an event nor its complement holds; ultimately, one of them must hold.  $\Gamma_D$  is the set of literals mentioned in a dependency  $D$  and their complements. For example,  $\Gamma_e = \{e, \bar{e}\}$ . For a set of dependencies  $\mathbb{D}$ , we define  $\Gamma_{\mathbb{D}}$  as  $\Gamma_{\mathbb{D}} = \bigcup_{D \in \mathbb{D}} \Gamma_D$ .

Let us consider a simple example. In the Shipping Protocol, let the *shipOrder* sent by the *sender* to the *shipper* be denoted by  $a$  and the shipment sent by the *shipper* in response to the *shipOrder* be denoted by  $b$ . Using these event literals, we can specify the constraint “A

*shipment* can only be received if a *shipOrder* has been sent in the Shipping Protocol” as a dependency  $E = a \cdot b \vee \bar{b}$ .

### 2.3.1 Constraint Language Semantics

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

$$M_1 \tau \models e \text{ iff } (\exists i : \tau_i = e)$$

$$M_2 \tau \models I_1 \vee I_2 \text{ iff } \tau \models I_1 \text{ or } \tau \models I_2$$

$$M_3 \tau \models I_1 \wedge I_2 \text{ iff } \tau \models I_1 \text{ and } \tau \models I_2$$

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

**Denotation.** The denotation  $\llbracket D \rrbracket$  of a dependency  $D$  is the set of runs that satisfy  $D$ , i.e.,

$$\llbracket D \rrbracket = \{\tau : \tau \models D\}.$$

For example, the denotation of the dependency  $a \cdot b \vee \bar{a}$  is the set of runs in which there is no  $b$  preceding  $a$  or, if there is a *shipment*, the *shipOrder* precedes the *shipment* ( $a \cdot b$ ).

### 2.3.2 Residuation

Another important concept in this specification language is residuation. The residual of a dependency  $D$  by an event  $e$  is denoted by  $D/e$  and corresponds to the largest set of runs satisfying the given dependency. Formally,  $\nu \in \llbracket D \rrbracket / e$  iff  $(\forall v : v \in \langle e \rangle \Rightarrow (v\nu \in \mathcal{U}_{\mathcal{I}} \Rightarrow v\nu \in \llbracket D \rrbracket))$ . The residual represents what remains in the dependency after a certain event has occurred for the dependency to be satisfied on any run. The following are residuation rules as given by Singh:

$$R_1 \ 0/e \doteq 0$$

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

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

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

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

$$R_6 D / e \doteq D \text{ if } e \notin \Gamma_D$$

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

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

Continuing with the above example, consider the dependency  $a \cdot b \cdot c$ , where  $c$  denotes a *shipmentAcknowledgement* sent by the *receiver* to the shipper. The above dependency encodes that “*shipOrder* ( $a$ ) is followed by *shipment* ( $b$ ) is followed by *shipmentAck* ( $c$ ).” The residual of this dependency with  $a$  is  $b \cdot c$ , which is what is left to be done to satisfy the dependency after  $a$  occurs. The residual of this dependency with some other literal  $x$ , however, is the dependency itself, since the dependency does not specify whether  $x$  occurs.

## 2.4 Running Example

We illustrate our approach with an example interaction in which a shipper ships some goods to a customer. The shipping protocol as shown in Figure 2.1 ends with a *Shipment* message from the *shipper* to the *receiver*. But this message surely does not capture the fact whether the shipment was actually delivered or not. For example, the *shipper* might have sent the package but it got misplaced on the way. Since the *shipper* is committed to shipping the package once the payment has been made by the *receiver*, it is important for him to monitor this commitment. This is where we incorporate event monitoring into the shipping protocol. At the point when the *Shipper* receives the *shipOrder* message from the *Sender*, it will initiate tracking of the shipment by subscribing to checkpoints (e.g., RFID sensors). When all such sensors return a positive response and finally the receiver too acknowledges the receipt of the shipment, the *shipper* can confirm that the shipment did really go through. Thus we refine the shipping protocol by including tracking in it. The protocol can now be represented as shown in Figure 2.2.

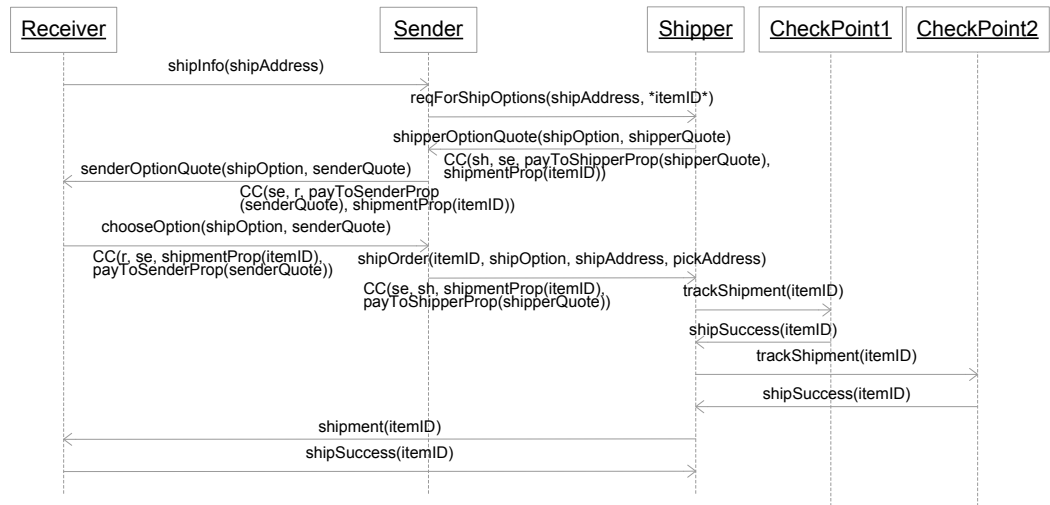


Figure 2.2: Shipping protocol refined to include tracking by sensors at two checkpoints

The protocol is hence complete only after a *shipSuccess* message is received from the receiver. This protocol is more robust since it takes into consideration the possibility of events that will provide extra information about the status of the shipment.

## Chapter 3

# Events in Business Processes

This chapter introduces our approach for achieving a proactive and reliable protocol-based business process enactment model. It introduces the Event Processing Language used to describe events and to incorporate them into business rules. It describes role-specific perspectives of commitments and how they can be used to deduce complex event patterns for fine-grained monitoring of processes.

In our approach, events are incorporated into the protocol-based enactment framework of OWL-P. The rules and policies are refined to accommodate events. The receipt of a message is a simple event. A simple event source could be any of the following:

- Agents participating in the protocol enactment
- Sensors (e.g., RFID sensors) which can provide information about external conditions that may have an effect on an interaction
- Other software applications (e.g., timers, databases etc)
- Other non-participating agents

Complex events are aggregations of simple events that corresponds to a significant business situation. Every agent decides the situations or interactions it wants to monitor based on its own perspectives of commitments. Each agent forms complex event patterns based on these situations. The agents are event processors themselves, and detect complex events in stream by making meaningful sense of lower-level events. They employ the pattern

matching technique to do so. Consequently, the agents react to detected complex events by triggering rules.

### 3.1 Event Processing Language

This is the specification language that combines event logic with rules. Event Processing Language (EPL) [Albek et al., 2005] comprises the following: Event Type, Complex Event Patterns formalized by an Event Pattern Language and Rules with Events incorporated into them.

As Chandy describes [2005] events are processed via rules of the form:

```
WHEN reality deviates significantly from from expectations
THEN respond
```

Let us elaborate this rule by stating that if a simple event occurs and its occurrence leads to the satisfaction of some condition or a complex event, then an action should be taken. The processing of an event is hence split into two steps: one, in which if a simple event occurs, a complex event pattern is evaluated to see if the occurrence of the simple event affected it; and two, in which if a complex event pattern is satisfied then an action is taken. In order to express the WHEN clause or the first part of the rule, we use an *Event Type* to represent a simple event and *Complex Event Patterns* formalized by an *Event Pattern Language* to represent the condition. The THEN clause is represented by business rules.

```
WHEN complex event pattern evaluates to true
THEN activate business rule
```

#### 3.1.1 Event Type

An *event type* specifies the template to represent a simple event. Events are represented as tuples of data as described in [Luckham, 2002]. We represent an event by the tuple  $e(\text{message}, \text{source})$  where message and source are *predefined public attributes* [Luckham, 2002], denoting the message that causes the event and the event source. Event could also be represented as  $e(\text{commitment}, \text{self})$  which denotes a commitment in the agent's own knowledge base. In accordance with our running example, a simple event occurs when a



shipment message is received from Checkpoint<sub>1</sub> by the shipper. (Checkpoint<sub>*i*</sub> is abbreviated  $P_i$  below.) This can be represented by event  $e_1$  as  $e_1(\text{shipSuccess}(\text{itemID}), P_1)$ . Or if we consider the commitment  $\text{CC}(\text{se}, \text{sh}, \text{shipment}(\dots), \text{pay}(\dots))$ , we can represent this by an event at the sender's KB as:

$$e_c(\text{CC}(\text{se}, \text{sh}, \text{shipment}(\dots), \text{pay}(\dots)), \text{self})$$

### 3.1.2 Event Pattern Language

We use the event-driven linear temporal logic of Section 2.3 as the *event pattern language* [Luckham, 2002] to formalize complex event patterns in our system. An example *complex event* or dependency is  $D = a \cdot b \vee \bar{a} \vee \bar{b}$ . The *complex event*  $D$  [Luckham, 2002] is generated from the *aggregation* of simple events  $e_1$ ,  $e_2$ , and  $e_3$ . Here,  $\wedge$  and  $\cdot$  are *pattern operators* that express the relationship between the events as introduced in Section 2.3. This expression checks for completeness of the event, or in other words, that  $E$  is valid. In our approach we differentiate between the occurrence of an event and the conditions that would constitute its non occurrence. Hence, we create a complementary pair,  $E$  and  $\bar{E}$ .

The above dependency  $D$  is split into an event  $E$  and its corresponding complement  $\bar{E}$  in order to distinguish between the desirable and the undesirable conditions. We get  $E = a \cdot b$ , and  $\bar{E} = \overline{a \cdot b} = \bar{a} \vee \bar{b}$ . However  $\bar{E}$  does not capture all the conditions under which  $E$  would fail. In order to ensure the non-occurrence of  $E$  we add  $b \cdot a$  to  $\bar{E}$ , hence resulting in  $\bar{E} = \bar{a} \vee \bar{b} \vee b \cdot a$ . The event  $E$  occurs only if  $a$  occurs, followed  $b$ .  $\bar{E}$  occurs if  $a$  does not occur or  $b$  does not occur or the occurrence of  $b$  is followed by occurrence of  $a$ . The second expression, hence, represents *exceptions* in the system.

Continuing with our running example, we could express a successful shipment by the complex event,

$$E = e_1(\text{shipSuccess}(\dots), P_1) \cdot e_2(\text{shipSuccess}(\dots), P_2) \cdot e_3(\text{shipSuccess}(\dots), \text{Receiver})$$

Or, more simply as  $E = e_1 \cdot e_2 \cdot e_3$ . An exception will be detected if this sequence of events does not happen.

We restrict our sequence expressions to have at most two operands for each operator. Hence the above expression can be reduced to  $E = e_1 \cdot e_2 \wedge e_2 \cdot e_3$ . The implication of this pattern is that if the events  $e_1$ ,  $e_2$ , and  $e_3$  occur sequentially, then a complex event  $E$

occurs, which has a significance and a consequent action. Patterns for such normal complex events are designed by protocol designer once he has decided the commitments it wishes to monitor. The corresponding exception will be  $\overline{E} = \overline{e_1} \vee \overline{e_2} \vee \overline{e_3} \vee e_2 \cdot e_1 \vee e_3 \cdot e_2 \vee e_3 \cdot e_1$ . Derivation of exception patterns from events have been explained further in Section 4.2.1.

### 3.1.3 Event Incorporated Rules

Business action rules are represented in JESS [Friedman-Hill, 2003] according to the pattern:

`condition  $\Rightarrow$  action`

The condition is the evaluation of a complex event pattern to true. The action is any subsequent process that follows. Examples of actions are:

- Instantiate a protocol. It could be the same or a different protocol. This leads to nested protocol enactment.
- Send messages to other participating agents.
- Call another application.

On incorporating events into the typical OWL-P rules for the Shipping Protocol introduced in Section 2.4, the rule would look like:

`contains(KB, shipmentEvent)  $\Rightarrow$  send(receiver, shipment(...))`

where *shipmentEvent* is the complex event formed by aggregating simple events *shipOrderEvent*, *P<sub>1</sub>SuccessEvent*, and *P<sub>2</sub>SuccessEvent*.

$$\text{shipmentEvent} = \text{shipOrderEvent}(\text{shipOrderMsg}, \text{se}) \cdot P_1\text{SuccessEvent}(\text{shipSuccessMsg}, P_1) \cdot P_2\text{SuccessEvent}(\text{shipSuccessMsg}, P_2)$$

A detected exception is handled by activating an exception handling policy. Exception pattern generation and exception handling rule generation have been described in more details in Section 4.2.

## 3.2 Monitoring Decisions

As discussed in Chapter 1, event monitoring is injected into a business protocol at specific points that require special attention. How does one decide what business events are significant enough to be monitored? What analysis is required to design these complex event patterns? To be able to take such design decisions, the designer must be aware of the business needs and goals of the agent. For example, a shipper who intends to serve preferred customers better, may choose to track the item through checkpoints. The same shipper might refrain from tracking shipments for an ordinary customer. Thus the same process may be enacted in several ways depending on the needs of the agent. This can be understood in terms of perspectives on commitments.

The *perspective* of a protocol for a particular agent is the enactment of the protocol under specific circumstances suiting the agent's own purpose. It is a customized version of the protocol that an agent enacts on the basis of the decisions it makes for monitoring commitments. The decision to monitor and hence the design of the complex event patterns depend on the agent's perspective of the protocol.

Every commitment in a protocol involves some condition that needs to be satisfied as discussed in Section 2.2. Since commitments are contracts or obligations, it is important to ensure that a commitment is satisfied. Satisfying the condition requires some interactions and actions by the agent. These interactions or actions are usually messages exchanged as part of the protocol. However the high-level interactions do not guarantee that the condition was satisfied. For example, shipping out a package does not guarantee that the package was delivered under circumstances that the parties agreed on. Again the shipper, sender and customer might deal with the same interaction, i.e., shipping the package, in different ways. These tailored versions of a protocol enactment are the agent's specific perspectives. Let us illustrate perspectives with the following scenario, continuing with our shipping example. In the shipping protocol a commitment is created stating that if the receiver makes a payment, the sender will be committed to sending the shipment to the receiver within a specific time. Another commitment is created between the sender and the shipper that specifies that the shipper, on receiving a ship order from the shipper, must send the goods to the receiver within the agreed upon time.

The protocol is enacted as follows: Once the receiver makes a payment, the sender sends a `shipOrder` message to the shipper, who then sends the shipment to the receiver. These high-level interactions are specified by the protocol rules. We break this entire piece of action into multiple subactions depending on the role the agent is playing. These set of subactions are simple events that are aggregated to form the high-level complex event patterns which are of significance to the business.

Each perspective gives rise to a set of complex events which the agent monitors and handles. Our tool aids the designer in deriving exception patterns from the event patterns automatically. Different perspectives specific to the role and instance of the protocol are described below.

### 3.2.1 Receiver's Perspective of the Shipping Protocol

The receiver's perspective might be to track the shipment with the aid of RFID sensors at intermediate points and a timer. This is illustrated in Figure 3.1. Depending on the outcome of this monitoring the action of the receiver may vary. There are three situations shown in the figure. If the shipment is received within the agreed time, it sends a receive acknowledgment (`shipRecvAck`) to the sender. If the shipment did not reach an intermediate checkpoint, it sends a delay alert (`shipmentDelayAlert`) to the sender. If the timer timed out, i.e., the shipment did not reach it within the agreed upon time, it will notify the sender of the failure by sending the `shipmentFailure` message. Using the event processing language introduced in Section 3.1, we can express the complex events and rules as follows:

Shipment received within the agreed upon time:

$$\begin{aligned} \text{shipmentReceivedEvent} = & \text{shipmentEvent}(\text{shipmentMsg}, sh) \cdot P_1\text{SuccessEvent}(\text{shipSuccessMsg}, P_1) \\ & \cdot P_2\text{SuccessEvent}(\text{shipSuccessMsg}, P_2) \end{aligned}$$

$$\text{shipmentReceivedEvent} \Rightarrow \text{shipmentRecvAck} \wedge \text{stopTimer}$$

$$\text{shipmentFailureEvent} = \text{shipmentEvent}(\text{shipmentMsg}, sh) \cdot \text{timeoutEvent}(\text{timeoutMsg}, \text{timer})$$

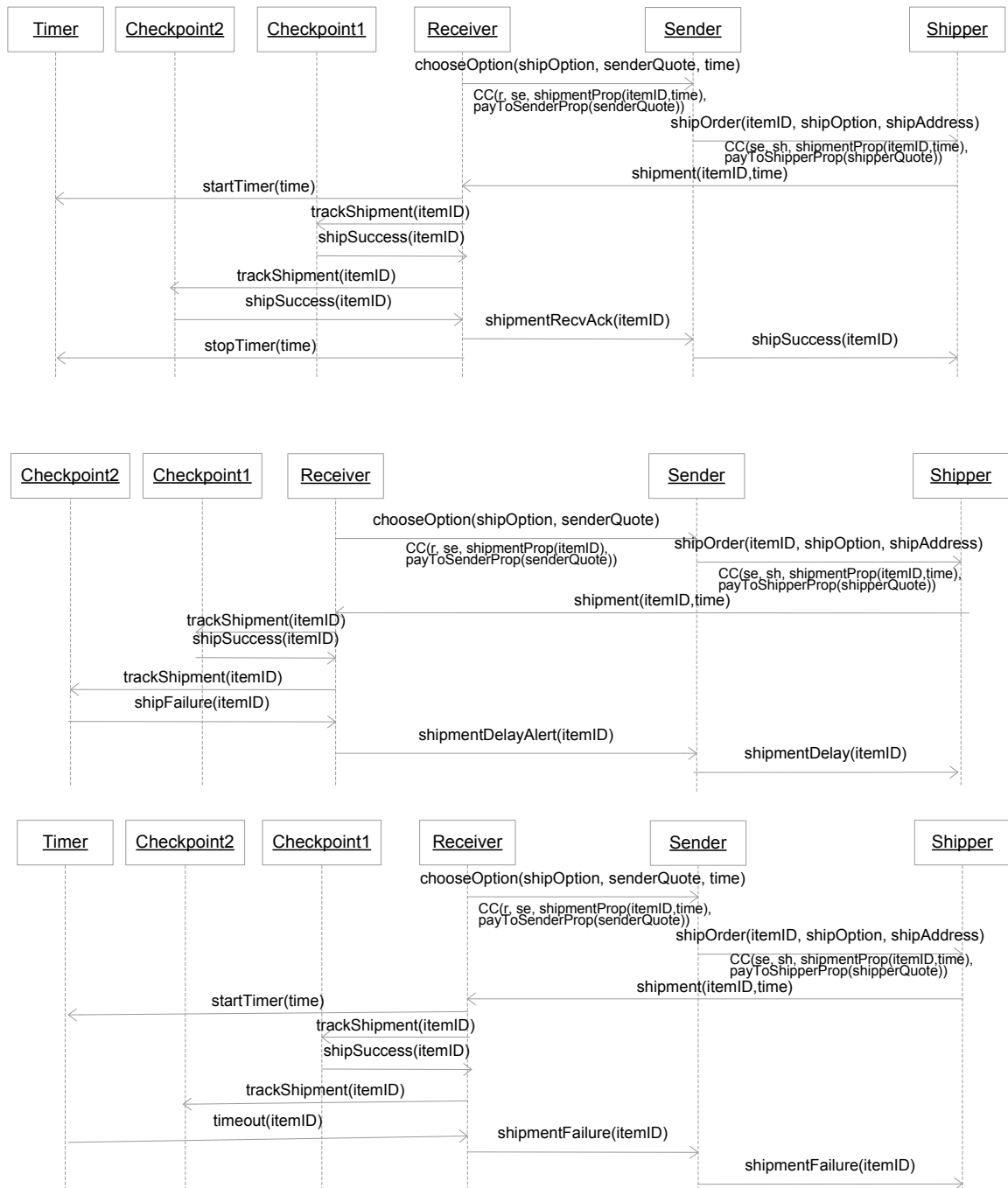


Figure 3.1: Receiver's perspective of shipment in the shipping protocol

### 3.2.2 Shipper's Perspective of the Shipping Protocol

The shipper's perspective of the same interaction is to set a timer and get an alert if the shipment was not delivered within the agreed upon time frame. In that case he would re-send the shipment. In another instance of the same protocol, the shipper might decide to take the responsibility of tracking the shipment through the inventory till its final destination. This is similar to the running example described in Section 2.4. Both these perspectives are shown in Figure 3.2. The events for the shipper would be as follows for the first instance:

*shipmentResendEvent* = *shipmentEvent(shipmentMsg, self)* ·  
*shipmentFailureEvent(shipmentFailureMsg, se)*

For the second instance the complex events and rules are defined as follows:

*shipmentSendEvent* = *shipOrderEvent(shipOrderMsg, se)* · *P<sub>1</sub>SuccessEvent(shipSuccessMsg, P<sub>1</sub>)* ·  
*P<sub>2</sub>SuccessEvent(shipSuccessMsg, P<sub>2</sub>)*

*shipmentSuccessEvent* = *shipOrderEvent(shipOrderMsg, se)* · *P<sub>1</sub>SuccessEvent(shipSuccessMsg, P<sub>1</sub>)* ·  
*P<sub>2</sub>SuccessEvent(shipSuccessMsg, P<sub>2</sub>)* ·  
*Receiver.SuccessEvent(shipRecvAckMsg, re)*

*contains(KB, shipmentSuccessEvent) ⇒ send(se, shipmentSuccess)*  
*contains(KB, shipmentFailureEvent) ⇒ shipmentFailurePolicy*

### 3.2.3 Sender's Perspective of the Shipping Protocol

The sender has a different perspective. The sender, who might be obligated to serve the receiver better since it is a premium customer, might decide to inquire with the shipper after half the time has elapsed if the shipment will make it within the remaining time. Or he might choose to remain passive and forward all messages from the receiver to the shipper. For the first scenario some of the events and rules are as follows:

*shipmentSuccessEvent* = *startTimerEvent(startTimerMsg, self)* ·  
*shipmentReceivedEvent(shipRecvAckMsg, Receiver)*

*contains(KB, shipmentSuccessEvent) ⇒ send(Shipper, shipmentSuccess)*

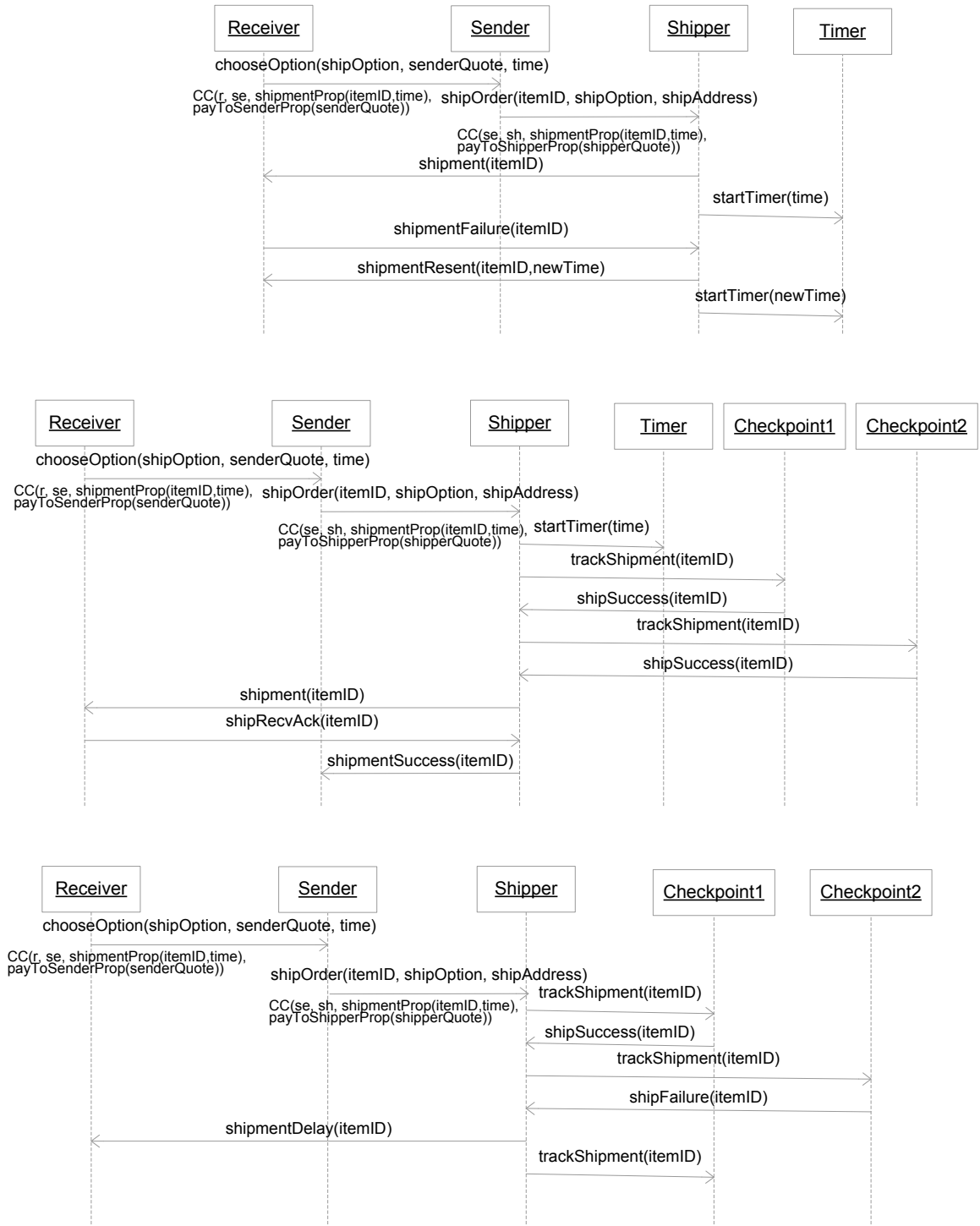


Figure 3.2: Shipper's perspective of shipment in the shipping protocol

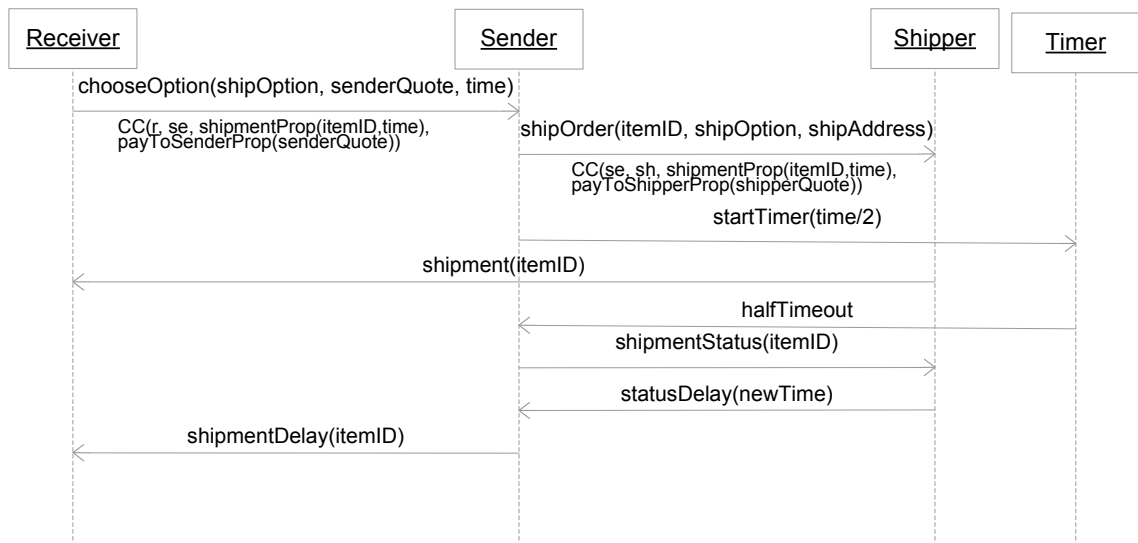
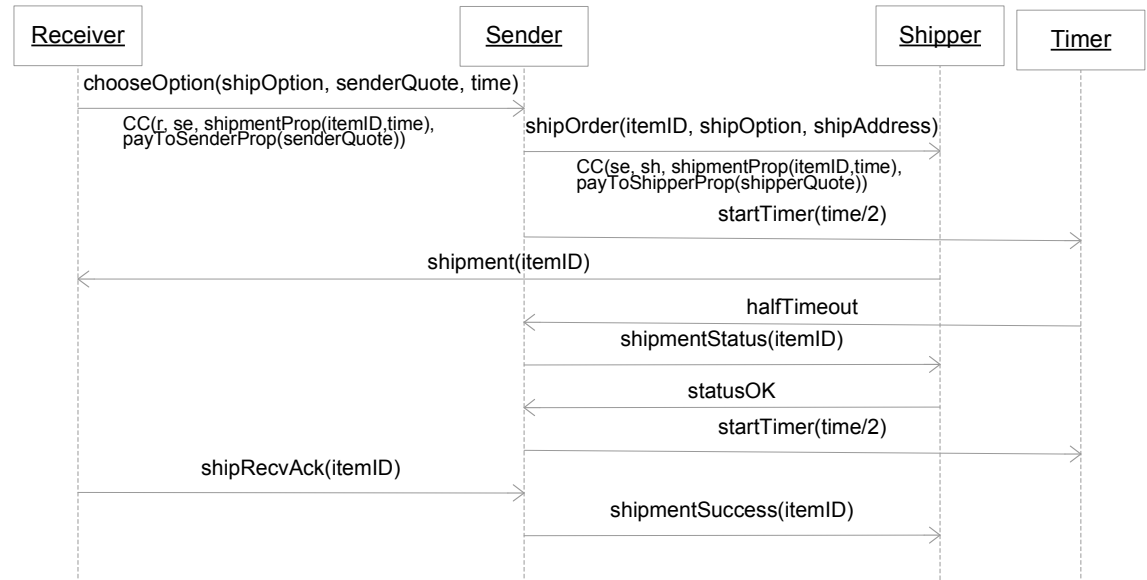


Figure 3.3: Sender's perspective of shipment in the shipping protocol



## Chapter 4

# Architecture and Process Monitoring

This chapter first describes the event-driven agent architecture. It then describes in detail how agents are configured to accommodate events and explains how process monitoring takes place.

### 4.1 Architecture

Our event-driven agent architecture is shown in Figure 4.1. The agent primarily consists of a rule base (RB), a knowledge base (KB) and an event processor (EP). Incoming messages from several event sources are processed by the EP. On detecting an event, the EP stores the event in the KB. The KB then activates rules in the rule base. In case of rules where events have not been introduced, the message is directly stored as a proposition in the KB which consequently activates protocol rules.

The event processor is the heart of the agent. It is similar to the concept of the Enterprise Service Bus [[Maréchaux, 2006](#)] except that the subscriptions in the event processor are based on state changes and these subscriptions can be optimally managed.

These are the tasks carried out by the event processor:

- Receives messages and transforms them into events conforming to the event type explained in Section 3.1.
- Aggregates lower level events to form complex patterns. Matches these patterns by

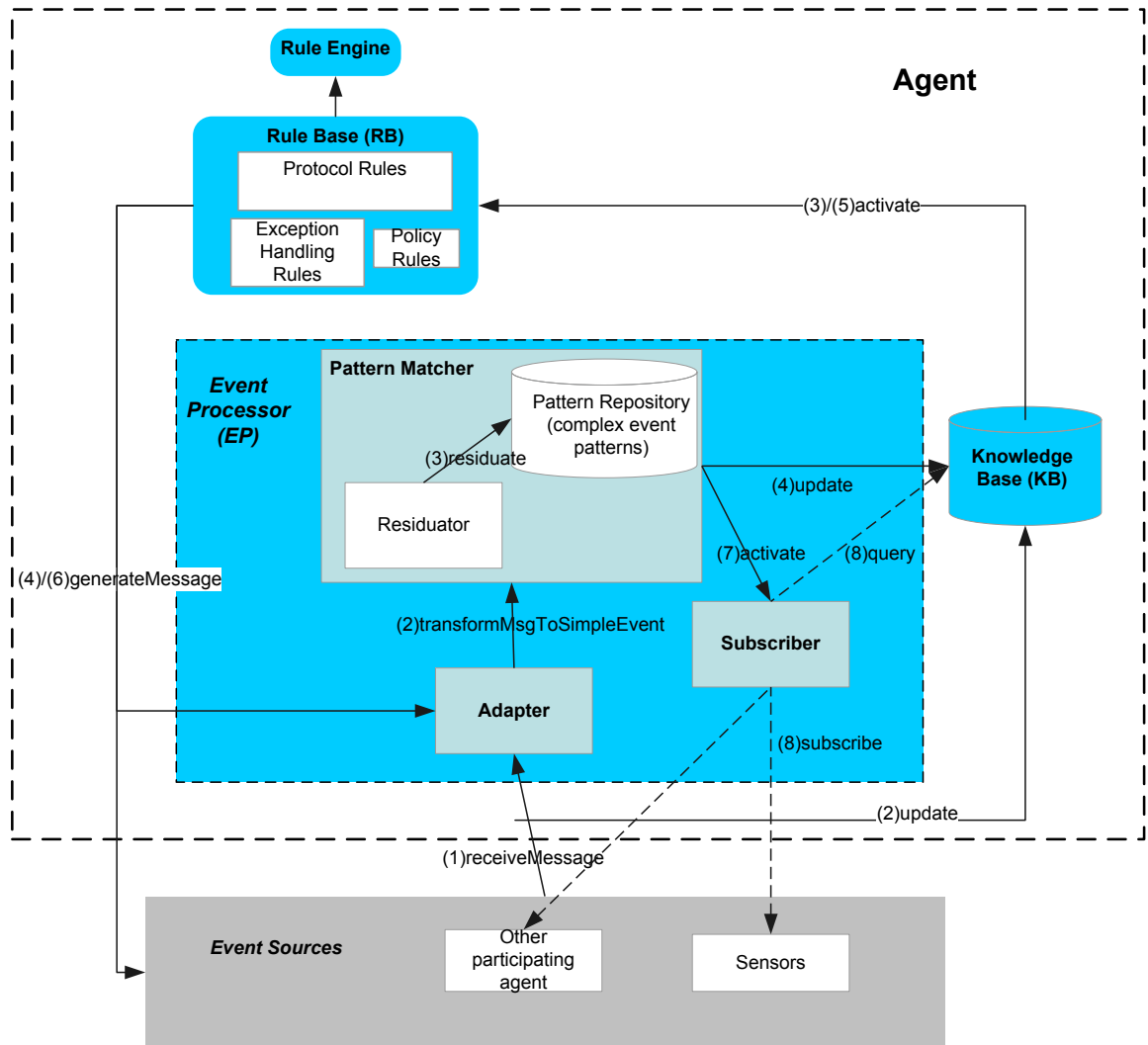


Figure 4.1: Event-driven agent architecture

residuation of the complex events with simple events.

- Manages subscriptions to event sources depending on the order in which it expects events to occur and depending on whether the agent chooses to subscribe selectively or not.

The following modules constitute the EP:

**Adapter.** The *adapter* is the entry point for messages. It transforms incoming messages into events conforming to the event type which will be described in Section 3.1. Such

events are then handed to the *pattern matcher*.

**Pattern Matcher.** The Pattern Matcher is responsible for aggregating simple events via complex event patterns available in the *pattern repository*. It is hence responsible for detecting expected patterns in the incoming events. It is made up of two components:

- The *pattern repository* maintains the current state of the event and exception patterns. It is initialized during the configuration of the EP with the new event patterns, as designed and exception patterns, as derived.
- The *residuator* performs residuation as explained in Section 2.3 on the event patterns in the pattern repository and updates its state. If any of the dependencies or event patterns evaluate to true, that event is asserted to the knowledge base. Further, the Pattern Matcher activates the subscriber if selective subscription is turned on.

**Subscriber.** The subscriber consults the pattern repository to determine the upcoming simple events that might lead to the occurrence of a complex event. It decides which source to listen to next, in order to get relevant information. Selective subscription is optional.

## 4.2 Event Configuration

At the beginning of the protocol enactment, the agent undergoes an initial setup phase called Event Configuration. Event configuration involves the following distinct tasks:

- Generation of complex exception patterns from high-level event patterns.
- Initialization of the Pattern Repository with event and exception patterns.
- Generation of exception handling rules.

Event configuration with the aid of our tool has been shown in Figure 4.2.

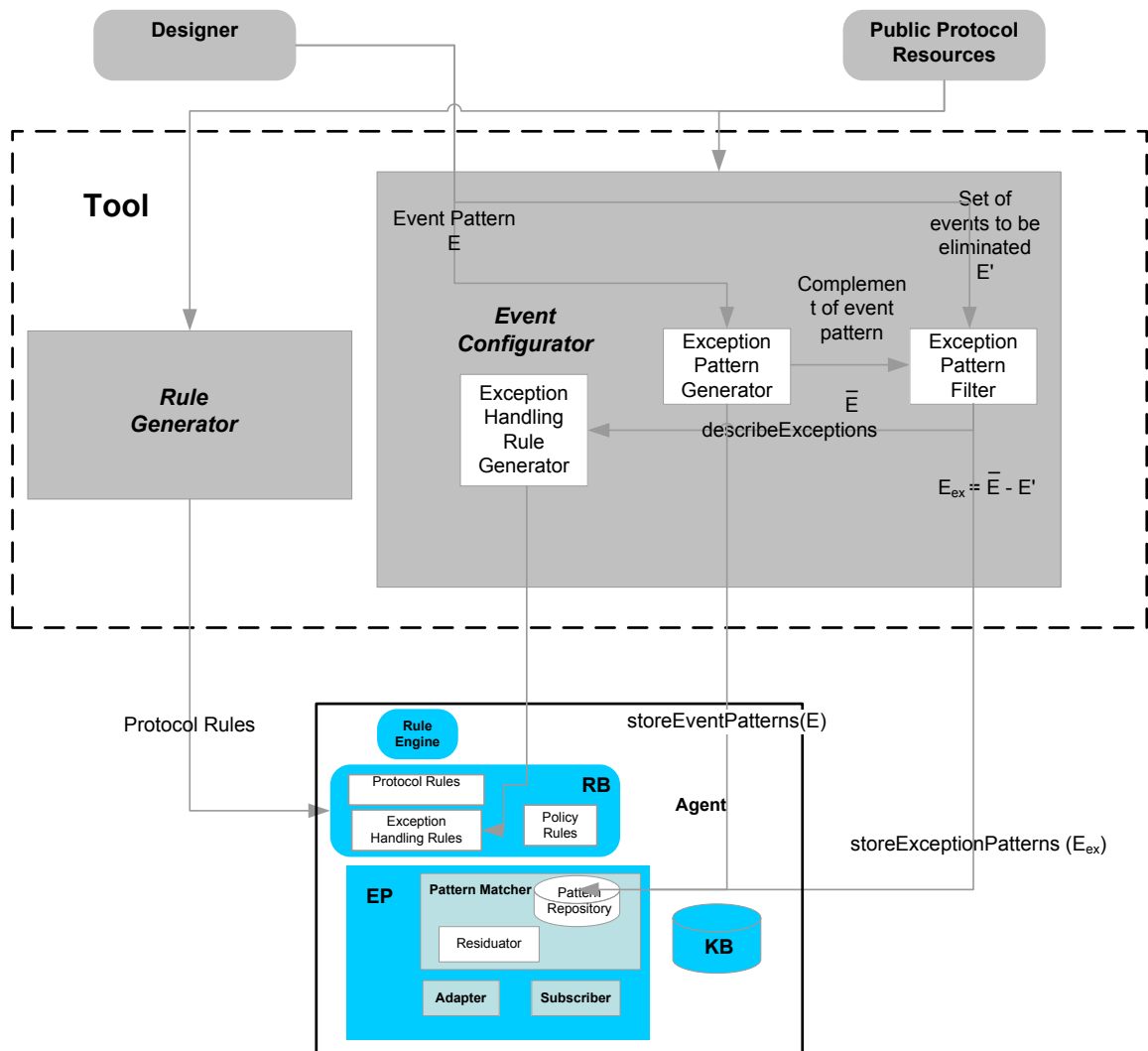


Figure 4.2: Event configuration

### 4.2.1 Deriving Exception Patterns

Complex event and exception patterns are formalized by the pattern language described in Section 3.1.2. The event processor derives patterns of exceptions from an event pattern  $E$  in two simple steps.

- Complementing the complex event  $E$  to generate all possible conditions corresponding to  $E$  not happening. This pattern is represented as  $\bar{E}$ , as explained in Section 3.1.2.

- Refining  $\overline{E}$  by eliminating conditions that cannot happen or are uninteresting. This step requires designer input.

The failure of the shipment event, i.e.,  $\overline{\text{ShipmentEvent}}$ , or  $\overline{E}$ , is computed by the *Exception Pattern Generator*  $\overline{E} = \overline{e_1 \cdot e_2} \vee \overline{e_2 \cdot e_3}$ . On simplification, we obtain  $\overline{E} = \overline{e_1} \vee \overline{e_2} \vee \overline{e_3} \vee e_2 \cdot e_1 \vee e_3 \cdot e_2$  according to our event pattern language.

This complex event pattern can further be simplified by filtering out events that have no possibility of occurrence. For example,  $e_2 \cdot e_1$  indicates that the  $P_2$  sends a success message before  $P_1$ . This is an impossible sequence. Or even if such a sequence occurs, under the circumstance that  $P_1$  somehow misread and gave an erroneous response, the occurrence of  $e_2 \cdot e_1$  would indicate an error. Thus this sequence can be removed from the exception pattern  $\overline{E}$ . Similarly  $e_3 \cdot e_2$  can also be eliminated. Another example where a simple event may be filtered out is where an agent has subscribed to a timer to receive an alert after some elapsed time to ensure event  $x$  occurs before the timeout occurs. This can be represented by the event patterns,  $E = x \cdot \text{timeout}$  and  $\overline{E} = \overline{x} \vee \overline{\text{timeout}} \vee \text{timeout} \cdot x$ . The event  $\overline{\text{timeout}}$  can be ignored since it has no significant effect. Hence  $\overline{E}$  would reduce to  $\overline{E} = \overline{x} \vee \text{timeout} \cdot x$ .

The agent may also choose to be specific about a particular exception. For example, it may decide to split the entire exception pattern into different simple exceptions. It may, hence, choose to monitor the occurrence of  $\overline{e_1}$ ,  $\overline{e_2}$  and  $\overline{e_3}$  as separate exceptions each leading to a different course of action. The task of elimination is performed by the *exception pattern filter* or EPF with the aid of human input.

The EPF takes a set of events  $E'$  as input from the protocol designer and performs  $\overline{E} - E'$  to generate  $E_{ex}$ . Here,  $E_{ex} = \overline{e_1} \vee \overline{e_2} \vee \overline{e_3}$ . This is the exception dependency that the agent will check for in addition to the normal event  $E$ . This exception can be called ShipmentException. The combination of ShipmentEvent ( $E$ ) and ShipmentException ( $E_{ex}$ ) completes the check for a successful shipment. The *Pattern Repository* of the *Pattern Matcher* module is initialized with such sets of event patterns and their corresponding exception patterns.

### 4.2.2 Generating Exception Handling Rules

Detected exceptions are handled by a new set of rules generated by the agent. The rules state that the agent's internal policies should be triggered on occurrence of an exception. Policy-driven exception management separates exception-handling from the normal business logic [Zeng et al., 2005]. The action taken is independent of the protocol and is the agent's own private decision. The policy might be a simple action like sending a message or could be something more complicated like initiating another protocol. The latter would lead to nested protocols arising from event occurrences. This gives a flavor of a truly event-driven protocol-based business process enactment.

For example, continuing with our previous example, the rule to handle the exception  $E_{ex}$  is

$\text{contains}(\text{KB}, E_{ex}) \Rightarrow E_{ex} \text{Policy}(\dots)$

or

$\text{contains}(\text{KB}, \text{ShipmentException}) \Rightarrow \text{ShipmentExceptionPolicy}(\dots)$

This rule means that if the event *ShipException* exists in the knowledge base the *ShipExceptionPolicy* of the agent should be consulted to take a decision of the necessary action. The *ShipmentExceptionPolicy* is an internal business decision of the shipper. It might decide to act in several ways, some examples of which are given below.

- Resend the package and thereby reinitiate this entire process.
- Send a message to the customer informing it of a delay.
- Delegate the control of action to  $P_1$  who reported an exception, who in turn might track the shipment via another checkpoint. This leads to a nested shipping protocol.

Algorithm 1 describes how event configuration takes place. It takes a set of complex event patterns  $E$  as input and produces a configured event pattern repository and an updated set of rules.

```

1 Procedure configureEvent ( $\mathbb{E}, \mathbb{E}', R$ ): Configures the event pattern
   repository with events and exceptions and generates exception handling rules
   input : A set of complex event patterns  $\mathbb{E}$ , set of exception filtering patterns  $\mathbb{E}'$ 
           and a set of rules  $R$ 
   output: A configured event pattern repository  $T$  and an updated set of rules  $R$ 
2  $T \leftarrow \{\}$ ;
3 foreach  $E_n$  in  $\mathbb{E}$  do
4    $T \leftarrow T \cup E_n$ ;
5    $E_{ex\_n} \leftarrow \overline{E_n} - E'_n$ ;
6    $R = \text{generateRule}(E_{ex\_n}, R)$ ;
7    $T \leftarrow T \cup E_{ex\_n}$ ;
8 Procedure generateRule ( $E_{ex}, R$ ): Generate exception handling rules for
   exception  $E_{ex}$  and add it to the set of rules  $R$ 
   input : Exception  $E_{ex}$  and a set of rules  $R$ .  $R = \{r\}$  where  $r = \langle E, A \rangle$ .  $E$  is an
           event that triggers the rule to take an action  $A$ 
   output: Updated set of rules  $R$ 
9  $r.E = E_{ex}$ ;
10  $r.A = E_{ex}Policy$ ;
11  $R \leftarrow R \cup r$ ;
12 return  $R$ ;

```

**Algorithm 1:** configureEvent( $\mathbb{E}, \mathbb{E}', R$ ): Configures the event pattern repository with events and exceptions and generates exception handling rules.

### 4.3 Process Monitoring

Process monitoring involves the following phases: Event Processing, Selective Subscription and Exception Recovery.

### 4.3.1 Event Processing

Once the event processor has been configured at the beginning of the protocol enactment, the agent is all set to begin monitoring. The runtime monitoring occurs in the order specified in Figure 4.1. Continuing with our running example, let us consider the shipper agent who has committed to ship an item to the receiver. It has sent out the shipment and is now beginning to track the package. After deriving its role in the shipping protocol, the events *ShipmentEvent* ( $E$ ) and *ShipmentException* ( $E_{ex}$ ) have been derived and stored in the pattern store as explained above in Section 4.2. The shipper needs to match the event pattern  $E = e_1 \cdot e_2 \cdot e_3$  and exception pattern  $E = \overline{e_1} \vee \overline{e_2} \vee \overline{e_3}$ . It has subscribed to  $P_1$  in order to get information about the status of the package. When  $P_1$  receives the package and forwards it to the next destination, it sends the message *shipSuccess*(*itemID*) to the shipper. The sequence of action that occurs after the receipt of the message (Step 1 in Figure 4.1) is described in the following subsections.

#### Transformation

Event transformation (Step 2 in Figure 4.1) is the process of converting a message received to a simple event conforming to the event template described in Section 3.1. The transformation of an incoming message  $M$  involves setting the message and source fields for an event appropriately based on the message. For example, message *shipSuccess*(*itemID*) from  $P_1$  is transformed into event  $e_1(\text{shipSuccess}(\text{itemID}), P_1)$ . The adapter then forwards this event to the pattern matcher asking if this message has any effect on any events in the pattern repository.

#### Pattern Matching

The pattern matcher performs residuation (Step 3 in Figure 4.1) on all complex event patterns in the pattern repository with this event. The details of residuation is explained in Section 2.3.2. The complex event pattern in the pattern repository thus reduce to:

$$E/e_1 \doteq (e_1 \cdot e_2)/e_1 \wedge (e_2 \cdot e_3)/e_1 \text{ or } E/e_1 \doteq e_2 \cdot e_3$$

If any complex event pattern evaluates to true due to the occurrence of  $e_1$ , then that event is stored in the KB as a proposition (Step 4 in Figure 4.1). If the occurrence the



complex event activates some rule in the rule base (Step 5 in Figure 4.1), the consequent action is taken. The action might involve sending messages to other agents as a part of the protocol (Step 6 in Figure 4.1). These sent messages are again fed to the adapter (Step 6 in Figure 4.1) to be transformed to simple events which might affect the complex patterns in the pattern repository. Next the pattern matcher activates the subscriber if selective subscription is turned on (Step 7 in Figure 4.1), analyze event trends, and selectively subscribe (Step 8 in Figure 4.1) to different event sources based on its prediction of what might happen next and who might be responsible for it. This concept is explained in more detail in the Section 4.3.2.

### Action

As discussed earlier in Section 3.1, the action to be taken is driven by business rules. These rules are stored in the rule base. On the occurrence of event  $E$ , if the rule base contains the rule  $E \Rightarrow x$ , action  $x$  will be implemented. For example, for a *shipmentEvent*,  $E$ , the following rule is activated.

```
contains(KB, E)  $\Rightarrow$  send(receiver, message)
```

This shows how the task of event monitoring is divided between the Event Processor and the Rule Base. The event processor evaluates and detects complex events  $E$  and the Rule Base executes business rules when  $E$  occurs.

The event processing steps as described in this section are summarized in Algorithm 2.

### 4.3.2 Selective Subscription

In our architecture the agent has the option to turn on selective subscription by which it can minimize the number of event sources it needs to subscribe to. While evaluating a complex event pattern, if a particular simple event that is part of the aggregation occurs, looking at the pattern the agent can tell what to expect next and from which event source to expect it. It will then subscribe to the expected event source and may unsubscribe itself from all sources that will have no further influence on its reasoning. The advantage of this is when the agent is part of a huge distributed network where there are thousands of

```

1 Procedure processEvent ( $T, M$ ): Event processing is done in three steps.
   Transform Message  $M$  to  $e$ . Residuate the set of event patterns and exception
   patterns in event repository  $T$  with  $e$ . Get a set of event sources to subscribe to.
input : An event pattern repository  $T$ , a message  $M$ 
output: A residuated repository  $T'$ , a set of simple event sources  $S_{sub}$  to
        subscribe to.

2  $e.m = M$ ;
3  $e.s = M.sender$ ;
4  $S_{sub} = \{\}$ ;
5 foreach  $E$  in  $T$  do
6    $E = E/e$ ;
7   if ( $E == \top$ ) then
8      $\text{updateKB}(E)$ ;
9      $S_{sub} = \text{subscribe}(E)$ ;
10 foreach  $Eex$  in  $T$  do
11    $Eex = Eex/e$ ;
12   if ( $Eex == \top$ ) then
13      $\text{updateKB}(Eex)$ ;
14    $S_{sub} = S_{sub} \cup \text{subscribe}(Eex)$ ;

```

**Algorithm 2:** processEvent( $T, M$ ): Process events with the help of the complex event patterns stored in the event repository  $T$ , on the receipt of a message  $M$

event sources, it does not need to waste time and memory on subscribing to sources that would not assist in its decision making. For example, consider an airline service that listens continually to a weather service that informs it of the current weather. Now suppose there is a serious threat like a hurricane that the weather service informs the airline service about. But the weather service does not know the details of the hurricane to predict the exact time and place based on which the airline will need to divert its flights. The airline can then subscribe to a specific service called the emergency service which has been set up for the emergency situation and gives more detailed information that the airlines needs. The

airline, however, does not need to subscribe to the emergency hurricane service on a regular basis.

We demonstrate selective subscription with our running example. Say  $e_1$  has occurred. On residuating  $E$  with  $e_1$ , the complex event pattern  $E$  reduces to  $E = e_2 \cdot e_3$ . The shipper can then subscribe to the source of  $e_2(\text{shipSuccess}(\text{itemID}), P_2)$ , which is  $P_2$  and may unsubscribe itself from the source of  $e_1$ , which is  $P_1$  since it does not need any more information from  $P_1$ . If  $e_1$  had not happened, i.e., the package did not reach the first checkpoint,  $P_1$ , it would not make any sense to listen to  $P_2$ . Since the events are specified in a particular order, it is easy to predict what to expect next and from which agent. In some cases the agent may need to query its own knowledge base to check for a particular event in order to residuate patterns in the pattern repository.

Algorithm 3 implements Selective Subscription using our event pattern language. Every event pattern is represented by a binary tree similar to the one shown in Figure 4.3. The dark arrows indicate the node that is being subscribed to. At every step, the set of nodes to be subscribed to is computed.

After the first round of residuation, the subscriber will subscribe to all those event sources that are the *left* children of a ‘.’ node, and to all other sources that are children of ‘^’ or ‘v’ nodes. This implies that if a complex event evaluates to true for a number of simple events AND’ed or OR’ed together, then subscribe to sources of all events that are operands of the AND or OR operator. If, however, there is a temporal operator ‘.’, then subscribe to the source of the event which is the left operand of the temporal operator, i.e., which will occur first. This management of subscriptions is described in Algorithm 3.

### Test Cases to Demonstrate Selective Subscription

We used different order of simple events on some complex event patterns to demonstrate event monitoring, selective subscription, and exception detection and recovery. We use the complex event pattern  $E = (e_1 \cdot e_2 \vee e_4 \cdot e_5) \wedge e_6$  and a refined exception pattern  $E = \overline{e_4} \vee e_2 \cdot e_1 \vee \overline{e_6}$ . We use different orders of occurrence of the simple events  $e_1$ ,  $e_2$ ,  $e_4$ ,  $e_5$ , and  $e_6$  to evaluate how selective subscription takes place. It is evident from the expression that if  $e_6$  occurs first, then the rest of the sources need not be subscribed to.

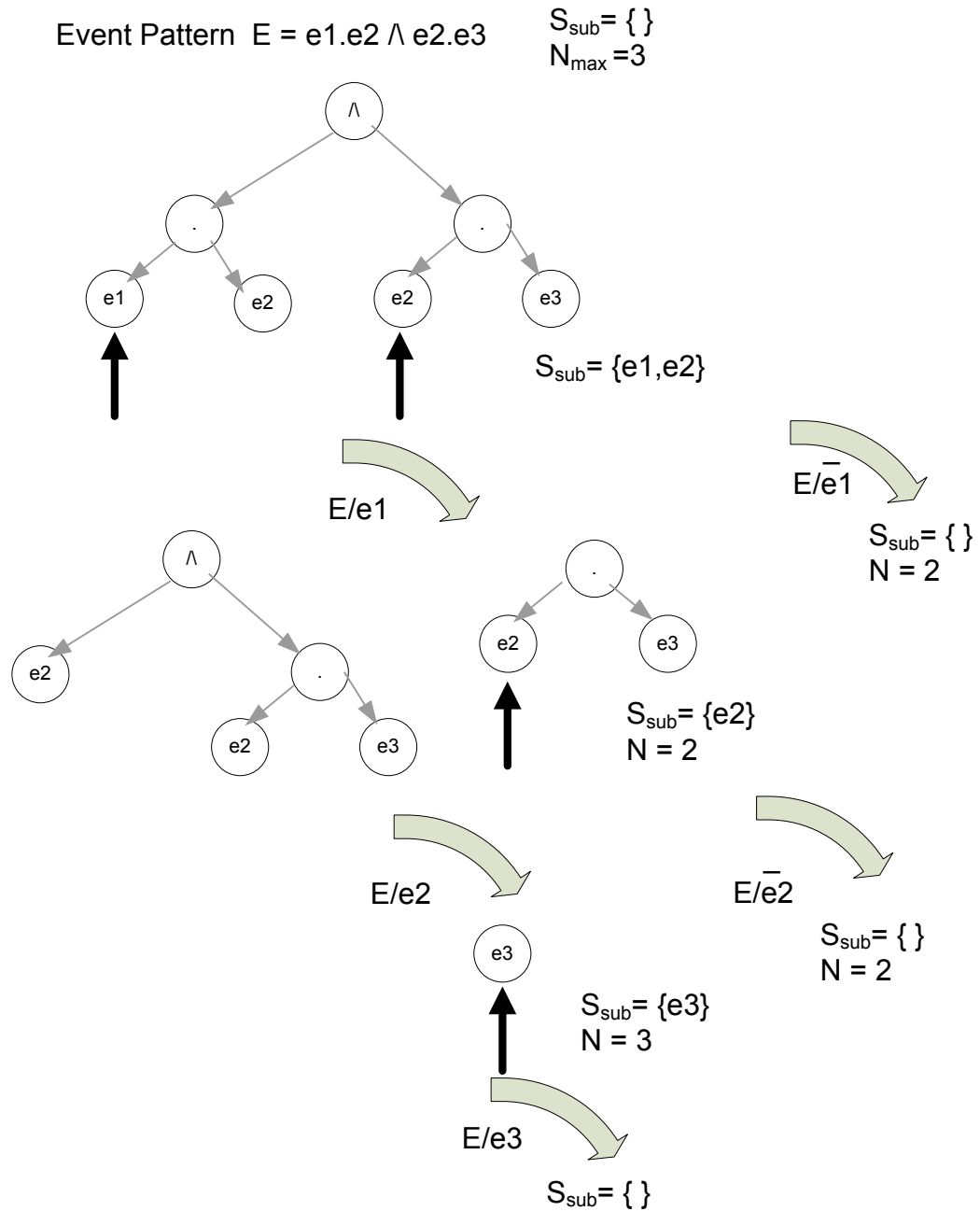


Figure 4.3: Selective subscription for a complex event pattern

Hence, though the agent could have subscribed to six event sources, it ends up subscribing only to one event source in order to enact the process.

Figure 4.4 shows the result of the evaluation of complex event  $E$  due to the occurrence

```

1 Procedure subscribe ( $E$ ): Find the set of event sources  $S$  to subscribe to for
  a Complex Event Pattern  $E$ 
  input : A complex event pattern  $E = \langle e_x, r_x \rangle$  where  $e = \langle m, s \rangle$ .  $e$  is a simple
    event represented in terms of message  $m$  and its source  $s$ 
  output: A set of simple event sources  $S = \{s_x\}$ 

2 foreach dependency  $D$  in  $E$  do
3   foreach sequence  $seq$  in  $D$  do
4     if ( $seq == e_1 \cdot e_2$ ) then
5        $S \leftarrow S \cup e_1.s$ ;
6 return  $S$ ;

```

**Algorithm 3:** subscribe( $E$ ): Find the set of event sources  $S$  to subscribe to for a Complex Event Pattern  $E$

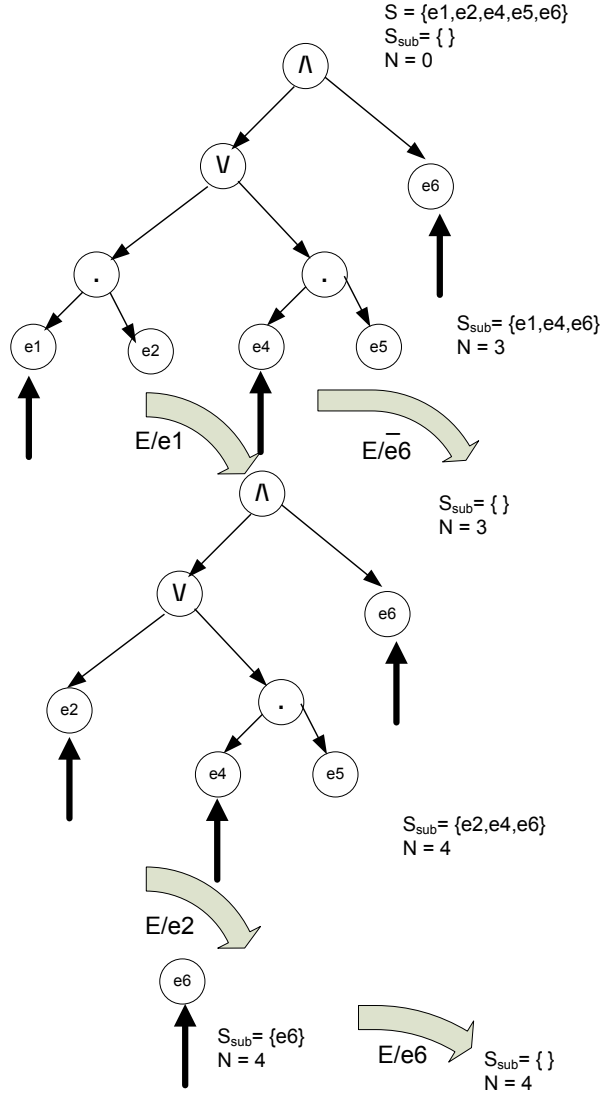
of the simple events in different orders. At every step we represent the total number of subscriptions made till that step, by the number  $N$ . In many cases the expression can be evaluated without the need of subscribing to all event sources. In other words, if the number of available event sources is  $N_{max}$ , in most of our tests  $N \leq N_{max}$ . Thus the resources and computing power that would be required to subscribe to  $N_{max} - N$  event sources, is saved.

### 4.3.3 Exception Detection and Recovery

Exceptions are detected as soon as any of the Exception patterns evaluate to true. Let us consider the circumstance in which after  $e_1$  occurs and then  $e_2$  fails due to a missing package at  $P_2$ , i.e.,  $\bar{e}_2$  occurs. On residuation of all the current patterns in the pattern repository with  $e_2$ , the exception  $E_{ex} = e_2 \vee e_3$ , will evaluate to true. Thus, a specific exception is detected by aggregating simple events.

Once an exception is detected, it is stored in the KB. That is  $E_{ex}$  is stored in the KB and it activates the  $E_{ex}Policy$  in the rule base. The policy reflects the agent's decision making, independent of the protocol. As explained in Section 4.2.2, the sender could resend

Event Pattern  $E = (e1.e2 \vee e4.e5) \wedge e6$   
 $N_{\max} = 5$



Exception  $E_{\text{ex}} = \bar{e4} \vee e2.e1 \vee \bar{e6}$   
 (after refining by EPF)  $N_{\max} = 4$

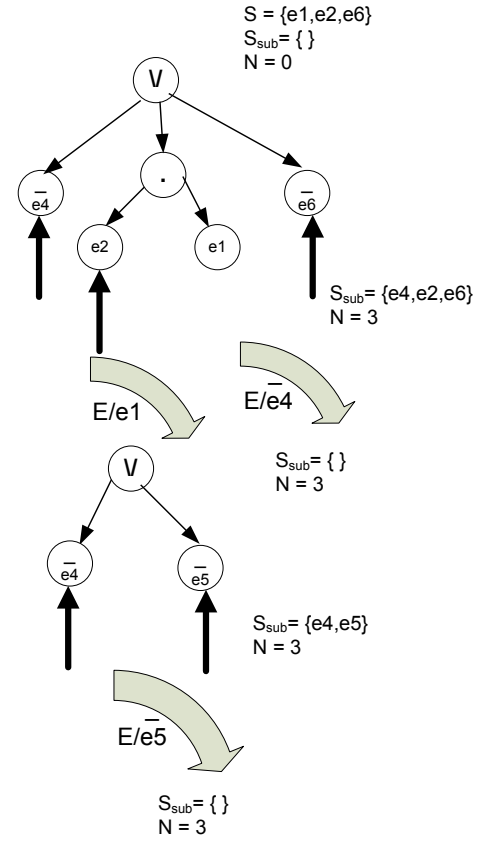


Figure 4.4: Selective subscription

the package and the shipper could start a whole new tracking protocol leading to nested protocol instantiation or the shipper could just send a message to the receiver notifying it of a delay.

This exception handling mechanism is forward error recovery i.e., it involves transforming the system components into any correct state [Cristian, 1989] as opposed to backward

error recovery, which is based on rolling system components back to the previous correct state. Forward error recovery, using exception handling mechanisms, is extensively exploited in the specification of composite web services in order to handle error occurrences. For instance, in BPEL4WS [BPEL, 2003], exception handlers (referred to as fault handlers) can be associated to a (possibly nested) activity so that when an error occurs inside an activity, its execution terminates, and the corresponding exception handler is executed.

The different types of exceptions, i.e., basic, expected and unexpected exceptions can be handled by our approach. This has been demonstrated by the different use-cases in Chapter 5.

#### 4.3.4 Nesting of Protocols

In order to achieve process monitoring we often find protocols being instantiated during the enactment of another protocol. Nesting of protocols may occur in order to monitor events or as a consequent action of an event.

For instance, in our running example, we introduce sensors to track a shipment. The interactions between the sensor and the shipper can be perceived to be specified by a protocol called the Tracking Protocol shown in Figure 4.5. In order to communicate with the checkpoints during the shipping protocol enactment, the shipper instantiates a tracking protocol with the checkpoints. Thus the tracking protocol is nested within the shipping protocol for process monitoring.

Nesting of protocols may also occur as a consequence of some action. For instance, in our running example, as a consequence of a shipment delay, the receiver might decide to instantiate a new shipping protocol by providing a new shipping address. However it may receive the shipment it is expecting before the second instance of the protocol can be enacted completely. In that case, it halts the second instance and completes the first instance of the protocol. This is demonstrated in Figure 4.6. The messages exchanged as a part of the first instance of the protocol are shown in solid lines while the interactions in the second instance are shown by dotted lines.

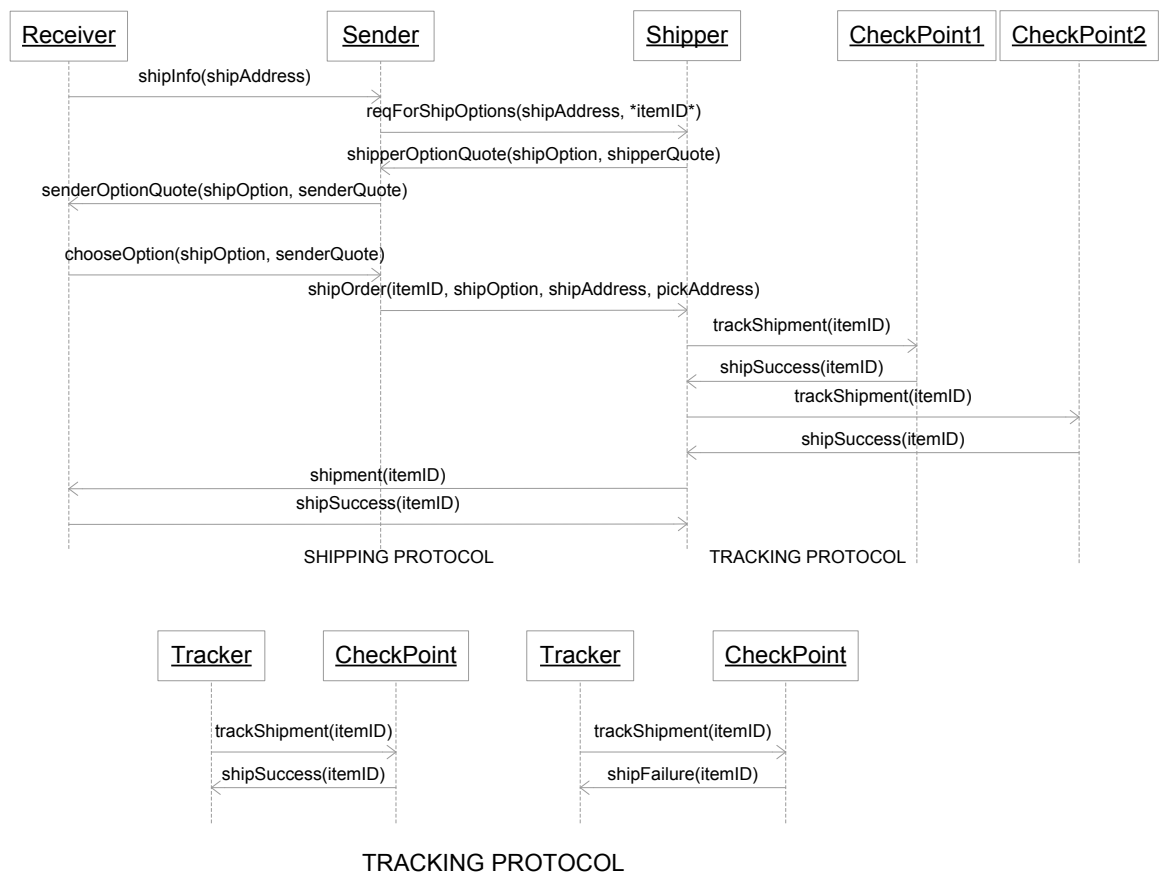


Figure 4.5: Tracking protocol nested within the shipping protocol





Figure 4.6: Shipping protocol nested within another instance of the shipping protocol

## Chapter 5

# Evaluation

This chapter describes the prototype implemented and the use cases to evaluate our approach. The use cases cover different types of exceptions and opportunities.

### 5.1 Prototype

We demonstrate the feasibility of the architecture by implementing a prototype. Each agent is an independent J2EE application running multiple subscription and point-to-point communication channels with other agents and sensors. The rule base and knowledge base have been implemented using JESS [[Friedman-Hill, 2003](#)]. We enact the local process of an agent by generating the Jess equivalent of its role skeleton and business logic stubs.

Figure [5.1](#) outlines the typical steps that need to be taken to configure an agent.

Initially, the protocol designer who wishes to incorporate process monitoring, needs to decide the specific events he wishes to monitor. He is aided by the Commitment Generator. The Commitment Generator takes input from the public protocol specifications and generates all the commitments that will be created during the protocol enactment. The designer uses this to decide which commitments it wishes to monitor depending on its perspective. Based on his analysis, he designs the complex event patterns.

The exception generator helps to derive exception patterns from the complex event patterns. The pattern repository is initialized by these patterns. Pattern matching is implemented using Java Regular Expressions [[Sun Microsystems, 2005b](#)].

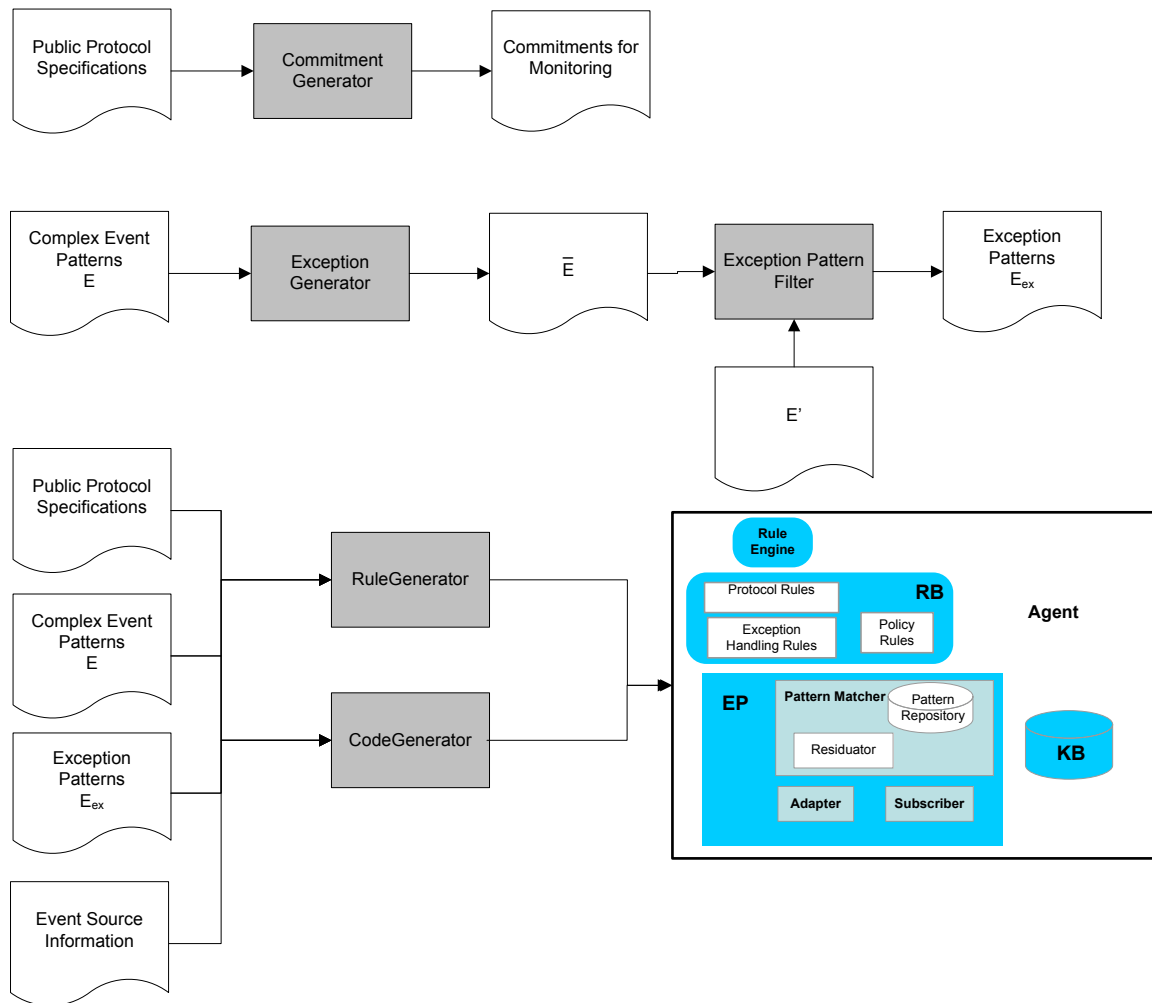


Figure 5.1: Tool aided agent configuration

Next, the event and exception patterns are fed to the Rule Generator and Code Generator along with the public protocol specifications and information about other event sources. This step may also need some designer input. The Rule Generator generates the protocol rules which are JESS equivalent of the OWL-P protocol specifications. It also generates exception handling rules and outlines policy rules. The policies need to be manually defined by the designer.

Once the rules have been generated, the policies written and events configured, the code generator generates a J2EE-based agent that loads these rules into the Jess engine, and communicates using a JMS [Sun Microsystems, 2005a] queue in a WebSphere Application

Server 6.0 [IBM Corp.]. The business process instance starts when all the participating agents have been deployed.

The sensors we have used to demonstrate our use cases are limited to respond with a “success” or “failure” message. The conditions under which they respond are simulated manually.

## 5.2 Use Case 1: Shipping Protocol

The Shipping Protocol introduced as running example in Section 2.4 has been implemented successfully. The different perspectives of the protocol as discussed in Section 3.2 have been implemented as different test cases. The various scenarios tested are missing item, lost message, no response from participating agent, failure at different checkpoints, timeouts and successful enactment. The test cases covered all kinds of exceptions like basic failures, expected and unexpected events. Some of the event incorporated rules generated for the same, and the log files which recorded the sequence of messages and events for each agent are shown in Appendix A.

Some special cases of the Shipping Protocol have been used to demonstrate how different types of exceptions and opportunities are handled.

### 5.2.1 Handling Conditions and Violations: Shipping protocol under special conditions for perishable items

This example demonstrates how event monitoring applies for shipping under special conditions. For example, consider the shipment of medicines or food that can be transported only through a specific temperature range failing which, the items could perish.

As we discussed earlier, the shipping protocol only outlines the rule that once a shipOrder is received, the shipper is committed to send the shipment to the receiver. Under special shipping conditions the shipper may introduce event monitoring to ensure that the temperature restrictions are obeyed. This is shown in Figure 5.2. The shipper routes the item through multiple stations (say StationA and StationB) in order to deliver to the final destination which is the receiver. Before it can route through a particular station, the shipper

checks with the station whether the temperature there is safe for the perishable items to pass. Only if it receives an OK from the station could the shipper proceed to ship through it. As soon as there is a change in route (route via StationC instead of StationB) due to unsuitable temperature, the shipper informs the receiver of the delay in shipment due to re-routing. The receiver in turn alters its actions accordingly.

The receiver, in his perspective, may decide to check the quality of the item once it is received to ensure that it has not perished. If his condition is violated, he may return the item and seek a refund.

This use case demonstrates how conditions can be adhered to and how violations can be detected and corrected.

### 5.2.2 Exploiting an Opportunity: Combining Orders to be Shipped

As we discussed earlier, monitoring business events in protocols not only helps in handling exceptions but also assists in capturing opportunities. For instance, consider a scenario where the shipper receives multiple orders for shipment to the same receiver from different senders. The shipper may choose to combine the shipments to the same destination. This is demonstrated in Figure 5.3. The Shipper subscribes to a database *shipOrderRecord* requesting it to notify it of any identical receiver i.e, the same shipping address and ship option. For every *shipOrder* it receives, it records it with the *shipOrderRecord*. On identifying two identical receivers, *shipOrderRecord* notifies the shipper who, consequently, combines the shipment of the two items.

## 5.3 Use Case 2: Money Wiring Protocol

A Money Wiring Protocol is shown in Figure 5.4. The three roles involved in the protocol are the *customer*, *bank*, and *destinationBank*. The customer sends a *wireReq* message providing the details of the transfer which includes his own account number (*accNo*), the destination account number (*destAcc*), the amount of money to be transferred (*amt*) and the currency (*currency*). The bank checks with the destination bank whether this wire request is valid i.e., if the account number is valid and reachable, the currency is permis-

sible and the wire is allowed without any special restrictions. Once the destination bank sends a *wireReqValid* message to the bank to proceed with the transaction, the bank sends a *quoteForWire* to the customer informing him about the charge (*wiringCharge*) and time (*time*) required for this transaction. If the customer accepts the quote (*acceptQuote*), a commitment is created between the bank and the customer specifying that if the customer pays the *wiringCharge*, then the bank is obliged to wire the money to the destination account as requested by the customer, within the agreed time frame. Then the bank wires the money and sends a *wireSuccess* message to the customer.

However sending out the money to the destination bank does not ensure that the transfer was completed successfully. There might be unforeseen situations like a new regulation that prohibits money transfer from the source country to the destination country, or the destination account being closed while the transfer is in progress, or insufficient balance in the customer's account on the day the money was to be transferred. Such circumstances are not the norm and hence are not included as a part of the protocol specification. This is where event monitoring comes in handy. We demonstrate the Money Wiring Protocol with events.

The Bank's perspective of the Money Wiring Protocol is now shown in Figure 5.5. Since the bank is committed to carrying out the wire transfer when the payment is made, it needs to ensure that the wiring took place successfully. Thus it introduces event monitoring once the commitment was created. The wiring takes place through several points. Let the original bank be bank *A* located in country *a*. First the money is sent out by the bank's headquarter *A'* located in country *a*. Now let us assume that the destination bank *B* is a local bank for country *b* and has no international branches. In order to reach *B*, *A* has to go through a bank *C* which is country *b*'s most well-networked bank and is easily reachable by country *a*. Thus *C* is the intermediate bank which forwards the transfer from *A'* to the destination bank's headquarter *B'*. *B'* will then transfer the money to *B* which is the local branch of our destination bank, where the account exists.

Thus bank *A* needs to track the wire through *BankHeadquarter (A')*, *IntermediateBank (C)*, *DestinationBank (B')* and *DestinationBankLocalBranch (B)*. If all of them return a *transferSuccess*, bank *A* will be ensured that the transfer went through and will then notify the customer that the wire was successful (*wireSuccess*). With this complex event

monitoring it will be easier to detect situations where things went wrong.

### 5.3.1 Detecting an Unexpected or Unanticipated Exception

Consider a scenario in which, while the transfer was being made, the destination account expired or closed. This information will be known by bank  $A$  only when it reaches the destination bank  $B'$ . Then it does not need to proceed to track the transfer any further and notifies the customer of a failure with the reason that caused it. The customer may choose to rewire to a different account. The consequence is shown in Figure 5.5. Another instance of an unanticipated exception would be a change in regulation that does not allow money wiring between the two countries that happens after the commitment to wire has been made and the money has been transferred to the intermediate bank. The exception could be a general exception to denote a wire failure by sending a generic *wireFailure* message to the customer or if the bank wishes to be specific, it may monitor separate exceptions pertaining to the failure of wiring at different agents and return not only a failure message but more information about the failure.

The events, exceptions and rules corresponding to this scenario are listed below:

$$\begin{aligned}
E &= \text{acceptQuote} \cdot A' \wedge A' \cdot C \wedge C \cdot B' \wedge B' \cdot B \\
E &\Rightarrow \text{send}(\text{Customer}, \text{wireSuccess}) \\
\overline{E} &= \overline{\text{acceptQuote}} \vee \overline{A'} \vee \overline{B'} \vee \overline{C} \vee \overline{B} \vee C \cdot A' \vee B' \cdot C \vee B \cdot B' \\
E' &= \overline{\text{acceptQuote}} \vee C \cdot A' \vee B' \cdot C \vee B \cdot B' \\
E_{ex} &= \overline{A'} \vee \overline{B'} \vee \overline{C} \vee \overline{B} \\
E_{ex} &\Rightarrow E_{ex}Policy \\
E_{ex}Policy &: \text{send}(\text{Customer}, \text{wireFailure}) \\
A_{ex} &= \overline{A'} \\
A_{ex} &\Rightarrow \text{send}(A', \text{retryWire}) \\
B'_{ex} &= \overline{B'} \\
B'_{ex} &\Rightarrow \text{send}(\text{Customer}, \text{failureAtB'}(\text{accountExpiredMsg}))
\end{aligned}$$

### 5.3.2 Detecting an Expected or Anticipated Exception

Consider a scenario in which, the Destination bank fails to identify the account to which the transfer is to be made. Hence, it sends a *wireReqInvalid* message to the bank. The

bank then informs the customer that its request was invalid. This kind of an exception may be expected and could arise due to factors like wrong input by the customer. Once the bank informs the customer of the invalid request, the customer may wish to retry or cancel the request as he pleases. This is shown in Figure 5.6.

The events, exceptions and rules corresponding to this are given below:

$$\begin{aligned}
E &= \text{wireValidityReq} \cdot \text{wireReqValid} \\
E \wedge \text{QuotePolicy} &\Rightarrow \text{quoteForWire} \\
\overline{E} &= \overline{\text{wireValidityReq}} \vee \overline{\text{wireReqValid}} \vee \text{wireReqValid} \cdot \text{wireValidityReq} \\
E' &= \overline{\text{wireValidityReq}} \vee \text{wireReqValid} \cdot \text{wireValidityReq} \\
E_{ex} &= \overline{E} - E' = \overline{\text{wireReqValid}} \\
E_{ex} &\Rightarrow E_{ex}Policy \\
E_{ex}Policy &: \text{send}(\text{Customer}, \text{invalidReq})
\end{aligned}$$

## 5.4 Use Case 3: Repair Protocol

Let us consider the auto insurance claim scenario described by Desai *et al.* [2006a]. In the Repair protocol as shown in Figure 5.7, the *Repairer* sends a *repaired* message to the *Owner*, on receiving a *repairReq* from the owner and once the repair is done. In order to know that the repair was completed, he might need to track the repair through several intermediate points. For example, the repair might involve an engine repair to be carried out by the car company, followed by tyre replacement to be carried out by a Tyre and Lube Department of his own organization, followed by cosmetic changes to its body carried out by a body shop. The successful completion of all these steps in sequence will complete the repair. Hence the Repairer will send the repaired message only once he has tracked the repair successfully through the various steps by interacting with different parties responsible for those steps. This is the Repairer's perspective of the Repair protocol and it introduces the complex event

$$\text{RepairedEvent} = \text{EngineRepaired} \cdot \text{TyresReplaced} \cdot \text{BodyRepaired}$$

The Repaired event hence captures a detailed view of how the repair is carried out. If any of the intermediate steps fail or are delayed, the owner would be notified. Say the owner has rented a car for a period of time specified by the repairer and a delay occurs, he can



extend his rental period. Such flexibility would not be there if events were not monitored and the agents adhered only to the protocol rules. In another instance of the same Repair protocol, the responsibility of tracking the repair through the different steps might fall on the Owner instead of the repairer. In that case, the Repairer would send him information about the repair process and it would track it with the different parties. Once the repair was completed successfully, it would send the repairOK message to the Repairer who would consequently inform the consultant.

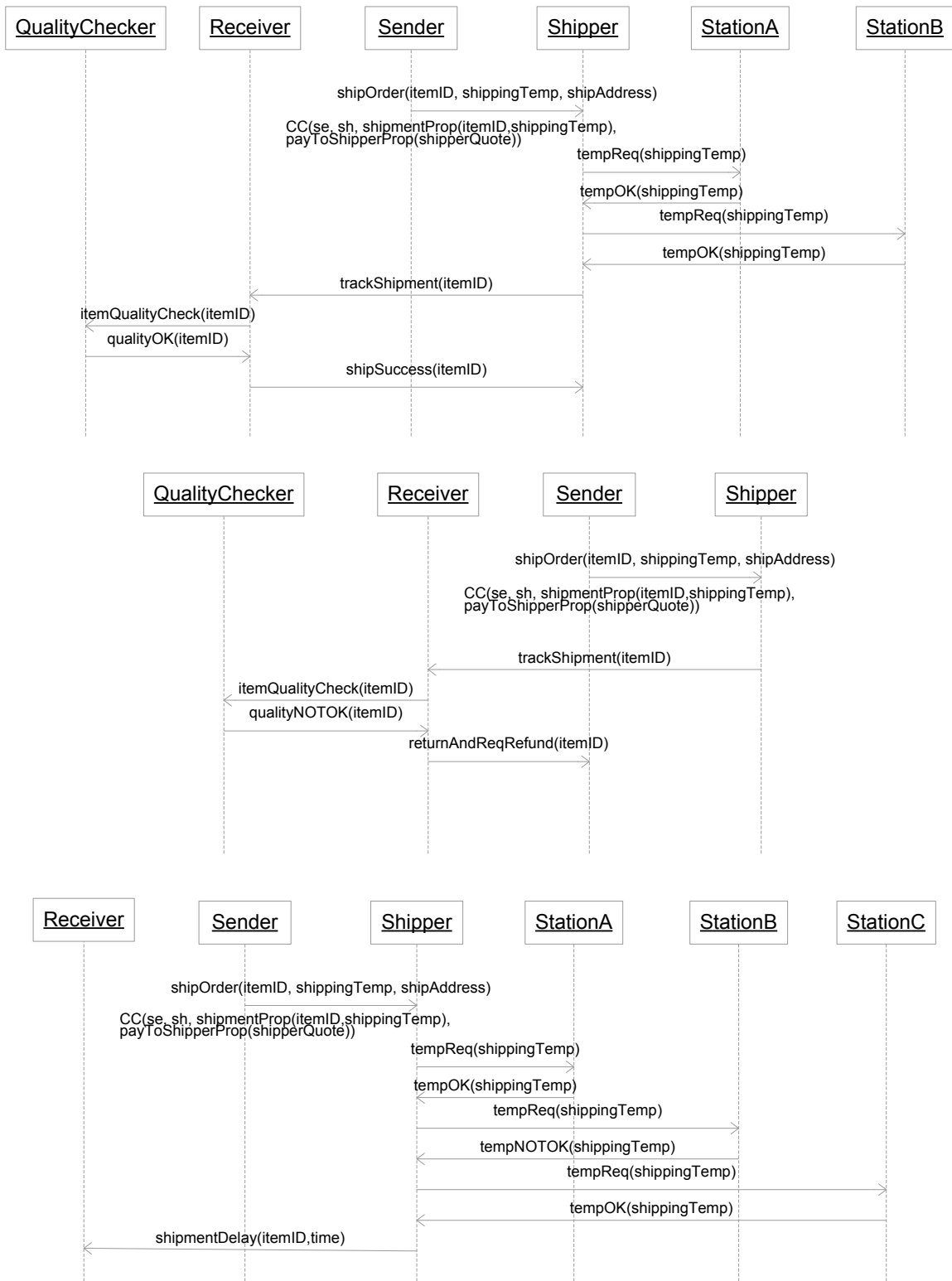


Figure 5.2: Handling conditions and violations: Shipping Protocol under Special Conditions for Perishable Items

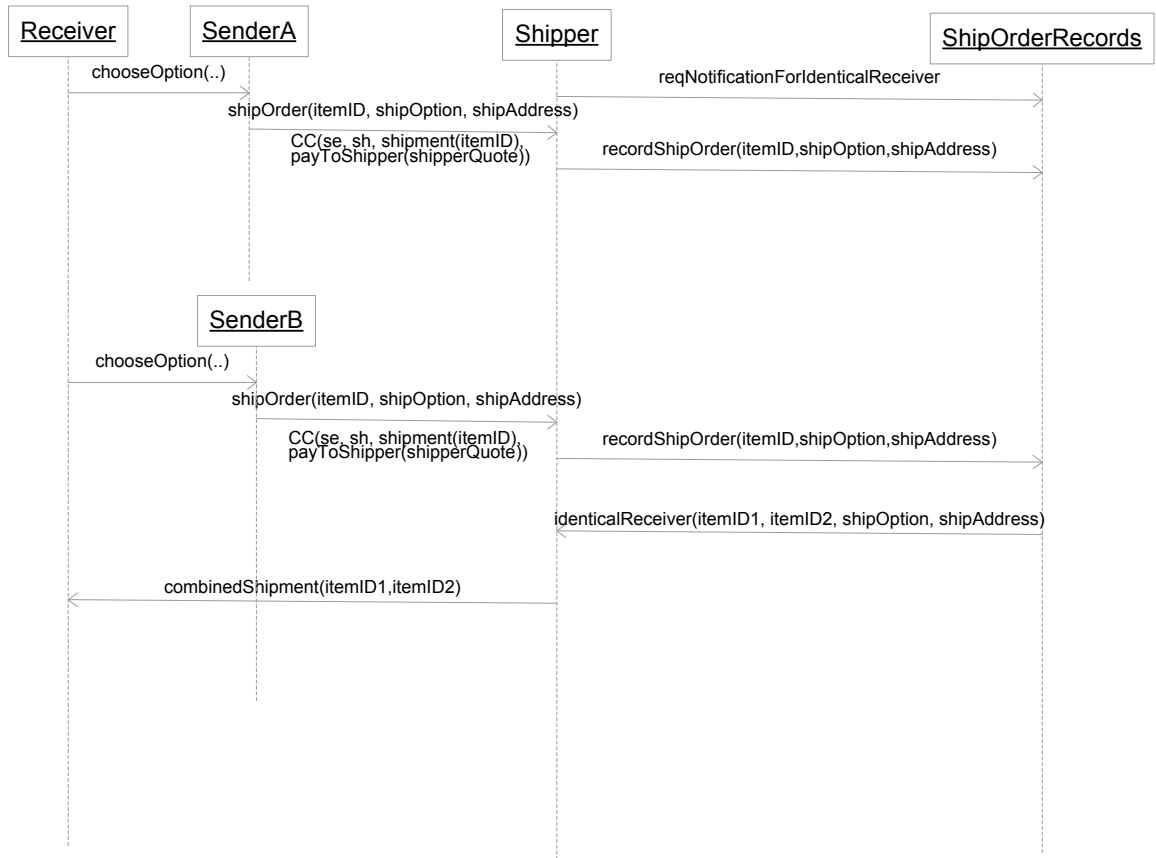


Figure 5.3: Exploiting an opportunity: Combining orders to be shipped

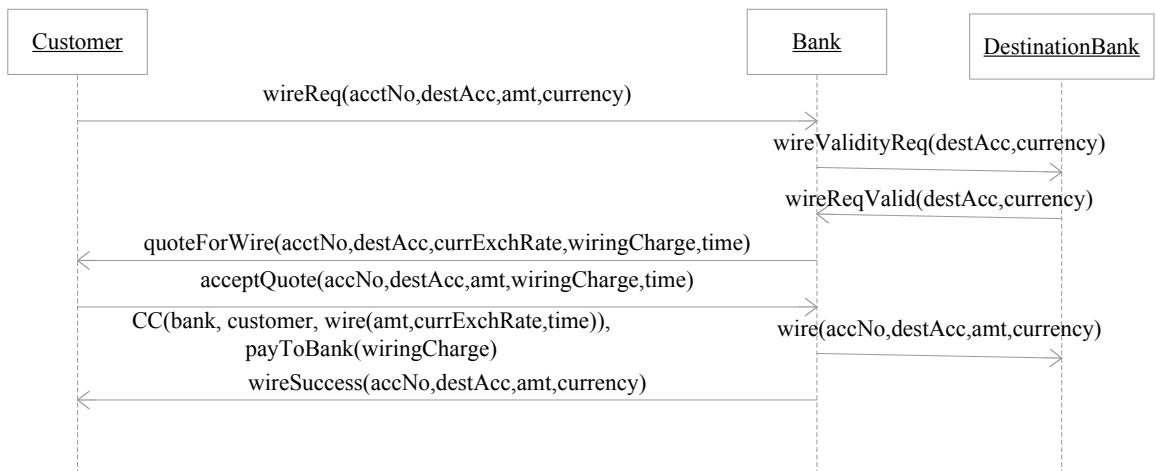
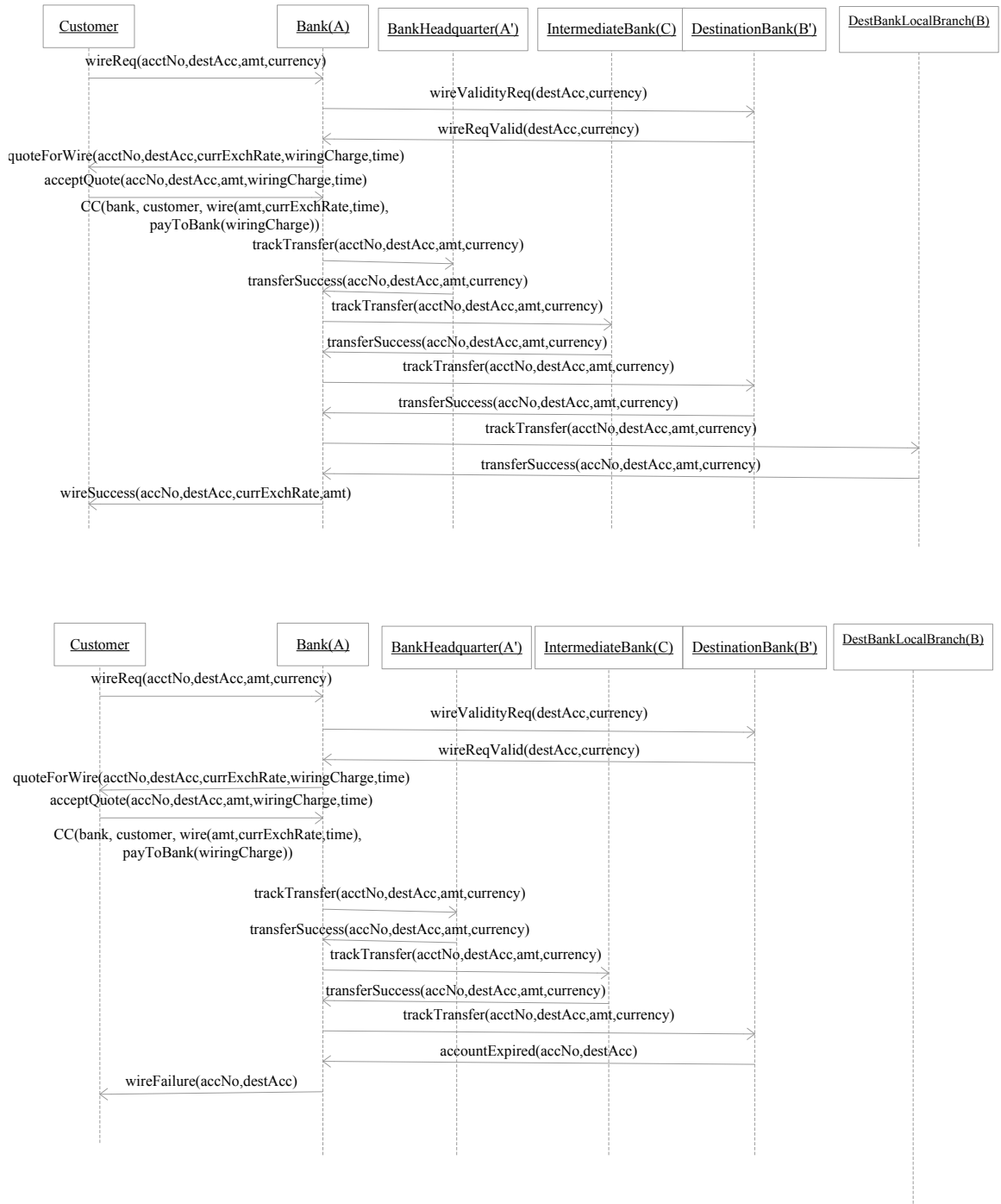


Figure 5.4: Money wiring protocol

Figure 5.5: Handling an unexpected exception from *Bank*'s perspective

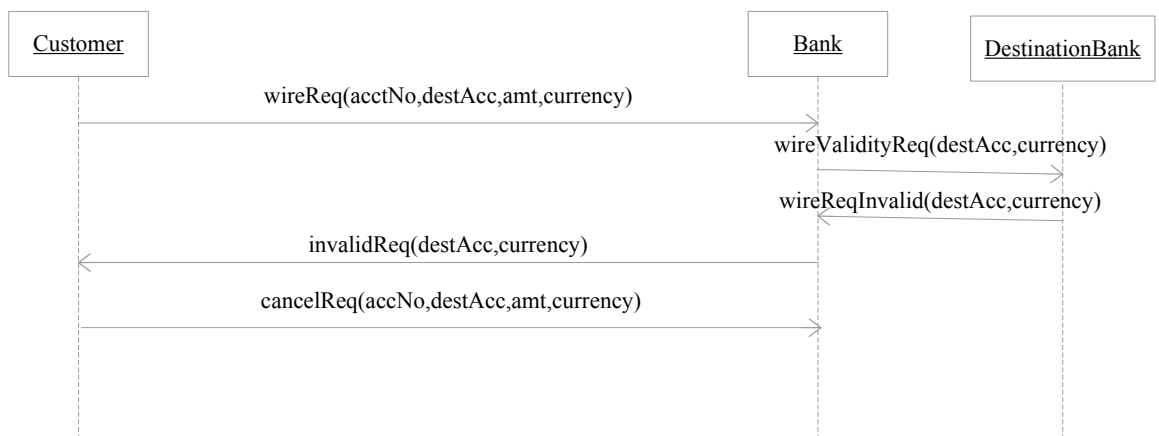
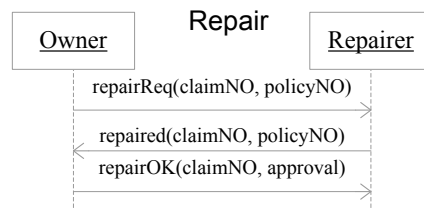


Figure 5.6: Handling an expected exception in money wiring protocol



### Repairer Perspective of Repair Protocol

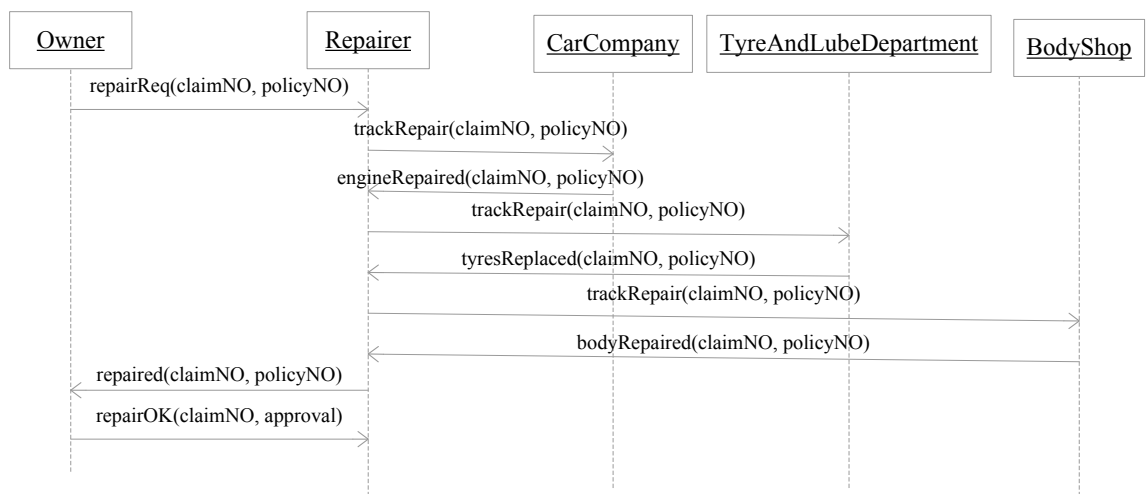


Figure 5.7: Repair protocol

## Chapter 6

### Discussion

Agents involved in business process enactment in large, distributed environments need to be sensitive to state changes or events. They should be able to detect exceptions and recover from them efficiently and spontaneously to prevent the enactment process from remaining in a state of error. High-level interactions between agents, in accordance with protocol specifications, are not sufficient to capture the fine-grained details of how an interaction is carried out and whether it was successful or not. We introduced event monitoring into agent-based business process enactment in order to ensure processes are enacted smoothly. A high-level interaction of a business protocol is split into several low-level interactions, which if they occur sequentially, will result in the successful completion of the high-level interaction. Hence high-level interactions have been modeled as complex events constituting low-level interactions or simple events.

The simple events that may lead to the occurrence of the complex event and the sequence that they might follow, can vary from agent to agent and instance to instance. Since commitments are significant points during the enactment of the process, event monitoring can be introduced for any commitment. The discretion is, of course, left to the agent. It will base its decision to monitor a commitment on several factors. For example, if the agent is a merchant who is committed to shipping some goods, he might choose to monitor scenarios where the customer is a privileged customer whose business is important to him, or if it is a time critical shipment or if there are some special instructions for the order like fragile goods or temperature-specific transport of goods. The different role-specific views of the

same commitment in a protocol, which forms the basis of the decision to monitor events, is called a perspective. Different perspectives of the Shipping Protocol have been discussed to demonstrate when and how events can be injected into the process enactment.

Exceptions are handled by activating private policies of the agent. Some of the actions that the agent might take in order to handle the exception are to invoke a new protocol thereby leading to nested protocol instantiation, or send messages other participating agents to notify of the exception. By recovering from the exception this way, the enactment process is brought to a stable state even if it is deviated from its routine path.

This thesis described an approach for agent-based business process enactment that includes fine-grained event monitoring. This results in a more proactive and robust agent, capable of detecting and handling exceptions beforehand resulting in a more reliable distributed system. We described an event-driven agent architecture and an event processing language that incorporates event logic into business protocol rules. We have placed this architecture in the protocol-based business process framework of OWL-P. A typical business protocol has been modified to accommodate events. We have designed algorithms to help a designer derive complex event patterns, to manage subscriptions to simple events and to monitor processes. We have also discussed role specific perspectives that helps in choosing event monitoring points during process enactment. Agents built using this architecture track complex patterns of events and exceptions by aggregating simple events in stream and generate rules in order to react to exceptions automatically.

## 6.1 Related Work

Agent-based business process modeling and enactment have been around for some-time. ADEPT [Jennings et al., 1996] and APMS [O'Brien and Wiegand, 1998] introduced multiagent systems coordinating and negotiating to achieve business goals via workflows. Protocol-based business process modeling proposed by Desai *et al.* [2004] was an improvement in terms of preserving autonomy since it separated the concerns for modeling and enacting the process. Mallya and Singh [2005] proposed an approach to handle protocol exceptions by identifying preferable protocol runs. Chopra and Singh [2006] discuss how



to make interactions in protocol-based business processes compliant. They propose ways to introduce additional messages into the protocol to accommodate deviations from their routine path, and to ensure that despite these deviations, the interactions of the agents are compliant with the protocol. However none of these approaches handle exceptions and opportunities by monitoring business events which may be the result of external as well as internal factors. The derivation of perspectives based on an agent's commitments and ensuring that the commitments are complied with by introducing fine-grained monitoring of the interactions, are a step further to making such systems flexible, reliable and compliant. Moreover, the sources of events are not restricted to the messages received from agents participating in the protocol. Additional information about the status of an interaction can be obtained from sensors, other applications like timers and databases, and other agents that are not a part of the protocol. These additional interactions and deviations from the normal execution path may also lead to nested instantiation of protocols.

Exception-handling in multagent enactment models with the help of ECA rules have been proposed by Brocks *et al.*[2005]. We incorporate complex events, formalized by temporal logic, into ECA rules to capture occurrences of events ordered in time.

Complex event processing and event-driven architecture have gained popularity in recent times through the works of Luckham [Luckham, 2002] and Chandy [Chandy, 2005]. Wang *et al.*[2005, 2002, 2002] have proposed and implemented multiagent systems that work toward monitoring business processes and workflow. However in none of these approaches, the agent enacting the business process, is itself event-driven and its monitoring decisions are based on their contractual obligations. Also, none of these architectures talk about derivation of exception patterns from complex event patterns. Subscription management to low-level events in order to optimize the resources used to set up subscriptions, is also an improvement in performance compared to current approaches.

## 6.2 Future Directions

In the current approach, perspectives on commitments are derived manually by the designer since it requires domain knowledge. The choice of the specific points in the process

where monitoring can be injected, are decided by the designer at configuration time. By studying the policies and business goals, these decisions can be automated to a certain extent. Making monitoring decisions dynamically at runtime is a possible enhancement of this work.

## Bibliography

- Elli Albek, Eric Bax, Greg Billock, K. Mani Chandy, and Ian Swett. An event processing language (EPL) for building sense and respond applications. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.
- BPEL. Business process execution language for web services, version 1.1, May 2003. [www-106.ibm.com/developerworks/webservices/library/ws-bpel](http://www-106.ibm.com/developerworks/webservices/library/ws-bpel).
- Holger Brocks, Henning Meyer, and Thomas Kamps. Flexible exception handling in a multi-agent enactment model for knowledge-intensive processes. In *IAT '05: Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, pages 479–482, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2416-8. doi: <http://dx.doi.org/10.1109/IAT.2005.76>.
- K. Mani Chandy. Sense and respond systems. In *Proceedings of the 31st International Computer Measurement Group Conference*, pages 59–66, 2005.
- Amit K. Chopra and Munindar P. Singh. Producing compliant interactions: Conformance, coverage, and interoperability. In *Declarative Agent Languages and Technologies IV (DALI), 4th International Workshop*, pages 1–15, 2006.
- F. Cristian. *Dependability of Resilient Computers*. Blackwell Scientific Publications, 1989.
- Nirmit Desai and Munindar P. Singh. Protocol-based business process modeling and enactment. *Second IEEE International Conference on Web Services*, page 35, 2004. doi: <http://doi.ieeecomputersociety.org/10.1109/ICWS.2004.1314721>.
- Nirmit Desai, Ashok U. Mallya, Amit K. Chopra, and Munindar P. Singh. Interaction protocols as design abstractions for business processes. *IEEE Transactions on Software Engineering*, 31(12):1015–1027, December 2005.
- Nirmit Desai, Amit K. Chopra, and Munindar P. Singh. Business process adaptations via protocols. In *IEEE International Conference on Services Computing (SCC)*, pages 103–110, Los Alamitos, CA, USA, 2006a. IEEE Computer Society. ISBN 0-7695-2670-5. doi: <http://doi.ieeecomputersociety.org/10.1109/SCC.2006.30>.

- Nirmit Desai, Ashok U. Mallya, Amit K. Chopra, and Munindar P. Singh. OWL-P: A methodology for business process development. *Agent-Oriented Information Systems III, Lecture Notes in Computer Science*, 3529:79–94, 2006b. doi: 10.1007/11916291.
- Johann Eder and Walter Liebhart. The workflow activity model (WAMO). In *Conference on Cooperative Information Systems*, pages 87–98, 1995.
- Ernest J. Friedman-Hill. *Jess in Action: Rule-Based Systems in Java*. Manning, Greenwich, CT, 2003.
- Michael N. Huhns and Munindar P. Singh, editors. *Readings in Agents*. Morgan Kaufmann, San Francisco, 1998.
- IBM Corp. Websphere application server. <http://www-306.ibm.com/software/webservers/appserv/was/>.
- Nicholas R. Jennings, Peyman Faratin, M. J. Johnson, Timothy J. Norman, P. O’Brien, and M. E. Wiegand. Agent-based business process management. *International Journal of Cooperative Information Systems*, 5(2&3):105–130, 1996.
- David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, Cambridge, MA, 2002.
- Ashok U. Mallya. *Modeling and Enacting Business Processes via Commitment Protocols Among Agents*. PhD thesis, North Carolina State University, Raleigh, NC, USA, 2006.
- Ashok U. Mallya and Munindar P. Singh. Introducing preferences into commitment protocols. *Proceedings of the 4th International Conference on Autonomous Agents and Multiagent Systems(AAMAS), Workshop in Agent Oriented Software Engineering (AOSE)*, 2005.
- Jean-Louis Maréchaux. Combining service-oriented architecture and event-driven architecture using an enterprise service bus, 2006. IBM DeveloperWorks.
- P. D. O’Brien and M. E. Wiegand. Agent based process management: applying intelligent agents to workflow. *Knowledge Engineering Review*, 13(2):161–174, 1998.
- Munindar P. Singh. Distributed enactment of multiagent workflows: Temporal logic for service composition. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 907–914. ACM Press, July 2003.
- Munindar P. Singh. An ontology for commitments in multiagent systems: Toward a unification of normative concepts. *Artificial Intelligence and Law*, 7:97–113, 1999.

- Munindar P. Singh and Michael N. Huhns. *Service-Oriented Computing: Semantics, Processes, Agents*. John Wiley & Sons, Chichester, UK, 2005.
- Sun Microsystems. JMS: Java messaging service, 2005a. <http://java.sun.com/products/jms/>.
- Sun Microsystems. Java Regex: Java 5.0 regular expressions, 2005b. <http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/package-summary.html>.
- Mahadevan Venkatraman and Munindar P. Singh. Verifying compliance with commitment protocols: Enabling open Web-based multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 2(3):217–236, September 1999.
- Feng Wan and Munindar P. Singh. Formalizing and achieving multiparty agreements via commitments. In *Proceedings of the 4th International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 770–777. ACM Press, July 2005.
- Huaiqing Wang, John Mylopoulos, and Stephen Liao. Intelligent agents and financial risk monitoring systems. *Communications of the ACM*, 45(3):83–88, 2002. ISSN 0001-0782. doi: <http://doi.acm.org/10.1145/504729.504733>.
- Minhong Wang, Huaiqing Wang, and Dongming Xu. The design of intelligent workflow monitoring with agent technology. *Knowledge Based Systems*, 18(6):257–266, 2005.
- Dongming Xu and Huaiqing Wang. Multi-agent collaboration for B2B workflow monitoring. *Knowledge Based Systems*, 15(8):485–491, 2002.
- Liangzhao Zeng, Hui Lei, Jun-Jang Jeng, Jen-Yao Chung, and B. Benatallah. Policy-driven exception-management for composite web services. *Seventh IEEE International Conference on E-Commerce Technology*, pages 355–363, 2005.

## APPENDICES

# Appendix A

# Appendix A

## A.1 Rules for Shipper in Running Example

```

(defrule shipInfoRulefor_shipper
  (and(startMsg (transID ?t1)) (shipInfo_Policy
    (shipAddress ?shipAddress-var1) ))
  =>
    (bind ?shipInfo-var (new shipInfo ?shipAddress-var1 ?t1))
    (definstance shipInfoMsgProp ?shipInfo-var static))

(defrule reqForShipOptionsRule_for_shipper
  (and(shipInfoMsgProp (shipAddress ?shipAddress-var1)(transID ?t1))
    (reqForShipOptionsMsg (shipAddress ?shipAddress-var2)
    (item ?item-var2)(transID ?t2)) (reqForShipOptions_Policy))
  =>
    (bind ?reqForShipOptions-var (new reqForShipOptions
    ?shipAddress-var2 ?item-var2 ?t2))
    (definstance reqForShipOptionsMsgProp
    ?reqForShipOptions-var static))

(defrule senderOptionQuoteRule_for_shipper
  (and(shipperOptionQuoteMsgProp (shipperQuote ?shipperQuote-var1)
    (shipOption ?shipOption-var1)(transID ?t1))
    (senderOptionQuote_Policy (senderQuote ?senderQuote-var1)))
  =>
    (bind ?cc-var(new JJConditionalCommitment "itemSender"
    "itemReceiver" "payToSender" "shipmentProp" ?t1))
    (definstance ccom ?cc-var static)
    (bind ?senderOptionQuote-var (new senderOptionQuote
    ?senderQuote-var1 ?shipOption-var1 ?t1))
    (definstance senderOptionQuoteMsgProp
    ?senderOptionQuote-var static))

(defrule shipperOptionQuoteRule_for_shipper

```

```

(and (reqForShipOptionsMsgProp (item ?item-var1) (shipAddress
?shipAddress-var1)
(transID ?t1))
(shipperOptionQuote_Policy (shipperQuote ?shipperQuote-var1)
(shipOption ?shipOption-var1) ))
⇒
(bind ?cc-var (new JJConditionalCommitment "shipper" "itemSender"
"payToShipper" "shipmentProp" ?t1))
(definstance ccom ?cc-var static)
(bind ?shipperOptionQuote-var (new shipperOptionQuote
?shipperQuote-var1 ?shipOption-var1 ?t1))
(definstance shipperOptionQuoteMsgProp
?shipperOptionQuote-var static)
(call JMSWrapper sendTo (new java.lang.String
"jms/Shipping/itemSenderQ") ?shipperOptionQuote-var)
(printout t "Shipper sending shipperOptionQuote to Sender"))

(defrule chooseOptionRule_for_shipper
(and (senderOptionQuoteMsgProp (senderQuote ?senderQuote-var1)
(shipOption ?shipOption-var1) (transID ?t1)) (chooseOption_Policy))
⇒
(bind ?chooseOption-var (new chooseOption ?shipOption-var1
?senderQuote-var1 ?t1))
(definstance chooseOptionMsgProp ?chooseOption-var static)
(bind ?cc-var (new JJConditionalCommitment "itemReceiver"
"itemSender" "shipmentProp" "payToSender" ?t1))
(definstance ccom ?cc-var static))

(defrule shipOrderRule_for_shipper
(and (chooseOptionMsgProp (senderQuote ?senderQuote-var1)
(shipOption ?shipOption-var1) (transID ?t1))
(shipOrderMsg (item ?item-var2) (shipAddress ?shipAddress-var2)
(pickAddress ?pickAddress-var2) (shipOption ?shipOption-var2)
(transID ?t2)) (shipOrder_Policy))
⇒
(bind ?shipOrder-var (new shipOrder ?item-var2 ?shipAddress-var2
?pickAddress-var2 ?shipOption-var2 ?t2))
(definstance shipOrderMsgProp ?shipOrder-var static)
(assert (item_prop (item ?item-var2)))
(assert (transID_prop (transID ?t1)))
(bind ?cc-var (new JJConditionalCommitment "itemSender" "shipper"
"shipmentProp" "payToShipper" ?t2))
(definstance ccom ?cc-var static))

(defrule shipmentRule_for_shipper
(and (shipmentEvent) (item_prop (item ?item-var1))
(transID_prop (transID ?t1)))
⇒

```



```
(bind ?shipment-var (new shipment ?item-var1 ?t1))
(definstance shipmentMsgProp ?shipment-var static)
(call JMSWrapper sendTo
 (new java.lang.String "jms/Shipping/itemReceiverQ")
 ?shipment-var)
(printout t "Shipper sending shipment to Receiver"))
```

## A.2 Complex Events for Shipper

```
shipmentEvent=(shipOrder.checkpoint1Success)&
(checkpoint1Success.checkpoint2Success)
shipmentException=(checkpoint1Failure |
checkpoint2Failure)
```

## A.3 Exception Handling Policy for Shipper

```
(defrule shipmentException_Policy
(and (shipmentException) (itemProp (item ?item-var1))
(transIDProp (transID ?t1)))
⇒
(bind ?shipmentDelay-var (new shipmentDelay ?t1 ?item-var1))
(call JMSWrapper sendTo (new java.lang.String
"jms/Shipping/itemReceiverQ") ?shipmentDelay-var)
(printout t "EXCEPTION DETECTED:
Shipper sending shipmentDelay to Receiver"))
```

## A.4 Log Records for Shipper

```
<!-- INITIALIZATION -->
Logger initialized
JessWrapper initialized
JMSWrapper initialized
Jess rule system initialized
Event Processor configured
shipper agent kicked-off
<!-- INITIALIZATION -->

shipper received:start
Stored:startMsg to KB

shipper received:reqForShipOptions from itemSender
Stored:reqForShipOptionsMsg to KB
```

shipper received:shipOrder from shipper  
 Stored:shipOrderMsg to KB

Complex event shipmentEvent=  
 (shipOrder.checkpoint1Success)&(checkpoint1Success.checkpoint2Success)  
 contains simple event shipOrder  
 E\_new = (checkpoint1Success)&(checkpoint1Success.checkpoint2Success)  
 Complex event shipmentEvent not matched yet  
 New event after residuation:  
 shipmentEvent=(checkpoint1Success)&  
 (checkpoint1Success.checkpoint2Success)  
 Replaced old event with new event  
 Events in list: checkpoint1Success  
 Events in list: checkpoint1Failure  
 Events in list: checkpoint2Failure  
 Subscribing to checkpoint1  
 Subscribing to checkpoint2

shipper received:checkpoint1Success  
 Stored:checkpoint1SuccessMsg to KB

Complex event shipmentEvent=(checkpoint1Success)&  
 (checkpoint1Success.checkpoint2Success)  
 contains simple event checkpoint1Success  
 E\_new = (checkpoint2Success)  
 Complex event shipmentEvent not matched yet  
 New event after residuation:  
 shipmentEvent=(checkpoint2Success)  
 Replaced old event with new event  
 Events in list: checkpoint2Success  
 Events in list: checkpoint1Failure  
 Events in list: checkpoint2Failure

shipper received:checkpoint2Failure  
 Stored:checkpoint2FailureMsg to KB

Complex event shipmentException=  
 (checkpoint1Failure|checkpoint2Failure)  
 contains simple event checkpoint2Failure  
 E\_new = (1)  
 Complex event shipmentException matched  
 Matched Event shipmentException. Updating KB  
 Asserted:shipmentException to KB  
 EXCEPTION DETECTED: Shipper sending shipmentDelay to Receiver